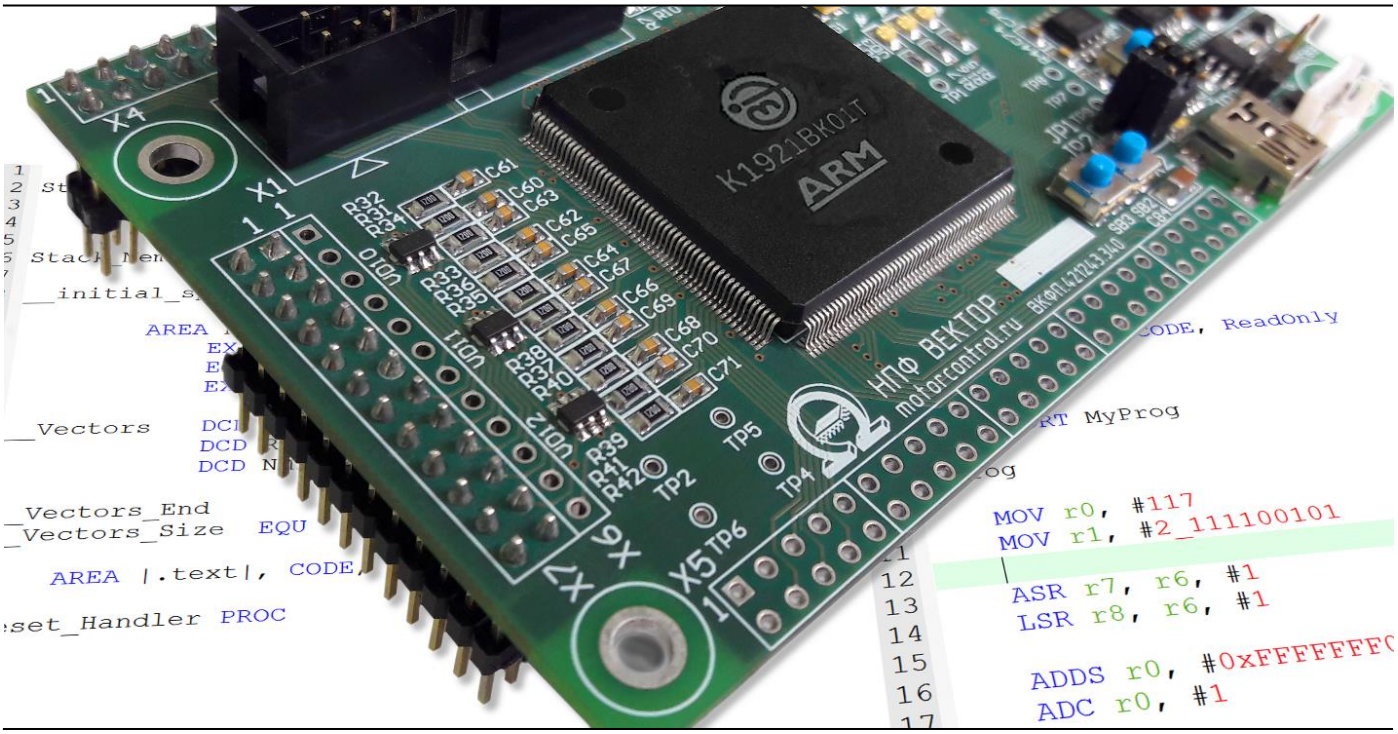
Козаченко В.Ф., Алямкин Д.И., Анучин А.С.,
Жарков А.А., Лашкевич М.М.,
Савкин Д.И., Шпак Д.М.



Практический курс

микропроцессорной техники на базе процессорных ядер ARM-Cortex-M3/M4/M4F

Архитектура, система команд, разработка и отладка программного обеспечения на Ассемблере в интегрированной среде Keil µVision

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МЭИ»

Практический курс микропроцессорной техники на базе процессорных ядер ARM-Cortex-M3/M4/M4F

по курсам «Микропроцессорные средства в электротехнике», «Компьютерная и микропроцессорная техника», «Микропроцессорная техника в электроприводе» и др. для студентов, обучающихся по направлениям «Электроэнергетика и электротехника» (13.03.02 – бакалавриат, 13.04.02 – магистратура), «Электроника и наноэлектроника» (11.03.04 – бакалавриат, 11.04.04 – магистратура).

Под общ. редакцией д.т.н., проф. В.Ф. Козаченко

Учебное пособие
Текстовое электронное издание

Москва
Издательство МЭИ
2019

УДК 621.398
ББК 32.973
П 692

Утверждено учебным управлением НИУ «МЭИ» в качестве учебного издания

Подготовлено на кафедре автоматизированного электропривода НИУ «МЭИ»

Рецензенты: докт. техн. наук, проф. каф. электротехники и промышленной электроники МГТУ им. Баумана А.Б. Красовский,
докт. техн. наук, проф. каф. АЭП НИУ «МЭИ» М.Г. Бычков

Авторы: В.Ф. Козаченко, А.С. Анучин, Д.И. Алямкин, А.А. Жарков, М.М. Лашкевич, Д.И. Савкин, Д.М. Шпак

П 692 Практический курс микропроцессорной техники на базе процессорных ядер ARM-Cortex-M3/M4/M4F [электронный ресурс]: учебное пособие – электрон. текстовые дан. (12 Мб) / В.Ф. Козаченко, А.С. Анучин, Д. И. Алямкин и др.; под общ. ред. В.Ф. Козаченко. – М.: Издательство МЭИ, 2019. – 1 электрон. опт. диск (CD-ROM).

ISBN 978-5-7046-2165-2

Практический интерактивный курс микропроцессорной техники для встраиваемых применений на базе микроконтроллеров с процессорными ядрами ARM-Cortex-M3/M4/M4F, выпускаемых в том числе отечественными предприятиями. Является одновременно учебником, сборником лабораторно-практических работ, самоучителем и справочником по архитектуре, системе команд и технологии разработки программного обеспечения на Ассемблере с использованием интегрированной среды разработки и отладки Keil μVision.

Ориентирован на разработчиков цифровых систем управления в энергетике, в транспорте, в станкостроении и робототехнике. Предназначен для студентов большинства электротехнических специальностей, в том числе: «Электропривод и автоматика», «Электрический транспорт», «Электрооборудование автономных объектов», «Промышленная электроника» и др.

Издано и опубликовано в авторской редакции.

Минимальные системные требования:

Компьютер: процессор x86 с тактовой частотой 500 МГц и выше; ОЗУ 512 Мб; 20 Мб на жестком диске; видеокарта SVGA 1280x1024 High Color (32 bit);
Операционная система: Windows XP/7/8 и старше
Программное обеспечение: Adobe Acrobat Reader версии 6 и старше.
Доступ к сети интернет.

Номер свидетельства о государственной регистрации: № 0321901060 от 12 апреля 2019 г.
ISBN 978-5-7046-2165-2

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ

ARM-процессоры для встроенного управления

Фирма ARM начала активное проникновение на рынок встраиваемых приложений и систем управления оборудованием с создания серии специализированных процессоров на базе ядра Cortex-M. Если процессоры ARM11, Cortex-A9 (семейство **Cortex-A**) больше ориентированы на рынок портативных цифровых устройств, таких как мобильные телефоны, планшеты, игровые приставки, где требуется большая вычислительная мощность, сравнимая с мощностью процессоров для современных компьютеров, при малом потреблении, то ядро **Cortex-M** предназначено для создания микроконтроллеров, *встраиваемых в промышленное оборудование* — от преобразователей частоты до роботов и автоматизированных линий. Третье направление развития процессоров ARM (семейство **Cortex-R**) ориентировано на создание высокопроизводительных систем реального времени для автомобильной промышленности, авиации и других, ответственных, применений, таких как оборудование энергетики и атомной промышленности.

Семейство Cortex-M появилось как альтернатива большому количеству 8- и 16-разрядных микроконтроллеров, присутствующих на рынке, преимуществом которого является существенный рост производительности вычислительных операций при сохранении низкого потребления электроэнергии. Типовые применения: управление силовыми преобразователями и источниками питания, электрическими двигателями, автомобильной электроникой, бытовыми устройствами, распределенное управление оборудованием «умных» домов, комплексная автоматизация производственных процессов и т.д.

На протяжении более чем двух десятилетий на рынке микроконтроллеров законодателями мод были хорошо известные изделия Intel 8051 и Motorola 68HC11/12/16. Начиная с 90-х годов значительное развитие получили 16-разрядные микроконтроллеры, в том числе сигнальные, которые имеют не только высокопроизводительное вычислительное ядро, но и широкий спектр встроенных на кристалл периферийных устройств. И вот, наконец-то, пришла очередь 32-разрядных микроконтроллеров завоевывать этот рынок. Фирма ARM поставила перед собой задачу создать процессорное ядро, которое одинаково хорошо будет работать и в системе управления холодильником или компрессором, и в современном мобильном телефоне.

Если тактовые частоты первых процессоров ARM были на уровне нескольких МГц, то сейчас они возросли на два и более порядка. Это позволило использовать архитектуру ARM в мобильных телефонах и портативных компьютерах, создав существенную конкуренцию традиционным производителям микропроцессоров, таким, как Intel. Оказалось, что мощный 32-разрядный процессор способен эффективно решать большинство задач, возникающих в сотовой связи — от интерфейса пользователя с клавиатурой и экраном до мониторинга состояния аккумулятора, декодирования MP3-музыки и поддержки компьютерных игр. И все это – при умеренном потреблении энергии.

По сути микропроцессоры ARM7 (встроенные впервые в планшет Apple iPod), кардинально изменили ситуацию на рынке мобильных устройств, сделав их доступными миллионам людей по всему миру.

На базе ARM7 и было разработано семейство процессорных ядер Cortex-M. Начиная с 2010 г. поставки микроконтроллеров этого семейства резко возрастают. Ожидается, что эта тенденция будет господствовать в текущем десятилетии, и к его концу традиционные 8- и 16-разрядные микроконтроллеры практически не будут использоваться в новых разработках. Не последняя роль при этим отводится языкам высокого уровня

C/C++, трансляторы с которых стали совершенными и доступными большинству разработчиков, что существенно ускорило процесс от начала разработки до выхода конечных изделий на рынок.

Если раньше использовались в основном однопроцессорные решения, то сейчас ситуация изменилась. Оказалось, что интеграция на кристалл сразу нескольких процессоров (от 2 до 8), возможно с разной архитектурой и встроенной периферией, позволяет предельно эффективно решать весь комплекс задач. Каждый процессор делает то, для чего оптимизирована его архитектура и система команд. Это направление особенно бурно развивается для использования в смартфонах, которые стали многофункциональными портативными компьютерами, в том числе для решения задач реального управления (например, оборудованием «умного» дома).

Мультипроцессорные системы получили название «Системы на кристалле» System-on-Chip (SoC) и могут содержать не только центральный процессор и сопроцессор поддержки вычислений с плавающей точкой, но и большое число системных периферийных устройств. Так, процессор фирмы Texas Instruments OMAP5 имеет на борту двухядерный процессор Cortex-A15 и два микроконтроллера Cortex-M4. При разработке программного обеспечения для таких сложных систем большое значение имеет унификация языков программирования отдельных процессоров, что может быть реализовано только в изделиях одного производителя.

Нужно ли современному инженеру знание архитектуры процессора, его системы команд, языка Ассемблер?

Резкое увеличение сложности процессорной техники ставит перед инженерами вопрос: «А нужно ли вообще изучать архитектуру и систему команд конкретного процессора, учиться программированию на Ассемблере? Может быть достаточно знать только язык высокого уровня СИ, чтобы эффективно работать?». Мы считаем, что актуальность знания основ процессорной техники только возрастает:

1. Не зная архитектуры и внутреннего устройства микроконтроллера невозможно создать на его основе микропроцессорный контроллер, встраиваемый в конкретное изделие, в котором будут эффективно использованы как возможности ядра процессора, так и встроенной в него периферии.
2. Пользуясь языком высокого уровня, Вы работаете с процессором как с «черным ящиком», который должен уметь делать все, что Вы для него запрограммировали. Однако, далеко не факт, что используемый в конкретном трансляторе режим работы процессора оптимален для Вашего приложения (например, используемый режим округления при работе с числами с плавающей точкой). Вы можете получить неожиданный результат и даже не понять в чем ошибка.
3. Система команд ARM-процессоров столь совершенна, что часто одной команды на Ассемблере достаточно для решения задачи, для которой на языке высокого уровня потребуется специализированная библиотека. Например, преобразование числа в формате с фиксированной точкой в число с плавающей точкой или обратно.
4. Работа с портами ввода/вывода и периферийными устройствами, встроенными на кристалл микроконтроллера, порой более эффективна на Ассемблере, чем на языке высокого уровня, особенно если для этого используется по-битово адресуемая память. Часто разработчики даже не подозревают, что используемый ими микроконтроллер имеет на борту «битовый сопроцессор», позволяющий предельно эффективно реализовывать логические контроллеры и дискретные управляющие автоматы.
5. Окончательная оптимизация микропроцессорной системы по быстродействию, объему используемой памяти и энергопотреблению возможна только на нижнем уровне. В

этом случае программа на языке C/C++ подвергается анализу с точки зрения наиболее узких мест и эти места оптимизируются с использованием макросов или подпрограмм, написанных на Ассемблере.

6. Не зная архитектуру процессора и его систему команд невозможно создать специальное программное обеспечение: компиляторы; драйверы периферийных устройств, специализированные библиотеки подпрограмм, например, для векторного управления электрическими двигателями.
7. Возможности современных процессоров и микроконтроллеров в части специальных команд цифровой обработки сигналов типа «умножение с накоплением» позволяют сегодня на Ассемблере в реальном времени решать в несколько раз более сложные системы дифференциальных уравнений, чем с использованием языка C/C++. Это открывает широкие возможности для использования принципиально новых методов цифрового управления оборудованием: с наблюдателями состояния, с прогнозом траектории движения и т.д.

Таким образом, инженер-аппаратчик и инженер-конструктор без знания основ процессорной техники не сможет оптимально проектировать прикладные устройства, а инженер-программист, который знает не только язык высокого уровня, но и архитектуру и систему команд процессора, будет тратить в разы меньше времени на оптимизацию кода и отладку конкретного проекта.

Итак, семейство микроконтроллеров с ядром Cortex-M4 в значительной степени опирается на архитектуру и систему команд ARM-процессора ARM7TDMI. Однако, в него включены существенные дополнения. В первую очередь – это команды *поддержки операций цифровой обработки сигналов и цифрового регулирования* как с числами в формате с фиксированной точкой, так и с плавающей точкой. Наличие математического сопроцессора позволяет во многих приложениях полностью отказаться от арифметики с фиксированной точкой в пользу арифметики с плавающей точкой — ускорить и упростить разработку и отладку программного обеспечения. Этим новым возможностям будет уделена значительная часть книги.

Что представляет собой эта книга?

Учебник для студентов

Книга написана на основе многолетнего опыта преподавания курсов «Компьютерная и микропроцессорная техника в электроприводе», «Микропроцессорные средства в электроприводе», «Системы управления электроприводов», «Электропривод с вентильными и шаговыми двигателями» в Национальном исследовательском университете «МЭИ» для студентов электротехнических специальностей, а также практического опыта создания серий цифровых систем управления электроприводов и силовых преобразователей для отечественной промышленности.

Предыдущие учебные курсы строились на основе 8- и 16-разрядных микроконтроллеров Intel, Motorola и 32-разрядных сигнальных микроконтроллеров Texas Instruments. Читателям предлагается курс микропроцессорной техники *на новой элементной базе* – 32-разрядных микроконтроллеров ARM с расширенными функциями обработки сигналов в форматах с фиксированной и плавающей точкой. Процессорные ядра ARM сегодня являются перспективными для использования во встроенных системах управления оборудованием, о чем свидетельствует разработка и начало серийного выпуска целого ряда отечественных микроконтроллеров на их основе.

Это учебник для студентов нескольких групп специальностей (в области электротехники, электроэнергетики, робототехники, автоматизации и др.), в задачу которых входит проектирование аппаратной части и программного обеспечения систем

цифрового управления оборудованием (от электрического двигателя/генератора, бензинового двигателя/дизеля, силового преобразователя энергии до электрического транспортного средства, энергетической установки, робота, станка, производственной автоматизированной линии).

Лабораторный практикум

Курс построен так, что теоретический материал сопровождается практической работой читателя с отладкой проектов в интегрированной среде разработки Keil µVision с использованием симулятора, то есть программно-логической модели процессора. Среда µVision для разработки и отладки микропроцессорных систем на базе ARM-ядер является общедоступной и может быть установлена на любой компьютер непосредственно с сайта фирмы ARM после процедуры регистрации.

Для более детального освоения материала книги вам так же могут понадобиться приложения к ней, которые распространяются бесплатно и доступны для свободного скачивания. Ознакомиться с кратким описанием этих приложений, а также узнать где можно скачать необходимые материалы, Вы можете в соответствующих разделах книги: приложение 1, приложение 2.

Все примеры программ, содержащиеся в книге, в виде готовых для выполнения проектов, представлены в приложении 2 к данной книге и могут быть немедленно запущены и отлажены. При этом преподаватель имеет возможность формирования списка лабораторно-практических работ с учетом специфики конкретной специальности: например, уделить больше времени программной реализации логических контроллеров и дискретных управляющих автоматов, или основам цифрового регулирования и цифровой фильтрации.

Работа за компьютером сопровождается вопросами и индивидуальными заданиями, позволяющими закрепить изучаемый материал и оценить уровень знаний. Все вопросы и задания снабжены вариантами возможных правильных ответов и решений.

Самоучитель

Микропроцессорная техника — это область знаний, которая очень быстро развивается. Поэтому, специалист, желающий оставаться в курсе ее последних достижений, должен постоянно самосовершенствоваться. Книга поможет в этом. Все, что нужно — это доступ к компьютеру, время и настойчивость. Любой желающий может работать самостоятельно дома, постепенно выполняя задания, расположенные в книге, отвечая на вопросы и сравнивая свои ответы с приведенными в книге.

Справочник

Книгой можно пользоваться как справочником по архитектуре и системе команд любых микроконтроллеров на базе процессорных ядер ARM-Cortex-M3/M4/M4F. Приложение к книге содержит подробное описание всех имеющихся команд процессорного ядра с графическими иллюстрациями алгоритмов выполнения наиболее сложных, комплексных команд, ограничениями и примерами использования команд на Ассемблере.

Курс разработки программного обеспечения на Ассемблере с использованием современных интегрированных сред разработки и отладки

В классическом варианте для описания языка Ассемблер, его директив и особенностей разработки программ на Ассемблере, а также для описания возможностей интегрированных сред разработки/отладки программного обеспечения должны быть изданы отдельные книги. Мы предлагаем другой путь: постепенно, по мере изучения системы команд, на *реальных практических примерах* знакомить читателя как с языком Ассемблера, включая его директивы, так и со средствами интерактивной разработки и отладки программ, включая возможности современных отладчиков (анализом содержимого регистров процессора, памяти, графическим отображением динамических процессов в логическом анализаторе и др.).

Это позволит читателю одновременно с изучением архитектуры и системы команд процессора освоить современную технологию разработки и отладки прикладного программного обеспечения.

Введение в микропроцессорную технику

Учебное пособие задумано и как «Введение в современную микропроцессорную технику» на базе процессоров ARM Cortex-M3/4/4F. Поэтому, оно содержит обзор наиболее употребительных терминов, краткие исторические сведения, сравнение разных типов процессорных архитектур, описание используемых в процессорной технике данных, основы двоичной арифметики и арифметики чисел с плавающей точкой. Система команд изучается не по алфавиту, а по функциональным группам, на примерах применения в реальных задачах. По мере освоения системы команд читатель получает необходимые *практические навыки*, отлаживая законченные программные модули и используя при этом как директивы Ассемблера, так и команды управления интегрированной средой отладки.

Перспективы

Данная книга предназначена для широкого круга читателей, поэтому специальные вопросы, такие как особенности создания цифровых систем управления электроприводов и источников питания, остались за кадром. Авторы планируют в ближайшее время подготовить к изданию отдельную книгу, которая включит в себя: описание возможностей системных и специализированных периферийных устройств микроконтроллеров на базе ARM-ядер Cortex-M3/M4/M4F; технику использования языка высокого уровня C/C++ и специализированных библиотек при разработке цифровых систем управления электрическими двигателями разных типов, управляемыми от силовых полупроводниковых преобразователей; методику совместного применения Ассемблера и C/C++; методы управления оборудованием в реальном времени по математическим моделям объекта с учетом специфики аппаратной части силовых преобразователей.

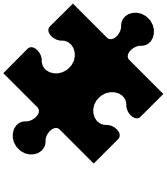
Авторский состав

Книга написана: Предисловие, гл.1,2,18 - совместно Анучиным А.С. и Козаченко В.Ф.; гл.5–6 – совместно Жарковым А.А. и Козаченко В.Ф.; гл.8 – совместно Козаченко В.Ф. и Лашкевичем М.М.; гл.9, 22 – совместно Козаченко В.Ф., Савкиным Д.И. и Шпак Д.М.; гл. 15 – совместно Алямкиным Д.И. и Козаченко В.Ф.; остальные главы – Козаченко В.Ф.

Рубрикация

Основной девиз книги – «От теории к практике через самостоятельную работу и текущий контроль знаний». Читатель может проверить ход усвоения материала сам на большом числе вопросов и заданий на самостоятельную проработку. Все вопросы и задания снабжены ответами, рекомендациями по поиску решения или примерами проектов, которые можно запустить и убедиться в их работоспособности.

Для удобства работы с книгой используются пиктограммы:

	Теория		Выводы. Важная информация
	Контрольные вопросы		Дополнительная информация для любознательных
	Ответы		Комментарии и пояснения к основному материалу
	Практическая работа. Приглашение в лабораторию или к самостоятельной работе дома	***ARM ASM***	Основные положения языка Ассемблер, в том числе директивы Ассемблера
	Индивидуальные практические задания с проверкой на компьютере		Техника работы в интегрированной среде Keil µVision5

Процессоры Cortex-M4 являются дальнейшим развитием ARM7TDMI, и при самостоятельной работе Вы можете использовать соответствующую документацию и технические руководства, как с сайта фирмы ARM, так и с сайтов фирм производителей микроконтроллеров (Texas Instruments, STMicroelectronics и др.).

Авторы благодарят сотрудников кафедры Автоматизированного электропривода НИУ «МЭИ», а также сотрудников ООО «Научно-производственная фирма ВЕКТОР» за помощь в подготовке книги к изданию. Особая благодарность – инженерам «Научно-исследовательского института электронной техники» (НИИЭТ, г. Воронеж), создавшим первую отечественную линейку микроконтроллеров на базе ARM-Cotrex-M4F со специализированной периферией для управления электродвигателями и силовыми преобразователями энергии.

Все замечания и рекомендации будут с благодарностью приняты авторами: kozachenkovf@yandex.ru, anuchin.alecksey@gmail.com, savkindmi@mpei.ru.

1 ПОЧЕМУ ARM?

Оглавление

1.1 Процессоры для компьютеров и встраиваемых применений

Развитие и совершенствование микропроцессорной техники на протяжении последних лет шло по двум основным направлениям: 1) процессоры *для персональных компьютеров и серверов*; 2) процессоры и микроконтроллеры *для встраиваемых применений*, в том числе *для цифрового управления оборудованием*. Конечно, эта классификация очень условна, так как процессоры для компьютеров тоже встраиваются в компьютеры. Эти два направления до последнего времени почти не конкурировали между собой.

Процессоры для компьютеров были на порядки быстрее, допускали подключение памяти большого объема, имели эффективную систему команд, адаптированную для выполнения вычислительных операций, в том числе с плавающей точкой. Но и стоили они от нескольких десятков до нескольких сотен долларов, что было основным препятствием для их использования во встраиваемых применениях, вместе с большими размерами, большим тепловыделением и необходимостью использования большого числа дополнительных микросхем окружения. Бесспорным мировым лидером в области процессоров для компьютеров и серверов была и остается фирма Intel.

Процессоры для встраиваемых применений, в основном микроконтроллеры, выпускались множеством фирм-производителей (той же Intel, Texas Instruments, Motorola, Analog Devices и т.д.). Главным требованием к ним было низкое энергопотребление и маленькая цена - от нескольких долларов до нескольких десятков долларов.

Ситуация резко изменилась в последние два десятилетия, когда на рынке массово появились процессоры ARM и микроконтроллеры на основе ARM-ядер. Оказалось, что они имеют не только малое энергопотребление и малую цену, но и высокую производительность. Это стало решающим фактором для производства на базе ARM-процессоров телефонов, смартфонов, планшетов, иными словами – портативных переносных компьютерных устройств, высокопроизводительных, но одновременно мало потребляющих и дешевых.

Если до начала 2000-х годов фирма Intel практически монополизировала рынок процессоров для компьютеров и не ощущала особой конкуренции других производителей,

то с появлением высокопроизводительных ARM-процессоров стала серьезно задумываться о снижении как энергопотребления, так и стоимости своих процессоров, чтобы не потерять не только рынок предельно дешевых микроконтроллеров для задач встроенного управления оборудованием (что фактически уже произошло), но и рынок портативных компьютеров, который все в большей мере заполнялся изделиями на базе ARM-процессоров. По статистическим данным процессоры ARM сегодня используются уже в 99% всех телефонов и смартфонов, а их доля в портативных компьютерах и планшетах быстро растет.

Сегодня мы являемся свидетелями настоящей битвы двух ведущих процессорных архитектур за рынок портативных компьютеров и планшетов. Наверняка, такой серьезный производитель, как Intel, сделает все возможное, чтобы сохранить за собой не только рынок процессоров для настольных компьютеров и серверов, но и максимально освоить рынок процессоров для портативных компьютеров и планшетов. Появились даже сообщения о перспективе создания фирмой Intel процессоров с совмещенной архитектурой (x86+ARM), которые будут одинаково успешно выполнять программное обеспечение, созданное для обеих архитектур. Производители портативных компьютеров выпускают сегодня свои изделия как на процессорах Intel, так и на процессорах ARM, внимательно отслеживая характеристики всех новинок. Какие процессоры окажутся успешнее, покажет ближайшее будущее, но это будут обязательно высокопроизводительные, но, вместе с тем, малопотребляющие и дешевые процессоры.

Несколько иная ситуация сложилась в области микроконтроллеров для встраиваемых систем управления. Быстрый рост в этом сегменте рынка ARM-микроконтроллеров уже стал свершившимся фактом. Сегодня практически все ведущие производители микроконтроллеров, такие как Texas Instruments, Freescale, Atmel, NXP, Samsung, LG, Sony Ericsson, nVidia и другие, выпускают свои линейки микроконтроллеров с процессорными ядрами ARM, которые отличаются предельно низким энергопотреблением, малой ценой и одновременно высокой производительностью.

Интересно, что и фирма Intel более 20 лет сотрудничает с фирмой ARM в области создания новых процессорных архитектур, являясь в том числе крупнейшим владельцем лицензированной интеллектуальной собственности ARM. Более того, в 2016 г. Intel предложила заказчикам производить на своих заводах процессоры на базе самых передовых ядер ARM, в том числе ядер Cortex, причем по новейшему 10 нм техпроцессу, что должно обеспечить уникальные показатели процессоров и микроконтроллеров как по энергопотреблению, так и по производительности. Крупнейшим заказчиком ожидается фирма LG Electronics, производящая широкий спектр цифровых устройств разного назначения – от планшетов и телефонов до преобразователей частоты и стиральных машин.

1.2 Отличительные особенности компании ARM

ARM (Advanced RISC Machine – усовершенствованная RISC-машина) – это название процессорной архитектуры и одновременно название компании, ее разработавшей. Сама по себе RISC-архитектура (Reduced Instruction Set Computer – компьютер с сокращенным набором команд) изначально предполагала уменьшенный набор команд - не несколько сотен, а всего несколько десятков. По замыслу создателей, это упрощает аппаратную часть процессора, уменьшает число транзисторов на кристалле и, соответственно, цену и энергопотребление процессора. Однако, сложность трансляторов и компиляторов заметно возрастает, так как в такой архитектуре именно они выполняют основную часть работы, создавая из относительно простых команд – комплексные, более сложные команды. Только тогда, когда были созданы качественные

трансляторы и компиляторы для RISC-процессоров, началось их стремительное внедрение в промышленность. Для этого потребовалось почти два десятилетия.

Если раньше ARM-процессоры могли работать только с целочисленной арифметикой, то сегодня они могут обрабатывать числа в формате как с фиксированной, так и с плавающей точкой, причем с производительностью, которая еще десятилетие назад была доступна только мощным компьютерам. Существенно расширилась и система команд – до нескольких сотен. Но при этом сохранилось основное преимущество RISC-процессоров: выполнение практически всех команд, в том числе таких сложных, как обработка чисел с плавающей точкой, всего за один процессорный цикл.

Компания ARM Limited уникальна. Это единственная в мире компания, которая занимается разработкой исключительно новых процессорных ядер и инструментария для их практического использования (трансляторов, компиляторов, отладчиков, интегрированных сред разработки и т.д.), но никак не производством самих процессоров. Не имея собственных заводов, она продает лицензии на производство ARM-процессоров всем желающим. Фактически фирма ARM предлагает оптимизированные процессорные ядра с детально выверенной и надежно работающей системой команд, плюс интегрированные в процессорное ядро системные периферийные устройства, например, такие как отладчики. На базе этих процессорных ядер любая компания, занимающаяся производством электроники, может разработать собственный процессор или микроконтроллер, добавляя к купленному о лицензии ядру ARM нужные объемы памяти программ и данных, а также свою специализированную периферию.

Большинство фирм производителей используют процессорные архитектуры ARM «как есть», не внося в архитектуру каких-либо изменений (Atmel, Samsung и т.д.). Это создает определенную унификацию, позволяющую конечным разработчикам создавать свои проекты на базе микроконтроллеров разных фирм, выбирая те, которые имеют нужные объемы встроенной памяти и нужные наборы периферийных устройств, в том числе специализированных, адаптированных к конкретной области применения (например, к медицине, автомобильной промышленности, электромеханике и т.д.).

Мощные компании, такие как Intel, DEC, Texas Instruments, купив лицензии ARM, создают свои собственные процессорные архитектуры и выпускают свои собственные процессоры на их основе.

Фактически ARM продает лицензии на полноценные системы на кристалле (или System on Chip – SoC), которые, помимо центрального процессора и сопроцессора поддержки вычислений с плавающей точкой, могут содержать системные таймеры, контроллеры прерываний, контроллеры памяти, порты ввода/вывода, графические сопроцессоры, системы гео-позиционирования (GPS), модули поддержки мобильной связи (3G, 4G), встроенные на кристалл средства интерактивной отладки программного обеспечения. Покупатель лицензии сам определяет список модулей, который нужен именно ему для создания своей продукции.

Такой подход очень эффективен, так как предполагает использование базовых ядер, работающих по одной и той же системе команд с гарантированной надежностью многократно протестированного ядра. Более того, можно использовать одно и то же, разработанное для ARM-процессоров, системное программное обеспечение (например, операционные системы реального времени), опирающееся только на системные ресурсы, общие для всех производителей микроконтроллеров с одинаковыми ARM-ядрами. Заметные преимущества получают и разработчики конечной продукции. Сокращаются затраты на обучение инженеров-аппаратчиков и программистов. Уже накопленный опыт быстро переносится на микроконтроллеры, имеющие то же ядро, но более высокую производительность, расширенную память или дополнительные периферийные устройства. Сокращается общее время от начала разработки до ее выхода на рынок, так как появляется возможность использовать унифицированные средства разработки и

системное программное обеспечение широкого назначения, созданное специализированными фирмами.

1.3 Краткая историческая справка

1983 г. – Компания Acorn Computers, разрабатывающая портативные компьютеры для образовательных целей, с участием Стива Фурбера и Софи Уилсона (Университет в Манчестере), приступила к разработке нового RISC-процессора ARM1 на базе опыта использования микропроцессоров 6502 и 68000 фирмы Motorola. В качестве изготовителя кристалла выступала фирма Philips Semiconductor, сегодня – NXP. Первые образцы процессоров были изготовлены в 1985 г. по технологии 3 мкм, имели на кристалле всего 25000 транзисторов, что было существенно меньше, чем в аналогичных по производительности изделиях конкурентов. Процессор работал на тактовой частоте 4 МГц и имел встроенный умножитель целых чисел, а также интерфейс для подключения внешнего сопроцессора обработки чисел с плавающей точкой. Интересно, что уже в первом ARM-процессоре поддерживались не только операции умножения, но и умножения с накоплением, считавшиеся ранее прерогативой исключительно сигнальных процессоров. Модифицированная версия этого процессора ARM-2 имела тактовую частоту уже 12 МГц и выпускалась по технологии 2 мкм. На базе этих процессоров были созданы настольные ПК «Архимед».

1989 г. – Доминирующее место в процессорах для компьютеров занимают изделия фирм Intel (x86) и Motorola (68000). Начинается процесс интеграции на плату процессора сопроцессоров ускорения вычислений с плавающей точкой и модулей эффективного управления памятью. Тактовая частота поднимается до 25 МГц. Разрабатывается процессор ARM3 с тактовой частотой 25 МГц, как альтернатива «грандам». Компания Acorn пытается конкурировать с IBM PC на компьютерном рынке. Ее процессорами заинтересовалась Apple для применения в своих изделиях. В конечном счете это приводит к созданию новой компании Advanced RISC Machine (ARM) – Усовершенствованные (продвинутые) RISC машины. Компания была создана на деньги Apple при участии ведущих инженеров Acorn.

С этого времени кардинально изменена бизнес-модель компании: поставлена задача – стать лидером в области разработки новых, самых перспективных процессорных архитектур, не производить процессоры, а продавать лицензии на их производство. Ставилась задача создавать перспективные процессорные ядра, своеобразные макро-ячейки, к которым фирмы производители процессоров и микроконтроллеров смогут добавлять свою собственную дополнительную логику и свои периферийные устройства. При этом процессорное ядро должно быть очень надежным, с тщательно отлаженной системой команд и, конечно, малопотребляющим.

1990 г. – Компания VLSI Technology (Технология БИС) становится первым лицензиатом, купив лицензию на производство процессора ARM6. Он незначительно отличался от процессора ARM3, поэтому переход сразу через два номера был чисто маркетинговым ходом. Фирма Apple использовала этот процессор (ARM610) в КПК «Ньютон».

1993 г. – Разработан процессор ARM7 и использован в планшетах Apple с развитыми мультимедийными возможностями. Именно с этого времени начинается победное шествие процессоров ARM по всему миру. На его основе фирма Acorn выпустила новую серию компьютеров и линейку КПК (карманных переносных компьютеров).

Преемником этого процессора стал процессор ARM7TDMI, который стал прототипом целого класса новых процессорных ядер под общим названием Cortex. Упрощенная блок-схема процессора ARM7TDMI представлена на рис. 1.1.

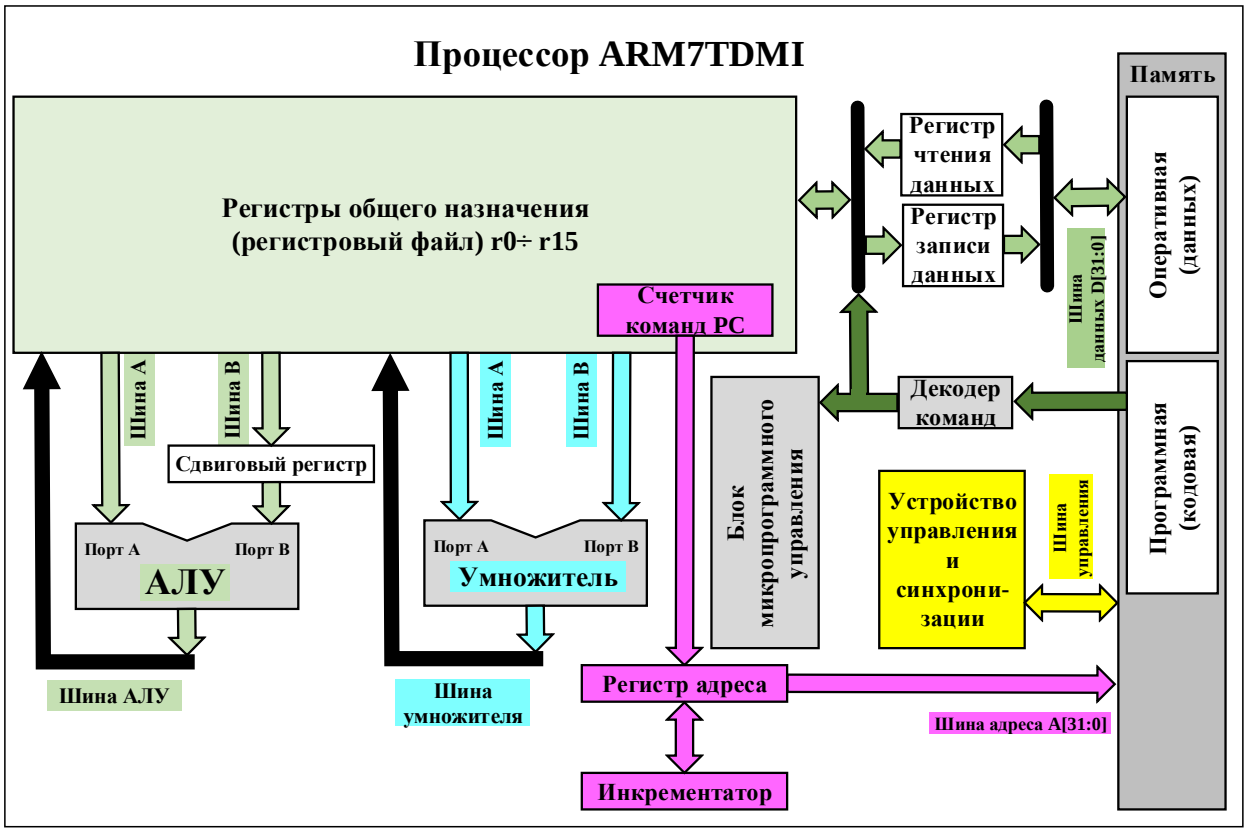


Рис. 1.1 Блок-схема процессора ARM7TDMI

Отладка микропроцессорных систем является довольно трудоемким делом. Для этой цели используются специальные аппаратные средства – отладчики (debugger) и так называемые внутрисхемные эмуляторы (In-Circuit Emulation – ICE). Процессор ARM7TDMI расширил возможности ARM7 с точки зрения отладки. В его состав включена аппаратная часть, поддерживающая подключение и эффективную работу внешних отладчиков (суффикс «D») и внутрисхемных эмуляторов (суффикс «I»).

Это был первый процессор ARM, рассчитанный на использование во встраиваемых системах:

- Для получения более компактного кода и уменьшения объема требуемой программной памяти был разработан новый набор 16-разрядных команд, получивший название **Thumb**, который мог применяться наряду с 32-разрядными командами **ARM**, требующими большего объема памяти. Эта возможность была быстро востребована в системах связи, в том числе, в сотовых телефонах.
- Суффикс «M» в обозначении процессора означает наличие на кристалле высокопроизводительного *аппаратного умножителя* (hardware multiplier), который переводит процессор в разряд сигнальных процессоров, обеспечивая эффективное выполнение алгоритмов *цифровой обработки сигналов* (DSP).

90-е годы – Фирма ARM разработала несколько еще более производительных процессорных ядер **ARM8**, **ARM9** и **ARM10**, изучение которых выходит за пределы данного учебника. На базе этих процессорных ядер вели собственные разработки такие лидеры компьютерной отрасли как DEC, Intel, Texas Instruments.

2000-е годы – Фирма ARM создала несколько линеек процессоров, среди которых **ARM11** и семейство **Cortex**, ряд многоядерных процессоров, а также специализированных высоконадежных процессоров для ответственных применений.

Как мы уже отмечали, сегодня в 99% сотовых телефонов используются процессоры ARM. Одной из первых начала их активное применение компания Nokia – один из лидеров рынка. Большинство ведущих производителей микроконтроллеров купили лицензии на процессорные ядра ARM и производят целые серии микроконтроллеров на базе этих ядер.

Каждый год по всему миру изготавливаются около 10 млрд. процессоров с ARM-ядрами и встраиваются в различные изделия – от мобильных телефонов, фотоаппаратов, видеокамер и планшетов до кондиционеров, роботов и автомобилей.

1.4 Версии процессорных архитектур в изделиях ARM

С точки зрения программиста определяющее значение имеет версия процессорной архитектуры (от v4 до v8), на базе которой разработано процессорное ядро.

Таблица 1.1 Базовые типы архитектур ARM-процессоров

Архитек-тура	Процессоры	Разряд-ность	Поддерживаются наборы команд			
			ARM-32 bit ISA	Thumb -16 bit ISA	Thumb2	ARM
v4	**ARM7TDMI SC100**	**32**	v	v		
v5	**ARM946 ARM968 ARM926**	**32**	v	v		
v6	**ARM1156T2 ARM1136 ARM1176 ARM11MP**	**32**	v	v	v	
v7	**Cortex-R4 Cortex-R5 Cortex-R7 Cortex-A5 Cortex-A8 Cortex-A9 Cortex-A12 Cortex-A15**	**32**	v		v	
v6-M	**Cortex-M0 Cortex-M1 SC000**	**32**		v		
v7-M	**Cortex-M3 Cortex-M4 SC300**	**32**		v	v	
v8	**Cortex-A53 Cortex-A57**	**64**		v	v	v

Именно версия процессорной архитектуры определяет набор команд, реализованный в процессоре. Так, процессорные ядра Cortex-M3, M4, которые будут изучаться в данном учебнике, имеют версию архитектуры v7-M (выделена в табл. 1.1 фоном) и поддерживают комбинированный набор 32-разрядных команд ARM и 16-разрядных команд Thumb, получивший название набора команд Thumb2. Это означает, что транслятор или компилятор сам выбирает, какую из двух возможных команд сгенерировать – 32-разрядную или 16-разрядную, позволяющую при той же выполняемой функции в два раза уменьшить требуемый объем кода.

Термином «транслятор» обычно называют программу, переводящую исходный код на языке Ассемблер конкретного процессора в машинный код. Термином «компилятор» – программу, переводящую программу, написанную на языке высокого уровня (например, СИ), в машинный код (возможно с созданием промежуточного файла на Ассемблере) – подробно в главе 8.

В каждом процессорном ядре имеются дополнительные опции, например, поддержка операций с плавающей точкой в процессорных ядрах Cortex-M4F.

1.5 Семейство процессоров Cortex

Все процессоры, ориентированные на *встраиваемые* применения, делятся на три группы в зависимости от требований конечных потребителей:

- Cortex-A
- Cortex-R
- Cortex-M

Каждое семейство постоянно совершенствуется – каждый год добавляются новые члены семейства, отличающиеся, прежде всего, производительностью и дополнительными функциональными возможностями.

Семейство **Cortex-A** предназначено для встраивания в изделия, требующие большой вычислительной мощности (тактовая частота до 3 – 4 ГГц), в том числе, в компьютеры и устройства связи. Если простой мобильный телефон с ограниченными функциями может быть реализован на Cortex-A5, то для планшета или компьютера автомобиля его производительности может не хватить. Для быстрой обработки больших объемов данных, например, в информационно-развлекательных устройствах, может потребоваться даже мультипроцессорная система Cortex-A15 (из 4-х процессоров).

Вместе с тем, контроллер управления приводом подачи станка не требует огромной вычислительной мощности и может быть реализован на более дешевом процессоре из семейства **Cortex-M**. Именно это семейство ориентировано на *микроконтроллерные применения, связанные с управлением оборудованием в реальном времени*. Интересно отметить, что как мощные процессоры семейства Cortex-A, так и процессоры семейства Cortex-M имеют много общего в системе адресации памяти и системе команд. Поэтому, изучив в этой книге семейство процессоров Cortex-M, вы сможете быстро освоить и другие более производительные процессоры.

1.5.1 Семейства Cortex-A и Cortex-R

Предназначены для высокопроизводительных устройств, требующих значительной вычислительной мощности: портативных компьютеров, планшетов, серверов. Имеют большую кэш-память, дополнительные арифметические блоки для поддержки графики, сопроцессоры вычислений с плавающей точкой и модули управления памятью, которые оптимизируют работу больших операционных систем (Windows, Linux, Android). В конце линейки этих процессоров есть изделия, которые способны поддерживать одновременную работу нескольких ядер (до 8). Процессоры Cortex-A5/7/8/9/12/15 являются 32-разрядными, а процессоры A53 и A57 – уже 64-разрядными. Свои системы на кристалле на базе этих ядер производят ведущие производители электроники, например, Texas Instruments (Davinci и Sitara на базе ARM9 и Cortex-A8, OMAP на базе Cortex-A15).

Процессоры Cortex-R4/5/7 имеют архитектуру, предназначенную для работы в очень ответственных устройствах, где надежность и предсказуемость имеет первостепенное значение: авиация, транспорт, энергетика и т.д. Это избыточные системы, реализованные на больше чем одном ядре. В них имеется система контроля и

предупреждения программного сбоя. Пример – линейка процессоров TMS570 от Texas Instruments.

1.5.2 Семейство Cortex-M

Семейство Cortex-M представляет собой линейку процессорных ядер для построения микроконтроллеров Cortex-M0/0+/1/3/4. Являясь 32-разрядными ядрами, изделия на их основе призваны заменить имеющиеся на рынке 8- и 16-разрядные микроконтроллеры. Области применения: от бытовой техники, систем управления приводами до систем комплексной распределенной автоматизации производства. Имеют маленькую потребляемую мощность и габариты, низкую цену. Лицензии на эти ядра приобрели более 200 компаний, занимающихся выпуском микроконтроллеров. За счет массового производства стоимость микроконтроллеров может находиться в диапазоне от 20 центов до 2 долларов.

В самом низу линейки процессор **Cortex-M0**, который имеет только 56 команд, малые габариты и предельно низкое потребление. Подходит для реализации простых логических контроллеров и автоматов. Реализован в линейке LPC1100 от NXP и линейке XMC1000 от Infineon. Ядро Cortex-M0+ отличается наличием модуля защиты памяти (MPU), перемещаемой таблицей векторов прерываний, одноцикловым интерфейсом ввода/вывода для задач быстрого управления и дополнительной отладочной логикой.

Ядро **Cortex-M1** разработано специально для реализаций на базе программируемых логических матриц (FPGA), содержит дополнительные интерфейсные модули памяти программ и данных, а также отладочную логику.

Процессорные ядра **Cortex-M3** разработаны для приложений, требующих быстрого отклика на внешние события, и содержат встроенный контроллер векторных приоритетных прерываний NVIC (имеется и в младших моделях), модуль защиты памяти MPU, модуль отладки с возможностями трассировки программы, аппаратный делитель и одноцикловой умножитель. Процессор имеет расширенную систему команд, статическую память SRAM и периферийный интерфейс.

Ядро **Cortex-M4** имеет еще большие возможности. В него включены команды поддержки цифровой обработки сигналов, а также (в версии **Cortex-M4F**) сопроцессор поддержки вычислений с плавающей точкой. Микроконтроллеры с этими ядрами производят многие фирмы, среди них Atmel, Freescale, Texas Instruments, в том числе и отечественные компании.

1.6 Использование ARM-ядер в отечественных микроконтроллерах

В последние несколько лет в России активно работает программа импорто-замещения, в рамках которой большое внимание уделяется созданию отечественных процессоров и микроконтроллеров, не уступающих по своим характеристикам лучшим западным аналогам. Возможный путь решения проблемы: покупка лицензий на базовые ARM-ядра у фирмы ARM и производство на этой основе микроконтроллеров с памятью и периферией собственной разработки. При этом объемы и типы встроенной памяти, а также встроенная периферия могут создаваться самостоятельно и адаптироваться для поддержки важнейших отраслей отечественного производства.

Почему бы не взять у ARM все самое лучшее, надежное и многократно поверенное, как это делают, не стесняясь, даже такие известные фирмы как Intel и Texas Instruments: одну из самых перспективных сегодня процессорных архитектур, поддержанную мощной системой команд и целым рядом встроенных в ядро системных периферийных устройств. При этом периферия будет оптимизироваться самостоятельно под потребности основных

российских потребителей микроконтроллеров. Преимуществом такого подхода является и то, что становится доступен весь арсенал специализированного программного обеспечения, уже разработанный в мире, в том числе интегрированные среды разработки и отладки программного обеспечения, специализированные библиотеки, драйверы типовых интерфейсов, операционные системы.

Приведем несколько примеров. Одна из ведущих отечественных электронных компаний АО «ПКК Миландр», г. Зеленоград, разработала, освоила в производстве и уже поставляет потребителям 32-разрядные микроконтроллеры **1986BE9x** с ядром на базе **ARM Cortex-M3** для широких областей применения взамен любых 8- и 16-разрядных микроконтроллеров импортного производства. В состав периферийных устройств включены три таймера с функциями генерации ШИМ-сигналов, что позволяет использовать микроконтроллер для создания относительно несложных, но массовых преобразователей частоты для управления асинхронными двигателями, например, в приводах насосов и вентиляторов ЖКХ. Вычислительных ресурсов микроконтроллера и встроенных коммуникационных интерфейсов достаточно для решения большинства практических задач (два CAN контроллера, контроллеры UART, SPI, I2C и USB).

Интерес для разработчиков представляют и новейшие микроконтроллеры **1986BE8T** этой фирмы, в основу которых заложено более производительное ядро **ARM-Cortex-M4F** с модулем аппаратной поддержки вычислений с числами в формате с плавающей точкой однократной точности. Они предназначены для создания сложных, высоко производительных информационно-измерительных систем с большим числом аналоговых входов и выходов. О существенно возросших возможностях российских производителей электроники говорит набор интерфейсов, интегрированных на кристалл (от контроллеров Ethernet и USB до специализированных интерфейсов, ориентированных на преимущественное применение в авиационной технике – ARINC, SpaceWire). Микроконтроллеры имеют также ряд встроенных систем контроля, обнаружения и исправления ошибок, в том числе при доступе к памяти и периферии, что позволяет использовать их в особо ответственных применениях.

В 2016 г. АО «НИИЭТ», г. Воронеж, завершил разработку и выпустил опытно-промышленную партию первых отечественных специализированных микроконтроллеров **К1921ВК01Т** типа «**Motor Control**» (Управление двигателями), «**Motion Control**» (Управление движениями) и «**Power Control**» (Управление силовыми преобразователями энергии). У отечественных специалистов, работающих в области машиностроения, энергетики, транспорта, электромеханики, наконец-то появилась отечественная элементная база на уровне лучших западных решений.

1.6.1 Технические характеристики микроконтроллера К1921ВК01Т

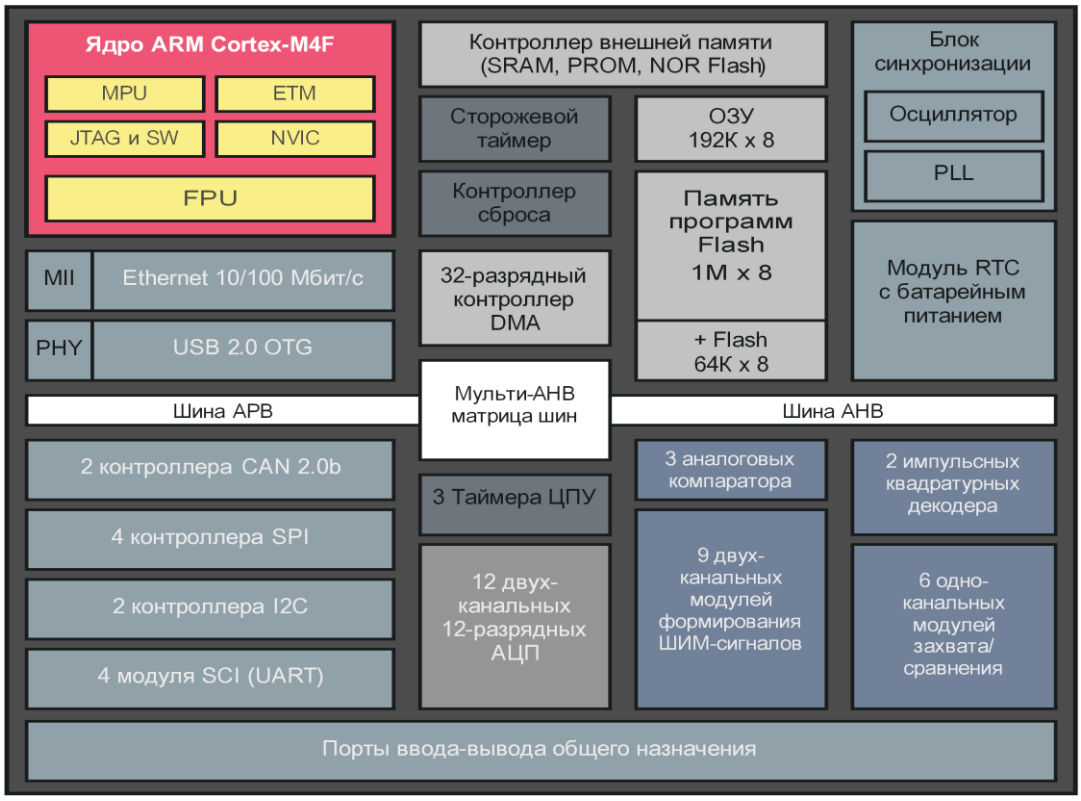


Рис. 1.2 Структурная схема отечественного микроконтроллера К1921ВК01Т фирмы АО «НИИЭТ»

В качестве ядра используется одно из самых перспективных микропроцессорных ядер **ARM-Cortex-M4F** со встроенным модулем поддержки вычислений с плавающей точкой FPU, модулем защиты памяти MPU, контроллером вложенных векторных прерываний NVIC и набором модулей для поддержки интерактивной отладки с выходом на два стандартных отладочных интерфейса JTAG и SWD. Процессор обеспечивает производительность до 125 MIPS (миллионов инструкций в секунду). В ближайшее время планируется поднять производительность микроконтроллера до 200 MIPS. Основные технические данные:

- Встроенная FLASH-память программ емкостью 1 Мбайт;
- Встроенное статическое ОЗУ (RAM) емкостью 192 Кбайта;
- Дополнительная FLASH-память данных объёмом 64 Кбайта;
- Возможность расширения памяти - контроллер внешней памяти (SRAM, PROM, NOR Flash);
- 32-канальный контроллер прямого доступа к памяти (DMA);
- Система сброса и сторожевой таймер (Watchdog);
- Часы реального времени (Real Time Clock) с батарейным питанием;
- Синтезатор таковой частоты микроконтроллера (PLL);
- Двенадцать 2-канальных 12-разрядных АЦП с временем выборки 2 МВыб/с, с режимами цифрового компаратора для каждого из каналов (равно или больше, равно или меньше, попадание в диапазон, выход из диапазона), с функцией автоматического запуска модулей ШИМ-генератора по событию в АЦП (окончанию преобразования), с дополнительным встроенным датчиком температуры (T-Sensor) для контроля температурного режима собственно микроконтроллера;

- Восемнадцать модулей ШИМ (PWM) или девять двухканальных модулей с комплиментарными выходами, из которых шесть могут работать в режиме «высокого» разрешения (HRPWM) с возможностью прецизионного изменения длительности импульсов на величину меньшую периода тактового сигнала процессора;
- Шесть модулей захвата/сравнения (CAP);
- Три аналоговых компаратора с функцией автоматического запуска модулей ШИМ по событию сравнения (равно или больше, равно или меньше);
- Три 32-битных таймера общего назначения;
- Два порта CAN 2.0b;
- Два интерфейса I2C с поддержкой режима High Speed (более 1 МГц);
- Два квадратурных декодера (enhanced Quadrature Encoded Pulse – eQEP) для обработки сигналов импульсных датчиков положения ротора и исполнительного органа в высокопроизводительных системах позиционирования и слежения (определение положения, направления и скорости вращения);
- Четыре порта синхронного периферийного интерфейса SPI;
- Четыре модуля коммуникационного интерфейса SCI (UART);
- Интерфейс USB 2.0 Device /Host с физическим уровнем PHY;
- Интерфейс Ethernet 10/100 Мбит/с с интерфейсом MII;
- Отладочный интерфейс (JTAG и ARM Serial Wire Debug (SWD));
- Не менее 88 выводов портов ввода/вывода общего назначения (GPIO), раздельно программируемых и мультиплексированных со спецфункциями периферийных устройств.

Встроенная периферия этого микроконтроллера по всем показателям не уступает самым лучшим западным образцам специализированных микроконтроллеров, например, периферии микроконтроллера TMS320F28335 фирмы Texas Instruments, который, по мнению специалистов, долгое время являлся лучшим в мире сигнальным микроконтроллером типа «Motor Control». По вычислительной производительности отечественный микроконтроллер пока несколько уступает аналогам (100 МГц вместо 150 МГц). Но уже сейчас его возможностей более чем достаточно для решения большинства практических задач в области отечественной энергетики, электромеханики, транспорта и комплексной автоматизации производства. При частоте 100 МГц гарантируется производительность не ниже 125 MIPS. А в конце 2017 года должна появиться модификация микроконтроллера с частотой 200 МГц, что позволит приблизиться к новейшим микроконтроллерам TMS320F28377 фирмы Texas Instruments.

В распоряжении отечественных разработчиков сегодня имеется большое число как импортных, так и отечественных микроконтроллеров на базе процессорных ядер ARM-Cortex-M3/M4/M4F, отличающихся высокой производительностью (тактовой частотой), объемами встроенной памяти программ и данных, наборами встроенных периферийных устройств и низкой ценой. Задача освоения этой новой и перспективной техники российскими специалистами и студентами является чрезвычайно актуальной. Это будет способствовать быстрому возрождению отечественного производства на принципиально новой цифровой элементной базе – созданию современного цифрового производства.

1.7 Преимущества архитектуры и системы команд ARM

Ниже перечислены важнейшие преимущества архитектуры и системы команд процессоров ARM, которые обеспечили завоевание рынка микроконтроллерных применений этими процессорами:

1) Обработка данных (арифметические, логические и другие операции), находящихся исключительно в сверхоперативной памяти центрального процессора (ЦПУ) – регистрах общего назначения с использованием самой компактной и быстрой *регистровой адресации* операндов. Обработка данных в формате с плавающей точкой исключительно в регистрах окружения сопроцессора. Полное отсутствие команд обработки, операнды которых расположены в памяти – уменьшение длины всех команд за счет исключения полей адресации операндов в памяти. *Унификация форматов команд* до 32- и 16-разрядных команд.

2) Ускорение выполнения операций за счет минимизации числа обращений к памяти, которая существенно медленнее, чем регистровая память центрального процессора. Быстрое выполнение большинства команд, в том числе команд сопроцессора, за один цикл: *одноцикловое выполнение* команд – визитная карточка процессоров с RISC-архитектурой. Даже такие комплексные команды, как умножение с накоплением чисел в формате с плавающей точкой, выполняются за один цикл, и только некоторые команды требуют большего числа циклов (например, деление, извлечение квадратного корня). Как следствие – высокая вычислительная производительность процессорных ядер.

3) Организация работы с памятью с помощью отдельных инструкций загрузки/сохранения содержимого регистров из памяти/в память. Применение для этого исключительно косвенной адресации операндов с использованием в качестве *регистров указателей любых регистров ЦПУ*.

4) Развитые средства *стековой адресации* и *относительной адресации* данных в памяти по содержимому счетчика команд. Возможность одной командой за один цикл выполнить загрузку/сохранение любых 32-разрядных данных из памяти/в память.

5) *Временное совмещение ряда операций*, таких как считывание кодов команд на конвейер команд и считывание данных из кодовой памяти при выполнении операций загрузки данных в регистры ЦПУ.

6) Использование самых *эффективных методов адресации массивов данных* в памяти, включая базовую адресацию со смещением, адресацию с пред- и пост-смещением указателя (авто-инкрементирование/декрементирование до или после доступа к данным), базовую адресацию с автоматическим вычислением смещения относительно базы в зависимости от длины данных в памяти (байт, полуслово, слово) и индекса. Поддержка эффективного доступа к любым структурам данных.

7) Использование прерываемых команд *множественной загрузки/сохранения* регистров ЦПУ или сопроцессора из памяти/в память.

8) Отсутствие регистра-аккумулятора, предназначенного для хранения промежуточных результатов вычислений, и использование для этой цели любого из регистров процессора или сопроцессора (при вычислениях с плавающей точкой).

9) Поддержка не выровненного по длине 32-разрядного слова доступа к памяти для экономии объема памяти при работе с байтами и полусловами.

10) Включение в архитектуру центрального процессора, кроме АЛУ и аппаратных умножителя и делителя, *кольцевого сдвигового регистра*, позволяющего выполнять

попутные операции масштабирования одного из операндов при сохранении времени выполнения команды всего за один цикл.

11) *Формирование флагов результатов операций* только тогда, когда это необходимо (опциональный суффикс установки флагов). Наличие команд сравнения операндов в формате с плавающей точкой.

12) Возможность *условного выполнения* всех команд процессора и сопроцессора по состоянию суффиксов условного выполнения, единых для ЦПУ и сопроцессора. Поддержка методов *линейного модульного программирования*, в том числе, на Ассемблере, практически без использования команд условного ветвления. Наглядность, простота отладки и сопровождения программных продуктов.

13) Наличие «семафорной памяти» с побитовой адресацией и неявно выраженного «битового сопроцессора».

14) Наличие команд преобразования форматов данных, существенно облегчающих интерфейс ввода переменных и вывода управляющих воздействий.

15) Программная поддержка операций цифровой обработки сигналов с числами как в формате с фиксированной, так и с плавающей точкой. Поддержка организации в памяти кольцевых буферов выборок данных.

16) Интеллектуальный транслятор с автоматической генерацией оптимальных команд вместо псевдокоманд Ассемблера. Автоматическая генерация транслятором блоков условного выполнения с числом условно выполняемых команд в блоке – до четырех.

Список рекомендуемой литературы

1) Техническая документация на микроконтроллер 1921ВК01Т1 фирмы АО «НИИЭТ». Доступно по ссылке.
2) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
3) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
4) ARM DDI 0403C_errata_v3, ARMv7-M Architecture Reference Manual, Arm Limited, 2010.
5) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
6) Электронный ресурс https://www.arm.com

2 ВВЕДЕНИЕ В МИКРОПРОЦЕССОРНУЮ ТЕХНИКУ

Оглавление

2.1 Цифровой мир

Компьютерная техника прочно вошла в нашу жизнь. Мы уже не мыслим свое существование без мобильного телефона, планшета или компьютера и порой не задумываемся о том, сколько усилий инженеров и программистов потребовалось для этого. Современный мир по-настоящему стал «цифровым», причем это касается не только бытовых устройств, но и промышленного оборудования. Сегодня каждое устройство от кондиционера до станка с числовым программным управлением и робота имеет по крайней мере один процессор, управляющий его работой. Процессоры стали непременным атрибутом любого оборудования.

Типичный пример – «умный дом», в котором каждая система (водоснабжения, кондиционирования воздуха, освещения, обогрева, охраны и т.д.), имея свой собственный процессор, объединена в общую локальную сеть, доступ к которой выполняется непосредственно со смартфона хозяина. На создание такой техники ушло почти полвека усилий инженеров и программистов. Современный планшет, например, представляет собой не только полноценный компьютер, но и мобильный телефон, имеет все необходимые интерфейсы для подключения к Интернету и периферийным устройствам (USB, Wi-Fi, Bluetooth). Более того, он становится центром контроля и управления как местными, так удаленными интеллектуальными устройствами, находящимися в распоряжении хозяина (от медиацентра до системы охраны дачи).

Инженер любой специальности сегодня должен обладать знаниями в области микропроцессорной техники, достаточными для разработки, отладки и сопровождения цифровой системы управления объектами и оборудованием в своей предметной области. Эта глава содержит необходимые начальные сведения, которые можно считать общими, не зависящими от типа используемого процессора или микроконтроллера. Тем не менее,

обратите внимание на некоторые попутные замечания, которые касаются основной темы книги – процессорных ядер ARM Cortex-M.

2.2 Структура микропроцессорной системы. Основные понятия

Базовым элементом любой цифровой системы (вычислительной или управляющей) является ***процессор*** – устройство для обработки информации и управления процессом обработки. Процессор может быть выполнен на одной или нескольких печатных платах, как это было еще полвека назад в первых компьютерах, а также в виде кристалла большой интегральной схемы (БИС), как сейчас. В последнем случае он называется ***микропроцессором (МП)***.

Процессор – это ***программно-управляемое устройство***, которое выполняет все действия в строгом соответствии с ***программой***, находящейся в ***памяти*** (рис. 2.1). Программа, которая может быть выполнена процессором, называется ***исполняемой программой*** или ***программой в машинных кодах*** (в двоичном коде).

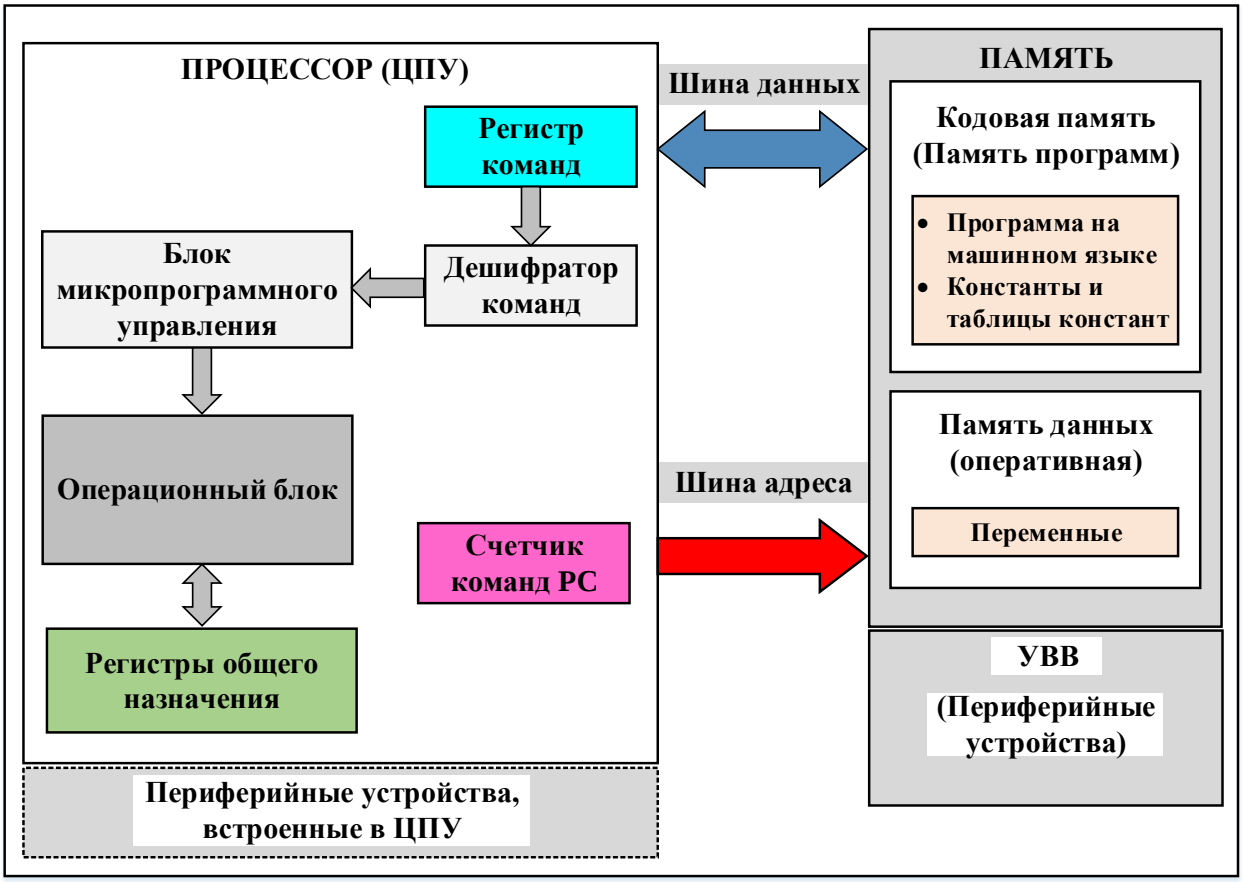


Рис. 2.1 Структура микропроцессорной системы

Процессор и память являются обязательными компонентами любой ***микропроцессорной системы (МПС)***. Для того, чтобы действия, выполняемые процессором, были «видны» пользователю или как-то воздействовали на объект управления, в состав микропроцессорной системы должен входить еще один обязательный компонент – ***устройства ввода/вывода*** (***УВВ***) или ***периферийные устройства (ПУ)***. В простейшем случае это могут быть просто ***порты ввода/вывода***

данных. В более сложном – необходимы специальные ***каналы связи*** **(*интерфейсы*)** и ***специальные периферийные устройства***. Так, ввести информацию о состоянии управляемого процесса можно только сняв показания с аналоговых датчиков (давления, расхода, температуры, тока, напряжения и т.д.) и преобразовав ее с помощью аналого-цифрового преобразователя (АЦП), в цифровую форму (цифровой код). Если объект управления (например, управляемый источник питания) имеет исполнительное устройство с аналоговым входом, то потребуется преобразование рассчитанного в цифровой системе управляющего воздействия в аналоговую форму – цифро-аналоговый преобразователь (ЦАП).

В последнее время получила развитие ***концепция прямого цифрового управления*** всеми компонентами оборудования, в соответствии с которой в состав МПС должны входить специализированные периферийные устройства, которые могут работать ***автономно***, минимально используя ресурсы процессора и памяти: многоканальные ШИМ-генераторы, модули захвата и сравнения, квадратурные декодеры, ЦАП, АЦП и т.д.

Все компоненты микропроцессорной системы располагаются на одной или нескольких ***печатных платах*** – printed circuit board (PCB), соединенных между собой «межплатными» интерфейсами. В состав МПС входит собственно процессор***, кодовая память*** или ***память программ (ПЗУ – постоянное запоминающее устройство)*** с ***доступом к данным только по чтению,*** содержащая также константы и таблицы констант, ***оперативная память (ОЗУ***) или ***память данных***, содержащая переменные, с ***доступом по записи и по чтению***, а также ***периферийные устройства***. На печатной плате устанавливаются и ***коммуникационные интерфейсы*** для сопряжения МПС с другими системами управления более высокого уровня. Они поддерживаются специальными интерфейсными БИС – ***контроллерами интерфейсов***.

Центральным процессорным устройством (ЦПУ) называется основной процессор в системе, который, как правило, работает с целыми числами и числами в формате с фиксированной точкой. Если необходима поддержка вычислений с плавающей точкой – она реализуется либо с помощью специальных библиотек, либо подключением сопроцессора вычислений с плавающей точкой. ***Сопроцессор*** – это отдельное устройство или опция производителя микроконтроллера, которая добавляется на кристалл по желанию заказчика и существенно расширяет возможности ЦПУ.

Некоторые, наиболее важные периферийные устройства, такие как ***системный таймер*** и ***контроллер прерываний,*** могут непосредственно встраиваться в ЦПУ. Так, в частности, поступает фирма ARM со своими процессорными ядрами (рис. 2.1). При таком подходе пользователи получают ряд преимуществ: достигается унификация процессорных ядер независимо от их производительности – при переходе на более производительное ядро наработки в области программного обеспечения (например, созданные ранее операционные системы реального времени – ОСРВ) сохраняются; так как ***системная периферия*** интегрирована в процессорное ядро, ее производительность уже оптимизирована разработчиком процессорного ядра (фирмой ARM).

Совокупность процессора, памяти, одного или нескольких сопроцессоров и периферийных устройств, размещенных на одном кристалле, в одном корпусе БИС, называется ***микроконтроллером*** или ***системой на кристалле*** system-on-chips (SoC). Системы на кристалле экономят место на плате и существенно уменьшают энергопотребление, что особенно важно для мобильных устройств с батарейным питанием. Современные системы на кристалле становятся все более сложными, объединяя порой процессоры разных архитектур для получения максимальных преимуществ каждой из них. Так, в двухъядерных микроконтроллерах Texas Instruments «Concerto», ARM-ядро выполняет ***коммуникационные функции*** (обмена данными по типовым интерфейсам), а функции ***управления в реальном времени*** двигателями и силовыми преобразователями –

ядро специализированного сигнального микроконтроллера C2000 фирмы Texas Instruments, оптимизированное для решения подобных задач.

Системы на кристалле – одна из главных тенденций развития современной процессорной техники. Они позволяют использовать освоенные ранее, надежные и качественные аппаратные и программные решения в новых разработках, ускоряя процесс создания все более производительных и функциональных устройств, не начиная каждый раз процесс разработки «с нуля». Мы не оговорились, эта тенденция касается и программных продуктов. Так, создать ***драйвер*** для поддержки интерфейса USB или Ethernet очень нелегкая работа, которая может потребовать нескольких «человеко-лет» труда высококвалифицированных программистов. Проще купить этот продукт, использовать его и быстрее выйти на рынок со своим новым изделием. Зачастую разработчики микроконтроллеров, заинтересованные в росте продаж собственных кристаллов, делают подобные драйверы или даже целые библиотеки специализированных подпрограмм общедоступными. Это на руку разработчикам ***конечного оборудования***, так как резко сокращает время от начала разработки до выхода изделия на рынок и, естественно, стоимость разработки.

Процессорная техника очень быстро совершенствуется. Производительность процессоров удваивается каждые два года. Это явление даже получило название «Закон Мура». Поэтому, важно сохранить сделанные ранее вложения в разработку. Это возможно только в том случае, когда сопряжение разных устройств реализуется по ***стандартным протоколам обмена и интерфейсам.*** Они разработаны для множества изделий, в том числе для сервоприводов станков и роботов. Если Вы создали инвертор для сервопривода со встроенной цифровой системой управления, но не совместимый со стандартом, то его вряд ли кто-то купит. Вы не сможете подключить его ни к одной системе управления верхнего уровня (системе числового программного управления ЧПУ или промышленному программируемому контроллеру ПК).

Под ***системой управления нижнего уровня*** обычно понимается встроенная система управления, которая интегрирована в оборудование (преобразователь частоты, робот, кондиционер, станок и т.д.), а под ***системой управления верхнего уровня*** – промышленный компьютер, промышленный программируемый контроллер, система ЧПУ, к которой по типовым интерфейсам подключаются встроенные системы управления нижнего уровня. Они, как правило, объединяются в ***локальную промышленную сеть***, образуя одну общую систему управления, в которой может быть несколько иерархий управления. Такая система управления может быть ***распределенной*** с целым рядом контроллеров нижнего уровня, встроенных в оборудование, контроллеров среднего и верхнего уровня. Это может быть ***система автоматического управления*** конвейерной или сборочной линией, производственным участком, электромобилем и т.д.

За последние годы для каждой области техники разработчики микроконтроллеров создали целый ряд ***специализированных периферийных устройств***, оптимизированных по режимам работы, выполняемым функциям и производительности именно к этой предметной области. Создана такая техника и для ***управления двигателями*** (Motor Control), ***системами питания*** (Power Control), ***автомобильной, авиационной электроникой*** и др. Такие микроконтроллеры или системы на кристалле получили название ***специализированных микроконтроллеров***. Целый ряд всемирно известных производителей, таких как Texas Instruments, ST Microelectronics и др., производят специализированные микроконтроллеры и на базе ядер ARM Cortex-M3/M4/M4F, изучению которых посвящена эта книга.

Выпускаются целые ***серии микроконтроллеров***, которые имеют общий центральный процессор, но разные объемы памяти и наборы периферийных устройств (ПУ) на кристалле. Разработчики систем управления нижнего уровня имеют возможность выбора микроконтроллера с нужной тактовой частотой (производительностью), объемами

памяти программ и данных, коммуникационными интерфейсами и встроенными периферийными устройствами, что позволяет оптимизировать конкретную систему управления по стоимости и потребляемой энергии.

Микроконтроллеры совершили революцию в современном управлении оборудованием – они *встраиваются* непосредственно в изделие (в стиральные машины, холодильники, кондиционеры, двигатели автомобилей и т.д.), обеспечивая прямое цифровое управление каждым элементом оборудования, вплоть до отдельного силового ключа, а также прямой интерфейс с датчиками аналоговых, дискретных и импульсных сигналов. Широкий набор встроенных интерфейсов обеспечивает сопряжение и с системами управления более высокого уровня, и с человеком-оператором.

2.3 Общие принципы работы процессора

Процессор, память и периферийные устройства объединяются между собой системой шин (см. рис. 2.1). Различают ***шину адреса***, по которой процессор информирует внешнюю память о том, к какой конкретно ячейке памяти последует обращение, а также ***шину данных***, по которой, собственно, и пересылаются данные. Шина адреса ***однонаправленная*** – от процессора к памяти и периферии, а шина данных – ***двунаправленная***.

Шина управления (не показана на рис. 2.1) предназначена, как минимум, для информирования внешней памяти или периферийного устройства о направлении передачи информации – чтение или запись. В любом случае, инициатором обмена данными с памятью или периферийным устройством является процессор. Именно он сначала выдает адрес ячейки памяти или регистра периферийного устройства, затем сигнал направления передачи данных и только потом получает с шины данных или выдает на нее нужную информацию.

Шины адреса и данных могут быть отдельными (изолированными) или совмещенными (***мультиплексированными)***. Это означает, что часть времени шина используется в качестве шины адреса, а часть – в качестве шины данных. При этом в начале каждого цикла обращения к памяти или периферии процессор сначала выдает адрес объекта доступа, который фиксируется в микропроцессорной системе (вне процессора), а затем та же шина переключается на использование в качестве шины данных.

Устройства ввода/вывода (периферийные устройства), как правило, являются ***программируемыми*** и содержат внутри себя целый набор регистров. Каждый из них имеет свой персональный адрес. В процессорах ARM реализована ***технология отображения адресов регистров периферийных устройств на память***. Это означает, что все регистры ПУ представляют собой как бы ячейки памяти, доступ к которым может производиться точно так же, как к обычным ячейкам памяти, с использованием всего мощного арсенала команд процессора. Эта технология имеет некоторые недостатки – часть общего адресного пространства приходится отводить под регистры периферийных устройств.

Альтернативное решение, которое часто применялось на заре процессорной техники – так называемый, ***изолированный ввод/вывод***, когда для работы с устройствами ввода/вывода используются отдельные шины адреса и данных. При этом не «расходуется» адресное пространство памяти, но для доступа к УВВ нужны специальные команды ввода/вывода типа Input (Ввод) и Output (Вывод).

Что определяет разрядность шины адреса? Это ***объем прямо адресуемой процессором памяти***. Так, если шина адреса 16-разрядная, как это было в первых

процессорах и микроконтроллерах, то объем доступной процессору памяти составит всего $2^n=2^{16}=65536=64\times1024$ = 64 К ячеек, если 32-разрядная – то уже $2^{32}=4,294,967,296=4,096\times1,048,576$ = 4096 М ячеек = 4×1,073,741,824 = 4 Г ячеек памяти.

Здесь и далее в книге для удобства читателей мы будем использовать запятую не в качестве десятичной точки, а в качестве разделителя цифр, чтобы длинные числа были более «читабельными». Десятичная точка будет использоваться для разделения целой и дробной части вещественного числа, как обычно.

В отличие от физики, где соответствующие весовые коэффициенты (К, М, Г) определяются степенью числа 10, в процессорной технике они определяются степенью числа 2, как основания двоичной системы счисления – табл. 2.1.

Это более удобно, так как адресная шина любой БИС памяти может иметь только целое число двоичных разрядов. Так, если она 10-разрядная, то может содержать только 2^{10}=1024 К ячеек, если 32-разрядная, как у процессоров ARM, – то 4 Г ячеек.

Таблица 2.1 Весовые коэффициенты в физике и процессорной технике

Весовой коэффициент	**В физике**		**В процессорной технике**	
	Степень 10	**Значение**	**Степень 2**	**Значение**
К - Кило	**10^3**	**1000**	**2^{10}**	**1024**
М - Мега	**10^6**	**1,000,000**	**2^{20}**	**1,048,576**
Г - Гига	**10^9**	**1,000,000,000**	**2^{30}**	**1,073,741,824**

Обычно разрядность ячейки памяти – байт (8 двоичных разрядов). Это означает, что процессоры ARM могут адресовать 4 Г байта памяти.

В большинстве ранее выпускаемых процессоров и микроконтроллеров разрядность шины данных совпадала с разрядностью памяти. В соответствии с этим различают 8-, 16- и 32-разрядные процессоры. Они обрабатывают за один ***цикл доступа к памяти*** либо байт, либо 16-разрядное полуслово, либо 32-разрядное слово.

В процессорах ARM шина данных 32-разрядная. Это означает, что фактически доступ к памяти 32-разрядный. Однако, внутри процессора возможна обработка не только ***полного 32-разрядного слова***, но и ***полуслова*** и даже ***байта***. В этом случае из памяти всегда считывается полное слово, но «лишняя» информация внутри процессора как бы отбрасывается. Таким образом, процессоры ARM, будучи настоящими 32-разрядными процессорами, допускают работу с данными и в более коротких форматах, присущих 8-и и 16-разрядным процессорам.

Кто управляет порядком выборки команд из памяти? В соответствии с концепцией, предложенной одним из основоположников компьютеров фон-Нейманом, за это отвечает специальный регистр внутри процессора, называемый ***счетчиком команд*** или ***программным счетчиком*** PC (Program Counter) (см. рис. 2.1). Он всегда содержит адрес команды, подлежащей выполнению. Когда очередная команда считывается из памяти и дешифрируется, в зависимости от ее длины счетчик команд автоматически инкрементируется нужное число раз. Так, если команда 32-разрядная и занимает в памяти 4 байта, то счетчик команд получает приращение +4. Счетчик команд является ***указателем адреса*** очередной команды, подлежащей выполнению. Его содержимое выставляется на шину адреса при каждом доступе процессора к очередной команде (пока для простоты предполагается, что в процессоре нет конвейера команд).

При сбросе процессора или при подаче на него питания содержимое счетчика команд PC автоматически устанавливается в состояние, соответствующее начальному адресу кодовой памяти. Именно с этого адреса начинается работа процессора, как

программно-управляемого устройства (в следующих главах этот простой механизм будет уточнен для процессоров ARM).

Считывая первые данные из памяти, процессор по умолчанию считает, что это – инструкция. Она поступает в специальный регистр, называемый ***регистром команд*** и расшифровывается. Так как в кодовой памяти могут находиться не только команды, но и данные (константы и таблицы констант), их выборка будет определяться только соответствующей командой процессора, в процессе ее выполнения. Применительно к ARM это может быть, например, команда загрузки данных из памяти с относительной адресацией по содержимому счетчика команд (текущему).

Каждый процессор имеет свой собственный ***набор команд***, то есть совокупность всех инструкций, которые могут быть выполнены процессором. Первые ARM процессоры имели всего несколько десятков команд, современные – более двух сотен.

Каждая команда из набора команд имеет свой персональный код, называемый ***кодом операции***. По этому коду процессор определяет, что именно он должен делать.

Функцию дешифрации команды выполняет устройство, называемое ***дешифратором команд***. Дешифратор команд по коду операции запускает соответствующий данной команде ***дискретный управляющий автомат***, задающий последовательность микроопераций, приводящих к выполнению команды.

Выполнение любой команды в процессоре разбивается на ряд ***микроопераций***, то есть минимально-возможных (атомарных для данного процессора) действий. Каждой команде соответствует своя собственная последовательность микроопераций. Если операция простая, то таких микроопераций мало, если сложная – много. Все микрооперации кодируются с помощью специального ***микрокода***. По существу, для каждой операции внутри процессора создается своя ***микропрограмма*** выполнения команды, которая запускается после расшифровки кода операции.

В составе любого процессора имеется так называемый ***операционный блок,*** который выполняет арифметические и логические операции (сложение, вычитание и т.д.). Его ядром является двухпортовое ***арифметико-логическое устройство*** (***АЛУ***). Каждый из портов АЛУ принимает по операнду, который, в соответствии с кодом операции, может поступать либо из ***регистров общего назначения (РОН) – сверхоперативной памяти процессора***, либо из памяти. В процессорах ARM в качестве операндов используются исключительно регистры ЦПУ. Как только операнды загружены, АЛУ получает команду на выполнение нужного действия. Результат сохраняется либо в специальном регистре – аккумуляторе, либо в любом из регистров общего назначения (как в процессорах ARM).

В составе процессора, как минимум, имеются: регистр команд, принимающий очередную команду из памяти на расшифровку; счетчик команд, следящий за порядком выборки команд из памяти; операционный блок, выполняющий предусмотренные системой команд процессора операции; регистры общего назначения, содержащие исходные операнды и принимающие результат операции; устройство управления и синхронизации, обеспечивающее выполнение всех операций по синхросигналам от тактового генератора; система шин для подключения памяти и периферийных устройств.

2.4 Что такое Ассемблер?

Исполняемая программа на машинном языке представляет собой последовательность цифровых двоичных кодов определенной разрядности, так как любая информация в цифровых системах хранится и обрабатывается только в ***двоичном коде***. Написать такую программу – непростая задача. Поэтому разработчики процессоров дают каждой команде из набора команд некоторое символическое имя – ***мнемокод***, который,

как правило, является отражением действия, выполняемого командой. Например, «MOV» – от англ. move – переместить, «ADD» - от англ. add – сложить и т.д. Программа, написанная на языке мнемокодов конкретного процессора, называется ***программой на языке Ассемблер.***

Так, строка программы на ассемблере (для процессоров ARM)

```
MOV r0, r1
```

означает: переслать данные из регистра-источника r1 в регистр-приемник r0.

Таким образом, каждая команда из набора команд процессора имеет свой ***персональный мнемокод*** и свой ***собственный синтаксис***, который определяет, что может быть указано в поле операндов этой команды и в какой форме. Например, возможна команда пересылки данных

```
MOV r0, #1
```

в которой выполняется загрузка непосредственными данными регистра r0, то есть его инициализация константой 1. ***Непосредственными*** называются данные, значения которых указаны в самой команде – они присутствуют в формате команды вместе с кодом операции.

Для каждого типа процессора разработчиками процессора создается свой собственный язык: **Ассемблер конкретного процессора** – язык его мнемокодов команд, отражающий архитектуру процессора, его систему команд, состав и символические имена встроенных в процессор регистров общего назначения. Так, один из ассемблеров, созданный для программирования процессоров ARM, называется ARM ASM. Для одного и того же процессора разными фирмами может быть создано несколько языков ассемблера, отличающихся по своим возможностям (обычно, по составу и функциям входящих в язык псевдокоманд – директив). Далее в тексте книги слово «Ассемблер», набранное с заглавной буквы, будет обозначать конкретную версию языка ассемблер, а набранное с маленькой буквы – любой язык мнемокодов процессора.

Регистры общего назначения образуют сверхоперативную память процессора, то есть являются принадлежностью процессора, в отличие от памяти программ, которая является внешней. Доступ к регистрам общего назначения предельно быстрый. Эти регистры могут использоваться в качестве операндов практически во всех командах процессора.

Типовые операции в разных процессорах похожи. Поэтому, изучив один раз язык ассемблера одного процессора, Вы сможете освоить и ассемблер любого другого процессора. Мнемокоды команд ARM-процессоров достаточно просты, например, «B» от англ. branch – ветвление, переход, передача управления. Однако, они могут содержать и целый ряд дополнительных суффиксов и префиксов, которые уточняют или меняют содержание команды, например, «BNE», - перейти, если «не эквивалентно». Префикс «V» будут иметь, в частности, все команды сопроцессора поддержки операций с плавающей точкой. Он будет как-бы их визитной карточкой. При разработке мнемоники системы команд создатели процессоров пользуются четкой логикой, поняв которую, Вы сможете быстро освоить ассемблер. Мы поможем Вам в этом, последовательно объясняя логику разработчиков. Более того, мы будем изучать программирование на ассемблере не с точки зрения «Какие команды есть в системе команд?», а с точки зрения «Зачем они нужны и как их использовать для решения наиболее часто встречающихся практических задач?».

Итак, Ассемблер – язык мнемокодов конкретного процессора. Сколько различных процессоров – столько же и языков Ассемблер. Более того, для одного и того же процессора может существовать несколько версий языка Ассемблер. Говорят, что Ассемблер – ***машинно-зависимый язык.*** Но, это уже язык мнемокодов, который понятен программисту, на нем можно писать и редактировать программы. Для перевода программы, написанной на языке Ассемблер, в машинный код, разработчики процессоров создают специальные программы – ***трансляторы*** с языка Ассемблер. Это программы-

переводчики, которые получают на входе файл, написанный на языке Ассемблер конкретного процессора (созданный в одном из текстовых редакторов на компьютере), и создают в результате трансляции выходной файл в машинном коде, который может быть загружен в кодовую память и выполнен. Процесс трансляции программы с конкретного языка Ассемблер в машинный код называется ***ассемблированием,*** сама программа на компьютере, которая выполняет этот перевод – ***ассемблером.***

Ассемблер – это язык программирования, который позволяет наиболее полно использовать все имеющиеся возможности конкретного процессора. Но, это достаточно сложный язык, требующий времени и значительных усилий для его освоения. В отличие от Ассемблера, языки высокого уровня, такие как C/C++, являются ***машинно-независимыми языками***. Преимущество их использования состоит в том, что программа, написанная на языке высокого уровня, может быть выполнена на любом процессоре. Правда, для этого необходима уже другая программа – ***компилятор*** с языка высокого уровня в язык используемого процессора. Обычно компилятор сначала «переводит» программу с языка высокого уровня в программу на языке Ассемблер определенного процессора, а затем транслирует ее в машинный код с помощью транслятора с языка Ассемблер. Хотя сам язык высокого уровня машинно-независим, программы-компиляторы являются машинно-зависимыми.

На современном уровне развития микропроцессорной техники для разработки, отладки и загрузки программы в память микропроцессорной системы используются так называемые ***интегрированные среды разработки – IDE***. Они включают в себя не только трансляторы и компиляторы для ***целевого процессора*** (выбранного разработчиком), но и целый ряд дополнительных программ, обеспечивающих программиста удобными средствами для написания и отладки программ: специализированные тестовые редакторы, отладчики, загрузчики и т.д. Эта книга написана с использованием одного из таких пакетов программ Keil µVision IDE, основные возможности которого для написания и отладки программ будут постепенно раскрываться.

Ассемблер – язык программирования низкого уровня, который позволяет максимально эффективно использовать все возможности, как архитектуры процессора, так и его системы команд. Для получения программы в исполняемом машинном коде используется транслятор с Ассемблера.

2.5 Системы счисления, используемые в процессорной технике

В быту и в физике мы пользуемся позиционной десятичной системой счисления – DEC (Decimal), в которой каждое число представляется десятичными цифрами (0,1,2…9), вес которых определяется позицией соответствующей десятичной цифры в числе - табл. 2.2.

Таблица 2.2 Веса десятичных разрядов числа

Позиция	**i**	**…**	**2**	**1**	**0**	**-1**	**-2**	**…**	**-q**
Вес	10^{i}	**…**	10^{2}	10^{1}	10^{0}	10^{-1}	10^{-2}	**…**	10^{-q}

Положение

Здесь i – номер десятичной цифры в целой части числа, q – в дробной части. Например, целое число 137 можно представить как $1\times10^2+3\times10^1+7\times10^0$, дробь 0.345 как $3\times10^{-1}+4\times10^{-2}+5\times10^{-3}$, а вещественное число 77.23 как $7\times10^1+7\times10^0+2\times10^{-1}+3\times10^{-2}$.

В микропроцессорной технике минимальной единицей информации, которая может храниться в памяти, является ***бит***. Однобитовая память статического типа (SRAM) представляет собой обычный D-триггер (триггер данных). Совокупность некоторого числа D-триггеров образует ячейку памяти заданной разрядности или регистр. Каждый ***разряд*** ячейки памяти или регистра является битом и может принимать только два возможных значения 0 или 1. Поэтому, в процессорной технике для представления любых данных используется исключительно ***двоичная система счисления*** – Bin (Binary). Двоичная система тоже является позиционной и в ней вес любого разряда числа определяется степенью основания системы счисления 2^n, где n – номер разряда, отсчитываемый от положения двоичной точки – табл. 2.3. Само ***положение двоичной точки никак не фиксируется*** аппаратными средствами (под нее не отводится ни один разряд числа) – это прерогатива исключительно программиста. Только он знает, между какими разрядами числа стоит эта «***виртуальная***» двоичная точка в данных, с которыми он работает.

Таблица 2.3 Веса двоичных разрядов числа

Позиция	**i**	**…**	**2**	**1**	**0**	**-1**	**-2**	**…**	**-q**
Вес	2^i	**…**	2^2	2^1	2^0	2^{-1}	2^{-2}	**…**	2^{-q}
Dec			**4**	**2**	**1**	**0.5**	**0.25**		

Положение двоичной

Как видите, в двоичных числах можно использовать только две цифры: 0 и 1, вес которых зависит от положения двоичной цифры внутри числа. Десятичные цифры в программах на Ассемблере обычно записываются с суффиксом «d» в конце числа или без какого-либо суффикса, двоичные – с суффиксом «b». Например, целое число 15 может быть представлено как 1111b = $1\times2^3+1\times2^2+1\times2^1+1\times2^0$ =$1\times$**8**$+1\times$**4**$+1\times$**2**$+1\times$**1**. Вещественное число 1.75d, в котором под целую часть отведено два разряда и два разряда под дробную, так: 01.11b. Для того, чтобы конвертировать двоичное число в десятичное, нужно сложить десятичные веса всех единичных разрядов этого числа.

Иногда система счисления, в которой записано число, указывается основанием системы счисления в виде подстрочного индекса в конце числа, например, 388_{10}, 11001_2. В языке Ассемблер ей соответствует следующий вариант записи: сначала указывается основание системы счисления, а после него, через символ нижнего подчеркивания – последовательность цифр в данной системе счисления, например, 10_4888, 2_1101001.

Конечно, многоразрядные двоичные числа не очень удобны для восприятия человеком. На помощь приходит ***восьмеричная и шестнадцатеричная системы счисления***. В восьмеричной системе можно использовать только 8 цифр (0,1, 2, …7), а в шестнадцатеричной – 16 (0,1, 2, …,8,9,A,B,C,D,E,F). Эти системы удобны тем, что каждые три бита двоичного числа можно заменить одной восьмеричной цифрой, а каждые четыре бита двоичного числа – одной шестнадцатеричной. Запись становится более компактной, например: 254d = 11111110b = 0FEh = $15\times16^1+14\times16^0$ = 240+14, или в восьмеричной системе счисления 254d = 376q = $3\times8^2+7\times8^1+6\times8^0$ = 192+56+6.

Шестнадцатеричное число записывается с суффиксом «h» в конце числа, а восьмеричное – с суффиксом «q». Для записи 32-разрядного адреса в шестнадцатеричной системе счисления потребуется всего 8 цифр. Так, адресный диапазон 32-разрядного процессора ARM можно записать в виде: 00000000h÷0FFFFFFFh. Заметьте, что, если первой шестнадцатеричной цифрой является буква, то перед ней в Ассемблере нужно обязательно указывать 0.

В Ассемблере и в языке C/C++ используется запись шестнадцатеричных чисел в виде: 0x(последовательность шестнадцатеричных цифр). Максимальный 32-разрядный адрес выглядит при этом так: 0xFFFFFFFF.

Пример конвертирования 32-разрядного двоичного числа в шестнадцатеричное путем замены каждых четырех бит одной шестнадцатеричной цифрой показан ниже:

1	0	1	0	1	0	0	0	0	1	1	1	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	0	1	1	0
A				8				7				F				3				B				F				6			

Полученное число записывается в виде 0A87F3BF6h или 0xA87F3BF6.

Заметим, что в шестнадцатеричном формате в процессорной технике могут быть представлены и числа с плавающей точкой, а также любая другая информация – символы, строки, то есть любая произвольная последовательность бит.

Данные в микропроцессорных системах представляются только в двоичном коде – последовательностью бит. Для их более компактного описания применяются восьмеричная и шестнадцатеричная системы счисления. В языке Ассемблер допускается запись констант в любой системе счисления, удобной для программиста. Функцию автоматического перевода таких констант в двоичный код берет на себя транслятор с Ассемблера.

2.6 Сопроцессоры

Для расширения диапазона чисел, которые могут обрабатываться в процессоре, разработан специальный формат их представления – ***формат чисел с плавающей точкой***. Обработка таких чисел возможна как программным путем с использованием специальных библиотек, так и аппаратным – с применением дополнительных модулей, интегрированных либо на процессорную плату, либо непосредственно на кристалл базового процессора, что на порядок быстрее. Эти модули получили название ***сопроцессоров***. Хорошо известны сопроцессоры поддержки вычислений с плавающей точкой фирм Intel 80387 и Motorola 68881, которые в свое время позволили существенно поднять скорость вычислений в персональных компьютерах.

Наиболее часто используются сопроцессоры ***поддержки вычислений с плавающей точкой (Floating Point Unit – FPU)***, хотя разработаны и другие типы сопроцессоров (графические, для векторных вычислений и т.д.). Сопроцессоры имеют свою собственную систему команд и определенный интерфейс с центральным процессором.

Процессорные ядра Cortex-M4F содержат интегрированный на кристалл ЦПУ ***сопроцессор поддержки вычислений в формате с плавающей точкой однократной точности (32 разряда)***. Этим они заметно отличаются от большинства существующих серий 8- и 16-разрядных микроконтроллеров, которые не имеют встроенного сопроцессора. На рынок вышли микроконтроллеры, вычислительные возможности которых уже сопоставимы с возможностями процессоров для современных компьютеров. В них обеспечивается поддержка ***цифровой обработки сигналов*** (DSP) как с числами в формате ***с фиксированной точкой***, так и ***с плавающей точкой***, которая ранее была доступна исключительно для так называемых ***сигнальных процессоров*** *и* ***сигнальных микроконтроллеров***.

Вычислительные возможности рассматриваемых в этой книге процессоров Cortex-M фирмы ARM не уступают, а порой и превосходят возможности так называемых сигнальных процессоров (DSP), которые до сих пор использовались для решения самых сложных задач управления объектами в реальном времени.

2.7 Прерывания и исключения

2.7.1 Прерывания

Процессор может работать, общаясь с внешним миром путем ***опроса регистров ввода*** данных с внешних датчиков, их обработки и вывода результатов обработки в виде управляющих сигналов через ***порты вывода*** на исполнительные устройства. Однако, такой режим работы «*по опросу*» крайне неэффективен. Программа должна постоянно опрашивать большое число датчиков, чтобы не упустить важную информацию, что требует значительных ресурсов процессора (памяти и времени).

Уже первые попытки использования компьютеров для целей управления промышленными объектами (в 50-х годах прошлого века) показали, что программный опрос датчиков, так называемый, «программный поллинг» – крайне неэффективен. Представьте себе, что в системе имеется ряд аварийных датчиков: короткого замыкания, пожара, превышения допустимого уровня какого-либо вещества, его концентрации и т.д. Они срабатывают крайне редко и даже могут никогда не сработать (при соблюдении правил эксплуатации системы). Тем не менее, состояние этих датчиков должно быть известно цифровой системе управления в любой момент времени, чтобы быстро принять соответствующие меры, например, отключить питание, включить сирену и т.п.

Поэтому, в состав любой микропроцессорной системы обязательно включается ***контроллер прерываний*** – устройство, которое, работая в автономном режиме, постоянно отслеживает некоторые ***события*** и при их возникновении определенным образом информирует о них процессор. События могут быть как ***внешними***, возникающими во внешнем по отношению к микроконтроллеру оборудовании (например, перегрев двигателя или насоса в результате перегрузки, перенапряжение, короткое замыкание), так и ***внутренними***, возникающими во встроенных в процессор или микроконтроллер периферийных устройствах. В любом случае устройство, детектирующее такие ситуации, должно сообщить об этом событии контроллеру прерываний, выставив ***сигнал запроса прерывания*** (ЗПР). Различают запросы прерывания, поступающие ***по уровню сигнала (статические входы)*** или ***по фронту*** **(*динамические*)**. Чаще всего в процессорной технике используются запросы прерываний по переднему фронту импульса.

Примером внутреннего события может быть завершение преобразования аналогового сигнала с внешнего датчика (например, давления, расхода, температуры и т.д.) в цифровой код. При этом АЦП выставляет сигнал ***готовности данных*** в контроллер прерываний. Что должен сделать контроллер прерываний? Информировать процессор, что имеется запрос на немедленную приостановку обычного хода выполнения программы –- ***прерывание*** и выполнение специальных действий по обслуживанию запроса. В случае с АЦП это достаточно простые действия: обратиться в выходной порт АЦП и считать из него преобразованные в цифровой код данные, сохранить их в соответствующей переменной в ОЗУ для последующей обработки. После выполнения этих действий управление можно вернуть основной программе, в которой рассчитываются управляющие воздействия (выполняется основной алгоритм управления).

Программа, выполняющая действия по обслуживанию запроса прерывания, называется ***подпрограммой обработки прерывания*** или ***обработчиком прерывания***. Начальные адреса всех обработчиков прерываний размещаются в так называемой

таблице векторов прерываний, расположение которой в памяти каждого конкретного процессора фиксировано. Для каждого запроса прерывания выделяется в памяти место, в котором должен размещаться начальный адрес его обработчика – ***вектор прерывания***.

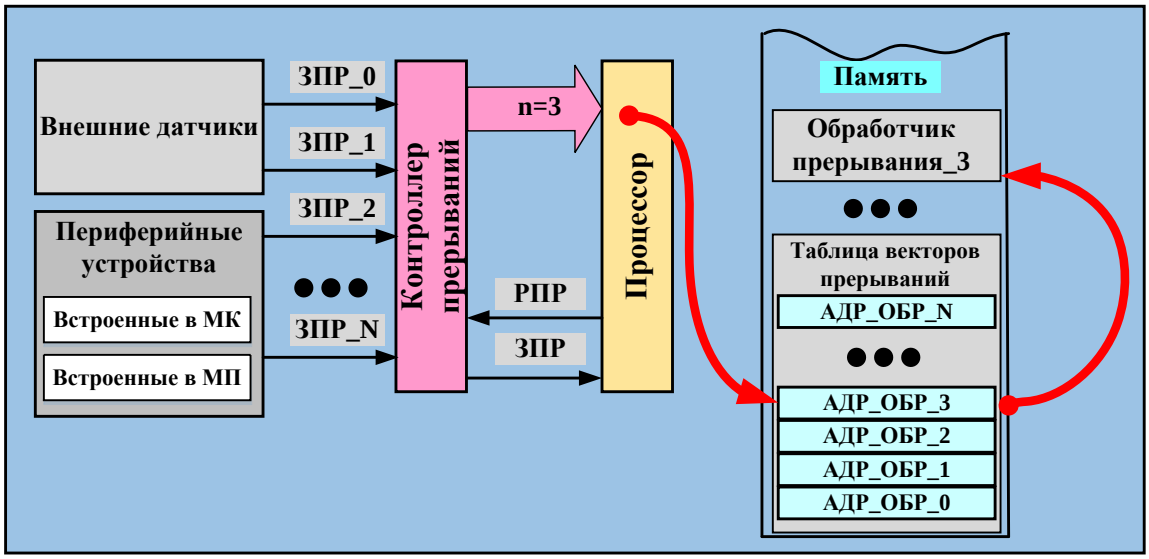


Рис. 2.2 Технология обслуживания прерываний в МПС

Тогда работу контроллера прерываний упрощенно можно представить так (рис. 2.2):

1. Возникает сигнал запроса прерывания от какого-то устройства.
2. Контроллер прерывания информирует процессор о возникшем запросе (ЗПР) и, получив от него разрешение на обслуживания прерывания (РПР), сообщает процессору номер возникшего запроса прерывания.
3. Работа основной программы приостанавливается. Содержимое важнейших регистров процессора (контекст) автоматически сохраняется в специальной области памяти (в стеке) вместе с адресом продолжения программы (адресом возврата).
4. По номеру поступившего запроса прерывания, процессор обращается к таблице векторов прерываний и считывает из нее адрес начала обработчика соответствующего прерывания.
5. Этот адрес загружается в счетчик команд PC и управление передается подпрограмме обслуживания прерывания.
6. В конце этой подпрограммы – обработчика прерывания – восстанавливается контекст, и управление возвращается в основную программу.

Что главное? То, что *в памяти процессора должна находиться таблица с начальными адресами всех возможных обработчиков прерываний*. По номеру запроса прерывания процессор должен *автоматически* обратиться к определенной позиции в этой таблице и считать из нее адрес соответствующего обработчика прерывания. Таблица должна быть запрограммирована пользователем, как и подпрограммы всех обработчиков прерываний.

Запросы прерываний могут поступать в контроллер прерываний от трех источников: 1) от внешних датчиков (например, короткого замыкания, перегрева и т.п.); 2) от периферийных устройств, интегрированных на кристалл микроконтроллера его производителем (таймеров, генераторов периодических сигналов и т.д.); 3) от периферийных устройств, встроенных в ЦПУ (например, от встроенных в процессоры ARM системных таймеров).

Схема, показанная на рис. 2.2, – упрощенная. Контроллер прерываний – сложное устройство, которое позволяет разрешить или заблокировать любой из запросов прерываний, назначить запросу определенный уровень приоритета и т.д. В процессорах ARM он называется ***контроллером вложенных векторных приоритетных прерываний*** – Nested Vectored Interrupt Controller (NVIC). Термин ***вложенные*** отражает тот факт, что более важные прерывания (имеющие более высокий приоритет) могут прерывать менее важные (с меньшим приоритетом).

2.7.2 Исключения

Термин «**исключение**» очень похож на термин «прерывание». В процессе выполнения программы пользователя могут произойти ***нештатные ситуации***, такие как:

- Обращение процессора за кодом операции к области памяти, в которой коды операций не могут находиться (в частности, к области памяти системной периферии процессора);
- Считывание из программной памяти кода несуществующей операции;
- Выполнение операции деления на ноль и другие.

Нештатные ситуации в процессе выполнения программы называются ***исключениями***. Для каждой из них в процессорах ARM предусматриваются свои собственные программы-обработчики. Их начальные адреса также, как и начальные адреса подпрограмм обработки прерываний, заносятся в ***таблицу векторов прерываний и исключений***. В аппаратную часть процессора включается специальный модуль – ***детектор исключительных ситуаций***.

Как только одна из таких ситуаций идентифицируется, происходит автоматическое обращение процессора к таблице векторов прерываний и исключений за получением начального адреса обработчика соответствующего исключения. Дальше аналогично обслуживанию прерываний: сохранение контекста и адреса возврата в основную программу (в стеке) и загрузка в счетчик команд PC начального адреса обработчика исключения с передачей ему управления.

В процедуре обработки исключения программист должен решить, что же делать в данной нештатной ситуации? Например, при делении на 0: можно вернуть в качестве результата максимально возможное значение и продолжить выполнение программы; или сообщить об ошибке оператору и прервать выполнение программы.

Заметим, что в большинстве 8- и 16-разрядных микроконтроллеров механизм обработки исключений вообще отсутствует. В микроконтроллерах с ядром Cortex-M он присутствует, что существенно *повышает надежность* работы программы и всей цифровой системы управления. Это особенно важно в ответственных объектах, где сбой может привести к серьезной аварии.

В системах управления оборудованием, работающих в реальном времени с большим числом встроенных периферийных устройств, не обойтись без контроллера прерываний. Надежность программного обеспечения существенно повышается, если в процессор встроена система детектирования нештатных ситуаций – исключений. Каждое исключение обрабатывается подобно запросу прерывания.

1) Какие три обязательных устройства должны входить в состав любой микропроцессорной системы?

2) В какой области памяти должна располагаться программа?

3) В каком регистре процессора находится код операции в процессе его декодирования?

4) Какой из регистров процессора отвечает за последовательность выборки команд из кодовой памяти?
5) В чем преимущество «отображения регистров периферийных устройств на память»?
6) В чем преимущество МПС с контроллером прерываний?
7) За счет чего повышается надежность программного обеспечения в МПС с обработчиками исключений?

1) Процессор, память, устройства ввода/вывода;
2) В области кодовой (программной) памяти – обычно в ПЗУ или ППЗУ (перепрограммируемом постоянном запоминающем устройстве), например, во флэш-памяти (допускает электрическое стирание и запись новых данных);
3) В регистре команд;
4) Счетчик команд PC всегда содержит адрес очередной команды, подлежащей выполнению. Автоматически инкрементируется на число байт, занимаемых в памяти текущей командой.
5) В том, что для доступа к регистрам можно использовать любые команды процессора, работающие с памятью.
6) Не нужно постоянно опрашивать состояние датчиков. Если событие произойдет, оно «уведомит» об этом контроллер прерываний, который, в свою очередь, сообщит процессору о том, что имеется запрос прерывания, и «уточнит» – от какого именно устройства или датчика.
7) Исключаются непредсказуемые ситуации в работе программы. Например, при считывании кода операции из области памяти, в которой код не может быть размещен или которой вообще не существует в системе. Идентифицируются некоторые аппаратные сбои, например, аварии шин МПС.

Список рекомендуемой литературы

1) Козаченко В.Ф. Микроконтроллеры: руководство по применению 16-разрядных микроконтроллеров Intel MCS-196/296 во встроенных системах управления. – М.: ЭКОМ, 1997. – 688 с.
2) Встраиваемые высокопроизводительные цифровые системы управления. Практический курс разработки и отладки программного обеспечения сигнальных микроконтроллеров TMS320x28xxx в интегрированной среде Code Composer Studio: учеб. пособие / А.С. Анучин, Д.И. Алямкин, А.В. Дроздов и др.; под общ. ред. В.Ф. Козаченко, – М.: Издательский дом МЭИ, 2010. – 270 с.
3) Анучин А.С., Козаченко В.Ф. Архитектура, система команд, технология проектирования и отладки специализированных сигнальных микроконтроллеров для управления двигателями: Лабораторный практикум. – М.: Издательство МЭИ, 2001. – 96 с.

3 ОСНОВЫ ДВОИЧНОЙ АРИФМЕТИКИ

Оглавление

В этой главе мы рассмотрим типы данных, которые могут обрабатываться в процессорах, а также основы двоичной арифметики. Эти знания позволят Вам лучше понять внутреннее устройство процессорных ядер Cortex-M, их архитектуру, а также оценить имеющийся в процессорах набор команд с точки зрения их использования для обработки данных разного типа.

Процессор работает с любой информацией как с простым набором бит. Никакого специального признака типа данных и даже места расположения двоичной точки в числе нет. Вся ответственность за содержимое данных и правильную их обработку, в том числе, за интерпретацию результата вычислений лежит на программисте. Только он знает, что представляют собой те или иные данные: целое число, вещественное, со знаком или без знака, или даже – символ или строка символов. В соответствии с конкретным типом данных, выбираются и нужные команды для их обработки.

Так же, как и в предыдущей главе, мы будем делать попутные замечания, касающиеся основной темы книги – процессорных ядер ARM Cortex-M.

Большинство приводимых в этой главе примеров будет касаться «как бы 8-разрядных процессоров» – в этом случае проще понять и проанализировать результат операции. Однако, правила двоичной арифметики являются общими для процессоров любой разрядности.

3.1 Целые числа

Целые числа могут быть любой разрядности 8-, 16-, 32-, 64-разрядными и более для представления больших по величине чисел. Обычно разрядность числа кратна восьми, числу бит в одном байте, хотя на практике возможны любые варианты, например, 10-разрядные целые. Часто это определяется разрядностью периферийных устройств, с которыми Вы работаете (например, информация, полученная с АЦП, может быть 10- или 12-разрядной). Обычно она размещается в младших разрядах соответствующих регистров периферийных устройств, являющихся 16-разрядными – в полусловах с точки зрения 32-разрядных процессоров ARM.

Целые числа могут быть как ***целыми числами без знака*** – ***Unsigned***, так и ***целыми со знаком*** – ***Signed***. Будем обозначать формат целого числа количеством разрядов и предшествующим ему префиксом «U» (для чисел без знака) или «S» (для чисел со знаком). Таким образом, возможны обозначения: U8, S8; U16, S16; U32, S32 и т.д.

Применительно к процессорам ARM данные первого типа называются **байтами**, второго – **полусловами**, третьего – **словами**. В 16-разрядных процессорах и микроконтроллерах словами называются данные типов U16, S16. Это естественно, так как именно эти числа обрабатываются в АЛУ 16-разрядных процессоров. В АЛУ процессоров ARM обрабатываются 32-разрядные числа. Именно они называются словами. Возможности процессоров ARM расширены так, что они могут обрабатывать не только слова, но и полуслова и даже байты, причем – параллельно: по два полуслова или по четыре байта одновременно.

3.2 Арифметика целых чисел без знака

Пример целого числа без знака 250 в формате U8 представлен ниже. Все восемь двоичных разрядов числа вносят свой вклад в значение числа в соответствии со степенью основания системы счисления 2^n, где n - номер разряда. Знаковый разряд отсутствует.

Целое число без знака 250							
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
27	**26**	**25**	**24**	**23**	**22**	**21**	**20**
128	**64**	**32**	**16**	**8**	**4**	**2**	**1**
1	**1**	**1**	**1**	**1**	**0**	**1**	**0**

Битовые операнды			**Результат**	
X	**Y**	**C**	**X+Y+C**	**C***
0	**0**	**0**	**0**	**0**
0	**1**	**0**	**1**	**0**
1	**0**	**0**	**1**	**0**
1	**1**	**0**	**0**	**1**
0	**0**	**1**	**1**	**0**
0	**1**	**1**	**0**	**1**
1	**0**	**1**	**0**	**1**
1	**1**	**1**	**1**	**1**

Сложение двоичных чисел выполняется по обычным математически правилам, «в столбик», начиная с младшего разряда. Если результат сложения двух бит оказывается равным основанию системы счисления (2), то он заменяется нулем и формируется ***флаг переноса*** в следующий (старший) разряд C. Сложение всех последующих бит, кроме младшего, выполняется «***по модулю 2***» с учетом переноса из предыдущего разряда. Это означает, что из результата, большего или равного 2, вычитается двойка и формируется перенос в следующий разряд C. В приведенной здесь таблице истинности показаны исходные значения битовых операндов X и Y и флага переноса C из предыдущего разряда, результат сложения X+Y+C, сохраняемый в соответствующем разряде суммы, и состояние флага переноса C* в следующий разряд суммы.

Пример операции сложения чисел 250 и 3 показан ниже. Состояние флага переноса C в следующий разряд отмечается левым надстрочным индексом рядом с битом суммы. Результат сложения числа 253 не выходит за максимально возможное значение целого 8-разрядного числа 255, следовательно, является достоверным:

	D7	D6	D5	D4	D3	D2	D1	D0
Первый операнд: 250d=0FAh	1	1	1	1	1	0	1	0
Второй операнд: 3d=03h	0	0	0	0	0	0	1	1
Результат сложения: 253d=0FDh	$^{0}1$	$^{0}1$	$^{0}1$	$^{0}1$	$^{0}1$	$^{0}1$	$^{1}0$	$^{0}1$

Если при сложении *двух самых старших бит числа* возникает ***перенос***, то это свидетельствует о **переполнении** – выходе результата за пределы разрядной сетки процессора (при работе с числами без знака).

Результаты всех арифметических и логических операций в АЛУ автоматически анализируются процессором и могут отображаться специальными флагами в регистре **статуса программы пользователя.** В процессорах ARM это происходит только тогда, когда программист указывает в команде специальную опцию «**S**» (Set – установить флаги результата операции). Один из таких флагов – флаг C (Carry – перенос) как раз и выставляется при сложении чисел без знака в результате возникшего переноса из последнего 31-го разряда.

Пусть выполняется арифметическая операция 254 + 3. Ее результат должен быть равен 257 (256 + 1). Он превышает максимально возможное целое 8-разрядное число. Поэтому, *в 8-разрядном процессоре* будет сформирован флаг переноса C. Единичное значение этого флага как раз и соответствует числу 2^8=256. Процессор сообщает: в рамках «моей» разрядной сетки получен результат 1, но не забудьте добавить к нему 256, если считаете, что складывали два числа без знака (цветами выделены ниблы операндов):

	C	D7	D6	D5	D4	D3	D2	D1	D0
Первый операнд: 254d=0FEh		1	1	1	1	1	1	1	0
Второй операнд: 3d=03h		0	0	0	0	0	0	1	1
Результат сложения:1d=01h	1	$^{1}0$	$^{1}0$	$^{1}0$	$^{1}0$	$^{1}0$	$^{1}0$	$^{1}0$	$^{0}1$

В 16- или 32-разрядом АЛУ флаг C будет выставляться только при выходе результата за пределы 16- или 32-разрядной сетки, соответственно. Следовательно, выполняя этот пример сложения на 16-разрядных или 32-разрядных процессорах, мы получим правильный результат, не требующий коррекции: 257d=0000000100000001b.

Операнды			Результат	
X	Y	B	X-Y-B	B*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	0
1	1	0	0	0
0	0	1	1	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	1

Вычитание целых чисел без знака выполняется аналогично сложению, начиная с младшего разряда. Отличие в том, что при вычитании может вырабатываться флаг «заема» из более старшего разряда числа в младший B (Borrow). Если «заем» возникает, то для следующего разряда из уменьшаемого нужно вычесть не только вычитаемое, но и флаг «заема». В таблице истинности битового вычитания показаны исходные значения операндов X, Y, а также бита «заема» B, сформированного при вычитании младших бит. В качестве результата указано значение X-Y-B, попадающее в соответствующий бит разности, а также значение флага «заема» B* из следующего (старшего) разряда.

Обратите внимание на вторую строку в таблице истинности: чтобы из 0 вычесть 1, нужно из более старшего разряда «занять» единицу; это будет соответствовать формированию флага заема B*=1; вес этой «занятой» единицы на самом деле вдвое больше – 2 в соответствии с весовыми значениями разрядов в двоичной системе счисления; вычитая (2-1), получим 1, что и отображено в качестве результата в таблице истинности.

Пример вычитания двух целых чисел без знака. Рядом с битовым результатом левым надстрочным индексом показано состояние флага B:

	D7	D6	D5	D4	D3	D2	D1	D0
Первый операнд: 83d=53h	**0**	**1**	**0**	**1**	**0**	**0**	**1**	**1**
Второй операнд: 58d=3Ah	**0**	**0**	**1**	**1**	**1**	**0**	**1**	**0**
Результат вычитания: 25d=19h	**^{0}0**	**^{0}0**	**^{1}0**	**^{1}1**	**^{1}1**	**^{0}0**	**^{0}0**	**^{0}1**

В большинстве современных процессоров отдельного флага «Заема» B нет. В качестве флага «Заема» *используется флаг переноса C*. Так же и в процессорах ARM: если при сложении возникает переполнение, то флаг C устанавливается. Если при вычитании возникает заем, то флаг C сбрасывается, а если заем не возникает, то, напротив, устанавливается. Можете для простоты считать, что при выполнении операции вычитания целых чисел состояние флага C формируется как инверсное значение флага «Заема».

3.3 Диапазон возможных целых чисел без знака

Диапазон возможных значений чисел без знака определяется разрядностью числа $0 \div (2^n-1)$, где n – число разрядов числа. Для типовых форматов чисел без знака получим:

Таблица 3.1 Диапазоны значений целых чисел без знака

Формат	**Диапазон возможных значений**
U8	**0÷255**
U16	**0÷65,535**
U32	**0÷4,294,967,295**

Как видите, в 8-разрядных процессорах диапазон возможных значений очень невелик, а в 32-разрядных вполне достаточен для большинства практических задач.

3.4 Целые числа со знаком. Арифметика чисел со знаком

Для представления в микропроцессорной системе ***целых чисел со знаком*** придется для представления знака выделить какой-то один из имеющихся разрядов, например, старший разряд числа. Теоретически для этого имеется несколько возможностей: 1) указать в специальном разряде знак числа (sign-magnitude), оставив для представления собственно значения числа все остальные разряды; 2) представить число в обратном коде (в виде дополнения всех разрядов числа до единицы - one's complement; 3) представить число в дополнительном коде - в виде дополнения до двух (two's complement).

Первый способ, на первый взгляд, кажется наиболее простым и называется представлением числа в **абсолютном коде**. Так, числа +9 и -9 в 8-разрядном формате S8 будут выглядеть так:

Число +9 в абсолютном коде							
S|7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	**0**	**0**	**0**	**1**	**0**	**0**	**1**

Число -9 в абсолютном коде							
S|7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	**0**	**0**	**0**	**1**	**0**	**0**	**1**

Значение самого старшего разряда определит знак числа: 0 – положительное, 1– отрицательное. То же отрицательное число -9 в 16-разрядном формате будет представлено так:

S|15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**

Обычные двоичные операции сложения и вычитания чисел в абсолютном коде *дают неверный результат*. Чтобы результат был правильным, необходимо либо

корректировать его, либо вводить в систему команд специальные операции типа: сложение, вычитание чисел со знаком в абсолютном коде. Поэтому, в процессорной технике этот способ представления знаковых чисел практически не используется.

Второй способ представления отрицательных чисел предполагает побитовое инвертирование соответствующего положительного числа – получение так называемого **обратного кода** положительного числа. Например, для числа -9, получим код 11110110b. Простое двоичное сложение таких одинаковых по величине положительных и отрицательных чисел даст единицы во всех разрядах – получим «отрицательный ноль». Значит, будут два нуля +0 и -0, что не очень удобно. Более того, результат двоичного сложения таких чисел опять дает результат, требующий коррекции или использования специальных команд для работы со знаковыми числами. Оба метода не удобны и редко применяются на практике.

В процессорной технике используется представление отрицательных чисел исключительно в **дополнительном коде** – третий метод. Это такое число, которое, будучи добавлено к своему положительному аналогу с использованием обычного двоичного сложения, даст *ноль в разрядной сетке*. По идее так и должно быть (+a)+(-a)=0. Этот метод называют также *дополнением до двух*. Что имеется в виду? То же самое, что и при сложении двух бит – результат должен обнулиться и возникнуть перенос в следующий разряд. Применительно к процессорам с разрядной сеткой любой длины: отрицательное число это такое, которое будучи прибавлено к соответствующему положительному числу с использованием обычной операции двоичного сложения, даст ноль в разрядной сетке и сформирует флаг переноса C за пределы разрядной сетки.

Технология получения отрицательного числа в дополнительном коде проста:

1) Получить обратный код положительного числа такой же величины путем побитового инвертирования всех разрядов, включая старший.
2) Добавить число с единицей в младшем разряде – 01b. Полученный результат и будет отрицательным числом в дополнительном коде.

Пример получения дополнительного кода числа (-2):

	D7	D6	D5	D4	D3	D2	D1	D0
Положительное число +2:	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
Обратный код числа +2:	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**
Прибавить единицу	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
Число (-2) в дополн. коде: 11111110b	^{0}**1**	^{0}**1**	^{0}**1**	^{0}**1**	^{0}**1**	^{0}**1**	^{0}**1**	^{1}**0**

При получении обратного кода используется обычная операция двоичного побитового сложения. Проверим, будет ли сумма (+2) + (-2) давать нулевой результат?

	C	D7	D6	D5	D4	D3	D2	D1	D0
Первый операнд: (+2)		**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**
Второй операнд: (-2)		**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**
Результат операции сложения: 0	**1**	^{1}**0**	^{1}**0**	^{1}**0**	^{1}**0**	^{1}**0**	^{1}**0**	^{1}**0**	^{0}**0**

Действительно, при сложении чисел с противоположным знаком в дополнительном коде, получаем внутри разрядной сетки ЦПУ ноль. Правда, при этом возникает флаг переноса C. Возникает вопрос, что с ним делать? Ответ такой: *ничего*, этот флаг имеет значение только в операциях с целыми числами без знака. Для контроля переполнений в операциях знаковой арифметики разработчики процессоров предусмотрели другой флаг V (Overflow) – **переполнение**. Именно он будет сигнализировать о том, что результат операции вышел за пределы представления знаковых чисел в рамках разрядной сетки процессора.

Главное преимущество дополнительного кода в том, что арифметические операции сложения и вычитания чисел без знака и со знаком можно выполнять ***по единой технологии, одними и теми же командами***. Есть только одно отличие – для контроля переполнений нужно анализировать не флаг C, а флаг V. Но, это уже дело не процессора, а программиста – он должен знать, с числами в каком формате работает!

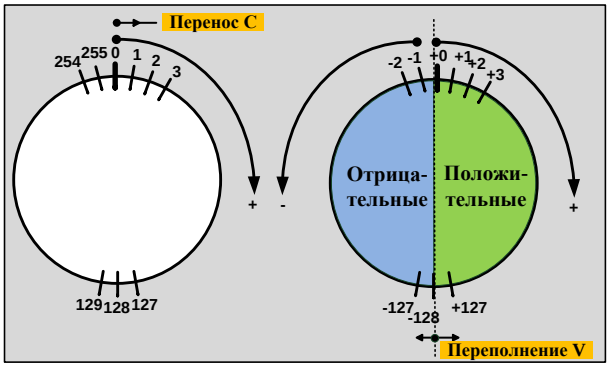


Рис. 3.1 Целые числа без знака U8 и целые числа со знаком в дополнительном коде S8

Графическая иллюстрация представления 8-разрядных чисел в виде чисел без знака U8 и со знаком в дополнительном коде S8 показана на рис. 3.1. В первом случае числа занимают весь диапазон возможных кодов (0-255), а во втором – диапазон делится ровно на две части: положительные числа (от +0 до +127) и отрицательные числа (от -1 до -128). Ноль считается условно положительным числом.

В 8-разрядном процессоре:

- При сложении чисел без знака и превышении (>=) суммой числа 256, из нее как бы вычитается число 256 и формируется флаг переноса C (см. рис. 3.1). Говорят, что сложение чисел без знака выполняется по модулю 256. Результат сложения любого числа байт будет правильным с точностью до 256.
- При сложении чисел со знаком, если результат больше максимально возможного положительного числа (+127) или меньше минимально возможного отрицательного числа (-128), формируется флаг переполнения V (см. рис. 3.1). Результат внутри разрядной сетки не корректируется – это дело программиста.

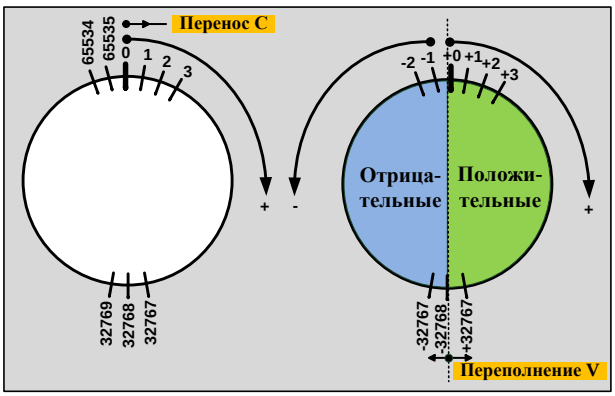


Рис. 3.2 Целые числа в 16-разрядных процессорах

Аналогично работают АЛУ в 16-разрядных и 32-разрядных процессорах. Отличие только в диапазонах представимых в этих процессорах целых чисел. На рис. 3.2 показаны диапазоны целых чисел без знака и со знаком в 16-разрядных процессорах.

3.5 Диапазоны целых чисел со знаком

Диапазоны возможных значений целых чисел со знаком в дополнительном коде определяются разрядностью числа от -2^{n-1} до $+(2^{n-1}-1)$, где n – число разрядов. Для типовых форматов получим (табл. 3.2):

Таблица 3.2 Диапазоны значений целых чисел со знаком в дополнительном коде

Формат	Диапазон возможных значений
S8	**-128 ÷ +127**
S16	**-32,768 ÷ +32,767**
S32	**-2,147,483,648 ÷ +2,147,483,647**
S64	**-2^{63}÷ $+(2^{63}-1)$**

В 16-разрядных процессорах, не говоря уже о 32- и 64-разрядных процессорах, диапазон представления целых чисел со знаком достаточен для решения множества практических задач. Задача программиста существенно облегчается – необходимость контроля переполнений возникает крайне редко.

Все арифметические операции над целыми числами без знака и целыми числами со знаком в дополнительном коде выполняются *одними и теми же командами процессора*. В первом случае для контроля переполнения используется флаг переноса C, а во втором – флаг знакового переполнения V.

1) Может ли процессор отличить, с каким числом в 8-разрядном формате он имеет дело: с числом без знака U8=255 или числом со знаком S8= -1?
2) Представьте числа +1 и -1 в дополнительном коде в форматах S8, S16, S32.

3) Может ли при сложении чисел в дополнительном коде с разными знаками возникнуть переполнение? А при сложении чисел с одинаковыми знаками?
4) Мы говорили о возможности выполнения в 32-разрядном АЛУ процессоров ARM одновременно двух арифметических операций над полусловами или четырех операций над байтами. Как, по-вашему, будут ли в этом случае формироваться 2 или 4 флага C и флага V?

1) Нет. Никаких признаков текущего формата числа нет. Об этом знает только программист.
2) Числа +1 и -1 в форматах S8, S16, S32 – см. рис. 3.3.

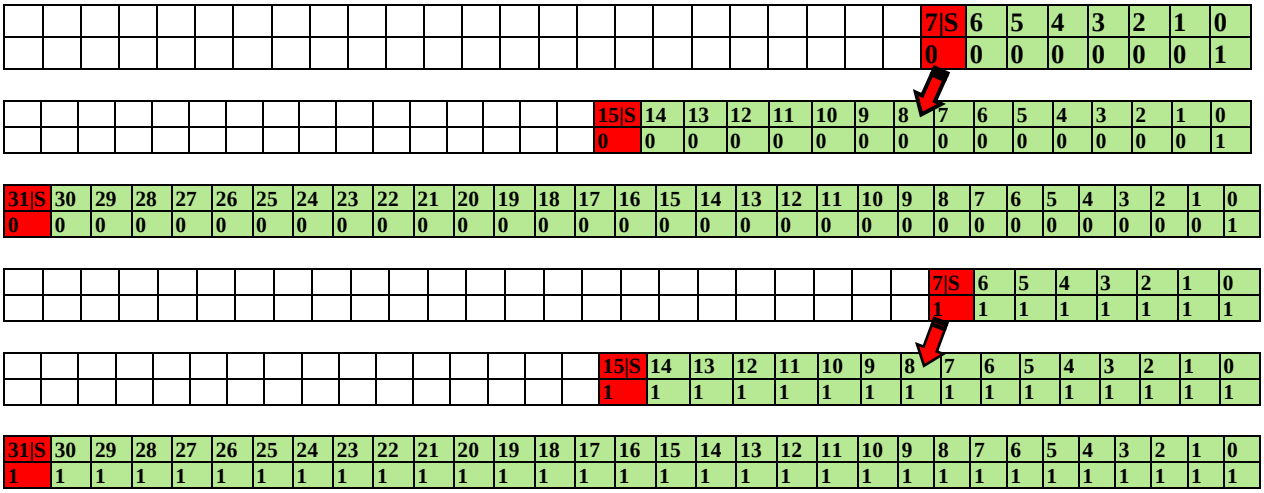


Рис. 3.3 Представление чисел +1 и -1 в различных форматах

3) Нет. При сложении чисел с разными знаками результат всегда будет по модулю меньше самого большого положительного или отрицательного числа. Переполнения исключаются. Именно при сложении чисел с одинаковыми знаками возможно «знаковое» переполнение, о чем и будет свидетельствовать возникший флаг V.
4) Нет. Эти флаги формируются исключительно при работе с числами, разрядность которых соответствует разрядности АЛУ. При параллельной работе с байтами выполняется сложение и вычитание по модулю 256, а при работе с полусловами – по модулю 65536. Все переносы и «заемы» игнорируются. Тем не менее, процессоры ARM предоставляют программистам дополнительные возможности анализа переполнений при выполнении параллельных арифметических операциях (см. приложение 1).

Для увеличения разрядности целого числа со знаком в дополнительном коде достаточно просто расширить знаковый разряд числа на все старшие разряды (см. рис. 3.3). Эта операция называется ***знаковым расширением числа***. Для ее выполнения предусматриваются соответствующие команды процессора, например, загрузки в регистр общего назначения (32-разрядный) байта или полуслова с автоматическим распространением знакового разряда в область старших разрядов.

3.6 Арифметика чисел большой разрядности

Разрядность арифметико-логического устройства (АЛУ) любого процессора фиксирована. Значит ли это, что он не может обрабатывать числа большей разрядности? Нет. В любом процессоре арифметические операции возможны над словами, имеющими любое число разрядов, кратное числу разрядов АЛУ: $n_{операнда}=k\times n_{АЛУ}$. Пусть складываются или вычитаются два операнда, каждый из которых состоит из нескольких слов от «Младшего слова» до «Старшего слова», и АЛУ процессора может обрабатывать слова данной разрядности. Тогда, сложение или вычитание «длинных» слов можно выполнить последовательно, как бы «в столбик», начиная с самого младшего слова:

- Младшие слова складываются с помощью обычной операции сложения ADD. При этом может сформироваться флаг переноса C. Все последующие (более старшие) слова складываются с помощью специальной операции сложения с учетом ранее возникшего переноса ADC.

- Младшие слова вычитаются с помощью обычной операции вычитания SUB. При этом может сформироваться флаг «заема» B. Все последующие (более старшие) слова вычитаются с использованием специальной операции вычитания с учетом ранее возникшего «заема» SBB.

C	C	C	C	C	
+	Ст. слово Оп1	…	Слово Оп1	Слово Оп1	Мл. слово Оп1
	Ст. слово Оп2	…	Слово Оп2	Слово Оп2	Мл. слово Оп2
	ADC или SBB	ADC или SBB	ADC или SBB	ADC или SBB	ADD или SUB

При такой технологии сложения/вычитания результат будет правильным как для целых чисел без знака, так и для чисел со знаком в дополнительном коде. Более того, результат будет правильным и для вещественных чисел с фиксированной точкой, при условии, что положение двоичной точки в исходных операндах одно и то же. Наличие флага C или флага V после операции сложения самых старших слов будет свидетельствовать о переполнении при работе с числами без знака или со знаком, соответственно.

Для работы с числами любой разрядности, превосходящей разрядность АЛУ, процессор должен иметь два типа команд сложения/вычитания: 1) без учета ранее возникшего переноса/заема; 2) с учетом ранее возникшего переноса/заема.

3.7 Как в процессорах с большой разрядностью обрабатывать числа меньшей разрядности?

Для этого необходимо предварительно преобразовать число в меньшем формате в число в большем формате, например, для 32-разрядных процессоров ARM: S8→S32 или S16→S32. Технология преобразования:

- Если число без знака (U8, U16), расширить его в область старших разрядов нулями.
- Если число со знаком в дополнительном коде (S8, S16), следует расширить его в область старших разрядов знаковым разрядом (см. рис. 3.3).

После такого преобразования форматов можно пользоваться любыми арифметическими командами, причем часто без опасности переполнения, если число операндов ограничено.

1) Сколько чисел без знака U8 можно сложить в 16-разрядном АЛУ без опасности возникновения переполнения? В 32-разрядном АЛУ?

2) Сколько чисел со знаком S16 (полуслов) можно сложить в 32-разрядном процессоре без опасности возникновения переполнения?

1) Разделим максимально возможное число без знака $U16_{max}$=65535 на максимально возможное число без знака U8max=255, получим 257. Для 32-разрядного процессора: $U32_{max}/U8_{max}$=4,294,967,296/255=16,843,009.
2) Разделим максимально возможное положительное число в формате S32 на максимально возможное положительное число в формате S16, получим: S32max/S16max=2,147,483,647/32,767=65538.

Как видно из приведенных выше примеров, в процессорах с большой разрядностью удобно обрабатывать массивы чисел с меньшей разрядностью. При этом за счет расширенной разрядной сетки процессора вероятность возникновения переполнения практически исключается.

Процессоры ARM, имея 32-разрядное АЛУ, могут эффективно обрабатывать и числа меньшей разрядности: 8- и 16-разрядные. Перед этим необходимо только выполнить операцию преобразования исходного формата числа в 32-разрядный формат.

Если арифметические операции с байтовыми числами (U8 или S8) или с полусловами (U16, S16) должны выполняться без контроля переполнений, то это можно сделать специальными командами *параллельного сложения/вычитания* сразу всех байт или всех полуслов исходных 32-разрядных операндов.

3.8 Арифметика действительных чисел с фиксированной точкой

Арифметика чисел с фиксированной точкой не отличается от арифметики целых двоичных чисел, *если двоичная точка в исходных операндах занимает одно и то же положение.* Если это не так, то операнды должны быть предварительно приведены к одному и тому же формату за счет сдвига числа влево или вправо на нужное число разрядов. При этом веса двоичных разрядов выравниваются, и операции сложения и вычитания будут корректными. Дадим краткий обзор форматов вещественных чисел.

3.8.1 Дробные числа без знака

Это числа, в которых двоичная точка «стоит» (*виртуально*) перед самым старшим разрядом, т.е. под целую часть числа не отведен ни один разряд. Весовые коэффициенты двоичных разрядов в направлении от старшего разряда к младшему равны: 2^{-1}, 2^{-2}, 2^{-3}, …, или 0,5; 0,25; 0,125… Пример 8-разрядной дроби без знака 0.625 показан ниже.

Дробное число без знака 0,625							
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
2^{-1}	**2^{-2}**	**2^{-3}**	**2^{-4}**	**2^{-5}**	**2^{-6}**	**2^{-7}**	**2^{-8}**
0,5	**0,25**	**0,125**	**0,0625**	**0,03125**	**0,015625**	**0,078125**	**0,00390625**
1	**0**	**1**	**0**	**0**	**0**	**0**	**0**

Для обозначения формата вещественных чисел будем по-прежнему использовать символ «U» или «S» для чисел без знака или со знаком, после которого указывать число бит, выделенное под целую часть числа I, включая знаковый разряд, и, через точку – число бит дробной части Q: U(I.Q) или S(I.Q). В соответствии с этим 8-разрядная дробь без знака будет иметь обозначение U(0.8).

3.8.2 Дробные числа со знаком в дополнительном коде

Как и для целых чисел, в дополнительном коде старший разряд числа отводится под знак (0 – положительное число, 1 – отрицательное), а все остальные разряды – под значение дроби. Естественно, что число разрядов дробной части при этом сокращается на один разряд. Пример числа +0.625 в 8-разрядном формате S(1.7) показан ниже:

Положительное дробное число +0,625							
D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
S	**2^{-1}**	**2^{-2}**	**2^{-3}**	**2^{-4}**	**2^{-5}**	**2^{-6}**	**2^{-7}**
	0,5	**0,25**	**0,125**	**0,0625**	**0,03125**	**0,015625**	**0,078125**
0	**1**	**0**	**1**	**0**	**0**	**0**	**0**

Технология получения отрицательных дробных чисел в дополнительном коде не отличается от уже описанной выше для целых чисел. Получаем обратный код положительного аналога дроби и добавляем 1 в самый младший разряд. Выполняем обычное двоичное сложение. Так, для числа (-0,625) имеем:

	D7	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Положительная дробь +0,625:	**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
Обратный код числа +0,625:	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**1**
Прибавить +1 в младшем разряде	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**
Число (-0,625) в дополнительном коде:	**1**	**0**	**1**	**1**	**^{1}0**	**^{1}0**	**^{1}0**	**^{1}0**

Проверим, будет ли сумма положительной и отрицательной дроби равна нулю?

	C	**D7**	**D6**	**D5**	**D4**	**D3**	**D2**	**D1**	**D0**
Первый операнд: (+0,625)		**0**	**1**	**0**	**1**	**0**	**0**	**0**	**0**
Второй операнд: (-0,625)		**1**	**0**	**1**	**1**	**0**	**0**	**0**	**0**
Результат операции сложения:	**1**	**^{1}0**	**^{1}0**	**^{1}0**	**^{1}0**	**0**	**0**	**0**	**0**

Да, при сложении дробей, одинаковых по модулю, но разных по знаку, результат в разрядной сетке процессора обнуляется и формируется флаг C, на который, как и при операциях с целыми числами со знаком не следует обращать внимания.

3.8.3 Действительные числа с фиксированной точкой

В этом случае определенное число старших двоичных разрядов I отводится для представления целой части числа (включая знаковый разряд), а оставшееся число разрядов Q используется для представления дробной части числа. Такой формат называется I.Q-форматом.

3.8.3.1 Действительные числа с фиксированной точкой без знака

В таких числах знаковый разряд отсутствует и все разряды перед «виртуальной» точкой используются для представления целой части числа U(I.Q). Ниже в качестве примера показано число 2,25 в формате U(3.5):

Вещественное число без знака 2,25							
D7	D6	D5	D4	D3	D2	D1	D0
2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
4	2	1	0,5	0,25	0,125	0,0625	0,03125
0	1	0	0	1	0	0	0

3.8.3.2 Действительные числа с фиксированной точкой со знаком

В формате S(I.Q) самый старший разряд отводится под знак. То же число 2,25 в формате S(3.5) представляется так:

Вещественное число со знаком 2,25							
D7	D6	D5	D4	D3	D2	D1	D0
S	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
	2	1	0,5	0,25	0,125	0,0625	0,03125
0	1	0	0	1	0	0	0

Технология получения дополнительного кода вещественных чисел с фиксированной точкой такая же, как для целых чисел.

1) Чем отличается формат числа U(16.16) от формата S(16.16)?
2) Нужно сложить два вещественных числа со знаком в форматах S(24.8) и S(16.16). Как выровнять положения двоичной точки в исходных операндах?
3) Будет ли при такой операции потеряна точность?

1) Первое число – без знака, под целую часть которого отведено 16 разрядов. Второе – число со знаком в дополнительном коде, под целую часть которого, включая знак, также отведено 16 разрядов.
2) Это не простой вопрос. Нужно сделать операцию *арифметического сдвига* второго операнда на 8 разрядов вправо. При этом положения двоичной точки станут одинаковыми, т.е. формат S(16.16) будет преобразован к формату S(24.8). При сдвиге вправо исходным знаковым разрядом будут заполнены все освободившиеся при сдвиге старшие разряды, а младшие 8 разрядов будут утеряны. В новом формате получим то же значение операнда, но с меньшим числом разрядов дробной части (вместо 16 разрядов – 8 разрядов).
3) Да, точность суммы будет на уровне точности операнда с меньшим числом разрядов дробной части.

Заканчивая эту главу, отметим, что процессоры Cortex-M4 имеют дополнительную поддержку операций по эффективной обработке сигналов, преобразованных в формат с фиксированной точкой (команды умножения с накоплением и насыщения), а процессоры Cortex-M4F – поддержку вычислений над числами в формате с плавающей точкой с помощью интегрированного на кристалл сопроцессора. Эти возможности подробно рассмотрены в последних главах книги.

Арифметические операции сложения и вычитания над вещественными числами с фиксированной точкой выполняются теми же командами, что и для целых чисел. Обязательным условием получения правильного результата является операция «выравнивания положения двоичной точки» в исходных операндах, реализуемая с помощью команд сдвига.

4 ОСНОВНЫЕ ТИПЫ ПРОЦЕССОРНЫХ АРХИТЕКТУР

Оглавление

4.1 Архитектура фон-Неймана

В современной компьютерной индустрии применяется несколько типовых архитектур построения процессоров. Познакомимся с ними. Это позволит оценить все достоинства и преимущества архитектуры процессоров ARM Cortex-M, изучаемых в этой книге (см. главу 5).

Базовые принципы построения процессоров были заложены в 1944 году коллективом ученых, разработавших первый в мире компьютер на электронных лампах «ЭНИАК» (Университет в Пенсильвании, США). По имени одного из них, фон-Неймана, и был назван *принцип совместного хранения в памяти кодов программ и данных*, известный сегодня как «Архитектура фон-Неймана». Процессоры с фон-Неймановской архитектурой отличаются следующими особенностями (рис. 4.1):

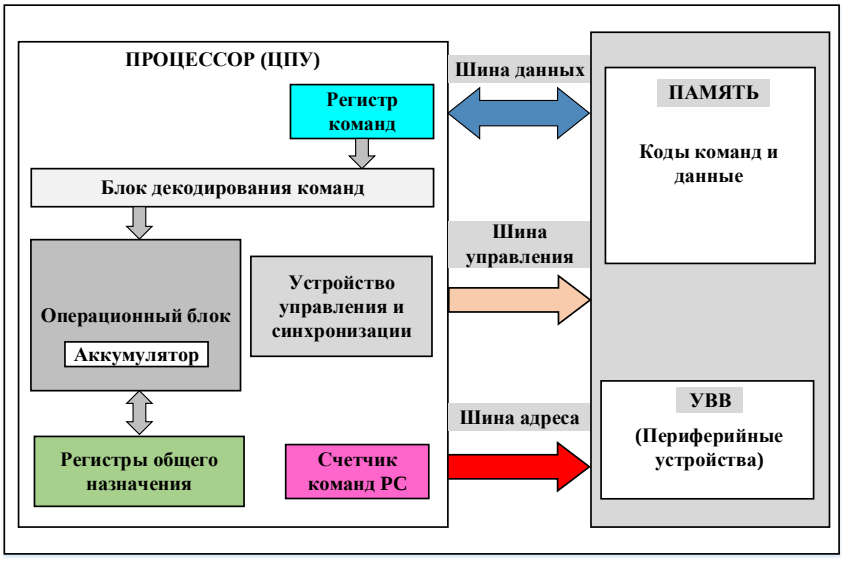


Рис. 4.1 Упрощенная фон-Неймановская архитектура

1) ***Программное управление***. Все действия, которые должен выполнить процессор, описаны в программе, расположенной в памяти. Программа представляет собой набор управляющих слов (кодов команд), которые «понятны» данному процессору, то есть могут быть декодированы и выполнены.

2) ***Последовательное выполнение команд.*** Команды считываются из памяти, расшифровываются и выполняются последовательно. За порядком выполнения команд следит специальный регистр процессора – *счетчик команд PC*, содержимое которого автоматически модифицируется процессором в зависимости от длины текущей команды. Он всегда содержит *адрес очередной команды, подлежащей выполнению*. Последовательное выполнение команд может нарушаться специальными командами условной или безусловной передачи управления, суть которых сводится к загрузке в счетчик команд PC нового адреса.
3) ***Память адресуется исключительно процессором***. Каждая ячейка памяти имеет свой персональный адрес, по которому процессор может обратиться к ней по чтению или записи, выставляя адрес этой ячейки на шину адреса. В памяти хранятся *слова информации в двоичном коде*, значения которых может интерпретировать только процессор, в зависимости от типа выполняемой в данный момент операции доступа к памяти (получение кода очередной команды или получение данных).
4) ***Направление передачи данных*** от памяти к процессору (чтение) или от процессора к памяти (запись) определяет только процессор, выставляя *на шину управления* либо сигнал чтения, либо записи данных.
5) ***Память является однородной***. Никакого специального признака типа хранимой в ней информации (данные или код операции) в памяти нет. Первый раз обращаясь к памяти, процессор «*по умолчанию*» считает, что расположенные там данные – код операции. Код операции автоматически попадает в *регистр команд* и подвергается расшифровке. Все последующие обращения процессора к памяти зависят от типа текущей команды. Если эта команда требует получения данных из памяти, то в процессе ее выполнения следует дополнительный цикл обращения к памяти, но уже за данными. Адрес этой ячейки памяти генерирует процессор и выставляет на шину адреса. Считанные из памяти данные попадают не в регистр команд, а в один из внутренних регистров процессора, обычно в один из регистров общего назначения.
6) ***Периферийные устройства могут адресоваться как память.*** Они могут быть подключены к тем же шинам, что и память. Регистры периферийных устройств имеют адреса, отображаемые на память, что позволяет работать с ними как с обычными ячейками памяти.

Большинство первых процессоров и микроконтроллеров имеют фон-Неймановскую архитектуру. Признаки этой архитектуры присутствуют и в большинстве других, современных процессорных архитектур.

Различают *аккумуляторную* и *регистр-регистровую* архитектуру. В первом случае результат любой операции в АЛУ сохраняется в специальном регистре- *аккумуляторе*. Более того, один из операндов может находиться в аккумуляторе по умолчанию. Следовательно, в формате команды будет адресоваться только второй операнд, что сокращает длину команды и делает ее выборку более быстрой. В ряде процессоров используется и двух-аккумуляторная архитектура. Это позволяет сохранить результат операции сразу в двух регистрах, например, при выполнении операции умножения, когда произведение имеет формат двух слов.

При увеличении объема сверхоперативной памяти процессора – числа регистров общего назначения, большая часть переменных может храниться не в памяти, а в регистрах (внутри процессора). Так как доступ к ним максимально быстрый, это повышает производительность МПС, одновременно сокращая формат команды (ее длину): число регистров *в регистровом файле* ЦПУ всегда ограничено; для их адресации может использоваться внутренняя шина адреса малой разрядности; поле адреса операнда, находящегося в одном из внутренних регистров ЦПУ, может быть коротким.

Рассмотрим простой пример. Если шина адреса памяти 16-разрядная и нужна команда сложения двух операндов в памяти, то при *прямой адресации операндов* для

указания их адресов в формате команды потребуется битовое поле длиной (16+16) бит. Это означает, что команда будет длиннее 4-х байт (с учетом кода операции) и будет долго извлекаться из памяти. Если те же операнды расположены в регистрах ЦПУ, общее число которых не превышает 256, то для указания адресов регистров-источников в формате команды потребуется битовое поле длиной только (8+8) бит. Формат команды становится существенно более коротким. Именно поэтому разработчики процессоров стремятся по минимуму использовать прямую адресацию операндов в памяти, заменяя ее *регистровой адресацией* операндов во внутренней памяти ЦПУ.

Регистр-регистровую архитектуру имеют многие современные 16-разрядные микроконтроллеры. Большая часть операндов извлекается из регистров общего назначения, в них же сохраняется результат операции. Как мы увидим далее, процессоры ARM также работают преимущественно с данными, расположенными в регистрах общего назначения, обращаясь к переменным в памяти только при необходимости (в основном при загрузке/сохранении данных).

4.2 Недостатки фон-Неймановской архитектуры

Наличие общих шин для обращения к памяти программ и памяти данных является *узким местом* первой процессорной архитектуры.

Представьте себе, что Вы работаете с 8-разрядным процессором Intel 8085, и выполняется команда загрузки в аккумулятор данных из ячейки памяти по адресу 8000h (шина адреса у этого процессора 16-разрядная): LDA 8000h (Load A –Загрузить аккумулятор). При 8-разрядной памяти формат этой команды следующий: первый байт – код операции; второй – младший байт 16-разрядного адреса ячейки-источника данных; третий – старший байт этого адреса). Процессор, считав в регистр команд код операции, расшифровывает ее и «понимает», что нужно загрузить данные из какой-то области памяти, адрес которой еще находится в памяти (в двух следующих ячейках) – это и есть так называемая *прямая адресация памяти*. Что он должен сделать? Сначала считать из памяти младший байт адреса, затем старший байт (в какие-то внутренние регистры процессора, обычно «теневые», невидимые для программиста) и только затем выставить полученный адрес на шину адреса и прочитать данные из нужной ячейки памяти (или из ОЗУ или из ПЗУ). Получается, что для выполнения команды процессору потребуется целых четыре цикла доступа к памяти. А если из памяти требуется извлечь два операнда – это уже 8 циклов. Все это время следующая команда будет ждать своей очереди на выполнение.

Говорят, что такая архитектура имеет *низкую пропускную способность канала связи между памятью и процессором* (интерфейса «Процессор» – «Память»). Этот недостаток становится особенно заметным при росте производительности процессора. Производительность процессора растет быстрее, чем быстродействие памяти, и именно *память становится тормозом* в повышении производительности всей микропроцессорной системы.

Часто фон-Неймановская архитектура процессора ассоциируется с архитектурой процессоров с полным набором команд – **CISC-архитектурой** (Complete Instruction Set Copmuter). Ее отличительная особенность – развитая система команд, которая допускает адресацию операндов непосредственно в памяти. Как следствие – длинные форматы команд (при прямой адресации), большое время, затрачиваемое процессором на считывание собственно кодов команд из памяти.

Наличие в процессорах с фон-Неймановской архитектурой общих шин для обращения и к памяти программ, и к памяти данных делает *одновременный, параллельный доступ к этим областям памяти невозможным*. Это означает, что считывать очередной

код команды из памяти и одновременно получать операнд из памяти для уже считанной команды, находящейся на этапе выполнения, невозможно.

Второй существенный недостаток связан с принципом однородности памяти. И память программ, и память данных находятся в общем адресном пространстве. Программа может располагаться в общем случае как в ПЗУ, так и в ОЗУ. При этом архитектура процессора не предполагает никаких аппаратных средств защиты области кодовой памяти с расположенной там программой от преднамеренного или непреднамеренного доступа по записи.

Расположенная в ОЗУ программа может быть подвержена действию вирусов или внешних атак. Защитой от них является только *размещение программного кода в постоянной памяти* (ПЗУ), запись в которую на стадии эксплуатации системы невозможна. Так и делается в микропроцессорных системах встроенного управления оборудованием. При этом программа размещается обычно в электрически программируемом постоянном запоминающем устройстве (ЭППЗУ).

К этому типу памяти относится и широко распространённая сегодня *флэш-память*, которая позволяет многократно стирать данные в памяти и записывать туда новые (до 10 тысяч циклов обновления без потери надежности). Ее отличительной особенностью является то, что для программирования не требуются дополнительные устройства – *программаторы*. Такой программатор встраивается в аппаратную часть самого микроконтроллера, а с помощью специальных программ – *загрузчиков флэш-памяти* – обеспечивает ее программирование, причем с обычного персонального компьютера по одному из стандартных интерфейсов (например, JTAG). Такие программы встраиваются и в интегрированные среды разработки, с которыми нам предстоит познакомиться.

В процессорах с фон-Неймановской архитектурой программа может располагаться не только в ПЗУ, но и в ОЗУ, например, в процессе отладки. Иногда некоторые функции могут специально размещаться в ОЗУ и выполняться оттуда, в попытке повысить производительность системы. Это возможно, так как быстродействие ОЗУ обычно выше быстродействия ПЗУ. Однако, гарантия «целостности» программного обеспечения исчезает. Фрагмент программы в ОЗУ не защищен от случайной записи, что может привести к сбою и даже к аварии. Поэтому, в системах управления оборудованием рекомендуется располагать программы преимущественно в ПЗУ (однократно программируемом при записи – ОППЗУ) или во флэш-памяти, допускающей многократную «перепрошивку» – перепрограммирование при модернизации программного обеспечения объекта. Только при таком подходе случайная запись в кодовую память на этапе выполнения программы становится аппаратно невозможной.

4.3 Гарвардская архитектура

Еще в 30-х гг. прошлого века (до второй мировой войны) в Гарвардском университете США была разработана процессорная архитектура, преимущества которой стали понятны лишь спустя несколько десятилетий. По иронии судьбы первый компьютер был реализован по архитектуре конкурентов из Принстонского университета – архитектуре фон-Неймана, описанной выше. Что же представляет собой Гарвардская архитектура?

1) ***Физически разная память для хранения команд и данных*** (кодовая память и память данных).
2) ***Физически разные интерфейсы «Процессор» – «Кодовая память» и «Процессор» – «Память данных».***

3) ***Физически разные интерфейсы «Процессор» – «Кодовая память» и «Процессор» – «УВВ».***

Выполнение первого пункта можно обеспечить и в процессорах с фон-Неймановской архитектурой. Два остальных пункта – отличительные. В этой архитектуре имеются три полностью изолированных друг от друга интерфейса процессора с кодовой памятью, памятью данных и устройствами ввода-вывода. При условии отображения адресов регистров периферийных устройств на память данных, последние два интерфейса становятся одним, общим.

Любая команда (например, сложение, умножение) требует получения двух операндов, выполнения действий над ними в АЛУ и сохранения полученного результата в памяти. Однако, код команды нужно сначала получить из памяти программ и расшифровать. Если кодовая память физически отделена от памяти данных, то операции получения/сохранения операндов и получения кода очередной команды *можно совместить,* повысив производительность процессора. При этом интерфейсы процессора с кодовой памятью и памятью данных могут существенно отличаться (по разрядности слова данных, тактовой частоте, протоколу обмена данными) – рис. 4.2. Более того, память данных может быть быстрой, а память программ – медленной, в том числе рассчитанной только на чтение (с аппаратной блокировкой записи). Память данных может быть малого объема (шина адреса малой разрядности), а память программ – большого (шина адреса большой разрядности).

Невозможность записи в память программ на стадии выполнения программы является средством *повышения надежности микропроцессорной системы*. Уменьшается вероятность как непреднамеренного, так и умышленного («хаккерская» атака) повреждения программы и программного сбоя.

В общем случае интерфейс процессора с периферийными устройствами может также отличаться от интерфейса с памятью данных, как по разрядности слова на шине данных, разрядности шины адреса, так и по протоколу обмена.

На рис. 4.2 не показана шина управления, которая, конечно, есть, но в этом случае она будет своя для кодовой памяти, памяти данных и периферийных устройств. В процессорах с Гарвардской архитектурой все три шины (шина данных, шина адреса и шина управления) условно объединяются в одну и называются *шинами интерфейса процессора с кодовой памятью, памятью данных, периферийными устройствами*. Для каждой из таких шин разработчики процессоров определяют разрядность данных, адресов, порядок обмена информацией, то есть *протоколы обмена данными*.

Итак, отличительная особенность Гарвардской архитектуры – *возможность параллельного выполнения нескольких действий сразу*: считывания кода очередной команды из кодовой памяти; чтения значений операндов из памяти данных или сохранения в ней результата предыдущей операции. Параллельно могут выполняться также операции получения очередной команды из кодовой памяти и чтения/записи в устройства ввода/вывода. Если память данных является *двухпортовой* (допускающей одновременную параллельную запись в нее данных по одному адресу и считывание данных по другому адресу), то возможен еще больший параллелизм: считывание очередного операнда для текущей операции и одновременное сохранение результата предыдущей операции (так и делается в современных сигнальных процессорах).

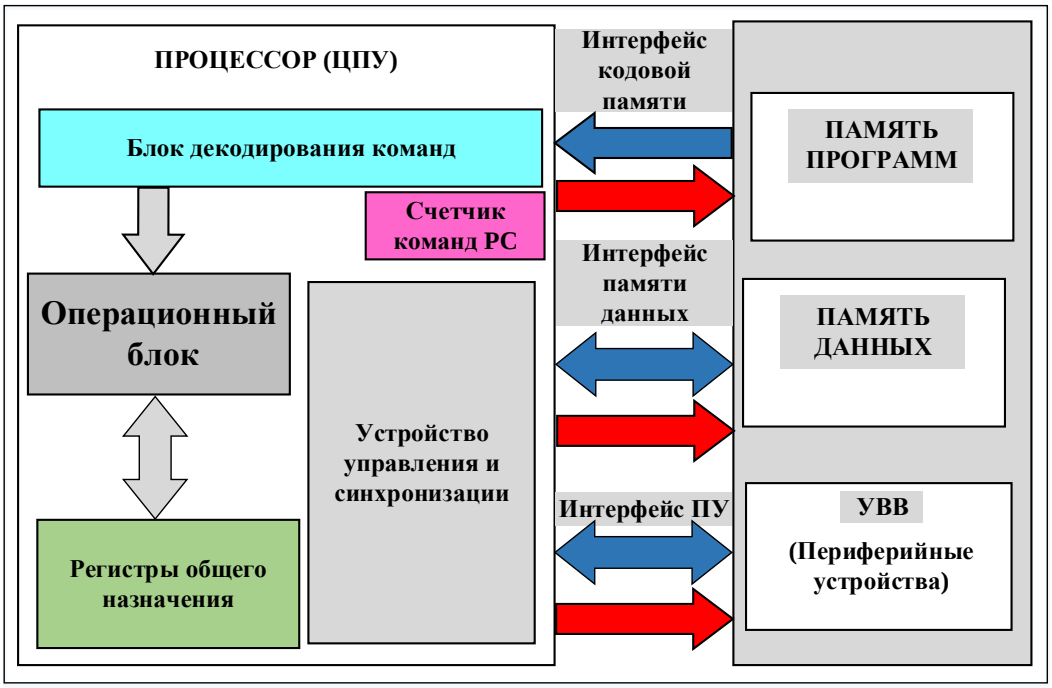


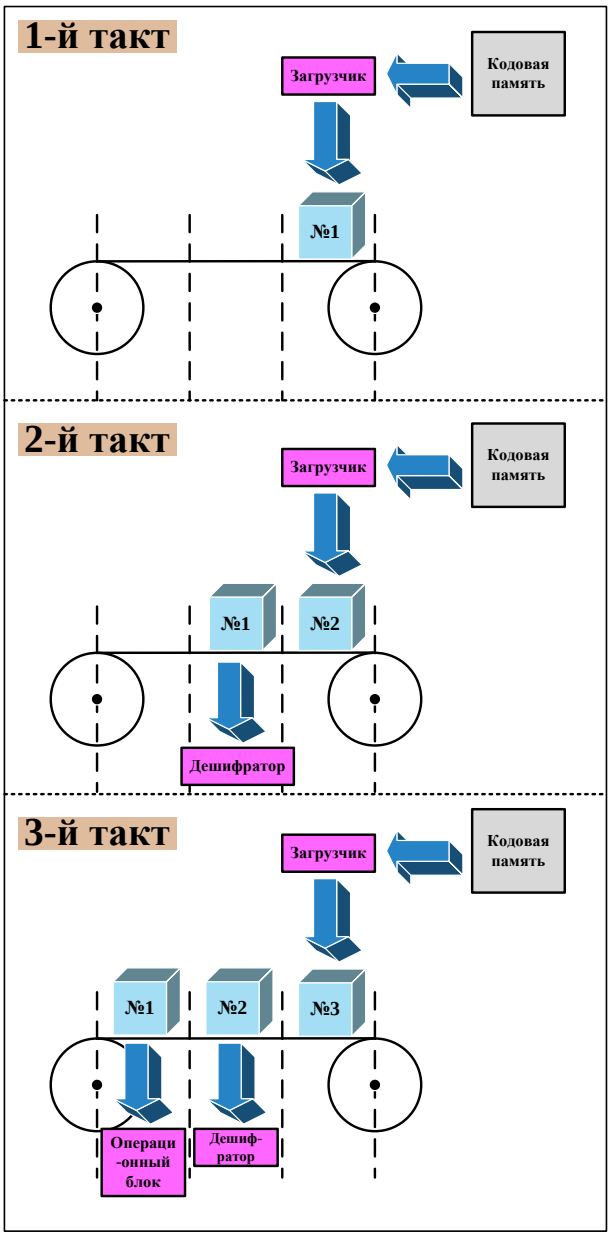
Рис. 4.2 Гарвардская архитектура процессоров

С одной стороны, использование этой архитектуры – существенное повышение быстродействия системы, а с другой – *значительное усложнение аппаратной реализации процессора*. Большое число шин означает также большое число выводов процессора, если элементы памяти – внешние, а также наличие для каждой из них своего собственного устройства управления и синхронизации. Именно усложнение аппаратуры задержало разработку процессоров с Гарвардской архитектурой на десятилетия. Она стала возможной только при резком повышении уровня интеграции транзисторов на кристалле и удешевлении процессорных БИС.

Более высокий уровень надежности определил области использования этой процессорной архитектуры – прежде всего встраиваемые в оборудование решения, где отказ в работе может привести к серьезным последствиям. Практически все разработанные в последние годы микроконтроллеры и сигнальные микропроцессоры имеют Гарвардскую архитектуру, в том числе и рассматриваемые в этой книге ядра ARM-Cortex-M3/M4/M4F.

Процессоры с Гарвардской архитектурой имеют отдельные оптимизированные интерфейсы с кодовой памятью, памятью данных и периферийными устройствами, что позволяет совместить несколько типовых операций по времени и резко поднять производительность процессора. Гарвардская архитектура предполагает (опционально) наличие дополнительных модулей управления памятью, которые имеют аппаратные средства блокировки несанкционированного доступа к заданным программистом областям памяти по записи для повышения надежности системы в процессе эксплуатации.

4.4 Конвейерная архитектура RISC-процессоров



Практически все современные процессоры, имеющие Гарвардскую архитектуру, обрабатывают команды на так называемом «*конвейере команд*» (рис. 4.3) и являются RISC-процессорами. Задача обработки команды разбивается на несколько этапов, на каждом из которых над командой выполняются строго определенные действия. Эти действия зависят от места расположения команды на конвейере. Каждое место (каскад конвейера) имеет свой собственный обработчик команды. В самом простом случае команду необходимо, как минимум:

1) ***выбрать*** из памяти по интерфейсу «Процессор» – «Кодовая память». Обработчик этого каскада извлекает очередную команду из памяти и размещает ее на конвейере – *загружает конвейер* новой командой;

2) ***декодировать*** код операции уже находящейся на конвейере команды – определить, что же должен делать операционный блок процессора, какая микропрограмма обработки данных должна быть запущена;

3) ***выполнить***. Обработчиком этого каскада конвейера является операционный блок процессора (включая АЛУ, умножитель, делитель, сдвиговый регистр, сопроцессор).

Рис. 4.3 Концепция конвейерной обработки команд

Все этапы конвейера выполняются за одно и то же время, называемое *тактом конвейера* (обычно равен одному *такту процессора* – периоду тактовых импульсов). Выше мы привели список задач, характерный именно для *3-х уровневого конвейера* процессорных ядер ARM-Cortex-M3/M4/M4F.

На 1-м такте на пустой конвейер загружается команда № 1. Далее (на 2-м такте) «конвейерная лента» продвигается на одну позицию влево и одновременно выполняются два действия: загружается новая команда № 2; ранее загруженная команда № 1 дешифрируется. «Конвейерная лента» сдвигается на одну позицию и на 3-м этапе одновременно выполняются уже три действия: декодированная команда № 1 выполняется в операционном блоке процессора; команда № 2 декодируется, а команда № 3 загружается на конвейер.

Далее на каждом очередном такте одновременно будут выполняться сразу три команды. Со стороны наблюдателя на выходе конвейера создается полное впечатление о том, что каждая команда выполняется всего лишь за один такт.

На самом деле каждая команда находится на конвейере три такта, но на разных этапах выполнения: загрузки, дешифрации, собственно выполнения.

Итак, конвейер команд очень похож на обычный автомобильный конвейер, на каждом этапе которого над автомобилем выполняются строго определенные действия (сварка кузова, окраска, установка двигателя, …, тестирование). На каждом этапе конвейера установлено особое оборудование (например, сварочные роботы) и работают специально обученные рабочие и инженеры. Команда в процессорной технике аналогична автомобилю, который последовательно продвигается по конвейеру и подвергается очередным этапам обработки – рис. 4.3.

Вот мы и подошли к понятию ***RISC-архитектура – Reduced Instruction Set Computer –*** компьютер (процессор) с сокращенным набором команд. Главной отличительной особенностью таких процессоров является выполнение большинства команд за *один такт процессора*. Это возможно только в *конвейерных архитектурах*, более того, в *многошинных Гарвардских*, когда отдельные интерфейсы с кодовой памятью и памятью данных позволяют центральному процессору независимо и одновременно обращаться к кодовой памяти для считывания кодов операций и к памяти данных для получения или сохранения результатов выполнения команд.

Сокращенное число команд означает, что число команд в наборе конкретного процессора не так велико и все они имеют одинаковый (или близкий) формат. Так, для процессоров ARM-Cortex – M3/M4/M4F – 32 или 16 разрядов, что заметно облегчает аппаратную организацию конвейера.

В более мощных процессорах, чем рассматриваемые в этой книге, например, в многоядерных ARM-процессорах, сигнальных процессорах и сигнальных микроконтроллерах число этапов конвейера может увеличиваться до 8 и более. Это могут быть специальные этапы, связанные с получением операндов из памяти данных и сохранением в ней результата предыдущей операции.

Команда на конвейере будет находиться столько тактов, сколько этапов имеет конвейер. Однако, с точки зрения наблюдателя на выходе конвейера (пользователя), каждая команда будет выполняться за один такт. Конвейер выполняет предварительную выборку команд из памяти и последовательную их обработку, существенно повышая производительность процессора.

В конвейерных архитектурах работа конвейера поддерживается несколькими регистрами ЦПУ. Обычно это следующие регистры:

- *Регистр – указатель адреса упреждающей выборки* содержит адрес очередной команды, которая будет считываться из кодовой памяти.
- *Регистр команд* – содержит код операции, подлежащий расшифровке в данный момент.
- *Регистр следующей команды в очереди* – содержит код операции очередной команды в очереди на дешифрацию.
- *Программный счетчик PC* – содержит адрес следующей, подлежащей выполнению команды.

Если для процессора без конвейерной архитектуры предположить, что время выполнения всех трех этапов (выборка, дешифрация, выполнение) одинаково и равно T, то общее время выполнения команды будет равно $t_{без\_конв} = 3T$. В процессорах с конвейерной архитектурой при полностью заполненном конвейере время выполнения

команды $t_{конв}$= T. Выигрыш в производительности процессора равен 3. На самом деле он несколько меньше (2.5 – 2.6) за счет дополнительных служебных издержек.

Если выполняется команда перехода или вызова подпрограммы, то *конвейер очищается* («все команды с него *смываются*») – загрузка конвейера начинается с нового адреса, как показано на рис. 4.3.

Дальнейшее совершенствование конвейерной архитектуры: *предсказание условных и безусловных переходов*, в которых задержки, связанные с очисткой конвейера, минимизируются.

4.5 Модифицированная Гарвардская архитектура

Любой современный микроконтроллер имеет процессор, встроенную память программ и данных определенного размера, ограниченный набор встроенных периферийных устройств. В большинстве практических применений список этих ресурсов микроконтроллера является основанием для его выбора. Это так называемая *«закрытая», нерасширяемая* архитектура микроконтроллера. Однако, в ряде приложений возникает необходимость при сохранении микроконтроллера либо расширить его память программ или данных, либо подключить к нему какие-то другие периферийные устройства. Такие микроконтроллеры являются *расширяемыми* и должны содержать шины для подключения внешнего оборудования. Они должны иметь так называемую *открытую архитектуру*. Но в Гарвардской архитектуре таких шин много. Сколько же потребуется дополнительных выводов, чтобы реализовать эту технологию?

Компромисс может состоять в том, что вся внутренняя архитектура микроконтроллера является настоящей многошинной Гарвардской, а *для сопряжения с внешними БИС памяти и внешними периферийными устройствами используются общие шины адреса и данных и общая шина управления*, содержащая сигналы чтения и записи, а также сигналы выборки нужной области памяти (селектирования). Такая архитектура получила название *модифицированной Гарвардской архитектуры*. Типичными представителями этой архитектуры являются сигнальные процессоры и сигнальные микроконтроллеры.

Наша книга посвящена изучению процессорных ядер ARM-Cortex-M3/M4/M4F, реализованных на базе Гарвардской архитектуры. Поэтому, более сложные варианты архитектур, используемые в высокопроизводительных процессорах, так называемые архитектуры многоядерных процессоров не рассматриваются.

Процессоры с Гарвардской архитектурой используются в наиболее часто применяемых на практике и относительно дешевых микроконтроллерах с определенным объемом памяти и фиксированным набором периферийных устройств (нерасширяемые МК). Процессоры с модифицированной Гарвардской архитектурой применяются в более сложных и дорогих микроконтроллерах, допускающих подключение дополнительной памяти и периферии (расширяемые МК).

1) Может ли программа в процессорах с фон-Неймановской архитектурой располагаться в ОЗУ?

2) А в процессорах с Гарвардской архитектурой?

3) Когда в процессорах с конвейерной архитектурой выполняется полная «перезагрузка» конвейера?

1) Может. Обычно это так и делается в процессе отладки. Программа может загружаться в ОЗУ и выполняться из него также в процессе эксплуатации (как в обычных компьютерах). В этом и состоит опасность случайной или преднамеренной ее «порчи».

2) Теоретически – нет, именно в целях максимальной защиты программы. На практике, как в процессорах фирмы ARM, разработчики процессоров предусматривают возможность обмена данными («мост») между интерфейсами памяти программ и памяти данных. Однако, в этом случае достичь такой же производительности, как при выполнении программы из кодовой памяти, невозможно. Этот режим используется только для отладки.

3) В том случае, когда последовательный ход программы нарушается, то есть при выполнении условного или безусловного перехода в другую точку программы, а также при вызове подпрограммы.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Козаченко В.Ф. Микроконтроллеры: руководство по применению 16-разрядных микроконтроллеров Intel MCS-196/296 во встроенных системах управления. – М.: ЭКОМ, 1997. – 688 с.
3) ARM DDI 0403C_errata_v3, ARMv7-M Architecture Reference Manual, Arm Limited, 2010.
4) ARM DDI 0337H, Cortex-M3. Technical Reference Manual. Arm Limited, 2010.
5) ARM DDI 0439D. ARM Cortex-M4 Processor. Technical Reference Manual. Arm Limited, 2013.
6) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User’s Manual. Texas Instruments Inc. – 2011.

5 АРХИТЕКТУРА ПРОЦЕССОРНЫХ ЯДЕР ARM CORTEX-M3/4/4F

Оглавление

5.1 Общие сведения

Процессорные ядра Cortex-M3/M4/M4F имеют во многом общую архитектуру и систему команд. Поэтому дальнейшее изложение будем вести на примере самого мощного ядра Cortex-M4F. Отличительные особенности процессорных ядер представлены в табл. 5.1.

Таблица 5.1 Особенности ядер Cortex-M

Ядро	Особенности
M3	**Ядро с Гарвардской архитектурой и трехступенчатым конвейером. Улучшенная базовая RISC-архитектура – ARMv7-M. Расширенная система команд, аппаратная поддержка умножения и деления. Система команд оптимизирована для типовых микроконтроллерных применений. Возможность замены обычных 8- и 16-разрядных микроконтроллеров с существенным повышением производительности за счет перехода к 32-разрядным операциям. Разработка программного обеспечения на Ассемблере и на языке высокого уровня C/C++.**

M4	**Добавлены команды цифровой обработки сигналов (DSP): умножение и умножение с накоплением 32-разрядных вещественных чисел с фиксированной точкой, насыщение. Возможность замены сигнальных микроконтроллеров с фиксированной точкой во встраиваемых приложениях.**
M4F	**Добавлен высокопроизводительный аппаратный сопроцессор поддержки вычислений с плавающей точкой однократной точности (32 разряда), регистровый файл для переменных в формате плавающей точки и развитая программная поддержка (система команд сопроцессора). Возможность замены сигнальных микроконтроллеров с плавающей точкой во встраиваемых приложениях.**

Процессорные ядра семейства Cortex-M разработаны фирмой ARM для использования в качестве *ядер высокопроизводительных микроконтроллеров*, представляющих собой законченные *системы на кристалле* System-on-Chip (SoC), ориентированные на разные области встроенного управления оборудованием. Фирма-производитель конечных изделий (микроконтроллеров) *самостоятельно добавляет к процессорному ядру* память программ, память данных и определенный набор периферийных устройств, ориентированный на предполагаемого потребителя. Фирма ARM, со своей стороны, *гарантирует полную работоспособность процессорного ядра* при условии, что все интерфейсы и протоколы обмена с дополнительными компонентами микроконтроллера будут соответствовать спецификациям фирмы.

Процессорное ядро является полностью 32-разрядным и имеет конвейерную RISC-архитектуру, рассчитанную на выполнение большинства команд за один такт, и низкое энергопотребление, что особенно важно во встраиваемых системах управления. Шины адреса и данных 32-разрядные, как и все регистры общего назначения. Тем не менее, поддерживается обработка данных в более коротких форматах – полуслово (16 разрядов), байт (8 разрядов) и даже бит (1 разряд). Явного «*битового сопроцессора*» в ядрах ARM Cortex-M нет. Однако, *его функции эффективно поддерживаются* общей системой команд и специальными областями побитово/побайтово-адресуемой памяти данных и периферийных устройств, так называемыми, битовыми лентами в памяти («семафорной» памятью). Технология bit-banding (битовых лент) подробно рассмотрена в книге. Доступ к биту в такой ленте выполняется всего лишь одной командой.

Одним из преимуществ процессорных ядер Cortex-M *является интеграция на кристалл центрального процессора важнейших периферийных устройств*, необходимых в каждом микроконтроллере для управления в реальном времени:

- Контроллера векторных приоритетных прерываний с минимизированными задержками на обслуживание прерываний;
- Системного таймера;
- Контроллера выхода процессора из режимов малого энергопотребления.

Конечные пользователи (разработчики прикладных систем) могут сами разрабатывать или покупать готовые *операционные системы реального времени и мониторы (*упрощенные операционные системы – специализированные программы верхнего уровня управления, рассчитанные на конкретную специфику области применения), использующие системные периферийные устройства. При этом обеспечивается независимость этих программных продуктов от объемов памяти и наборов периферийных устройств в конкретных типах микроконтроллеров.

Процессор имеет мощную встроенную систему отладки, значительно облегчающую и ускоряющую разработку программного обеспечения, конкретный состав блоков которой зависит от требований производителя микроконтроллера. Все современные стандарты интерактивной отладки, в том числе с использованием компьютерных интегрированных сред разработки, поддерживаются на аппаратном уровне процессорного ядра.

Особая гордость разработчиков – поддержка в Cortex-M4 команд цифровой обработки сигналов в формате с фиксированной точкой, а в Cortex-M4F – аппаратная интеграция в ЦПУ *сопроцессора вычислений с плавающей точкой* однократной точности со своей мощной системой команд, большинство из которых, в соответствии с RISC-технологией процессора, также выполняются за один такт. Эта опция переводит любой микроконтроллер с данным процессорным ядром *в разряд сигнальных высокопроизводительных микроконтроллеров*, причем, по предельно низкой цене и при минимальном уровне потребления энергии.

Как и любой RISC-процессор, ядра Cortex-M имеют систему команд, оптимизированную для использования языков высокого уровня и соответствующих компиляторов. Система команд процессоров настолько мощная, что, *зачастую, реализовать нужную функцию на Ассемблере даже проще, чем на Си*. Особенно это касается сопроцессора вычислений с плавающей точкой. Мы познакомим читателя с этими уникальными возможностями, позволяющими создавать предельные по быстродействию программы управления оборудованием.

Основная область применения процессорных ядер Cortex-M – системы реального времени управления, в том числе двигателями, силовыми преобразователями энергии, роботами, узлами автомобильной техники, распределенными системами автоматизации технологических процессов, медицинским оборудованием и т.п.

5.2 Архитектура процессорного ядра Cortex-M4

Упрощенная блок-схема процессора Cortex-M4 изображена на рис. 5.1. Предполагается, что в состав операционного блока включен опциональный модуль поддержки вычислений в формате с плавающей точкой FPU (ядро Cortex-M4F).

5.2.1 Операционный блок ЦПУ

1) Арифметико-логическое устройство (АЛУ) для выполнения арифметических, логических и других операций над 32-разрядными операндами с дополнительным кольцевым сдвиговым регистром, позволяющим выполнять так называемые «попутные» операции сдвига при выполнении большинства обычных операций.
2) Аппаратный умножитель 32-разрядных чисел с возможностью получения 64-битного результата.
3) Аппаратный делитель 32-разрядных чисел.
4) Файл регистров общего назначения (РОН) r0-r15, каждый из которых может быть источником или приемником операции.
5) Сопроцессор для обработки чисел в формате с плавающей точкой – модуль FPU - Floating Point Unit.
6) Файл регистров сопроцессора s0-s31, каждый из которых может быть источником или приемником операции с плавающей точкой.

Один из регистров общего назначения играет особую роль (R15), являясь счетчиком команд PC, содержимое которого задает адрес очередной команды в кодовой памяти, подлежащей выполнению.

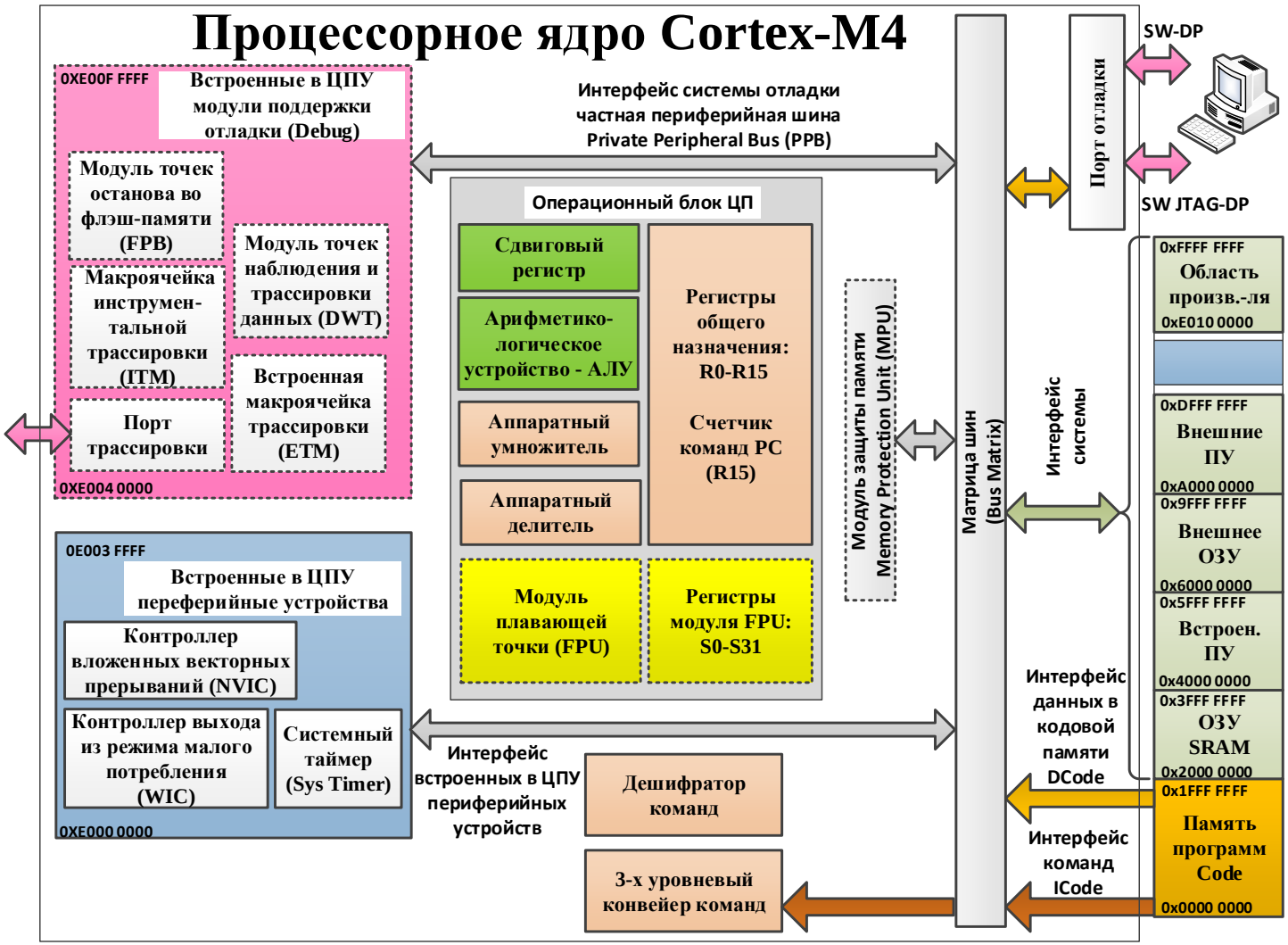


Рис. 5.1 Структура процессорного ядра Cortex-M4

5.2.2 Периферийные устройства, встроенные в ЦПУ

Мы уже отмечали, что одним из преимуществ процессорных ядер Cortex-M является интеграция непосредственно в процессор ряда важнейших периферийных устройств:

1) Контроллера вложенных векторных прерываний Nested Vectored Interrupt Controller (NVIC).
2) Системного таймера (Sys Timer).
3) Контроллера выхода из режима малого потребления энергии (WIC).

Интеграция такой периферии непосредственно в ЦПУ позволяет предельно минимизировать затраты времени на обращение к этим устройствам, в том числе, время реакции на прерывания, что является очень важным для систем управления реального времени.

С использованием встроенной периферии процессорного ядра специализированные фирмы разрабатывают и продают конечным пользователям так называемое *программное обеспечение верхнего уровня* (мониторы, операционные системы реального времени и т.д.). Это ПО может работать совместно с пользовательскими программами, являющимися *программным обеспечением нижнего уровня*, что позволяет создавать, так называемые, многозадачные приложения, в которых системы верхнего уровня координируют запуск и выполнение нескольких задач прикладного уровня.

5.2.3 Встроенные модули отладки ЦПУ

На современном этапе развития микропроцессорной техники базовые *средства поддержки отладки* интегрируются непосредственно на кристалл микроконтроллера или процессора. Фирма ARM не передает эту важную функцию разработчикам микроконтроллеров, а реализует ее сама в архитектуре процессорного ядра. Такой подход позволяет унифицировать все отладочные средства, методы отладки и отладочные интерфейсы любых микроконтроллеров, реализованных на процессорных ядрах ARM любыми производителями.

С их помощью поддерживаются:

1) *Доступ* в режиме отладки *ко всей памяти системы и регистрам* процессора, включая доступ к регистрам периферийных устройств;
2) *Установка* в отлаживаемой программе *точек останова*, при достижении которых выполнение программы приостанавливается и управление передается отладчику. Программист имеет возможность оценить промежуточные результаты работы программы, изучить состояние памяти и регистров процессора;
3) Установка *точек наблюдения*, при достижении которых значения переменных сохраняются для последующего анализа.
4) *Трассировка* программы, когда она выполняется либо в пошаговом режиме, либо от одной точки останова до другой с возможностью контроля значений переменных.
5) *Профилирование* программы. Оценка быстродействия (времени выполнения) всей программы или отдельных ее фрагментов (подпрограмм), частоты обращения к подпрограммам с целью последующей оптимизации наиболее часто используемых фрагментов кода или определения вообще не используемых участков кода.
6) *Вывод отладочной информации* на дисплей системы верхнего уровня (компьютер), используемый для отладки.
7) *Сопряжение с анализаторами данных* в порту трассировки Trace Port Analyzer (TPA).

8) *Доступ к встроенным в процессор отладочным устройствам* через два внешних интерфейса, к которым подключаются компьютерные *системы отладки верхнего уровня:*
   - *Последовательный отладочный JTAG-порт Serial Wire JTAG Debug Port* (SWJ-DP), являющийся стандартом, де-факто, для производителей отладочного оборудования и микроконтроллеров;
   - *Последовательный отладочный порт Serial Wire Debug Port* (SW-DP), широко используемый производителями ARM-процессоров и микроконтроллеров;
   - В конкретном микроконтроллере может быть либо один из этих интерфейсов, либо оба для сопряжения со средствами отладки разных производителей.
   - Часто JTAG-порт является универсальным и совмещается с SW-DP-портом.

При выборе микроконтроллера убедитесь в том, что по крайней мере один из указанных выше отладочных портов имеется на кристалле. Только в этом случае Вы будете иметь шанс на полноценную отладку своего программного обеспечения.

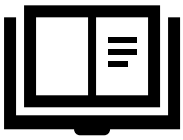
Конечному пользователю не требуется детальное знание специфики каждого отдельного отладочного модуля, входящего в процессорное ядро, в том числе протоколов обмена этих модулей с ЦПУ. Эта информация – для разработчиков и производителей микроконтроллеров (поставляется вместе с лицензией на процессор). Тем не менее, при выборе микроконтроллера желательно поинтересоваться, какие из возможных отладочных модулей включены в состав процессора при покупке лицензии у фирмы ARM.

Отмеченные выше отладочные функции поддерживаются несколькими модулями отладки, которые могут быть заказаны разработчиками микроконтроллеров в качестве опций при покупке лицензии на процессор (см. табл. 5.2).

Таблица 5.2 Набор отладочных функций

Модуль отладки	**Назначение модуля**	**Возможные варианты комплектации**					
DWT	**Data Watch point and Trace Unit – Модуль точек наблюдения и трассировки программы**	-	**Трассировка не поддерживается**	V	V	V	V
ITM	**Instrumentation Trace Macrocell Unit –Макро-ячейка (модуль) инструментальной трассировки для вывода данных вместо интерфейса JTAG через порт трассировки, что быстрее**	-		V	V	V	V
ETM	**Embedded Trace Macrocell – Макро-ячейка встроенной трассировки**	-		-	V	V	V
HTM interface	**AHB Trace Macrocell interface – Интерфейс макро-ячейки трассировки**	-		-	-	V	V
TPIU	**Trace Port Interface Unit – Модуль интерфейса порта трассировки для вывода трассировочной информации**	-					V
Debug Port	**Debug Port AHB-AP interface – Интерфейс отладочного порта AHB-AP**	-		V	V	V	V

Более подробную информацию не приводим. Для обычных пользователей достаточно знать, что все нужные средства отладки интегрированы на кристалл процессора и реализованы в выбранном Вами микроконтроллере.



Для того, чтобы внешняя аппаратура имела доступ к встроенным в процессор средствам отладки, в составе отладочных модулей процессора должен обязательно присутствовать модуль *порта отладки* Debug Port. Именно через него обеспечивается подключение внешних отладчиков, управление отладкой и доступ к памяти и регистрам процессора даже во время работы пользовательской программы – «на лету».

Обычно управление отладкой и получение всей отладочной информации выполняется через порт отладки (например, по интерфейсу JTAG). Если в микроконтроллере будет дополнительно присутствовать *порт трассировки*, то к нему могут быть подключены специальные устройства – *трассировщики*, собирающие и быстро выводящие отладочную информацию на экран компьютера. Это дополнительные аппаратные средства ускорения отладки.

В книге мы познакомим читателя с важнейшими приемами отладки, которые предоставляют современные *интегрированные среды разработки*, выполняющие свои функции во взаимодействии со встроенными в процессор модулями отладки.

Для сложных проектов требуется *отладка в реальном времени*, когда на фоне выполнения пользовательской программы, без остановки работы всех встроенных в процессор периферийных устройств, выполняется наблюдение за состоянием тех или иных переменных процесса, возможно, с графическим выводом результатов трассировки на экран отладчика. Для этой цели необходимы дополнительные внешние программные и/или аппаратные средства, описание которых выходит за рамки этой книги.

5.2.4 Система шин. Сопряжение процессора с другими модулями микроконтроллера

Процессорные ядра Cortex-M имеют Гарвардскую архитектуру, отличающуюся *раздельными областями памяти для хранения программного кода и данных* – рис. 5.1. Шина адреса для доступа к памяти и периферийным устройствам 32-разрядная. Все имеющееся адресное пространство проектировщиками процессора *жестко разделено на области*, в каждой из которой могут адресоваться *только определенные устройства*: 1) память программ (кодовая); 2) память данных – статическое ОЗУ; 3) регистры периферийных устройств, интегрированных на кристалл микроконтроллера фирмой-производителем микроконтроллера; 4) внешнее по отношению к микроконтроллеру ОЗУ; 5) внешние периферийные устройства, подключаемые к микроконтроллеру разработчиком конкретной (прикладной) микропроцессорной системы; 6) область памяти производителя микроконтроллера, в которой он может разместить любые дополнительные устройства по своему усмотрению.

В соответствии с поддерживаемой концепцией отображения адресов всех регистров периферийных устройств, встроенных в микроконтроллер на область памяти, определенные диапазоны адресов выделены и для регистров встроенных в процессор периферийных устройств (для системной периферии), а также для регистров модулей поддержки отладки. На рис. 5.1 для удобства читателей для каждого из блоков памяти и периферии указаны начальные и конечные адреса, в соответствии с принятым фирмой ARM унифицированным распределением памяти.

Процессор Cortex-M4 разработан в соответствии с архитектурой ARMv7E-M для микроконтроллерных применений и имеет три основных интерфейса:

1) **ICode memory interface** – для выборки кодов команд (I – instruction) из кодовой памяти;
2) **DCode memory interface** – для выборки данных (D – Data) из кодовой памяти;
3) **System interface** – системный интерфейс со встроенной памятью и периферийными устройствами микроконтроллера;

Назначение первого интерфейса – выборка кодов команд из памяти программ и загрузка их на конвейер команд для расшифровки и выполнения.

Назначение третьего интерфейса – доступ процессора к переменным, расположенным в оперативной памяти как по записи, так и чтению. Так как регистры всех периферийных устройств «отображены на память», то для доступа к памяти данных и периферийным устройствам используется одна общая шина. Одновременный доступ к данным в оперативной памяти и к периферийным устройствам не поддерживается.

Назначение второго интерфейса требует комментария. Дело в том, что в процессорах ARM используются как 32-разрядные шины адреса, так и 32-разрядные шины данных. Более того, сами команды являются 32-разрядными. В этой ситуации проблемой оказывается загрузка внутренних регистров ЦПУ константами (непосредственными данными). Константы тоже 32-разрядные. Как разместить их в 32-разрядной команде? Понятно, что это невозможно. Разработчики архитектуры ARM предложили оригинальное решение:

- Предусмотрели механизм компрессии (сжатия) констант, когда некоторая группа констант все же может разместиться в 32-разрядной команде (в 10 или 12 битах);
- Для всех же остальных констант предусмотрели механизм автоматического размещения их транслятором в свободных областях кодовой памяти с последующим доступом к ним с использованием относительной адресации по текущему значению счетчика команд PC и смещению (далее подробно в 9.3).

Именно для считывания таких, расположенных в кодовой памяти данных, и предусмотрен интерфейс DCode. Он автоматически начинает работать, когда встречается команда загрузки данных с относительной адресацией по счетчику команд. Более того, считывание данных выполняется настолько быстро, что команда-инициатор этого процесса (загрузки регистра константой), получив нужное значение из памяти, выполняется за один цикл, как и положено в процессорах с RISC-архитектурой. Можно сказать, что *считывание кода операции из кодовой памяти выполняется параллельно со считываем из нее данных*.

Все три внутренние шины выполнены в одном стандарте, разработанном фирмой ARM – Advanced High-performance Bus (AHB)-Lite – *расширенной высокопроизводительной шины* (AHB), *упрощенной* (Lite).

Четвертая шина Advanced Peripheral Bus (APB) – *расширенная периферийная шина* предназначена для интерфейса встроенной в процессор системы отладки с внешним отладочным оборудованием. Это обеспечивает независимую работу системы отладки от работы программы пользователя (отладку «на лету»).

Как видите, в процессоре Cortex-M4 полностью реализованы преимущества Гарвардской архитектуры, когда выборка команд может выполняться параллельно с выборкой данных по независимым шинам (как из кодовой памяти, так и из памяти данных), что и обеспечивает высокую производительность процессора.

5.2.4.1 Интерфейс выборки инструкций из кодовой памяти ICode

Обеспечивает доступ к кодам команд *только в кодовой памяти* (Code memory) в диапазоне адресов с нулевого адреса 0x00000000 по 0x1FFFFFFC (адреса последнего 32-разрядного слова в зоне кодовой памяти). Выборка кодов команд (инструкций) из этой области выполняется *исключительно 32-разрядными словами*.

Реальное количество кодов команд, извлеченных из кодовой памяти за один доступ, зависит от длины команд: поддерживается доступ как к 32-разрядным командам набора ARM, так и к 16-разрядным командам набора Thumb-2. В процессорах Cortex-M3/M4/M4F используется единая, общая система команд, объединяющая достоинства как 32-разрядных, так и 16-разрядных команд. Если команды 16-разрядные, то возможно считывание из программной памяти на конвейер сразу двух команд.

Процессор имеет 3-уровневый конвейер команд, на который и поступают извлеченные из памяти команды. Принцип работы конвейера команд подробно описан нами ранее (см. 4.4).

5.2.4.2 Интерфейс выборки данных из кодовой памяти DCode

В кодовой памяти могут располагаться не только коды инструкций, но и данные – константы и таблицы констант. Очень часто здесь находятся 32-разрядные адреса меток или точек входа в подпрограммы. По мере выполнения программы адреса должны считываться и загружаться в регистры процессора (должна выполняться инициализация 32-разрядных регистров-указателей).

Данные в кодовой памяти могут быть не только словами, но и полусловами и байтами. Поэтому, диапазон адресов доступа к данным в кодовой памяти 0x00000000 – 0x1FFFFFFF (адрес последнего байта). Всего в кодовой памяти 0,5 Гбайт.

Доступ к данным в этой области может запрашивать как процессорное ядро, так и отладочная система. Реально, при считывании байта или полуслова доступ всегда выполняется к полному 32-разрядному слову.

Считывание данных из кодовой памяти *имеет более высокий приоритет* по сравнению со считыванием кодов команд ICode, так как данные нужны для уже считанных и выполняемых в данный момент команд. Например, для команд доступа к данным с использованием относительной адресации по счетчику команд. Далее мы рассмотрим технологию доступа к так называемым «литеральным пулам» данных (константам) в кодовой памяти более подробно (см. 9.3).

5.2.4.3 Системный интерфейс (System interface)

Через этот 32-разрядный интерфейс обеспечивается доступ к следующим областям памяти микроконтроллера:

- 0x20000000 – 0xDFFFFFFF – область памяти встроенного в микроконтроллер статического ОЗУ, встроенной периферии, внешнего ОЗУ, внешних периферийных устройств.
- 0xE0100000 – 0xFFFFFFFF – область памяти, назначение которой определяет разработчик микроконтроллера.

Интерфейс системной шины содержит специальную логику, которая обеспечивает:

- Возможность не выровненного по границе 32-разрядного слова доступа к данным;
- Доступ к специальным областям по-битово-/по-байтово- адресуемого ОЗУ (к так называемым «битовым лентам») по 32-разрядным «псевдоадресам» битов;
- Выборку кодов команд из ОЗУ с последующей загрузкой на конвейер команд;
- Доступ к данным в памяти или регистрам периферийных устройств, запрошенным системой отладки.

Обратите особое внимание на то, что программный код можно при отладке размещать не в программной памяти, (например, во встроенной флэш-памяти), а в ОЗУ. Такую программу можно выполнить и отладить, но *время ее выполнения будет несколько больше,* так как системная шина не оптимизирована для работы с программными кодами (в частности нет отдельной шины считывания данных).

5.2.4.4 Собственная периферийная шина ЦПУ

Для того, чтобы обеспечить предельное быстродействие и минимизацию временных задержек при работе со встроенными в процессор модулями отладки, они подключаются к нему по отдельной расширенной высокопроизводительной шине

Advanced High-performance Bus (AHB), которая называется *собственной периферийной шиной процессора* – Private Peripheral Bus (PPB).

Все отладочные модули, являющиеся, как и процессор, собственной разработкой инженеров ARM, имеют оптимизированный по быстродействию протокол обмена, отличающийся от стандартного протокола обмена данными с внешними периферийными устройствами (протокола шины системного интерфейса). Эти устройства в области памяти от 0xE004 0000 до 0xE00F FFFF должны быть полностью совместимы с ядром процессора (Core-compatible). Поэтому обычно поставляются вместе с процессором. При обмене данными с ними имеются некоторые специальные ограничения, которые нежелательно использовать при работе с обычными периферийными устройствами, поставляемыми производителями микроконтроллеров.

Все собственные периферийные устройства процессора, подключенные к шине PPB в диапазоне адресов (от 0xE004 0000 до 0xE00F FFFF), могут быть доступны *только в привилегированном режиме* работы процессора (privileged mode) – *см. 5.4*. Доступ к ним выполняется только программами-отладчиками, создаваемыми специализированными фирмами. Обычные пользователи имеют доступ к этим устройствам только опосредованно – через купленные ими аппаратно-программные средства поддержки отладки.

Доступ к системным периферийным устройствам, таким, как контроллер прерываний, также выполняется по периферийной шине процессора. Отличие состоит в том, что он может выполняться и в *непривилегированном режиме*. Это связано с тем, что контроллер прерываний широко используется практически всеми разработчиками конечных изделий в любой прикладной программе и, поэтому, должен быть максимально доступен.

5.2.4.5 Матрица шин

Вы уже обратили внимание на сложность шинной организации процессора, например, на то, что программа может выполняться из *оперативной памяти*. Как достигается такая универсальность? За счет использования своеобразного шинного мультиплексора (переключателя) – *матрицы шин* (Bus Matrix). Она позволяет направить поток данных в нужное в данный момент устройство, выполняя функцию шинного соединителя.

Итак, каждая шина по физической структуре и протоколу обмена оптимизирована для выполнения своих специализированных операций, а *мостом между ними* является матрица шин, которая выполняет не только физическое соединение шин между собой, но и согласовывает протоколы обмена данными по этим шинам.

1) Почему процессорные ядра Cortex-M относятся к процессорам с Гарвардской архитектурой?
2) Вспомните, какие этапы имеет 3-уровневый конвейер команд?
3) Какие операции в процессорных ядрах Cortex-M могут выполняться параллельно?
4) В чем преимущество процессорных ядер ARM, содержащих внутри системную периферию и модули поддержки отладки?
5) Какие области памяти не обслуживаются системным интерфейсом?

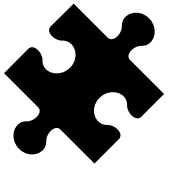
1) Они имеют физически разные области памяти для хранения кодов команд и данных и физически разные шины для доступа к ним, оптимизированные по быстродействию и протоколам обмена.
2) Выборки команды, ее дешифрации, выполнения.

3) Операции считывания кода команды из кодовой памяти с загрузкой ее на конвейер; декодирования ранее выбранной команды; выполнения уже декодированной команды, в том числе с доступом к переменным в оперативной памяти или к регистрам периферийных устройств. Возможно также параллельное обращение за данными, расположенными в кодовой памяти – по шине DCode.
4) Системное программное обеспечение мало зависит от частоты работы конкретного микроконтроллера, объема встроенной в него памяти и набора встроенных периферийных устройств. Оно сохраняется, например, при переходе на более производительный микроконтроллер с большим объемом памяти или отличающийся другой периферией.
5) Это область памяти программ и область памяти, отведенная под собственные периферийные устройства процессора – контроллер прерываний, системный таймер, модули поддержки отладки (адресный диапазон 0xE000 0000 – 0xE00F FFFF).

5.2.5 Опциональные модули процессора Cortex-M4

Некоторые блоки процессора Cortex-M4 являются опциями и поставляются по требованию производителя микроконтроллера:

- Опциональный *модуль защиты памяти* Memory Protection Unit (MPU), предназначенный для защиты определенных областей памяти от случайного или несанкционированного доступа. Используется только в достаточно сложных системах, как правило, имеющих операционную систему реального времени. Области памяти, которые использует программное обеспечение верхнего уровня, могут быть заблокированы для доступа со стороны обычных пользовательских программ, что повышает надежность работы всей системы.
- Опциональный *модуль обработки чисел в формате с плавающей точкой однократной точности* Floating Point Unit (FPU). Одна из важнейших опций, существенно расширяющих вычислительные возможности и производительность процессора. Дальше рассматривается подробно (см. 20.1).
- Модуль Flash Patch Breakpoint (FPB) – *установки точек останова во встроенной флэш-памяти микроконтроллера* и локальных вставок («заплаток») во флэш-памяти.
- *Модуль выхода процессора из режимов малого потребления энергии* Wake-up Interrupt Controller (WIC) – контроллер «пробуждающих» прерываний. Позволяет отключать питание процессора с сохранением его состояния и быстро возвращать в работу при возникновении прерывания.
- Debug Port – *Интерфейс отладочного порта* AHB-AP.
- Bit-banding – *Аппаратная поддержка «битовых лент»* в памяти данных с возможностями доступа индивидуально к каждому биту, ориентированная на эффективную обработку бит, фактически поддержка «битового сопроцессора».
- Constant AHB control – *Средства постоянного контроля шин* AHB.



Прежде чем выбрать для применения определенный микроконтроллер, конкретной фирмы-производителя убедитесь в том, что его процессорное ядро Cortex содержит нужные Вам опции.

Если Вы используете микроконтроллер с ядром Cortex-M4F, то это не значит, что модуль поддержки вычислений с плавающей точкой готов к работе «по умолчанию». Его работу необходимо предварительно разрешить, как и любого сопроцессора. Это же относится к любым опциональным модулям. Далее мы покажем, как выполняется инициализация процессора – устанавливается нужный режим работы и конфигурируется его окружение.

5.3 Основные технические характеристики ядра Cortex-M4

Ядро **Cortex-M4 core** выполняет функции быстродействующего центрального процессора со следующими возможностями:

1) Единый набор команд Thumb, разработанный для архитектуры процессоров фирмы ARM®v7-M, объединяющий преимущества двух ранее существовавших наборов команд: 32-разрядного набора команд ARM и 16-разрядного набора Thumb-2. Обеспечение более высокого быстродействия и одновременно более высокой плотности кода. Автоматический выбор более компактной команды в процессе ассемблирования программы;
2) Работа с операндами, расположенными исключительно в регистрах ЦПУ R0-R15;
3) Выполнение большинства команд за один такт.
4) Основной пользовательский режим работы Thread (или User Mode) и режим обработчика исключений Handler Mode, в котором может работать специализированное системное ПО типа операционной системы реального времени;
5) Два состояния процессора: выполнение команды единого набора Thumb и состояние отладки Debug, в котором обеспечивается доступ отладчика ко всем ресурсам;
6) Аппаратный умножитель 32-разрядных чисел в формате с фиксированной точкой с поддержкой операций умножения-накопления;
7) Аппаратный делитель целых чисел без знака и со знаком;
8) Банк из двух указателей стека Stack Pointer (SP) для разных режимов работы процессора (пользовательского и обработки исключений). Возможность работы системного и пользовательского ПО с разными стеками для повышения надежности;
9) Автоматическое сохранение и восстановление состояния процессора при обработке прерываний с минимальными задержками вызова процедур обработки прерываний и возврата в основную программу;
10) Поддержка прерываемых операций множественной пересылки данных из регистров в память и обратно (команды LDM, STM, PUSH и POP) с минимальными задержками входа и выхода в/из процедур обслуживания прерываний Interrupt Service Routine (ISR);
11) Поддержка двух вариантов расположения байт в словах: прямого и обратного;
12) Поддержка не выровненного доступа к байтам для более эффективного использования памяти данных.

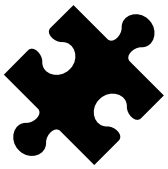
Модуль **аппаратной поддержки вычислений** в формате с плавающей точкой Floating Point Unit (FPU) выполняет функцию сопроцессора и обеспечивает:

1) Обработку операндов, расположенных в 32-разрядных регистрах модуля плавающей точки S0-S32, которые представляют собой дополнительное сверхоперативное ОЗУ модуля FPU. Регистры S0-S31 могут быть доступны и как шестнадцать 64-разрядных регистров, содержащих двойные слова (double-word);
2) Операции обработки 32-разрядных чисел с плавающей точкой однократной точности: преобразования форматов, сложения, вычитания, сравнения, умножения, умножения с накоплением, деления, извлечения квадратного корня;
3) Операции обычного последовательного умножения с накоплением (Multiply and Accumulate) и одновременного умножения с накоплением повышенной точности – (Fused MAC);
4) Расширение диапазона чисел с плавающей точкой однократной точности за счет поддержки операций с денормализованными (denormals) числами и всех режимов округления, предусмотренных международным стандартном IEEE 754.

5) Извлечение и выполнение команд обработки чисел с плавающей точкой на трехстадийном конвейере с независимой работой стадий конвейера.
6) Выполнение большинства команд сопроцессора за один такт.

Контроллер вложенных векторных прерываний Nested Vectored Interrupt Controller (NVIC) полностью интегрирован в ядро процессора для предельного уменьшения задержек в обслуживании прерываний. Он обеспечивает:

1) До 240 внешних запросов прерываний с номерами от 1 до 240;
2) Поле битов приоритетов прерываний размером 3÷8 бит, позволяющее задать от 7 до 256 уровней прерываний, соответственно;
3) Динамическую смену приоритетов прерываний – (dynamic reprioritization);
4) Группировку запросов прерываний по приоритетам (priority grouping) для выделения групп прерываний, имеющих преимущественный приоритет, и групп прерываний, не имеющих преимущественного приоритета;
5) Автоматическое сохранение контекста при вызове процедуры обработки прерывания и восстановление при возврате из нее;
6) Поддержку последовательной обработки нескольких запросов прерываний (цепочки запросов) без дополнительных затрат времени на сохранение и восстановление контекста.
7) Опциональный контроллер прерываний «с будильником» Wake-up Interrupt Controller (WIC), который обеспечивает ультранизкий уровень потребления питания в режиме «сна» микроконтроллера (sleep mode) и «пробуждение» по сигналу запроса прерывания.



Принципиальным отличием процессоров ARM от многих других процессоров является интеграция на кристалл процессора контроллера прерываний, который обрабатывает не только *запросы прерываний* от внешних источников, встроенных и внешних периферийных устройств, но и *исключения*, которые могут возникнуть при некорректном выполнении программного кода, таком как попытка выполнения несуществующей операции или делении на ноль. Эти возможности позволяют существенно *повысить надежность программного обеспечения*, особенно в цифровых системах управления ответственным оборудованием.

Для поддержки обработки прерываний и исключений, в памяти процессора должна располагаться *таблица векторов перехода на процедуры обработки прерываний и исключений*. Организация этой таблицы в процессорах Cortex-M с указанием источников прерываний/исключений рассматривается в следующей главе. Место расположения этой таблицы в памяти *строго определено архитектурой процессора*.

Прежде, чем написать даже простенькую прикладную программу, нужно, как минимум, инициализировать процессор и, по крайней мере, начальную часть таблицы векторов прерываний/исключений. Об этом – следующая глава.

5.4 Режимы работы процессора. Привилегии доступа к регистрам специального назначения

Процессор поддерживает два возможных режима работы: 1) *Пользовательский – User mode*, называемый также режимом выполнения потока команд, потоковым режимом – *Thread mode* и 2) *Обработчика исключений – Handler mode.* В первом режиме обычно выполняется любая программа приложения. Во второй режим работы процессор

переходит только при обнаружении исключения и автоматической передачи управления соответствующему обработчику исключения.

Эти режимы отличаются тем, какие *привилегии* предоставляются программному коду в отношении доступа к специальным регистрам процессора (управляющим работой собственно процессорного ядра и встроенной в процессор периферией).

Различают:

- *Привилегированный уровень доступа* (*Privileged* access level) – все ограничения на доступ к системным регистрам процессора снимаются и *за все последствия отвечает программист*, который в этом режиме должен действовать исключительно внимательно и ответственно, как системный администратор компьютера. В этом режиме доступа выполняется инициализация или переинициализация процессора и встроенных периферийных устройств.
- *Непривилегированный уровень доступа* (*Unprivileged* access level) – такой, когда код программы не может получить доступ к некоторым специальным регистрам процессора, которые отвечают за функционирование ядра процессорной системы. Такой уровень доступа исключает возможные плачевные последствия при неправильном программировании со стороны неквалифицированного программиста.

Непривилегированный код содержат обычно программы конечных пользователей, которые решают свои конкретные прикладные задачи (ПО нижнего уровня). Впрочем, любой, написанный прикладным пользователем, «обработчик исключения», будет выполняться уже в привилегированном режиме. Программное обеспечение верхнего уровня, если оно присутствует в конкретном приложении (например, монитор или операционная система реального времени), обычно выполняется в привилегированном режиме, имея доступ ко всем ресурсам процессора.

Напомним, что *исключения* – это *нестандартные ситуации*, на которые реагирует процессорное ядро, вызывая соответствующие обработчики (подпрограммы). Типичный пример – случайное считывание кода команды из области памяти, которая предназначена для регистров периферийных устройств, где программный код находиться просто не может.

Как происходит *управление режимами работы процессора*?

- Процессор переходит в пользовательский режим работы (User Mode) сразу после сброса (Reset) или в результате возврата из процедуры обработки исключения. В этом режиме может выполняться как привилегированный (Privileged), так и непривилегированный (Unprivileged) код. Сразу после сброса процессора *«по умолчанию» устанавливается привилегированный режим доступа*, чтобы пользователь имел возможность проинициализировать процессор в соответствии со спецификой своего приложения.
- Процессор переходит в режим обработчика исключений (Handler mode) только в результате возникновения исключения и передачи управления соответствующему обработчику. Весь код в этом режиме работы является только привилегированным (Privileged). В процедуре обслуживания исключения, в самом ее конце, последней командой процессор переводится в режим нормальной работы – в пользовательский режим User Mode.

Графическая иллюстрация возможных режимов работы процессора и допустимых переходов из одного режима в другой показана на рис. 5.2.

Режим обработки исключений Handler Mode всегда является привилегированным – в нем возможен доступ ко всем ресурсам процессора, так как процедура обслуживания исключения «должна разобраться», что же произошло? И «найти» корректный выход из

сложившейся ситуации. Он может быть либо очень простым – остановить ход выполнения программы, либо более сложным. Так, при исключении «деление на ноль», если это возможно с точки зрения конкретного приложения, можно продолжить выполнение программы, выдав в качестве результата операции максимально возможное значение с соответствующим знаком.

Для управления режимами работы процессора используется регистр управления – Control Register. Далее на примерах мы покажем, как выполняется инициализация процессора, как разрешить/запретить доступ к специальным регистрам и как обеспечить возврат из процедуры обслуживания исключения в основную программу.

Рис. 5.2 Режимы работы процессоров Cortex-M

Процедура инициализации процессора очень важна. Только после ее выполнения можно передать управление пользовательской программе. При инициализации процессора выполняется доступ к специальным регистрам – это можно сделать только в привилегированном режиме работы, который и устанавливается «по умолчанию» при сбросе или включении питания.

Переход в непривилегированный режим работы выполняется исключительно программным путем.

Переход в режим обработчика исключений Handler Mode выполнятся автоматически, в момент детектирования процессором нестандартной ситуации и обращения к программе – обработчику этого исключения.

5.5 Операционные состояния процессора

Процессор может находиться в двух *операционных состояниях*: либо выполнять инструкции, либо находиться в состоянии останова (в точке останова при отладке программы):

- Thumb state – *состояние выполнения* команды набора Thumb. Это нормальный режим работы, когда выполняются 16- или 32-разрядные команды единого набора команд Thumb, выровненные по границе слова или полуслова. Это общее состояние, независимо от типа конкретной команды (32-разрядной ARM) или 16-разрядной команды (Thumb-2). Напомним, что используемый в рассматриваемых процессорах набор команд Thumb объединяет оба набора команд в один общий.

- Debug State – *состоянии отладки*, когда процессор находится в состоянии останова для выполнения любых отладочных операций. Управление всеми внутренними ресурсами процессора при этом передается отладчику.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Козаченко В.Ф. Микроконтроллеры: руководство по применению 16-разрядных микроконтроллеров Intel MCS-196/296 во встроенных системах управления. – М.: ЭКОМ, 1997. – 688 с.
3) ARM DDI 0403C_errata_v3, ARMv7-M Architecture Reference Manual, Arm Limited, 2010.
4) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
5) ARM DDI 0337H, Cortex-M3. Technical Reference Manual. Arm Limited, 2010.
6) ARM DUI 0068B. ARM Developer Suite. Assembler Guide. Arm Limited, 2001.
7) ARM DUI 0553A. Cortex-M4 Devices. Generic User Guide. Arm Limited, 2010.
8) ARM DDI 0439D. ARM Cortex-M4 Processor. Technical Reference Manual. Arm Limited, 2013.
9) ARM DDI 0479B. Cortex-M System Design Kit. Technical Reference Manual. Arm Limited. – 2011.

6 УНИФИЦИРОВАННАЯ КАРТА ПАМЯТИ ПРОЦЕССОРНЫХ ЯДЕР ARM

Оглавление

6.1 Общие сведения

Карта памяти процессора определяет возможную структуру памяти микроконтроллера, изготовленного на его основе, т.е. возможное распределение памяти на области: встроенной кодовой памяти (памяти программ); встроенной памяти для чтения и записи (оперативной памяти); дополнительной внешней памяти; памяти периферийных устройств; памяти специального назначения.

Под *внутренней* памятью понимается *встроенная* в кристалл микроконтроллера память, а под *внешней* – дополнительно подключаемая к нему в случае необходимости (обычно для расширения объема внутренней памяти в больших и сложных проектах).

Процессоры Cortex-M3/M4/M4F имеют *унифицированную карту памяти*, задающую максимально возможные диапазоны адресов всех возможных областей памяти,

типы (назначение) и особенности использования этих областей памяти (атрибуты). Производители конкретных микроконтроллеров *должны следовать этим общим правилам*, размещая реальную память только в предназначенных для этого областях. Только в этом случае все внутренние интерфейсы процессора будут работать в оптимальных режимах.

Фактическая карта памяти конкретного микроконтроллера используется при разработке программного обеспечения. Она задается в виде допустимых диапазонов адресов соответствующих областей памяти и позволяет программе *компоновщику* (*линкеру*) размещать код программы и данные только в реально существующих областях памяти программ и данных, контролируя возможные переполнения этих областей.

Фактическая карта памяти микроконтроллера будет содержать также список реальных адресов регистров встроенных в микроконтроллер периферийных устройств. Эта информация обязательно содержится в техническом описании каждого выпускаемого микроконтроллера (в его Data Sheet).

6.2 Начальные сведения о проекте

Любой *программный проект* предназначен для решения конкретной прикладной задачи и состоит из определенного числа *программных модулей*, написанных одним или несколькими программистами. Каждый программный модуль представляет собой программу на языке ассемблера или на языке высокого уровня, которая выполняет определенные функции, обращаясь к переменным, расположенным в оперативной памяти микропроцессорной системы. Для каждого из программных модулей могут резервироваться свои собственные переменные в ОЗУ. Часть переменных может быть общей для всех или нескольких программных модулей. Все переменные проекта делятся на два типа:

- *Инициализируемые*, начальные значения которых перед запуском программы должны быть определены (установлены);
- *Неинициализируемые*, начальные значения которых устанавливать не требуется. Об этом позаботится программа пользователя.

В языке Ассемблер или в любом языке высокого уровня должны быть и имеются средства не только для написания собственно программы модуля, но и для резервирования в памяти переменных – инициализируемых и неинициализируемых.

После трансляции исходных файлов, написанных на ассемблере или компиляции исходных файлов, написанных на языке высокого уровня, которая завершается созданием машинного кода каждого модуля, все коды объединяются в *один программный код проекта* компоновщиком (линкером) и размещаются в последовательной области кодовой памяти микроконтроллера. Одновременно *объединяются и области памяти инициализированных и неинициализированных переменных* в ОЗУ, как показано на рис. 6.1. Основная задача компоновщика – выполнять это объединение и следить за тем, чтобы имеющихся в МК ресурсов памяти было достаточно, в противном случае – информировать разработчика о недостатке объема кодовой памяти или памяти данных.

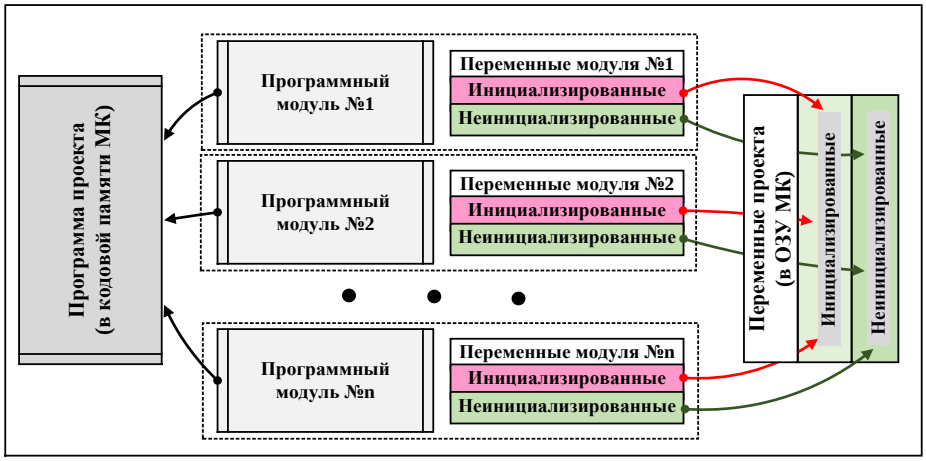


Рис. 6.1 Понятие проекта

Любой проект состоит из определенного числа программных модулей, написанных на языке ассемблера или на языке высокого уровня. После трансляции или компиляции каждого из модулей кодовые части модулей объединяются в один код проекта для размещения в памяти программ, а области инициализированных и неинициализированных переменных – в общие области для размещения в ОЗУ.

Компилятор сначала преобразует программу, написанную на языке высокого уровня, в программу на языке Ассемблер конкретного процессора, а затем запускает процесс трансляции с языка Ассемблер в машинный код этого процессора.

6.3 Порядок расположения данных в памяти

Процессоры ARM имеют 32-разрядную адресную шину, по которой могут адресовать в памяти данные (объекты) разной длины:

- 8-разрядные – байты;
- 16-разрядные – полуслова;
- 32-разрядные – слова.

С объектами большей длины, например, 64-разрядными двойными словами, процессор работает последовательно, извлекая из памяти младшее, а затем старшее слово двойного слова.

Процессоры ARM рассматривают память как линейную область 8-разрядных чисел (байт), каждый из которых имеет свой персональный адрес, начиная с нулевого адреса до 0xFFFF FFFF. Два последовательно расположенных байта образуют 16-разрядное полуслово (Halfword), а два последовательно расположенных полуслова – полное 32-разрядное слово (Word). Адреса полуслов, как правило, выравниваются по четным адресам (кратным 2), а адреса слов – по дважды четным адресам (кратным 4).

Каждое слово состоит из четырех байт, при этом используется так называемый *прямой порядок расположения байт и полуслов в слове*: по младшему адресу в памяти располагается младший байт и, соответственно, младшее полуслово.

Ниже показан порядок расположения байт в слове 0x55AAFE02, расположенном по адресу 0x2000 0000. Для обозначения старшего полуслова используется мнемоника MSHW (старшее значащее полуслово), а для обозначения младшего – LSHW (младшее значащее полуслово). Аналогичные сокращения используются для обозначения старшего значащего байта MSB и младшего значащего байта LSB (см. табл. 6.1).

Таблица 6.1 Порядок расположения байт в слове

Адреса ячеек памяти	Байты Hex	Байты Bin	Полуслова Halfword		Слова Word	
0xFFFF FFFF						
…						
0x2000 0003	55	0101 0101	MSHW=0x55AA	MSB	W=0x55AAFE02	MSB
0x2000 0002	AA	1010 1010		LSB		
0x2000 0001	FE	1111 1110	LSHW=0xFE02	MSB		
0x2000 0000	02	0000 0010		LSB		LSB
…						
0x0000 0000						

Прямой порядок расположения бит используется также в байте, полуслове и слове, т.е. вначале располагается самый младший бит, затем – более старшие биты.

Каждый байт содержит 8 бит двоичной информации. Биты нумеруются, начиная с младшего бита: 0, 1, 2, … Таким образом, самый старший бит байта имеет номер 7, полуслова 15, слова 31:

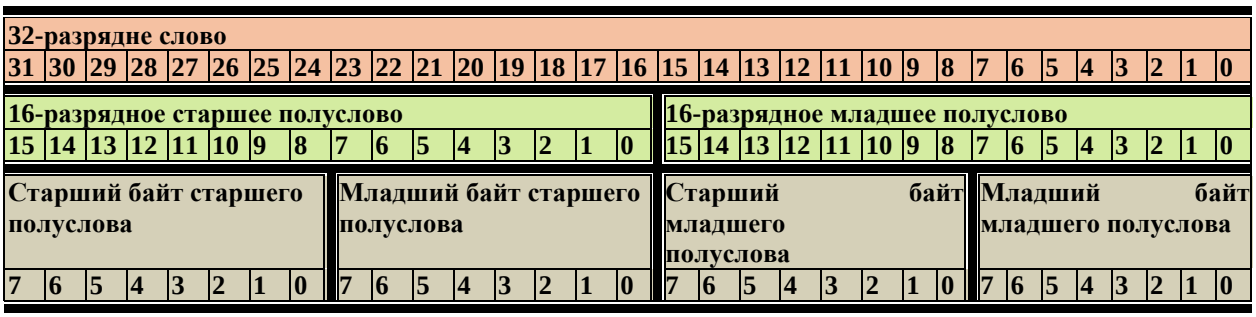
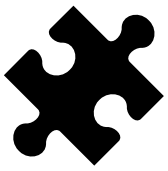


Рис. 6.2 Порядок расположения бит в слове



В процессорах ARM свой адрес имеет каждый байт, полуслово и полное слово. Исключение составляют некоторые специальные области памяти, в которых содержатся так называемые *битовые ленты* – области памяти, которые адресуются как побайтово или пословно, так и побитово. Работа с «битовыми лентами» рассматривается отдельно (глава 13).

Любые массивы и структуры данных адресуются по начальному адресу их расположения в памяти.

6.4 Объем прямо-адресуемой памяти в процессорах Cortex

Он измеряется в единицах информации, которые могут быть записаны или считаны из памяти. Такой минимальной единицей в данном случае является байт. Имея 32-разрядную шину адреса процессоры ARM могут адресовать 232= 4 Гбайт данных, 2 Г полуслов или 1 Г слов. Напомним, что в процессорной технике принято обозначать масштабирующие коэффициенты так:

- К – Кило = 1024;
- М – Мега = 1024 х 1024 = 1 048 576;
- Г – Гига = 1024 х 1024 х 1024 = 1 073 741 824.

Имеющиеся объемы памяти перекрывают потребности большинства разработчиков встраиваемых в оборудование цифровых систем управления. Однако, Вы должны иметь в

виду, что распределение этого объема памяти по назначению жестко регламентируется фирмой ARM. Так, разработчик конкретного микроконтроллера не может отдать весь объем прямо-адресуемой памяти под кодовую память. Часть общего объема займет оперативная память, внешняя память и периферийные устройства. Но даже такого объема, который фирма ARM резервирует для кодовой памяти (0.5 Г байта), больше чем достаточно для большинства приложений.

6.5 Адресация регистров периферийных устройств

В процессорах ARM применяется механизм адресации регистров периферийных устройств (ПУ), называемый отображением адресов регистров ПУ на память. Часть адресного пространства выделяется для периферийных устройств и называется областью памяти периферии. При этом одна область памяти предназначена для размещения встроенных на кристалл микроконтроллера периферийных устройств, а вторая – внешних, подключаемых при необходимости уже разработчиком конкретной микропроцессорной системы. Преимущества:

1) Для доступа к регистрам периферийных устройств можно использовать весь арсенал команд процессора, предназначенный для работы с памятью. Никаких специальных команд ввода/вывода нет.
2) Это касается и непосредственной работы с битами регистров периферийных устройств (команд так называемого «битового сопроцессора», – можно устанавливать и сбрасывать биты при инициализации или изменении режимов работы ПУ, считывать и обрабатывать битовую информацию из портов ввода, выводить информацию в порты вывода обычными командами работы с памятью (глава 13). Это, правда, возможно только в том случае, если периферийные устройства, интегрированные на кристалл разработчиком микроконтроллера, расположены в зоне адресов, допускающих побитовую адресацию в соответствии с унифицированной картой памяти ARM.
3) Системные периферийные устройства (например, контроллер прерываний, таймер), относящиеся к периферии, встроенной в ядро ARM-процессора, располагаются в своей персональной области памяти. Их адреса не меняются при переходе разработчика на микроконтроллер с большей производительностью или другим объемом встроенной памяти, другим набором специализированной периферии. Это обеспечивает как аппаратную, так и программную совместимость при модернизации и разработке новых устройств. Без каких-либо изменений можно использовать программное обеспечение верхнего уровня – операционные системы реального времени и мониторы, использующие в своей работе исключительно системные периферийные устройства.

6.6 Унифицированная карта памяти

Ниже представлена унифицированная карта памяти процессоров Cortex-M3/M4/M4F. Разработчикам и производителям микроконтроллеров предписывается размещать свою кодовую и оперативную память в строго определенных адресных диапазонах, а также использовать в качестве адресов регистров периферийных устройств специально выделенный для этой цели диапазон адресов памяти (см. табл. 6.2).

Таблица 6.2 Унифицированная карта памяти процессоров Cortex-M

Диапазон адресов	Область памяти	Объем	Тип	Атри-бут	Назначение
0xFFFF FFFF 0xE010 0000	**Vendor specific memory** ***Память, определяемая производите-лем МК***	**511 Мбайт**	**Device**	**XN**	**Память, назначение которой определяется исключительно производителем микроконтроллера (МК)**
0xE00F FFFF 0xE000 0000	**Private Peripheral Bus** ***Собственная периферийная шина процессора***	**1 Мбайт**	**Strongly-ordered**	**XN**	**Регистры периферийных устройств центрального процессора:** **контроллера прерываний NVIC;** **системного таймера – System timer;** **Блока управления системой Block system control**
0xDFFF FFFF 0xA000 0000	**External device** ***Внешние устройства***	**1 Гбайт**	**Device**	**XN**	**Регистры внешних периферийных устройств**
0x9FFF FFFF 0x6000 0000	**External RAM** ***Внешнее ОЗУ***	**1 Гбайт**	**Normal**	**-**	**Данные исполняемой программы**
0x5FFF FFFF 0x4000 0000	**Peripheral** ***Встроенная периферия***	**0.5 Гбайта**	**Device**	**XN**	***Регистры встроенных в микроконтроллер периферийных устройств.*** **Содержит также область побитово-, побайтово-адресуемых регистров**
0x3FFF FFFF 0x2000 0000	**SRAM** ***Встроенное статическое ОЗУ***	**0.5 Гбайта**	**Normal**	**-**	***Данные исполняемой программы.*** **Может содержать код программы.** **Содержит также побитово-, побайтово-адресуемую память.**
0x1FFF FFFF 0x0000 0000	**Code** ***Память программ (кодовая)***	**0.5 Гбайта**	**Normal**	**-**	***Исполняемый код программы.*** **Могут храниться данные (константы и таблицы констант)**

Понятие типа и атрибута области памяти поясняется ниже. Цветным фоном выделены области памяти, наиболее часто используемые прикладными программистами. Обратите внимание на то, что области встроенного статического ОЗУ (SRAM), а также встроенных регистров периферийных устройств (Peripheral) содержат специальные разделы памяти для поддержки операций с битами (битовые ленты). Если разработчики конкретного микроконтроллера разместили свои периферийные устройства в другой области, например, в области внешних периферийных устройств, то побитовый доступ к ним станет невозможным.

1) Проверьте, действительно ли общий объем памяти в соответствии с унифицированной картой памяти равен 4 Г байта?
2) Вспомните, какие из периферийных устройств являются встроенными в центральный процессор?
3) С какой целью все периферийные устройства разделены на две

группы: принадлежащие собственно процессору; принадлежащие микроконтроллеру?

4) В какую область памяти попадают модули поддержки отладки?

1) Да, это так.
2) Прежде всего контроллер прерываний и системный таймер, а также модули, поддерживающие интерактивную отладку.
3) С целью обеспечения предельного быстродействия встроенных в процессор периферийных устройств, которые используются всеми разработчиками.

4) В область устройств на собственной периферийной шине процессора.

6.7 Типы и атрибуты областей памяти

Каждая область памяти предназначена для размещения в ней данных определенного *типа*. Она может иметь дополнительные *атрибуты*, связанные со спецификой доступа к данным в этой области, что и отмечено в соответствующих позициях карты памяти.

Информация о типах и атрибутах областей памяти (табл. 6.3, 6.4) предназначена, главным образом, для разработчиков микроконтроллеров. Конечные пользователи могут только проверить, что используемые ими микроконтроллеры соответствуют требованиям фирмы ARM.

Таблица 6.3 Типы области памяти

Тип области памяти	Особенности
Normal – Нормальная	**Процессор может переупорядочить операции доступа к данным в этой области или выполнить упреждающее чтение.**
Device – Устройство	**Процессор сохраняет порядок операций по пересылке данных (транзакций) при обращении к устройству. Буферирование данных при записи возможно.**
Strongly-ordered Строго по очереди	**Все транзакции выполняются строго в порядке очереди. Буферирование при записи невозможно.**

В процессоре имеется специальная *система автоматического упорядочивания доступа к памяти*, которая оптимизирует этот процесс. Поэтому, фактический порядок доступа может отличаться от предусмотренного программой пользователя, если это не влияет на конечный результат. Если результат выполнения программы жестко зависит от последовательности доступа к памяти (например, при инициализации периферийных устройств или самого процессора), то программист должен разделить инструкции обращения к памяти *специальной командой разделения доступа* memory barrier.

Тип памяти «Строго по очереди» имеют только системные периферийные устройства. Если при последовательном обращении к двум областям памяти:

- Одна из областей имеет тип «Нормальная», то программный порядок обращения не гарантируется. Процессор может выполнять автоматическую оптимизацию доступа (изменить порядок, определенный программой) с целью повышения производительности всей системы.
- Обе области памяти имеют тип «Строго по очереди», то команды выполняются в порядке их расположения в программе. При работе со встроенной периферией процессора это дает уверенность, что все действия будут выполнены строго в соответствии с желанием программиста.

Дополнительные атрибуты областей памяти описаны в табл. 6.4.

Таблица 6.4 Дополнительные атрибутам областей памяти

Атрибут	**Что означает**
Shareable – Совместное использование	**Обеспечивается необходимая синхронизация при доступе к этой области как со стороны системы управления памятью процессора, так и со стороны контроллера прямого доступа к памяти DMA.**
Execute Never (XN) – Никогда не выполнять	**Процессор предотвращает выполнение кода из этой области памяти. При такой попытке генерируется исключение сбоя (fault).**

Атрибут «Совместного использования» имеют некоторые подобласти памяти периферийных устройств (не указаны в приведенной выше карте памяти). Он имеет значение только для мульти-микропроцессорных систем, которые в этой книге не рассматриваются.

Атрибут «*Никогда не выполнять*» имеет большое значение. Если в результате неправильного программирования или программного сбоя последует считывание программного кода, например, из области памяти периферийных устройств, то будет сгенерировано *исключение fault* (ошибка). Такая мера защиты существенно повышает надежность систем встроенного управления оборудованием.

Обратите внимание на то, что все области памяти периферийных устройств, встроенных в процессор, встроенных в микроконтроллер и даже внешних ПУ, автоматически защищены от попытки считывания из них программного кода. В этом случае будет генерироваться исключение по ошибке fault.

6.8 Особенности доступа к различным областям памяти

Каждая область памяти на карте памяти процессора имеет определенные особенности доступа к расположенной там информации:

1) *Код программы пользователя может размещаться в трех областях*: Code, SRAM, и external RAM. Однако, фирма ARM рекомендует размещать код программы всегда в области кода Code. Теоретически код программы может располагаться и во внутреннем ОЗУ – SRAM и во внешнем – external RAM, что удобно при отладке. Однако, выполнение программы из ОЗУ замедляется. Дело в том, что для доступа к программной памяти в области Code используются две отдельные шины – для считывания кодов команд на конвейер команд и для считывания данных. Гарвардская архитектура центрального процессора обеспечивает одновременный и независимый друг от друга доступ к кодам команд (из секции Code) и к данным как из других секций, так и из секции Code, что существенно повышает производительность процессора.
2) В состав центрального процессора может включаться дополнительный модуль защиты памяти Memory Protection Unit (MPU), который предназначен для изменения типов и атрибутов областей памяти, указанных на карте памяти и являющихся параметрами областей памяти по умолчанию.
3) С помощью этого блока могут устанавливаться также дополнительные ограничения на доступ к памяти для совместного использования (shared memory) в мультипроцессорных системах. При этом сама область памяти может разделяться на подобласти со своими индивидуальными типами и атрибутами доступа.

6.9 Размещение в памяти таблицы векторов прерываний и исключений

В процессорах Cortex-M контроллер прерываний вместе с детектором исключительных ситуаций интегрированы на кристалл процессора. Обработка всех прерываний и исключений выполняется соответствующими подпрограммами, обработчиками прерываний/исключений, написание которых – задача программиста. Начальные адреса обработчиков *должны быть записаны* программистом в *таблицу векторов прерываний и исключений*, с которой далее в автоматическом режиме будет работать контроллер прерываний. При возникновении прерывания или исключения (если оно разрешено) контроллер прерываний считывает из таблицы начальный адрес соответствующего обработчика и загружает его в счетчик команд PC, передавая управление обработчику прерывания/исключения.

Сразу после включения питания или сброса процессора предполагается, что таблица векторов прерываний и исключений *находится в самом начале памяти программ, начиная с нулевого адреса*. При необходимости ее положение в памяти системы можно изменить. Для этой цели предназначен один из регистров встроенного контроллера приоритетных прерываний NVIC – VTOR. Его значение после сброса процессора – ноль.

В табл. 6.5 приведена структура таблицы векторов прерываний и исключений процессоров Cortex-M. Цветом выделены различные функциональные группы запросов прерываний.

Таблица 6.5 Таблица векторов прерываний и исключений (сразу после сброса ЦПУ)

Номер искл.-я/ прер.-я	Адресное смещение	Обозначение запроса	Содержит начальный адрес обработчика запроса прерывания /исключения
18…255	0x48…0x3FF	IRQ №2…239	От периферийного устройства, встроенного в микроконтроллер
17	0x44	IRQ №1	
16	0x40	IRQ №0	
15	0x3C	SYSTICK	От системного таймера
14	0x38	PendSV	Адрес супервизора или ОС РВ
13	0x34	Зарезервировано	
12	0x30	Debug Monitor	Адрес отладочного монитора
11	0x2C	SVCall	Адрес супервизора или ОС РВ
7…10	0x1C…0x28	Зарезервировано	
6	0x18	Usage Fault	Обработчика ошибки использования
5	0x14	Bus Fault	Обработчика ошибки шины
4	0x10	Mem Manage Fault	Обработчика ошибки управления пам.-ю
3	0x0C	Hard Fault	Обработчика аппаратной ошибки
2	0x08	NMI	Обработчика немаскируемого прерывания
1	0x04	Reset	Обработчика сброса процессора
0	0x00	MSP	Начальное значение указателя стека MSP

6.9.1 Код инициализации указателя стека

Первый вектор в таблице – особый. Это не начальный адрес какого-либо обработчика, а *значение указателя стека* микропроцессорной системы SP. По сути – это код инициализации *вершины стека в системе* – адреса последней «как бы занятой данными» ячейки памяти, отведенной пользователем под стек. Мы подробно рассмотрим назначение стека и особенности работы с ним в следующей главе. Пока же отметим, что стек – это часть памяти данных, специально отведенная программистом для временного

хранения информации, в том числе адресов возврата при вызове подпрограмм и обработчиков прерываний/исключений. При записи данных в стек содержимое указателя стека SP автоматически декрементируется (на 4), так как в процессорах Cortex-M используется так называемый «автодекрементный» стек (см. 7.7.3). После этого нужное слово (4 байта) сохраняется в памяти.

Итак, по начальному адресу в кодовой памяти программист *должен записать код инициализации указателя стека.* Сразу после сброса процессор *автоматически считает этот код и загрузит его в указатель стека SP*.

Зачем это делается? Если уже в самом начале работы программы пользователя возникнет какая-то непредвиденная ситуация, например, аппаратная ошибка, связанная с аппаратной ошибкой при проектировании системы, то процессор должен вызвать соответствующий обработчик ошибок. Но, прежде нужно записать адрес возврата в стек. А если он не определен? Именно для повышения надежности программного обеспечения инициализация стека выполняется автоматически в самом начале работы процессора, причем еще до инициализации самого процессора и памяти данных (если это требуется).

Стек располагается в ОЗУ. В соответствии с унифицированной картой памяти процессоров ARM Cortex-M область ОЗУ: 0x20000000-0x3FFFFFFF (512 М байт). Зная реальный объем ОЗУ Вашей системы, Вы можете в конце этой области памяти разместить стек. Так, если объем ОЗУ 256 Мбайт, то указатель стека можно проинициализировать числом 0x30000000 (первая запись будет в ячейку памяти 0x2FFFFFFC).

6.9.2 Начальный адрес процедуры инициализации процессора

Второй вектор тоже особый. Это *начальный адрес процедуры инициализации центрального процессора* и, в общем случае, содержимого оперативной памяти системы (установки значений инициализируемых переменных). Фактически это адрес одной из важнейших программ любого проекта – *стартовой программы Startup*. Ее задача – выполнить все необходимые начальные установки в системе и затем передать управление программе пользователя (более подробно – в разделе 9.1). Каждый раз, когда Вы будете включать питание или нажимать на кнопку «Сброс», будет выполняться переинициализация системы.

Стартовая программа Startup должна обязательно присутствовать в любом проекте. Именно она будет содержать код начальной инициализации процессора. Более того, в ней будут инициализироваться нужные пользователю периферийные устройства (как встроенные в процессор, так и в микроконтроллер), даваться разрешение на работу сопроцессора (если это необходимо), а также инициализироваться области памяти данных, предназначенные для переменных.

Обратите особое внимание на то, что при включении питания состояние ячеек памяти данных не определено – там может быть произвольная информация.

Стартовая программа обычно пишется на языке Ассемблер, даже если разработка программного обеспечения выполняется на языке высокого уровня С/С++. В ее конце выполняется безусловная передача управления на начало программы пользователя, написанной на ассемблере, или на начало функции main на СИ. При использовании СИ функция main может содержать процедуры начальной инициализации нужных конкретному пользователю периферийных устройств.

6.9.3 Начальный адрес обработчика немаскируемого прерывания

Третий вектор – это начальный адрес процедуры обслуживания так называемого *немаскируемого прерывания* Non-Maskable Interrupt (NMI). Все прерывания (внешние и от встроенных в микроконтроллер периферийных устройств) делятся на два класса:

маскируемые и *немаскируемые*. Первые могут быть запрещены – замаскированы, путем записи соответствующей информации в регистры контроллера прерываний, а вторые нет, – их обработку нельзя «отключить». К числу немаскируемых прерываний относятся прерывания *от аварийных датчиков*, реакция на срабатывание которых обеспечивает *«живучесть» объекта управления*.

Это могут быть датчики аварийного состояния и самой микропроцессорной системы управления, например, отказа памяти или какого-либо важного периферийного устройства. Если датчик такой аварии подает сигнал на вход NMI контроллера прерываний, то в процедуре обслуживания прерывания выполняется немедленное отключение оборудования во избежание дальнейшего развития аварии.

На вход немаскируемого запроса прерывания микроконтроллера обычно подается сигнал с датчиков, сигнализирующих о невозможности дальнейшей работы оборудования в штатном режиме. В процедуре обслуживания запроса делается все возможное для предотвращения аварии.

Места в таблице векторов прерываний с 3 по 14 (12 позиций) отведены под начальные адреса процедур обработки исключений. Обработка исключений – это специальная тема, которая должна изучаться только после изучения системы команд процессора и общих приемов программирования. Она сложна не только для начинающих, но и для квалифицированных программистов. Поэтому поступим так: дадим здесь лишь краткий обзор возможных источников исключений и прерываний в процессорах Cortex-M, чтобы *завершить описание таблицы* векторов. Это позволит нам понять структуру стартового программного модуля Startup, предлагаемого фирмой ARM в качестве шаблона для использования прикладными программистами. Приведенная ниже информация при первом чтении может быть бегло просмотрена.

6.9.4 Аппаратные ошибки

В начале таблицы располагаются адреса обработчиков ошибок при выполнении команд, когда процессор не может корректно завершить выполнение операции. Возникновение этих неисправностей контролируется на аппаратном уровне ARM-ядра процессора соответствующими детекторами. Приведем кратное описание причин возникновения этих ошибок:

- **Hard fault** – *Аппаратная ошибка* в схемотехнике микроконтроллера (МК) или микропроцессорной системы (МПС).
- **Memory management fault** – *Ошибка управления памятью*. Может возникнуть только тогда, когда в состав ЦПУ включен опциональный блок защиты памяти MPU (Memory Protection Unit), который в состоянии идентифицировать некорректный доступ к памяти.
- **Bus fault** – *Ошибка шины*. Может проявиться на стадии разработки МК или МПС: при невозможности считывания кода команды на конвейер команд; при невозможности доступа к данным по записи/чтению; при некорректном доступе к периферии (например, при обращении к байтовому регистру периферийного устройства командой чтения слова).

Первые три ошибки сигнализируют о некорректности или неисправности в аппаратной части МК или МПС. Их обработчики используются, прежде всего, инженерами-аппаратчиками на стадии тестирования новых микроконтроллеров на базе ядер ARM Cortex-M и микропроцессорных систем на их основе. Если Вы покупаете микроконтроллер известного производителя, то вероятность возникновения таких ошибок

минимальна. Они могут быть следствием некорректного подключения к микроконтроллеру внешних дополнительных периферийных устройств или некорректного расширения памяти.

6.9.5 Программные ошибки

Usage fault – *Ошибка использования*. Это *программная ошибка*, связанная с некорректными действиями программиста. Именно она позволяет обнаружить такие ошибки, как деление на ноль, извлечение квадратного корня из отрицательного числа и т.д.

Ошибка использования не имеет отношения к аппаратной части. Причина ее возникновения – неправильный алгоритм или неправильная его реализация программистом. Поэтому очень важно, чтобы обработчик этой ошибки уже существовал на стадии отладки программного обеспечения. Его важнейшая задача – сигнализировать программисту о некорректности работы программы. В простейшем случае это останов программы и светодиодная индикация причины возникновения ошибки на плате МПС. В более сложном случае – попытка устранения ошибки и продолжение выполнения программы.

Так, деление на ноль чисел с фиксированной точкой – это фатальная ошибка. Можно сообщить о ней программисту и «заставить» его модифицировать программу. При делении на ноль чисел с плавающей точкой – ситуация иная. Обработчик ошибки может не прерывать выполнение программы, а вернуть в качестве результата операции положительную или отрицательную бесконечность. Вычислительный процесс или процесс управления может быть продолжен.

Одна из наиболее частых причин возникновения ошибки использования – считывание из кодовой памяти *неопределенной инструкции* (undefined instruction). Это возможно, если код, поступивший на конвейер команд, не может быть декодирован.

Ошибка использования – это ошибка программиста. Самое простое – обработчик ошибки должен сигнализировать о ее появлении программисту. Он должен присутствовать в составе пользовательского ПО уже на стадии отладки.

6.9.6 Программный запрос вызова супервизора

Одна из позиций в таблице векторов прерываний/исключений может содержать начальный адрес специальной программы, называемой *супервизором* или *монитором*. Это может быть и *операционная система реального времени* (ОС РВ), которая куплена разработчиком конкретных приложений, в том числе, с целью использования *типовых, оптимизированных функций ввода/вывода данных по стандартным интерфейсам*.

Чтобы из пользовательской программы (нижнего уровня) обратиться к супервизору (программе верхнего уровня) в системе команд Cortex-M предусмотрена специальная команда так называемого *программного вызова супервизора* SVC #imm (SerVice Call). Это не аппаратное прерывание, а *программное*, сгенерированное специальной командой в пользовательской программе. При ее выполнении автоматически считывается адрес в позиции 11 таблицы векторов прерываний, загружается в счетчик команд и управление передается программе-супервизору.

В команде SCV задается также 8-разрядный непосредственный операнд #imm, который определяет *номер услуги* (0÷255), которую хочет получить программист нижнего уровня от программы-супервизора. Он идентифицируется супервизором, после чего вызывается нужная функция. Список имеющихся функций определяется разработчиком супервизора.

В чем преимущества такого подхода? В том, что приложение пользователя может «не задумываться» о том, как фактически выполняется нужная функция и какие при этом используются ресурсы. Оно не обращается к конфигурационным регистрам процессора и регистрам периферийных устройств. Четкое разделение между службами разного уровня – гарантия надежной работы программного обеспечения, когда программы нижнего уровня не допускаются к системным ресурсам во избежание случайного малоквалифицированного доступа.

6.9.7 Отладочный монитор

Ядро процессоров Cortex-M3/M4/M4F имеет встроенную систему аппаратной поддержки разработки и отладки программного обеспечения, называемую *системой отладки* (debug unit). Эта система позволяет подключать внешние компьютерные системы отладки (например, по интерфейсу JTAG) и не требует дополнительной программной поддержки. Все нужные для отладки средства уже реализованы на кристалле процессора. Система отладки позволяет полностью взять на себя управление кодом программы, в том числе исполнять его пошагового, от одной точки останова до другой с контролем состояния регистров и памяти. Выход из режима отладки возможен только по сбросу процессора Reset или при формировании запроса немаскируемого прерывания NMI.

Недостаток простых систем отладки – не обеспечивается *режим отладки в реальном времени*. Например, в системах управления приводами необходимо, чтобы в процессе отладки работали все или часть периферийных устройств, в частности, многоканальный генератор периодических сигналов. Более того, необходимо, чтобы функционировала система прерываний и запросы прерываний от генератора обслуживались. Только при этом параметры управления генератором будут постоянно обновляться (период или скважность выходных ШИМ-сигналов) и программист сможет наблюдать за изменением управляющих сигналов и реакцией на них объекта управления в «реальном времени».

Отладку в реальном времени в процессорах Cortex-M могут поддерживать специальные программы, написанные самостоятельно или купленные у специализированных фирм – *отладочные мониторы* Debug Monitor. Начальный адрес размещения таких программ соответствует 12-му исключению в таблице векторов прерываний/исключений. Любые события, требующие отладки, не будут полностью останавливать процессор и периферию (состояние «Halt»), а будут вызывать переход к программе монитора по запросу прерывания trap.

Контроллер прерываний позволяет разрешить или запретить использование такого отладочного монитора. Запуск или останов монитора выполняется с помощью конфигурационных регистров отладочных модулей ядра процессора. Написание программы «Монитор» по силам только очень квалифицированному коллективу разработчиков. Мы в этой книге будем пользоваться исключительно стандартными средствами отладки, поддерживаемыми ядром процессора.

6.9.8 Сервисная служба PENDSV

Это может быть служба верхнего уровня (супервизор, монитор, ОС РВ) – та же или аналогичная службе SVCall. Отличие состоит в том, что она вызывается не командой программного прерывания SVC #imm, а установкой *специального бита ждущего прерывания* в одном из регистров контроллера прерываний.

Вторая возможность вызова супервизора связана с тем, что программное прерывание не всегда может быть активизировано. Это нельзя сделать, в частности, из программы-обработчика прерывания/исключения с более высоким приоритетом. Самыми высокоприоритетными являются запросы, расположенные в нижней части таблицы

векторов (Reset, NMI). Так что, если обслуживается прерывание NMI, то программный вызов сервиса невозможен. Такая попытка вызовет аппаратный отказ (hard fault).

О назначении сервисной службы мы уже упоминали – это возможность приложения низкого уровня (пользовательского) обратиться к процедурам программы высокого уровня, в частности, для передачи/приема данных по одному из интерфейсов.

6.9.9 Прерывание от системного таймера

Позицию 15 в таблице векторов прерываний/исключений занимает начальный адрес обработчика запроса прерывания от системного таймера SisTick. Системный таймер предназначен для точного отсчета временных интервалов – организации так называемых «Системных часов» процессора. Именно это устройство позволяет реализовывать многозадачные операционные системы и ядра реального времени (real-time kernel). В них используется специальный механизм распределения процессорного времени между отдельными задачами. Приложение пользователя также может использовать системный таймер для аппаратного отсчета нужных временных интервалов. Идея такова: таймер генерирует запрос прерывания, например, каждую 1 мс. В процедуре обслуживания запроса прерывания от таймера контролируется любое число временных интервалов, заданных пользователем в количестве миллисекунд. С каждым «тиком» системных часов заданная ненулевая уставка времени уменьшается на 1. Как только она становится равной 0, выдается сигнал об истечении соответствующей выдержки времени. Число таких одновременно работающих *программно-аппаратных таймеров* не ограничено.

6.9.10 Прерывания от встроенных в микроконтроллер периферийных устройств

Начиная с позиции 16 в таблице векторов прерываний/исключений должны находиться начальные адреса обработчиков запросов прерываний от встроенных в микроконтроллер его производителем периферийных устройств IRQ № 0…239. Всего предусмотрено 240 запросов прерываний от периферийных устройств. Конкретное назначение каждого запроса прерывания IRQ определяется фирмой – производителем микроконтроллера. Оно зависит и от состава встроенной в микроконтроллер периферии, и от ее сложности. Одно периферийное устройство может генерировать несколько разных запросов прерываний.

В начале кодовой памяти должна располагаться таблица векторов прерываний/исключений. Она обязательно должна содержать первые два вектора для авто-инициализации указателя стека и передачи управления программе Startup начальной инициализации микропроцессорной системы (и процессора и памяти). Крайне желательно, чтобы в составе прикладного ПО был и обработчик программных ошибок. Это позволит своевременно идентифицировать некачественные алгоритмы и программы.

1) Какие два обязательных программных модуля входят в любой проект?
2) Какие переменные называются неинициализированными?
3) Инициализированными?

1) Стартовый программный модуль Startup с инициализацией процессора и собственно программный модуль пользователя (пользовательская программа) Main.

2) Те переменные, для которых в памяти данных просто резервируется место (закрепляется за ними). Начальное состояние этих областей памяти после сброса процессора или включения питания не определено.

3) Такие переменные, начальные значения которых должны быть установлены до выполнения пользовательской программы. Это происходит так: компоновщик объединяет все инициализированные секции в памяти данных в одну и располагает последовательность кодов инициализации переменных в определенном месте кодовой памяти (делает копию инициализированной секции данных в программной памяти); задает начальный адрес расположения инициализированной секции данных в ОЗУ. Далее вступает в действие программа Startup начальной инициализации процессора и памяти. Она копирует массив кодов инициализации из ПЗУ в ОЗУ. Тем самым переменные в ОЗУ становятся инициализированными.

При программировании только на Ассемблере обычно резервируют место под переменные в ОЗУ, а их инициализацию выполняют в пользовательской программе по мере необходимости. Это существенно упрощает структуру стартовой программы Startup.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер.с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) ARM DDI 0403C_errata_v3, ARMv7-M Architecture Reference Manual, Arm Limited, 2010.
3) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User’s Manual. Texas Instruments Inc. – 2011.
4) ARM DDI 0479B. Cortex-M System Design Kit. Technical Reference Manual. Arm Limited. – 2011.

7 ПРОГРАММНАЯ МОДЕЛЬ ПРОЦЕССОРОВ CORTEX-M

Оглавление

Под *программной моделью* процессора понимается технология доступа к данным, которые могут рассматриваться в качестве операндов в командах процессора, входящих в его *систему команд* (*набор команд*). Речь идет прежде всего о регистрах, расположенных в сверхоперативной памяти процессора, особенностях их использования в качестве *операндов* команд, а также в качестве *регистров-указателей* для доступа к данным во внешней по отношению к процессору кодовой памяти и памяти данных. В программную модель процессора включаются также *регистры специального назначения* (управления режимами работы процессора, сопроцессора, статуса программы), к отдельным битовым полям которых или отдельным битам программисту приходится обращаться при написании и отладке любой программы.

При создании системы команд любого процессора разработчики руководствуются некоторыми утвержденными ими общими принципами, которые реализуются и в *технологии кодирования операций* – присвоении командам определенных символических имен – *мнемокодов* и в *синтаксисе* ассемблерных команд, то есть в способах возможного формирования поля мнемокода и поля операндов команды. Знание этих принципов, неразрывно связанных с архитектурой процессора, позволяет быстро и эффективно изучить систему команд процессора и начать применять ее на практике.

7.1 Структура программы на языке Ассемблер. Формат строки

Ассемблер – это язык программирования самого *низкого уровня*, в котором каждая строка программы может содержать мнемокод одной, конкретной команды процессора. Совокупность таких строк и является *программой*, написанной на Ассемблере. Технология создания мнемокодов команд разрабатывается очень тщательно, чтобы, с одной стороны, быть лаконичной (по возможности краткой), а с другой – отражать смысл и последовательность действий, выполняемых командой. Все строки в Ассемблерной программе имеют один и тот же *унифицированный формат* (см. табл. 7.1):

Таблица 7.1 Унифицированный формат команды

<Метка> (<Label>)	**Разде-литель**	**Мнемокод операции или директивы Ассемблера (псевдо-команды)**	**Разде-литель**	**Поле операндов команды или поле операндов директивы Ассемблера**	**Разде-литель**	**Поле комментария**
Обязательно с первой позиции строки	**Любое число пробелов или символ табуляции Tab**		**Пробел**		**;**	

Начиная с первого символа строки может располагаться *метка* (Label) – символическое имя, созданное программистом, обязательно начинающееся с буквы, в котором могут использоваться как буквы, так и цифры, причем, как в верхнем, так и в нижнем регистрах. В именах меток допускается использование и некоторых специальных символов, в частности, символа нижнего подчеркивания «_», который обычно применяется в качестве разделителя в составных именах для их большей «читабельности».

Метка – это символическое имя, значение которого на стадии трансляции (ассемблирования программы) соответствует *текущему адресу расположения* в памяти кода команды или данных, помеченных этой меткой. Метками помечаются начальные адреса подпрограмм или фрагментов основной программы, к которым планируются обращения в программе (безусловные или условные переходы), а также адреса расположения в памяти отдельных констант или таблиц констант. Метка является *опциональной* – ставится в случае необходимости.

Для размещения в памяти констант, выделения места в памяти под переменные, управления процессом ассемблирования и многих других целей применяются не команды процессора, а так называемые *директивы* языка Ассемблер – *псевдокоманды*. Они не транслируются в машинные коды, а являются «указаниями» для транслятора выполнить те или иные действия, например, разместить по текущему адресу заданную программистом константу. Ассемблер процессоров ARM имеет большое число директив, с назначением которых и особенностями их использования мы будем постепенно знакомить читателя.

Сразу после метки, через разделитель (любое число пробелов или символ табуляции), в ассемблерной строке размещается *мнемокод команды процессора* или *мнемокод директивы Ассемблера*.

Обратите особое внимание на то, что мнемокод команды не может начинаться с первого символа строки – *ему обязательно должен предшествовать, по крайней мере, один пробел*.

После мнемокода команды или псевдокоманды, через разделитель в виде, по крайней мере, одного пробела, следует *поле операндов команды или директивы Ассемблера*. Имена операндов отделяются друг от друга *запятой*. Число операндов зависит от конкретной команды/директивы.

Ассемблерная строка может завершаться *комментарием* (поясняющим текстовым сообщением), который следует после разделителя в виде точки с запятой «;».

При программировании на Ассемблере используются следующие соглашения «по умолчанию»:

1) Метка может отсутствовать, но мнемокоду команды или директивы должен предшествовать, по крайней мере, один пробел. Правилом хорошего тона считается размещать все мнемокоды команд и директивы после символа Tab. Это позволяет четко отделять метки от команд и хорошо структурировать программу.
2) В качестве меток не допускается использование так называемых «зарезервированных» имен – мнемокодов команд или директив Ассемблера, имен регистров.
3) Все метки в программе должны иметь разные имена.
4) В программе допускаются пустые строки, которые не содержат ни кода, ни комментария. Они являются своеобразными разделителями фрагментов программы, повышая ее «читабельность».
5) Комментарий может начинаться с первой позиции строки сразу после символа «;». Это удобно, если он содержит заголовок расположенного ниже фрагмента программы или подпрограммы (функции). В таком заголовке обычно присутствует краткое описание функции, способ передачи ей параметров и возврата результата, назначение используемых в подпрограмме регистров.
6) Комментарий должен отражать основную суть алгоритма, заложенного в программу. Он необходим для последующего сопровождения и модификации программы, если это потребуется.

Здесь и далее в книге примеры программного кода будут набраны моноширинным шрифтом:

```
ADD r1, #1          ; Увеличить содержимое регистра r1 на 1
MOV r2, #2          ; Инициализировать регистр r2 числом 2
M5 SUB r0, r2, r1   ; Вычесть из содержимого регистра r2
                    ; содержимое регистра r1, сохранить
                    ; разность в регистре r0
                    ; Повторить фрагмент
B M5                ; Перейти на метку M5
END
```

Первая команда примера **ADD** (от англ. Addition – сложение) выполняет инкрементирование текущего содержимого регистра r1, причем величина приращения #1 задается в коде самой команды – используется так называемая *непосредственная адресация* операнда (ее визитной карточной является символ «#», предшествующий самому числу). Значение непосредственного операнда автоматически размещается в формате команды транслятором с Ассемблера. Оно считывается из регистра команд ЦПУ на стадии выполнения команды и используется в качестве второго операнда АЛУ (первый извлекается из регистра общего назначения r1).

Непосредственная адресация операндов применяется в основном для инициализации начальных значений в регистрах процессора, как это показано во второй команде примера **MOV** (от англ. Move – перемещение).

Команда **SUB** (от англ. Subtraction – вычитание) вычитает из содержимого регистра r2 содержимое регистра r1 и сохраняет разность в регистре r0.

Следующая строка содержит только комментарий. Он может просто разделять разные фрагменты программы, повышая ее «структурированность».

Команда **B M5** (от англ. Branch – ответвление, переход) прерывает ход последовательного выполнения программы и передает управление на метку M5.

В последней строке программы находится не команда процессора, а директива Ассемблера **END**. Она указывает транслятору, что дальше ассемблерного текста нет и процесс трансляции в машинные коды нужно прекратить. Эта директива должна быть последней в любом программном модуле, написанном на Ассемблере и расположенном в отдельном файле.

Имеются различные версии языка Ассемблер, разработанные разными компаниями. Все они обеспечивают 100-процентную совместимость языком мнемокодов команд собственно процессора. Однако, по составу директив (псевдокоманд) и их синтаксису версии могут существенно отличаться. Так, в некоторых версиях Ассемблера после имени метки обязательно должно следовать двоеточие, а директива END может отсутствовать. Тем не менее, все Ассемблеры очень похожи друг на друга, в том числе, по составу директив и выполняемым ими функциям.

В этой книге мы пользуемся языком Ассемблера, разработанным фирмой ARM – ARM ASM. Именно он поддерживается в интегрированной среде разработки Keil µVision EDI, используемой в книге.

7.2 Программная модель процессора Cortex-M

Представим программную модель процессора в виде имеющихся в его распоряжении операционных блоков, обеспечивающих выполнение команд, и *регистрового окружения* этих блоков, содержащих регистры, которые могут использоваться в командах процессора в качестве регистров-источников Rs и регистров-приемников Rd данных, то есть в качестве операндов команд – рис. 7.1.

На рис. 7.1 изображена программная модель процессоров Cortex-M4F, имеющих на кристалле встроенный модуль поддержки вычислений с плавающей точкой – FPU (сопроцессор). Она справедлива и для младших моделей Cortex-M3/M4, но с учетом отсутствия в них сопроцессора и его «регистрового окружения».

7.3 Регистровое окружение центрального процессора

В состав центрального процессора (ЦПУ) входят три независимых устройства обработки данных (*операционный блок процессора*):

1) Арифметико-логическое устройство (АЛУ);
2) Аппаратный умножитель;
3) Аппаратный делитель.

Операционный блок обрабатывает операнды, расположенные исключительно в регистрах общего назначения процессора r0÷r15. Они являются своеобразным «*регистровым окружением*» ЦПУ. Перед выполнением любой операции в регистры-источники должны быть предварительно загружены исходные данные – операнды. Результат операции также сохраняется в одном из «регистров окружения ЦПУ» r0÷r15 (см. черные стрелки с выходов операционных блоков на рис. 7.1).

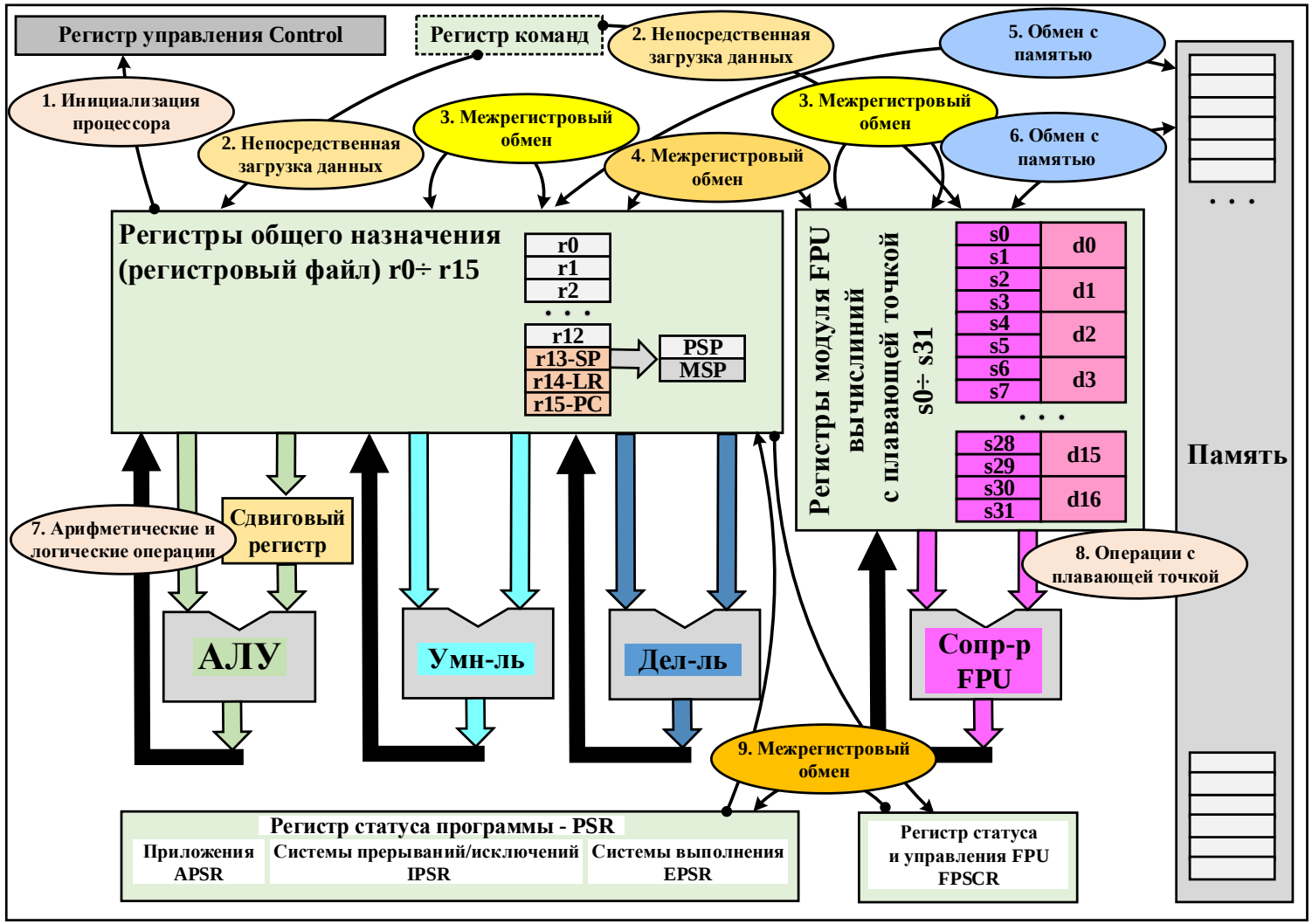


Рис. 7.1 Программная модель процессора Cortex-M

Ниже (табл. 7.2) представлен список регистров окружения ЦПУ с указанием их альтернативных обозначений и краткими рекомендациями по использованию.

Таблица 7.2 Регистры общего назначения

<table>
<tr><th>Имя регистра</th><th>Второе обозначение</th><th colspan="2">Использование</th></tr>
<tr><td>R0</td><td>-</td><td rowspan="8">Младший банк регистров общего назначения ЦПУ</td><td rowspan="8">Нет никаких ограничений в использовании этих регистров в качестве операндов-источников или операндов-приемников</td></tr>
<tr><td>R1</td><td>-</td></tr>
<tr><td>R2</td><td>-</td></tr>
<tr><td>R3</td><td>-</td></tr>
<tr><td>R4</td><td>-</td></tr>
<tr><td>R5</td><td>-</td></tr>
<tr><td>R6</td><td>-</td></tr>
<tr><td>R7</td><td>-</td></tr>
<tr><td>R8</td><td>-</td><td rowspan="5">Старший банк регистров общего назначения ЦПУ</td><td rowspan="5">Для некоторых команд имеются ограничения в использовании (см. приложение 1).</td></tr>
<tr><td>R9</td><td>-</td></tr>
<tr><td>R10</td><td>-</td></tr>
<tr><td>R11</td><td>-</td></tr>
<tr><td>R12</td><td>-</td></tr>
<tr><td>R13</td><td>SP</td><td colspan="2">Stack pointer register – регистр-указатель стека
Состоит из двух регистров:
<table>
<tr><td>PSP</td><td>SP_process – указатель стека процесса пользователя (программ нижнего уровня)</td></tr>
<tr><td>MSP</td><td>SP_main – указатель стека обработчиков исключений и программ верхнего уровня</td></tr>
</table></td></tr>
<tr><td>R14</td><td>LR</td><td colspan="2">Link register – линк-регистр, регистр связи
Содержит адрес возврата в основную программу после команды вызова подпрограммы</td></tr>
<tr><td>R15</td><td>PC</td><td colspan="2">Program counter – программный счетчик (счетчик команд) Всегда содержит адрес очередной, подлежащей выполнению команды.</td></tr>
<tr><td></td><td>PSR</td><td colspan="2">Program Status Register – Регистр статуса программы
Состоит из трех регистров:
<table>
<tr><td>APSR</td><td>Application Program Status Register – регистр статуса программы-приложения</td></tr>
<tr><td>IPSR</td><td>Interrupt Program Status Register – регистр статуса системы прерываний/исключений</td></tr>
<tr><td>EPSR</td><td>Execution Program Status Register – регистр статуса системы выполнения команд</td></tr>
</table></td></tr>
</table>

Дальше (см. 7.6) мы подробно остановимся на особенностях применения каждого из этих регистров. В программе на Ассемблере зарезервированные имена регистров можно вводить как в верхнем (R0÷R15), так и в нижнем (r0÷r15) регистрах (в общем случае это зависит от версии конкретного языка Ассемблер).

7.4 Регистровое окружение сопроцессора

Модуль поддержки вычислений с плавающей точкой FPU может выполнять операции только над операндами, расположенными в его регистрах окружения. Будем называть их *регистрами сопроцессора* или *регистрами модуля FPU*. Как показано на рис. 7.1, имеется 32 регистра s0÷s31, предназначенных для хранения данных *в формате с плавающей точкой однократной точности*. Все регистры 32-разрядные. Каждая пара регистров s0÷s31 с точки зрения программиста может рассматриваться как один 64-разрядный регистр. Он может содержать данные *в формате с плавающей точкой двойной точности* или два слова с плавающей точкой однократной точности. Всего в окружении сопроцессора 16 таких регистров с именами d0÷d15.

При написании программы имена регистров сопроцессора, как и имена регистров ЦПУ, можно набирать как в нижнем (s0÷s31 или d0÷d15), так и в верхнем регистрах (S0÷S31 или D0÷D15).

Сопроцессору и технологии вычислений с плавающей точкой посвящены специальные разделы книги (главы 19, 20, 21).

7.5 Регистр управления Control

Регистр специального назначения, который позволяет установить нужный режим работы процессора, задать тип доступа к памяти (привилегированный или непривилегированный), разрешить или запретить работу встроенных в процессор устройств, в частности, модуля FPU поддержки вычислений с плавающей точкой.

Обычно пользователей вполне устраивают значения в этом регистре «по умолчанию» (которые устанавливаются автоматически при сбросе процессора Reset или при включении питания). Содержимое этого регистра и технология его программирования рассматриваются при описании файла начальной инициализации системы Startup (см. 9.1).

7.6 Влияние RISC-архитектуры процессора на его программную модель

Во всех архитектурах современных процессоров в состав процессорного ядра входит сверхоперативная память, содержащая регистры общего назначения. В процессорах Cortex-M такая память имеется как в окружении центрального процессора, так и в окружении сопроцессора. Особенность процессоров с RISC- архитектурой в том, что их система команд поддерживает *вычислительные операции исключительно с регистрами окружения* и не поддерживает операции с данными, расположенными в памяти (кроме операций пересылки данных).

Операционный блок процессора и сопроцессор *оптимизированы на работу с регистрами окружения*. В кодах команд при доступе к регистрам окружения процессора и сопроцессора достаточно указать их внутренние адреса, а не 32-разрядные адреса ячеек памяти, как это было бы при расположении операндов в памяти. Так, для адресации 16 регистров общего назначения ЦПУ достаточно использовать поле адреса всего из четырех бит (адреса от 0 до 15). Следовательно, для задания двух адресов регистров-источников и одного адреса регистра-приемника в формате команды достаточно 4*3=12 бит. При этом остальные биты (в 16- или 32-разрядной команде) могут использоваться для задания кода операции. Такой способ адресации операндов в команде называется *регистровой адресацией*. Он позволяет создавать короткие команды (16 или 32 бита), быстро считывать их из кодовой памяти и выполнять. При этом никакого обращения к внешней памяти программ или данных за операндами не требуется.

Как доказывает успех фирмы ARM, именно такой подход позволяет создать максимально компактную систему команд и обеспечить максимально высокую производительность вычислительного процесса в конвейерных архитектурах процессоров (глава 6).

Естественно, что ввиду ограниченности числа регистров ЦПУ и сопроцессора обращения к памяти неизбежны. Однако, они выполняются помимо операционного блока процессора через также оптимизированный по быстродействию интерфейс процессора с памятью данных. При этом используется эффективный механизм *косвенной адресации* памяти по содержимому любого регистра общего назначения процессора, в том числе счетчика команд PC (относительная адресация) и указателя стека SP (стековая адресация). Любой регистр общего назначения ЦПУ может выполнять функцию *регистра-указателя*

адреса расположения данных в памяти. Это чрезвычайно эффективный механизм доступа к памяти. Он поддерживает *пост- и пред- автоинкрементную и автодекрементную адресацию*, а также *групповую загрузку/сохран*ение содержимого сразу нескольких регистров (глава 11).

Таким образом, архитектура процессора Cortex-M и его система команд эффективно поддерживают все три стадии вычислительного процесса:

1. **Загрузка регистров ЦПУ или сопроцессора исходными данными**:
   1.1. Непосредственная – константами, расположенными в формате самой команды (поз. 2 на рис. 7.1). Число возможных констант ограничено, так как применяется специальный механизм «сжатия» данных;
   1.2. Из других регистров окружения ЦПУ (поз. 3) или сопроцессора (поз. 4), уже содержащих нужные данные;
   1.3. Из кодовой памяти или памяти данных (поз. 5 и 6). Естественно, что эти данные должны быть предварительно размещены там, для чего используются специальные директивы Ассемблера.
2. **Реализация вычислительного алгоритма** – арифметических или логических операций над целыми числами и числами с фиксированной точкой в операционном блоке ЦПУ (поз.7) или над числами в формате с плавающей точкой – в сопроцессоре (поз. 8).
3. **Сохранение результатов вычисления** (при необходимости) в оперативной памяти микроконтроллера (поз. 5 и 6).

Особенность процессорных ядер Cortex-M в том, что в процессе пересылки данных возможна *попутная обработка этих данных*, например, преобразование числа из формата с фиксированной точкой в формат числа с плавающей точкой. Поэтому, стрелки на рис. 7.1, иллюстрирующие межрегистровый обмен, показаны условно. На самом деле данные в процессе пересылки могут подвергаться обработке в операционном блоке ЦПУ или в сопроцессоре.

Еще раз внимательно изучите рис. 7.1. По существу, он содержит *классификацию команд процессора по их функциональному назначению*. Мы будем придерживаться этой классификации при изучении команд процессора в последующих главах книги.

7.7 Назначение и особенности использования регистров окружения ЦПУ

7.7.1 Регистры общего назначения

Регистры *младшего регистрового банка* (R0÷R7) не имеют каких-либо ограничений по использованию и могут применяться в качестве операндов в любых командах процессора. Использование регистров *старшего банка* (R8÷R12) в некоторых командах не допускается. Поэтому, при написании своих программ, старайтесь, по возможности, обойтись регистрами младшего регистрового банка.

В процессорах Cortex-M два набора команд, существовавших в предыдущих версиях процессоров этой фирмы (набор 32-разрядных команд ARM и набор 16-разрядных команд Thumb), объединены в единый набор команд Thumb2. При этом никаких программных переключений из одного набора команд в другой, как это было ранее, не требуется. Транслятор сам выбирает оптимальный способ кодирования команды: если он сможет сгенерировать более компактную 16-разрядную команду – делает это, в противном случае – генерирует 32-разрядную.

Транслятор с Ассемблера является «интеллектуальным» и оптимизирует объем программного кода. Команды набора ARM (32-разрядные) генерируются, как правило, для много-операндных команд и команд, использующих в качестве операндов регистры старшего регистрового банка.

7.7.2 Счетчик команд PC. Относительная адресация в командах передачи управления

Регистр R15 в процессорах Cortex-M выполняет специальную функцию – определяет порядок выполнения команд в программе и называется *счетчиком команд* PC (Program Counter) или *программным счетчиком*. Он всегда содержит адрес очередной команды, подлежащей выполнению, то есть адрес команды, которая уже считана из кодовой памяти на конвейер команд и прошла первые две стадии обработки на конвейере – выборку и дешифрацию. Работа конвейера команд описана нами ранее (см. 4.4).

Так как в унифицированном наборе команд Thumb2 присутствуют и 32-разрядные и 16-разрядные команды, то в зависимости от формата конкретной команды, в процессе ее выполнения, счетчик команд PC автоматически инкрементируется на +4 или на +2 (размер слова или полуслова в байтах).

Большая часть любого программного кода выполняется последовательно. Однако, часто требуется либо *безусловная передача управления* в другую точку программы, либо *условная передача управления* – по некоторому условию, которое может быть истинным или ложным в зависимости от результата предыдущих вычислений. В этих случаях счетчик команд PC с помощью соответствующих команд загружается адресом точки входа в нужный фрагмент программы, и выполнение программы продолжается с нового адреса.

Точка программы, в которую должен быть выполнен переход, помечается меткой. Значение метки соответствует адресу размещения в памяти соответствующей команды и автоматически идентифицируется транслятором с Ассемблера. Для безусловной передачи управления в процессорах ARM используется команда **`B label`**, где мнемокод «B» соответствует английскому слову branch (ответвление, передача управления), а «label» – является именем *метки*, содержащим адрес точки входа в нужный фрагмент программы. Вы скажете, а как же 32-разрядный адрес ячейки памяти помещается в 32-разрядной команде? Ведь, в этом случае, не остается ни одного бита для размещения собственно кода операции? Может быть, такая команда будет занимать в кодовой памяти не одно, а целых два слова – 64 бита? Но, это противоречит концепции RISC-архитектуры процессора, в соответствии с которой все команды должны быть не длиннее одного слова.

Разработчики ARM-процессоров для исключения этого противоречия предложили в командах передачи управления использовать не *прямую адресацию* (когда адрес перехода указывается отдельным словом в формате команды), а *относительную адресацию* по содержимому счетчика команд PC и *дополнительному короткому смещению* #offset, которое должно присутствовать в формате таких команд, как непосредственная константа.

Суть метода иллюстрирует рис. 7.2. В начале выполнения любой команды счетчик команд PC автоматически инкрементируется так, что всегда показывает на адрес очередной команды, подлежащей выполнению. Если по адресу M1 находится команда безусловной передачи управления **`B M5`**, то в начале ее выполнения значение счетчика команд будет равно M2 (адресу следующей команды в программе), то есть будет известно процессору. Транслятор с Ассемблера, «зная» адрес метки M5, на которую должен быть сделан переход, и адрес метки M2, может, на стадии ассемблирования программы,

рассчитать величину смещения offset = (M5-M2) и записать его в виде непосредственного операнда #offset в формат команды **`B M5`**.

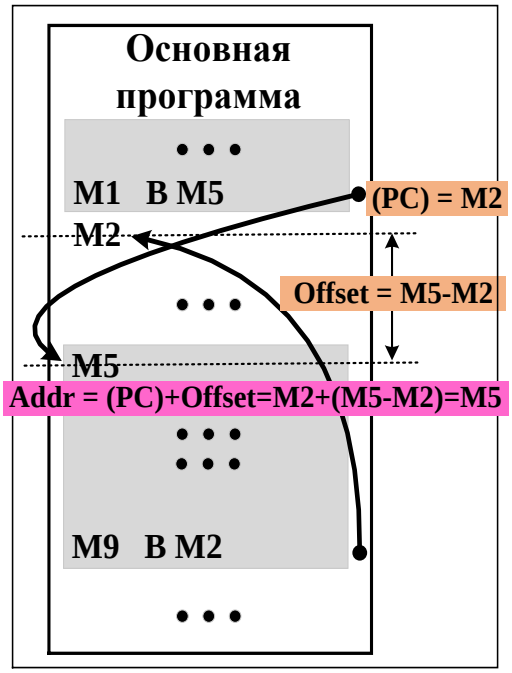
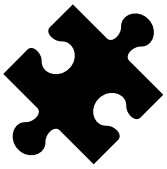


Рис. 7.2 Передача управления по содержимому счетчика команд PC и непосредственному смещению #offset.

На стадии выполнения команды смещение просто добавляется к текущему значению счетчика команд: (PC) = M2+(M5-M2) = M5, в результате чего счетчик команд перезагружается нужным адресом перехода addr = (PC) + #offset. Операции по расчету величины смещения и кодированию команды перехода полностью берет на себя транслятор с Ассемблера. Вы, как программист, должны лишь поставить метку в нужном месте программы и написать команду перехода на нее B label.

Передача управления может выполняться как в направлении «Вперед» (в сторону возрастающих адресов), так и в направлении «Назад» (в сторону убывающих адресов), что необходимо при организации в программе циклов. В этом случае транслятор с Ассемблера закодирует в команде перехода отрицательное значение смещения (см. команду **`B M2`** на рис. 7.2).



Будьте осторожны при использовании регистра R15 (PC)! Хотя он является регистром общего назначения и в большинстве команд может использоваться как регистр-источник или регистр-приемник данных, помните, что в последнем случае это вызовет передачу управления по адресу, равному новому содержимому R15. Вы можете случайно запрограммировать передачу управления, причем «в никуда»! По этому адресу может вообще не быть кода – поведение Вашей программы будет *непредсказуемым*!

Имейте также в виду, что в некоторых командах использование регистра R15(PC) в качестве операнда-приемника и даже операнда-источника – запрещено.

При переходах с использованием относительной адресации по текущему значению счетчика команд возможны «короткие переходы» (адресное смещение в диапазоне ±1024) и «средние переходы» (адресное смещение ±4096). Выбор нужного кода операции перехода транслятор делает автоматически.

При необходимости «длинного перехода» на удаленную метку придется воспользоваться командой непосредственной загрузки счетчика команд 32-разрядным адресом перехода (глава 10).

На какой адрес области памяти всегда показывает содержимое счетчика команд PC?
1) Чему окажется равно содержимое счетчика команд PC после выполнения команды `B M2` (см. рис. 7.2)?

1) На адрес следующей, подлежащей выполнению команды.
2) Адресу метки M2.

7.7.3 Регистр-указатель стека. Понятие стека.

Регистр R13 имеет особое значение. Он является *регистром-указателем стека* SP. *Стеком* называется специально выделенная для временного хранения информации *линейная область памяти данных* (ОЗУ). Она используется как для сохранения *адресов возврата* в основную программу из подпрограмм (функций), так и для временного хранения любых других данных из регистров процессора. Одно из основных назначений стека – *сохранение контекста* основной программы путем записи значений некоторых нужных регистров в стек в начале подпрограммы или процедуры обработки прерывания/исключения и *восстановление контекста* перед операцией возврата из подпрограммы или процедуры обработки прерывания/исключения в основную программу. Это позволяет внутри таких процедур «безбоязненно» использовать эти регистры.

Указатель стека SP содержит *адрес последней, «занятой» данными ячейки памяти* в области стека. При очередной записи данных в стек указатель стека предварительно декрементируется, указывая на адрес очередной свободной ячейки памяти, после чего в нее записываются нужные данные. Все команды работы со стеком выполняют запись и извлечение из стека только полных 32-разрядных слов (4-байтовых), поэтому указатель стека всегда модифицируется на -4 (при записи) и на +4 (при чтении). Напомним, что в процессорах ARM собственный адрес имеет каждый байт памяти.

Стек такого типа называется LIFO (Last In, First Out) – «Последний вошел, первый вышел». Его принцип действия аналогичен «магазину» патронов пистолета или автомата: последний, поступивший в «магазин» патрон, извлекается в ствол первым.

Порядок записи данных в стек или извлечения из него определяет текущее содержимое указателя стека SP. Во всех командах работы со стеком значения указателя стека SP модифицируется автоматически. На рис. 7.3 иллюстрируется принцип работы стека и авто-модификации содержимого указателя стека. Предполагается, что выполняются четыре операции сохранения в стеке значений переменных VAR1, VAR2, VAR3 и VAR4, расположенных в регистрах общего назначения ЦПУ (команды PUSH {Rn}) и две операции извлечения из стека ранее записанных данных (команды POP {Rn}).

Использование в поле операндов команд PUSH {Rn} и POP {Rn} фигурных скобок означает задание *списка регистров* Reglist, содержимое которых должно быть сохранено в стеке или восстановлено из него. По существу, это команды группового сохранения/восстановления данных (более подробно в главе 15).

При включении питания или сбросе процессора *указатель стека SP автоматически инициализируется начальным значением*, задающим положение так называемой *вершины стека*. В примере на рис. 7.3 это значение равно 0x20005000 (находится в конце области памяти данных в соответствии с унифицированной картой памяти), следовательно, все ячейки памяти с большими адресами считаются «как бы занятыми», а с меньшими – «свободными».

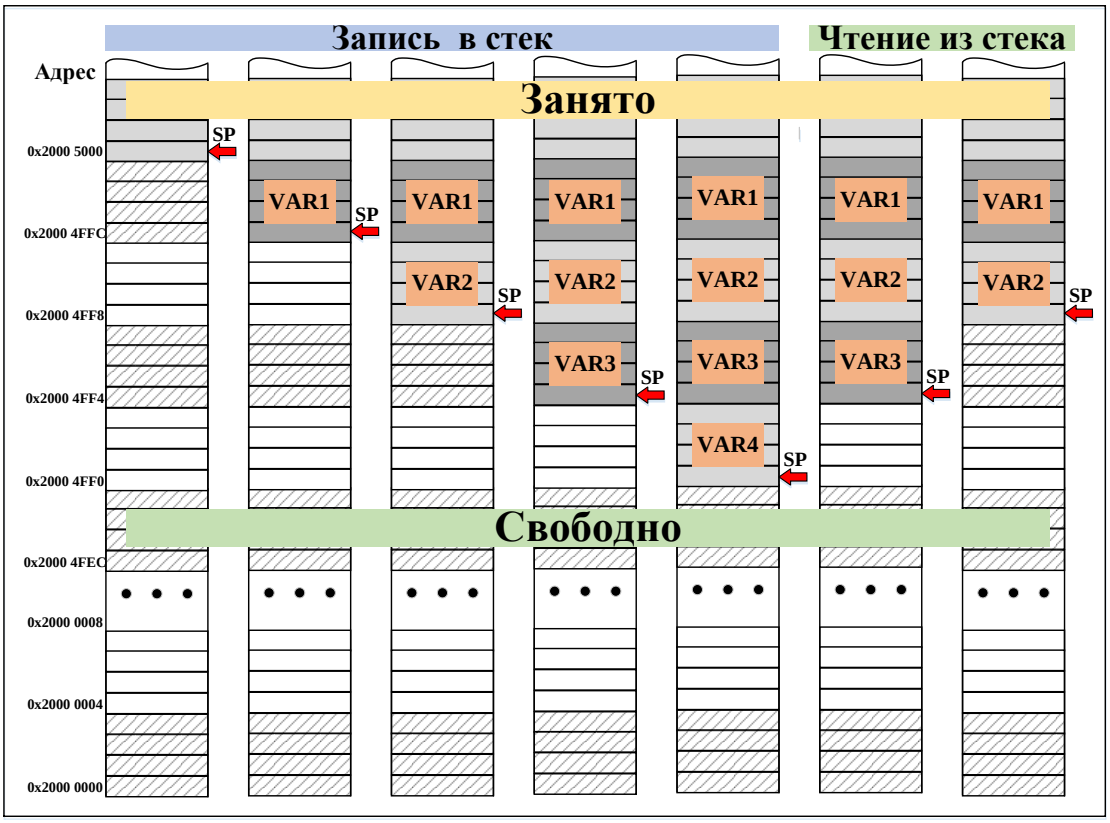


Рис. 7.3 Запись и извлечение данных из стека

Как видите, при записи в стек значение в указателе стека SP автоматически декрементируется (-4) – стек заполняется данными в направлении убывающих адресов памяти – это так называемый *автодекрементный стек по записи*. При извлечении данных из стека в регистры ЦПУ вначале выполняется считывание данных, после чего указатель стека автоматически инкрементируется (+4), как бы высвобождая текущую ячейку памяти для новой записи (ранее записанное в нее значение сохраняется неизменным, пока не состоится новая запись).

В действительности указатель стека SP состоит из двух регистров: PSP – *указателя стека процесса пользователя* и MSP – *указателя стека обработчиков исключений*. К какому именно указателю стека в данный момент обращается процессор, зависит от текущего режима работы процессора (непривилегированный или привилегированный). Это позволяет физически иметь два стека: стек для пользовательских программ (нижнего уровня), работающих в непривилегированном режиме, и стек для программ более высокого уровня – операционных систем реального времени и мониторов, работающих в привилегированном режиме. Такое разделение стеков обеспечивает независимость программ разного уровня друг от друга и повышает надежность микропроцессорной системы в целом.

1) Возможна ли запись и извлечение из стека байта, полуслова?
2) Как понимать термин «автодекрементный стек по записи»?
3) Можно ли использовать указатель стека SP в качестве обычного операнда в командах?

1) Нет, только 32-разрядного слова целиком.
2) Это означает, что перед записью очередных данных в стек значение указателя сначала декрементируется (на 4) и только затем выполняется запись.
3) Да, в большинстве команд. Только для некоторых команд использование указателя стека SP и счетчика команд PC ограничивается.

7.7.4 Регистр связи. Внутренняя модульность программного обеспечения.

Особую роль при программировании играет регистр R14, который называется *регистром связи* или линк-регистром LR. Понятие *внутренней модульности* программы, написанной на Ассемблере и расположенной в отдельном файле, предполагает, что она состоит из *основной программы* и некоторого числа *подпрограмм (функций)*, которые могут вызываться из основной программы в произвольном порядке любое число раз.

Каждая подпрограмма предварительно отлаживается на всем возможном наборе значений аргументов. Перед вызовом подпрограммы ей передаются конкретные значения аргументов, после чего следует вызов подпрограммы. Технология передачи параметров в подпрограммы и особенности работы с вложенными подпрограммами (одна вызывает другую) рассматривается отдельно (глава 15).

В большинстве процессоров для вызова подпрограмм используется специальная команда типа **`CALL addr`**, которая вначале запоминает в стеке следующий после этой команды адрес, называемый *адресом возврата*, а затем загружает в счетчик команд PC значение addr – адрес *точки входа в подпрограмму*. В результате чего управление передается подпрограмме. В конце подпрограммы обычно используется команда **`RET`** (Возврат), которая извлекает из стека адрес возврата и записывает его в счетчик команд PC. Тем самым управление возвращается в основную программу.

Отличительной особенностью процессоров ARM является то, что в их архитектуре для сохранения адреса возврата используется не стек, расположенный во внешней по отношению к процессору памяти данных, а специально предназначенный для этой цели регистр общего назначения – *регистр связи* LR, который можно рассматривать как *регистр обратной связи* при вызове подпрограмм. Преимуществом такого подхода является то, что адрес возврата сохраняется не в памяти, доступ к которой может быть медленным, а в одном из регистров окружения ЦПУ, доступ к которому – предельно быстрый, что обеспечивает существенное *повышение вычислительной производительности процессора при работе с подпрограммами*.

Рис. 7.4 иллюстрирует механизм вызова подпрограмм и возврата в основную программу, реализуемый с использованием двух регистров ЦПУ – регистра связи LR и счетчика команд PC. В отличие от простого перехода по метке (команда **`B label`**) для вызова подпрограммы используется команда *передачи управления с обратной связью* Branch with link – **`BL label`**: сначала адрес возврата (адрес следующей команды в программе) сохраняется в регистре связи LR, а затем счетчик команд PC загружается адресом метки label.

Последней командой подпрограммы должна быть команда *косвенной загрузки счетчика команд* PC (Branch indirect) текущим содержимым регистра связи LR – **`BX LR`**. Это аналог команды **`RET`** («Возврат») в других процессорах. В мнемокоде этой команды первая буква «B» от англ. **B**ranch (ответвление), а вторая от англ. E**x**change (обмен). В команде выполняется «как бы обмен» данными между двумя регистрами, один из которых указан в качестве операнда-источника – LR, а второй – «по умолчанию» является регистром-приемником – счетчиком команд PC.

В общем случае команда косвенной передачи управления имеет следующий синтаксис:

```
BX Rm
```

По написанию она является одно-операндной командой – переслать данные из регистра-источника Rm в счетчик команд PC, имя которого в качестве регистра-приемника опущено и предполагается «по умолчанию». Это действительно *косвенная передача управления* по адресу, расположенному в регистре общего назначения Rm. Если операндом-источником является регистр связи LR, то сохраненный в нем ранее адрес возврата загружается в счетчик команд – выполняется возврат в основную программу.

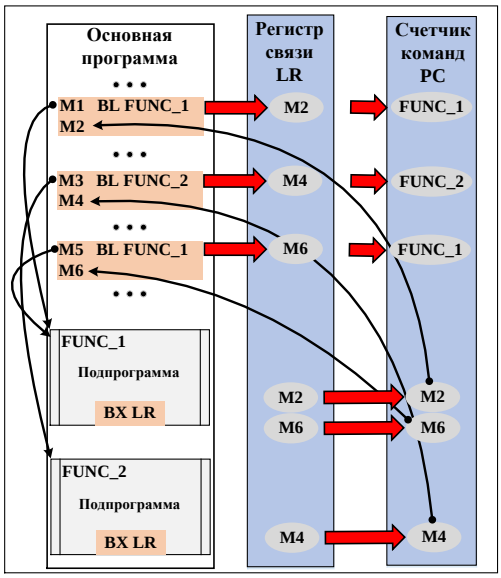


Рис. 7.4 Внутренняя модульность программного обеспечения

На самом деле мнемоника команды не совсем точна. Полноценного обмена данными PC↔Rm не происходит, выполняется обмен только в одну сторону PC←Rm.

На Рис. 7.4 красными стрелками показано состояние регистра связи LR и счетчика команд PC после выполнения команд вызова подпрограмм **`BL label`** и команд **`BX LR`** возврата из подпрограммы в основную программу.

К любой подпрограмме можно обращаться любое число раз (на Рис. 7.4 – к подпрограмме FUNC_1).

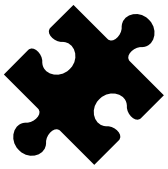
Пара команд **`BL Label`** и **`BX LR`** выполняет в процессорах ARM функции вызова подпрограммы **`Call Label`** и возврата из нее в основную программу **`Ret`**. Вызов и возврат выполняются предельно быстро, так как для сохранения адреса возврата используется не стек в памяти данных, а специальный регистр обратной связи LR, являющийся регистром общего назначения ЦПУ.

При использовании вложенных подпрограмм для сохранения адресов возврата все же придется использовать стек (глава 15).

1) Можно ли вместо команды BX LR использовать команду BX R14?
2) А команду пересылки данных из регистра-источника LR в регистр-приемник PC: MOV PC, LR или ее аналог MOV R15, R14?
3) Как Вы думаете, что означает команда BLX Rm?

1) Да, R14 всего лишь альтернативное обозначение регистра связи LR.
2) Да, оба варианта. Однако, привыкайте использовать общепринятую в процессорах ARM мнемонику команды возврата из подпрограммы BX LR.
3) Так как в ее мнемокоде есть корень **BL**, можно предположить, что это вызов подпрограммы с сохранением адреса возврата в регистре связи LR. Суффикс **X** указывает на то, что новая загрузка счетчика команд будет не по адресу метки label, как в команде **`BL label`**, а по адресу, расположенному в регистре Rm, указанному в

поле операндов команды («с обменом»). Так что это команда косвенной передачи управления по содержимому регистра Rm с одновременным сохранением адреса возврата в регистре связи LR (Branch indirect with Link).



Исторически команды передачи управления использовались в процессорах ARM также для переключения между возможными двумя наборами команд ARM и Thumb. Переключение выполнялось младшим битом адреса перехода. В процессорах Cortex-M используется единый набор команд Thumb-2. Однако, младший бит адреса перехода по-прежнему отвечает за текущий набор команд процессора. Поэтому, при выполнении косвенных переходов приходится следить за тем, чтобы он был равен 1 (набор Thumb). Переход всегда будет выполняться по правильному адресу – с автоматическим очищением младшего бита адреса перехода.

7.8 Регистр статуса программы. Флаги результатов операций в операционном блоке ЦПУ

7.8.1 Регистр статуса программы и его компоненты

Любой вычислительный процесс и, тем более, процесс управления оборудованием, предполагает анализ текущих значений переменных (больше, меньше, равно и т.д.), а также состояния внешних датчиков и оперативное принятие решения: «Что делать дальше?». Поэтому, важнейшим элементом любого операционного блока процессора является так называемый *регистр признаков результатов операций* или *регистр флагов операций*.

В процессорах Cortex-M этот регистр называется *регистром статуса программы-приложения* APSR (Application Program Status Register). Вместе с *регистром статуса системы прерываний и исключений* IPSR (Interrupt Program Status Register), а также *регистром статуса системы выполнения команд* EPSR (Execution Program Status Register) эти три регистра образуют один 32-разрядный регистр статуса программы PSR (Program Status Register).

На рис. 7.5 показано распределение битовых полей между тремя регистрами статуса, приведены символические названия этих полей.

Регистр статуса программы PSR																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Регистр статуса программы приложения APSR																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	**Z**	**C**	**V**	**Q**																											

Регистр статуса системы прерываний/исключений IPSR																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																							ISR_NUMBER									

Регистр статуса системы выполнения EPSR																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					IC/IT		**T**									**IC/IT**																

Рис. 7.5 Регистр статуса программы PSR и его три составляющие APSR, IPSR, EPSR

Отметим, что несмотря на то, что все три регистра статуса являются частью одного и того же регистра PSR, в Ассемблерной программе к каждому из них можно обращаться по его персональному имени: APSR, IPSR или EPSR. Возможно и обращение по символическому имени PSR. При этом будут считаны состояния всех трех статусных регистров одновременно.

При чтении содержимого статусных регистров в регистры общего назначения ЦПУ будут считаны только те битовые поля, которые относятся к указанному Вами регистру. Отметим также, что регистры статуса являются *регистрами специального назначения*. Доступ к ним возможен только с помощью *специальных команд* обмена данными между регистрами общего назначения ЦПУ и регистрами специального назначения (эти команды рассматриваются в последующих главах).

7.8.2 Флаги результатов операций в ЦПУ

Регистр статуса приложения APSR содержит пять флагов результатов операций, расположенных в самых старших битах регистра APSR – (31÷27) (см. табл. 7.3):

Таблица 7.3 Описание флагов

Флаг	Название. Порядок формирования
N	**Флаг отрицательного результата N (Negative). Устанавливается в 1 (N=1), если в результате операции получено число, старший бит которого (31-й, знаковый) равен 1. Если старший бит равен нулю, то флаг N сбрасывается в 0 (N=0).**
Z	**Флаг нуля Z (Zero). Если в результате арифметической или логической операции получен нулевой результат (все 32 бита результата нулевые), то флаг выставляется (Z=1), в противном случае – сбрасывается (Z=0).**
C	**Флаг переноса C (Carry).** **Устанавливается (C=1), если в результате операции сложения двух 32-разрядных чисел произошел *перенос* за пределы разрядной сетки. В противном случае – сбрасывается (C=0). Свидетельствует о *переполнении при сложении чисел без знака*.** **Сбрасывается (C=0), если в результате операции вычитания двух 32-разрядных чисел произошел «*заем*» и выставляется (C=1) в противном случае (нет «заема»).**
V	**Флаг переполнения V (Overflow). Выставляется (V=1) в том случае, когда при выполнении арифметической операции сложения или вычитания чисел, рассматриваемых процессором как числа со знаком в дополнительном коде, возникло *переполнение*, т.е. полученный 32-разрядный результат вышел за пределы диапазона 32-разрядных чисел со знаком, в противном случае – сбрасывается (V=0).**
Q	**Флаг насыщения Q (saturation). Вырабатывается только двумя командами процессора – насыщение содержимого регистра как числа без знака USAT и насыщение содержимого регистра как числа со знаком SSAT. Если насыщение (ограничение на уровне максимального значения) имело место, то флаг устанавливается (Q=1), в противном случае – сбрасывается (Q=0).**

Мы уже отмечали (глава 3), что операции двоичного сложения/вычитания чисел без знака и чисел со знаком в дополнительном коде выполняются в АЛУ одними и теми же командами. Более того, процессор не может отличить число без знака от числа со знаком в дополнительном коде. О том, с какими числами в данный момент выполняется операция сложения или вычитания знает только программист. Результат операции всегда будет правильным, если он не выходит за пределы разрядной сетки процессора (в данном случае 32-разрядной). О выходе за эти пределы при работе с числами без знака свидетельствует флаг переноса C, а при работе с числами со знаком – флаг переполнения V. Поэтому, процессор всегда формирует оба флага (и C, и V). Нужный флаг анализируется программистом исходя из того, с какими числами он работает в данный момент (со знаком или без знака).

1) Работаем с числами без знака. Какое значение должна превысить сумма, чтобы возник флаг переноса C?
2) Работаем с числами со знаком в дополнительном коде. В каком диапазоне должен находиться результат операции сложения или вычитания целых чисел, чтобы флаг переполнения V не возник?
3) Когда устанавливается флаг нулевого результата Z?

1) Максимальное целое число без знака $(2^{32}-1)= 4,294,967,295$
2) В диапазоне $-(2^{31})\div(2^{31}-1)$, то есть от -2,147,483,648 до +2,147,483,647
3) Тогда, когда результат тождественно равен нулю, т.е. во всех двоичных разрядах, без исключения, нули.

7.8.3 Программное управление формированием флагов результатов операций

В большинстве процессоров флаги результатов операций формируются автоматически после завершения каждой арифметической или логической операции. В процессорах ARM – *это не так*. Они выставляются *только тогда, когда программист этого требует*. Если такого требования нет, то флаги не модифицируются (их значения остаются прежними).

С какой целью это делается? Для повышения надежности программного кода.

Если программист желает выполнить команду условной передачи управления (или условно выполнить одну или группу команд), то он *должен в явном виде указать*, по результатам какой именно операции должны быть установлены флаги. Это повышает ответственность программиста за написанный им код: прежде чем написать команду, которая может существенно изменить ход вычислительного процесса или процесса управления, необходимо определиться, на основании результатов какой конкретной операции это изменение должно быть сделано.

Указание на установку флагов делается путем добавления к мнемонике любой команды суффикса «S» (Set – Установить флаги). Итак, чтобы заставить процессор сформировать флаги результатов операции в регистре APSR (N, Z, C, V, Q), нужно добавить к основному мнемокоду команды суффикс «S»: <Основной мнемокод команды><S>. Ниже в качестве примера показана команда сложения двух операндов в регистрах r3 и r4 с сохранением суммы в регистре r0 в двух вариантах: 1) без установки флагов результата операции сложения; 2) с установкой флагов:

```
ADD r0, r3, r4      ; Без установки флагов
ADDS r0, r3, r4     ; С установкой флагов
```

Здесь и далее в книге суффиксы, которые модифицируют операцию, *будут выделяться цветом*. Это делается исключительно *в учебных целях*. При написании реальных программ цветовое деление внутри мнемокода команды не используется.

Единственным исключением из приведенного выше правила *«принудительного» формирования флагов результатов операций* являются команды сравнения CMP и тестирования: TST (Логическое И с маской); TEQ (тестирование на эквивалентность – Логическое Исключающее ИЛИ двух операндов). Команда сравнения предназначена только для того, чтобы сравнить два операнда путем вычитания из первого операнда второго. По результату операции сравнения выставляются флаги в регистре APSR, а сам результат вычитания нигде не сохраняется (отбрасывается). Приведенные ниже две

команды с точки зрения установки флагов эквивалентны. Разница заключается в том, что в команде вычитания результат вычитания сохраняется в регистре-приемнике r1, а в команде сравнения – просто теряется:

```
SUBS r1, r2     ; [(r1)-(r2)] → r1, установить флаги
CMP r1, r2      ; [(r1)-(r2)], установить флаги
```

Обратите внимание на то, что в команде вычитания программист сделал явное указание процессору на необходимость формирования флагов результата операции с помощью дополнительного суффикса «S». В команде сравнения этот суффикс не требуется. Заметьте, что в примере использована двух-операндная команда вычитания. Функцию регистра-приемника автоматически выполняет первый операнд-источник (регистр r1).

7.9 Условная передача управления и условное выполнение команд

Процессоры фирмы ARM имеют одну очень важную особенность, которая принципиально отличает их от процессоров других фирм – большинство команд, входящих в систему команд и выполняемых как в операционном блоке ЦПУ, так и в сопроцессоре, *может выполняться условно*. Такие команды будут иметь мнемокод, расширенный специальным *кодом условного выполнения* cond:

```
<Мнемокод команды><cond>
```

Код условного выполнения представляет собой суффикс, который добавляется к мнемокоду команды без каких-либо пробелов или разделителей. Если команда уже имеет в своем обозначении какой-то/какие-то суффиксы, то код условного выполнения указывается после них (в самом конце мнемокода команды). В справочнике по системе команд те команды, которые могут быть выполнены условно, имеют в описании синтаксиса опциональный код условного выполнения {cond}.

7.9.1 Коды условной передачи управления и условного выполнения команд

Ветвление программы или условное выполнение команд возможно в следующих случаях:

1) Сразу после любой арифметической или логической операции, в мнемокоде которой присутствует суффикс «S» (установить флаги результата операции), или после любого числа следующих за ней команд, которые сами не модифицируют флаги.
2) Сразу после операции сравнения 32-разрядных чисел с использованием команды сравнения CMP или одной из команд тестирования TST, TEQ;
3) После сравнения чисел в формате с плавающей точкой командой VCMP.F32 и дополнительного копирования флагов из регистра FPSCR (статуса и управления модуля плавающей точки), в регистр статуса программы приложения APSR – для этого предусмотрена специальная команда (глава 21).

Коды условного выполнения строго соответствуют определенным состояниям флагов результатов операций – либо отдельных флагов, либо их комбинации. Это соответствие приведено в табл. 7.4 с указанием значения суффикса при условии, что была выполнена операция сравнения двух 32-разрядных чисел. Заметим, что по результату сравнения чисел в формате с плавающей точкой также можно использовать указанные

ниже суффиксы (они общие для ЦПУ и для сопроцессора). Для этого необходимо предварительно скопировать флаги результата сравнения чисел в формате с плавающей точкой из регистра FPSCR сопроцессора в регистр APSR – подробно в главах 20, 21.

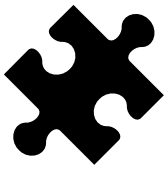
Таблица 7.4 Суффиксы условного выполнения команд и их значение

Суффикс	Флаги	Значение
EQ	**Z=1**	**«Эквивалентно» (Equal)**
NE	**Z=0**	**«Не эквивалентно» (Not Equal)**
HS или CS	**C=1**	**«Выше или то же самое» (Higher or same) или «Флаг Carry установлен» (Carry Set) Больше или равно (≥) при сравнении чисел без знака (unsigned)**
LO или CC	**C=0**	**«Ниже»(Lower) или «Флаг Carry сброшен» (Carry Clear) Строго меньше (<) при сравнении чисел без знака (unsigned)**
MI	**N=1**	**«Минус» (Negative) - Отрицательное**
PL	**N=0**	**«Плюс» (Positive or Zero) – Положительное или ноль**
VS	**V=1**	**«Переполнение знаковое» (Overflow)**
VC	**V=0**	**«Нет знакового переполнения» (No Overflow)**
HI	**(C=1) and (Z=0)**	**«Выше» (Higher) Строго больше (>) при сравнении чисел без знака (unsigned)**
LS	**(C=0) or (Z=1)**	**«Ниже или то же самое» (Lower or same) Меньше или равно (≤) при сравнении чисел без знака (unsigned)**
GE	**N=V**	**«Больше или эквивалентно» (Greater then or equal) Больше или равно (≥) при сравнении чисел со знаком (signed)**
LT	**N!=V**	**«Меньше чем» (Less then) Строго меньше (<) при сравнении чисел со знаком (signed)**
GT	**(Z=0) and (N=V)**	**«Строго больше» (Greater then) Строго больше (>) при сравнении чисел со знаком (signed)**
LE	**(Z=1) or (N!=V)**	**«Меньше чем или эквивалентно» (Less then or equal) Меньше или равно (≤) при сравнении чисел со знаком (signed)**
AL	**Любые значения флагов**	**«Всегда». Суффикс по умолчанию, который используется всегда, когда код условного выполнения не специфицируется.**

Для сравнения переменных без знака и со знаком в дополнительном коде используются разные суффиксы. Так, математическому символу «>» (Больше) для чисел без знака соответствует суффикс «HI» (Выше), а для чисел со знаком? – символ «GT» (Строго больше).

Процессор при выполнении операции сравнения (или вычитания) не «знает» с какими числами он имеет дело: с числами без знака или со знаком в дополнительном коде. В любом случае, на основании значений всего 4-х флагов N, Z, C, V он может оценить истинность или ложность всех возможных условий сравнения (=, ≠, >, ≥ , ≤, <) и для чисел без знака и для чисел со знаком в дополнительном коде, формируя значения всех кодов условного выполнения, приведенных в табл. 7.4. Программист, зная с какими числами он работает, должен выбрать команду условного ветвления или условного выполнения с соответствующим суффиксом:

- Для сравнения любых чисел на равенство или неравенство - EQ или NE
- Для оценки знака числа – MI или PL
- Для сравнения только чисел без знака - HS, LO, HI, LS (выделены зеленым)
- Для сравнения только чисел со знаком – GE, LT, GT, LE (выделены голубым)



1. Символ восклицательного знака в таблице использован для задания операции инверсии флага N!;
2. Напомним, что при вычитании двоичных чисел флаг Carry (C) – выполняет также функцию флага «заема». Если «заема» нет, то флаг C устанавливается в 1, в противном случае сбрасывается в 0. Наличие флага C свидетельствует о том, что при вычитании двух чисел без знака «заема» нет, следовательно, первое число «Выше или равно», т.е. «≥» для чисел без знака;
3. Условие «GE» (Строго больше) при сравнении чисел со знаком в дополнительном коде проверяется по равенству флагов отрицательного результата N и знакового переполнения V: V=N. Приведем несколько простых примеров *байтовых* операций вычитания, чтобы Вы могли убедиться в том, что такая проверка допустима (результаты представлены для условно 8-разрядного АЛУ, хотя на самом деле оно 32-разрядное) (см. табл. 7.5).

Таблица 7.5 Иллюстрация работы условия «GE»

Операция сравнения чисел со знаком	**Состояние флагов N и V**	**Флаги эквивалентны?**	**Результат сравнения**
(+3)-(+2)=+1	**N=0, V=0**	**Да**	**≥**
(+2)-(+3)=-1	**N=1, V=0**	**Нет**	**<**
(+3)-(+3)=0	**N=0, V=0**	**Да**	**≥**
(-1)-(-3)=+2	**N=0, V=0**	**Да**	**≥**
(-3)-(-1)=-4	**N=1, V=0**	**Нет**	**<**
+127-(-3)=+130	**N=1, V=1**	**Да**	**≥**
(-3)-(+127)=-130	**N=0, V=1**	**Нет**	**<**

4. Из этой же таблицы следует, что условие «LT» (Строго меньше) при сравнении знаковых операндов можно проверять по взаимной инверсии флагов: N!=V

При выполнении операции сравнения или любой команды с суффиксом обязательного формирования флагов «S» процессор автоматически формирует значения всех четырех флагов N, Z, C, V, состояние которых однозначно определяет все возможные результаты сравнения чисел без знака и со знаком. Вслед за командой сравнения должна применяться команда условного ветвления или условного выполнения с конкретным суффиксом условного выполнения (с тем условием, которое проверяется программистом).

7.9.2 Блоки условного выполнения команд

В процессорах Cortex-M технология условного выполнения команд усовершенствована по сравнению с ранее применявшейся в процессорах ARM: команды условного выполнения (от одной до 4-х) автоматически помещаются транслятором с Ассемблера в так называемый *блок условного выполнения команд*, который начинается *специальной командой условного выполнения блока* IT (If-Then). Этот механизм, подробно рассмотренный далее (см. 12.4), позволяет поднять скорость выполнения команд условного выполнения и производительность процессора. Автоматическая генерация блоков условного выполнения транслятором позволяет нам пока не вдаваться в детали этого процесса. Заметим только, что блоки условного выполнения не генерируются для команд условного ветвления программы. Они появляются только при условном выполнении других операций (арифметических, логических, пересылки и т.д.).

Число условно выполняемых команд в одном блоке – не более четырех. Если команд больше, то генерируется последовательность из нескольких блоков.

7.9.3 Примеры условной передачи управления и условного выполнения команд

Пример 1. Требуется сравнить два 32-разрядных числа без знака в регистрах r3 и r4. Если число в регистре r3 окажется «Выше» (Higher), «>», чем число в регистре r4, - нужно поменять местами содержимое регистров r3, r4.

Первое, что мы должны сделать – это сравнить числа в регистрах. Для этого проще всего использовать команду сравнения, которая не требует суффикса «S» установки флагов и выполняет их модификацию автоматически:

```
CMP r3, r4      ; (r3)-(r4)
```

Решим задачу сначала с использованием команды условной передачи управления. Так как мы имеем дело с числами без знака, то условию «Выше» соответствует суффикс «HI». При выполнении этого условия мы должны поменять местами содержимое регистров r3, r4, а в противном случае, оставить все «как есть». Соответствующий фрагмент программы будет иметь вид:

```
; Вариант №1
        BHI  Change  ; Перейти на метку Change при условии HI
        B    Next    ; В противном случае - продолжить программу

                     ; Поменять местами содержимое
                     ; в регистрах r3, r4
Change MOV r0, r3    ; Временно сохранить содержимое r3 в r0
        MOV r3, r4   ;(r3) ← (r4)
        MOV r4, r0   ;(r4) ← (r0)
Next                 ; … Продолжение программы
```

Этот фрагмент программы хоть и решает задачу, но показывает, как «не надо» программировать на Ассемблере! В нем используется сначала команда условного перехода по нужному нам условию, а затем – команда безусловной передачи управления на метку Next продолжения программы. Если этого не сделать, то блок кода, в котором делается обмен содержимого регистров r3, r4, помеченный меткой Change, будет выполняться при любом результате сравнения исходных чисел.

Как правильнее решать подобные задачи? Мы должны определить так называемое *оппозитное* условие (*противоположное*). В нашем случае противоположным к условию HI (>) «Выше» будет условие LS (≤) «Ниже или то же самое». Если оппозитное условие будет истинным, нам ничего не нужно делать – просто перейти к следующему фрагменту программы Next. В противном случае – нужно выполнить требуемый обмен данными:

```
; Вариант №2
        BLS Next     ; Перейти на метку Next при условии LS
                     ; по оппозитному условию!
                     ; Поменять местами содержимое
                     ; в регистрах r3, r4
                     ; если условие не выполняется
Change MOV r0, r3    ; Временно сохранить содержимое r3 в r0
        MOV r3, r4   ; (r3) ← (r4)
        MOV r4, r0   ; (r4) ← (r0)
Next                 ; …  Продолжение программы
```

Покажем, как та же задача решается с использованием механизма условного выполнения команд. Сразу после операции сравнения можно воспользоваться любыми командами с нужными нам кодами условного выполнения:

```
; Вариант №3
        CMP r3, r4          ; (r3)-(r4)
                            ; Поменять местами содержимое
                            ; в регистрах r3, r4
                            ; если условие HI истинно

Change MOVHI r0, r3         ; Сохранить содержимое r3 в r0
        MOVHI r3, r4        ; (r3) ← (r4)
        MOVHI r4, r0        ; (r4) ← (r0)
Next                        ; … Продолжение программы
```

Как мы уже отмечали, транслятор автоматически добавит перед блоком условного выполнения (в данном случае из трех команд), команду IT (ее назначение см. в 12.4). Если условие HI «истинно», все три команды будут последовательно выполнены, в противном случае – пропущены. Согласитесь, что последнее решение существенно проще и понятнее программисту. Причем, как видите, можно вообще обойтись без команд условной или безусловной передачи управления.

Возможности условного выполнения команд являются важным преимуществом процессоров фирмы ARM. Они могут кардинально изменить стиль программирования на Ассемблере, существенно уменьшив число безусловных и условных переходов в программе. Это значит, что конвейер команд практически не будет перезагружаться, что повысит общую производительность системы.

1. Нужна ли метка Change в двух последних вариантах? Можно ли ее опустить?
2. Как будет работать программа в варианте № 3, если условие HI не выполняется?
3. Что нужно изменить в приведенных выше версиях кода, если сравниваться будут два 32-разрядных числа со знаком в дополнительном коде?

1. Не нужна. Ее можно опустить, но лучше этого не делать! Она выполняет роль дополнительного комментария, структурирующего код.
2. Все три команды в блоке, помеченном меткой Change, будут выполнены как команды NOP (Нет операции).
3. Нужно использовать суффикс условного выполнения «Строго больше» GT для первого и третьего вариантов и суффикс оппозитного условия «Меньше или равно» LE – для второго варианта.

Пример 2. В регистре r3 находится число со знаком в дополнительном коде. Получить абсолютное значение этого числа r3 = ABS(r3).

Попытка найти команду получения абсолютного значения числа (целого или с фиксированной точкой) в системе команд процессора не приведет к успеху. Такая команда есть только для обработки чисел в формате с плавающей точкой VABS.F32. Как же быть? Обратите внимание на то, что в зависимости от состояния флага N в регистре статуса программы-приложения формируются два кода условного выполнения команд MI

(«минус» – результат отрицательный) и PL («плюс» – результат положительный). Следовательно, можно проанализировать знак исходного операнда в регистре r3, и если он окажется отрицательным, то получить абсолютное значение числа путем вычитания из нуля этого числа: 0-(r3). В противном случае, значение в регистре r3 можно не менять.

Для того, чтобы установить знак исходного операнда, можно воспользоваться разными командами. Главное, чтобы они выставляли флаги результатов операции. Приведем несколько возможных вариантов:

```
; Вариант №1.
; Сравнить содержимое регистра r3 с непосредственным
; операндом нулем #0.
CMP r3, #0      ;(r3)-#0. Выставить флаги

; Вариант №2.
; Сложить содержимое регистра r3 с нулем.
; Не забыть указать опцию «S» установки флагов
ADDS r3, #0     ;[(r3)+#0]→ r3. Установить флаги

; Вариант №3.
; Переслать содержимое регистра r3 в свободный регистр,
; например, r0, указать опцию «S» установки флагов
MOVS r0, r3     ; (r0)←(r3). Установить флаги
```

Как видите, вариантов много – выбирайте любой, удобный для Вас. Обратите внимание на то, что в отличие от многих других процессоров, команда межрегистровой пересылки данных MOV может вырабатывать флаги (если суффикс «S» указан).

Итак, флаги установлены. Прямое условие – MI («минус»), а оппозитное – PL («плюс»). Варианты решения:

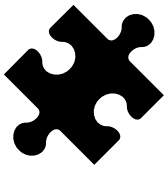
Вариант 1:

```
; Если число положительное, оставить все «как есть»
     BPL  Next
; В противном случае - изменить знак числа на обратный
     MOV r0, #0      ; r0 ← #0
     SUB r3, r0, r3 ; r3 ←[(r0)-(r3)]
                     ; Здесь использована трех-операндная команда
                     ; r3 – регистр-приемник результата операции
                     ; r0 – 1-й регистр-источник
                     ; r3 – 2-й регистр-источник
Next                 ; …  Продолжение программы
```

Вариант 2

```
; Изменить знак числа на обратный, если число отрицательное
MOVMI r0, #0         ; r0 ← #0
SUBMI r3, r0, r3     ; r3 ←[(r0)-(r3)]
```

Очевидно, что и в этом примере имеет смысл использовать команды условного выполнения вместо команд условной передачи управления. Программа получается более наглядной и «читабельной».



Для решения подобных задач очень удобно использовать команду так называемого *реверсивного вычитания* RSB {Rd}, Rn, #0 – Reverse Subtract. Если второй операнд в команде – непосредственные данные, то отпадает необходимость использования дополнительной команды загрузки регистра константой (#0). При прямом вычитании из значения первого операнда-источника вычитается значение второго операнда-источника, а при реверсивном вычитании: из значения второго операнда-источника вычитается значение первого-операнда источника. Результат можно сохранить в любом регистре приемнике Rd или, если он опущен, то в первом регистре-источнике. Применительно к нашей задаче – достаточно будет всего одной команды:

```
; Вариант 3
; Изменить знак числа на обратный, если число отрицательное
RSBMI r3, r3, #0    ; r3 ← [#0-(r3)]
```

1. Мнемокод любой команды процессора формируется по одним и тем же общим принципам. Что за корень имеет команда RSBMI? Какой у нее префикс и что он обозначает? Какой у нее суффикс, каково его значение?
2. Можно ли использовать не трех- а двух-операндную команду `RSBMI r3, #0`?
3. Как Вы думаете, почему команда `SUBMI r3, #0, r3` является недопустимой?

1. Корень **SB** – от **SUB** (Вычитание), сокращенный на одну букву. Префикс R от англ. Reverse (Реверсивный порядок операции): из вычитаемого вычитается уменьшаемое. Префикс специфицирует тип операции, в данном случае – реверсивное вычитание. Суффикс – MI («Минус») – код условного выполнения команды.
2. Да.
3. Дело в том, что только второй операнд в команде является универсальным и на его месте можно указать не только имя регистра ЦПУ, но и непосредственный операнд – константу (см. 7.11).

7.10 Унифицированный синтаксис команд процессора и сопроцессора. Основные принципы кодирования команд

При создании системы команд Cortex-M специалисты фирмы ARM старались унифицировать синтаксис команд, чтобы облегчить программистам восприятие мнемокодов команд, их запоминание и использование. Для большинства команд, обрабатываемых в ЦПУ и в сопроцессоре, применяется *унифицированный синтаксис команд*, основанный на следующих положениях:

1. В командах обработки данных в качестве операндов могут использоваться в основном регистры окружения ЦПУ или сопроцессора.
2. Второй операнд-источник является особым: универсальным операндом, допускающим, в частности, задание в качестве операндов - констант.
3. Для доступа к регистрам специального назначения (управления, статуса и т.д.) используются только специальные команды.

4. Для доступа к памяти используются только специальные команды, имеющие особый синтаксис кодирования адреса нужной ячейки памяти.
5. Для доступа к битам и битовым полям в регистрах ЦПУ используются специальные команды.
6. Большинство команд выполняют операции над словами (32-разрядными операндами) с получением 32-разрядного результата. Некоторые исключения:
    a. Поддерживаются параллельные арифметические операции над одноименными байтами, расположенными в регистрах-источниках;
    b. Поддерживаются параллельные арифметические операции над полусловами, расположенными в регистрах-источниках;
    c. Поддерживаются операции над 32-разрядными словами с получением длинного 64-битного результата (умножение, умножение с накоплением).
7. В поле мнемокода команды должен быть указан *основной мнемокод операции*, который может быть расширен слева *опциональным префиксом* и справа – *опциональным суффиксом* (возможно, несколькими суффиксами). Между префиксом и основным мнемокодом операции, между основным мнемокодом и суффиксом *разделителей нет*. Полученный таким образом *составной мнемокод* полностью соответствует назначению команды и операциям, ею выполняемым.
8. *Префикс* специфицирует конкретный тип операции или является указателем места ее выполнения (в ЦПУ или сопроцессоре). Так, для всех команд, выполняемых в сопроцессоре поддержки вычислений с плавающей точкой FPU, используется префикс «V». Для команд, выполняемых в ЦПУ, он может отсутствовать или уточнять тип операции, например, «S» (Signed) – операция с числами со знаком в дополнительном коде; «U» (Unsigned) – операция с числами без знака.
9. *Суффикс* – специфицирует одну из отличительных особенностей операции. Суффиксов может быть несколько, в соответствии с несколькими особенностями конкретной операции. Примеры в табл. 7.6.

Таблица 7.6 Суффикс специфики операции

Суффиксы, отражающие специфику операции	**Значение**
S	**Set – Установить флаги в регистре статуса APSR**
8	**Исходные операнды – байты (8-разрядные). Параллельное выполнение 4-байтовых операций в АЛУ**
16	**Исходные операнды – полуслова (16-разрядные). Параллельное выполнение двух операций с полусловами.**
L	**Long – Получение длинного 64-битного результата умножения или умножения с накоплением.**
W	**Использовать младшее полуслово в исходном 32-разрядном регистре**
T	**Использовать старшее полуслово в исходном 32-разрядном регистре**
{cond}	**Любой из суффиксов условного выполнения команды**

10. Поле мнемокода операции отделяется от поля операндов по крайней мере одним пробелом.
11. В поле операндов могут быть указаны в порядке слева направо: операнд-приемник результата операции Rd или Sd (от англ. Destination – назначение), первый операнд-источник Rn или Sn, второй операнд-источник Rm или Sm. Имя операнда-приемника всегда указывается первым.
12. В качестве разделителя между операндами используется запятая.
13. В зависимости от числа операндов все команды можно разделить на четырех-операндные, трех-операндные, двух-операндные, одно-операндные и не имеющие операндов (например, NOP).

14. В двух-операндной команде функцию регистра-приемника Rs (от англ. Source – источник) данных выполняет первый регистр-источник Rn, имя которого ближе к мнемокоду операции. При этом исходные данные, расположенные в нем, автоматически замещаются результатом операции.
15. В одно-операндной команде единственный регистр, указанный в поле операндов, выполняет функцию операнда-источника данных. Регистр-приемник не указывается, а подразумевается «по умолчанию». Обычно это счетчик команд PC в командах косвенной передачи управления.
16. Если в мнемокоде команды используется суффикс «L» (Long), то приемником операции является «длинный регистр», состоящий из двух 32-разрядных регистров общего назначения RdLo (младшее слово результата) и RdHi (старшее слово). В поле операндов они должны быть указаны именно в таком порядке: RdLo, RdHi. В этом случае поле операндов должно содержать имена 4-х регистров.
17. В командах умножения с накоплением в поле операндов может присутствовать четыре регистра: регистр-приемник Rd, регистр-аккумулятор Ra, регистр первого исходного операнда Rn (множителя), регистр второго операнда Rm (множителя). В операциях умножения с накоплением, кроме самой последней операции, имена регистров Rd и Ra должны совпадать.

Четырех-операндные команды используются в операциях цифровой обработки сигналов (DSP). Их назначение и синтаксис подробно рассматриваются в соответствующих главах книги. Представим здесь только синтаксис большинства, наиболее часто применяемых команд.

7.10.1 Синтаксис трех-операндных команд

Трех-операндная команда								
Поле мнемокода команды				**Поле операндов**				
<Префикс>	**Основной мнемокод операции**	**<Суффикс/ы>**	**␣**	**Имя регистра-приемника Rd или Sd**	**,**	**Имя 1-го регистра-источника Rn или Sn**	**,**	**Имя 2-го регистра-источника Rm или Sm**

Примеры:

```
ADD r0, r3, r4      ; Сложить содержимое регистров r3 и r4.
                    ; Сохранить сумму в регистре r0.
ADDS r0, r3, r4     ; То же самое, но, дополнительно выставить
                    ; флаги результата операции в регистре
                    ; статуса приложения APSR.
SMULBT r0, r3, r4   ; Выполнить знаковое умножение –
                    ; префикс «S» (Signed) младшего (B - Bottom)
                    ; полуслова первого операнда в r3
                    ; на старшее полуслово (T - Top) второго
                    ; операнда в r4 и сохранить результат в
                    ; регистре r0.
VMUL.F32 s0, s3, s4
; Выполнить умножение в модуле плавающей точки FPU
; (префикс V) чисел с плавающей точкой однократной точности
; (суффикс «.F32»)в регистрах источниках s3, s4
; с сохранением произведения в регистре s0.
```

Как видите, корневая часть мнемокода операции определяет ее основное содержание: ADD – от англ. addition – сложение, MUL – от англ. multiplication – умножение. Префикс определяет главную специфику команды. Так, в третьем примере он указывает на то, что операция умножения будет знаковой – над операндами со знаком в дополнительном коде. В последнем примере указывает на то, что операция будет выполнена в сопроцессоре над операндами в формате с плавающей точкой. Для большинства команд, выполняемых в операционном блоке ЦПУ, префикс отсутствует, как в первых двух командах.

Суффикс подчеркивает особенности в выполнении команды. Один из важнейших суффиксов «S» (вторая команда) заставляет процессор выставить по результату текущей операции флаги результата операции и сохранить их в регистре статуса приложения APSR. Суффикс «.F32» в последней команде специфицирует формат чисел с плавающей точкой, которые будут обрабатываться – 32-разрядные числа однократной точности. Хотя сопроцессор Cortex-M4F и не поддерживает арифметические операции над числами с плавающей точкой двойной точности (64-разрядными), он поддерживает операции загрузки и сохранения таких чисел. Поэтому суффикс «.F32» указывается для определенности.

Приведем еще один пример:

```
SMULL r0, r1, r7, r8
```

Это четырех-операндная команда по синтаксису (в поле операндов перечислены имена четырех регистров) по сути является трех-операндной, в которой содержимое первого регистра-источника r7 умножается (корень мнемокода MUL) на содержимое второго регистра-источника r8 и длинное (суффикс L –Long) 64-битное произведение сохраняется в паре регистров r0, r1 (в r0 – младшее слово RdLo, в r1 – старшее RdHi). Префикс «S» свидетельствует о том, что операция умножения выполняется над числами со знаком в дополнительном коде.

7.10.2 Синтаксис двух-операндных команд

Двух-операндная команда							
Поле мнемокода команды				**Поле операндов**			
<Префикс>	**Основной мнемокод операции**	**<Суффикс/ы>**	**␣**	**Имя 1-го регистра-источника и одновременно регистра-приемника Rn или Sn**	**,**	**Имя 2-го регистра-источника Rm или Sm**	

Примеры:

```
SUB r3, r4          ; Вычесть из содержимого регистра r3
                    ; содержимое регистра r4.
                    ; Сохранить результат в регистре r3.
SUBS r3, r4         ; То же, но по результату операции вычитания
                    ; выставить флаги в регистре статуса APSR.
UDIV r5, r6         ; Разделить 32-разрядное число без знака,
                    ; префикс «U» (Unsigned), в регистре r5 на
                    ; 32-разрядне число без знака в регистре r6.
                    ; Сохранить частное в регистре r5.
VDIV.F32 s1, s2     ; Разделить число с плавающей точкой
                    ;(префикс «V»), однократной точности
                    ;(суффикс «.F32»)в регистре s1 на число
                    ; с плавающей точкой в регистре s2.
                    ; Сохранить частное в регистре s1.
```

В этих примерах корневая часть мнемокода операции строго соответствует ее сути: SUB – от англ. Subtraction – Вычитание, DIV – от англ. Divide – Деление. Значения префиксов и суффиксов отражены в комментариях к командам. Обратите внимание на то, что во всех двух-операндных командах первый регистр-источник одновременно является и регистром-приемником данных. Следовательно, с одной стороны – эти *команды могут быть более короткими* (более компактными), чем трех-операндные (входить в набор 16-разрядных команд Thumb), а с другой – будут замещать («затирать») данные в первом регистре-источнике. Если исходные данные больше не нужны – это не критично. В противном случае лучше использовать трех-операндную команду, которая и сохранит исходные данные и направит результат операции по месту назначения.

1) Как Вы думаете, допустима ли такая команда:
   `ADD r1, r1, r2?`
2) А такая:
   `ADD r1, r1, r1?`
3) Почему последнюю команду лучше заменить на команду:
   `ADD r1, r1?`

1) Да. Никаких ограничений на использование регистров ЦПУ или сопроцессора в поле операндов нет.
   Эта команда выполнит операцию [(r1)+(r2)] → r1.
2) Тоже допустима. Это просто удвоение содержимого регистра r1.
3) Выполняет те же действия, что и предыдущая команда. Более проста по синтаксису.

Транслятор с Ассемблера, являясь «интеллектуальным», всегда пытается сгенерировать более компактный код (16-разрядного набора Thumb), если это возможно. Не удивляйтесь, если вместо написанной Вами трех-операдной команды Вы увидите полностью соответствующую ей двух-операндную. Если все три регистра в поле операндов разные, то транслятор сгенерирует нужную Вам трех-операндную команду.

7.10.3 Синтаксис одно-операндных команд

Одно-операндная команда					
Поле мнемокода команды				**Поле операндов**	
<Префикс>	**Основной мнемокод операции**	**<Суффикс/ы>**	**␣**	**Имя регистра-источника Rm**	

Примеры:

```
BX   Rm     ; Косвенный переход по адресу в регистре Rm
            ; (Rm)→ PC
BLX  Rm     ; Косвенный переход по адресу в регистре Rm
            ; с предварительным сохранением адреса возврата
            ; в регистре связи LR
```

Представленные выше команды являются командами этого класса условно. Ведь на самом деле регистр приемник в них присутствует, но «по умолчанию» – это счетчик команд PC.

Итак, большая часть команд процессора двух- и трех-операндные. Даже такие команды, как получение абсолютного значения числа в формате с плавающей точкой или вычисление квадратного корня из него формально двух-операндные: результат операции над одним операндом в регистре-источнике можно сохранить либо в том же самом регистре, либо в любом другом Sd:

```
VABS.F32 Sd, Sm     ; |(Sm)|→ Sd
VSQRT.F32 Sd, Sm    ; [√(Sm)] → Sd
```

Имеется и небольшая группа команд, не имеющих никаких операндов. Это специальные операции, одна из которых – команда NOP (No Operation – Нет операции).

Процессоры Cortex-M имеют разветвленную систему команд, которая, с одной стороны, достаточно сложна для изучения, а с другой – настолько мощная, что позволяет решать практически любые задачи обработки данных, причем в любых форматах (бита, байта, полуслова, слова). *Совет*: старайтесь понять логику разработчиков системы команд, используемую при их кодировании. Тогда система команд не покажется «чрезмерно сложной», а наоборот – логически стройной и понятной. Мы постараемся помочь Вам в этом.

7.11 Гибкий второй операнд-источник. Попутные операции в сдвиговом регистре

7.11.1 Гибкий второй операнд

Для ряда арифметических, логических команд и команд пересылки, которые выполняются с использованием АЛУ процессора, возможности второго операнда, поступающего во второй порт АЛУ, существенно расширены. Если первый операнд может извлекаться только из регистра общего назначения, то второй – из трех возможных источников:

1) Из *кода операции*, когда *непосредственный операнд* **#Const** содержится в формате самой команды;
2) Из *регистра общего назначения* ЦПУ без какой-либо обработки – **Rm**;
3) Из *регистра* общего назначения ЦПУ с предварительной обработкой в специальном *сдвиговом регистре процессора* – **Rm, shift**.

Такой операнд, в отличие от обычного операнда-регистра, имеет в системе команд специальное обозначение **Op2** и называется «*гибким вторым операндом-источником*» (Flexible Second Operand). Синтаксис команд процессора с гибким вторым операндом продемонстрируем на примере команды сложения:

```
ADD {Rd}, Rn, Op2
```

По-прежнему, сразу после мнемокода операции опционально (не обязательно) указывается имя регистра-приемника Rd (заключено в фигурные скобки), затем имя первого регистра-источника Rn, а после него – имя второго гибкого операнда Op2. При использовании во втором операнде содержимого регистра Rm, синтаксис Op2 выглядит так: Rm {,shift}. Фигурные скобки указывают на то, что предварительный сдвиг содержимого регистра Rm – опция. Вы можете выполнить *предварительную попутную обработку* содержимого регистра Rm с использованием специальной операции в сдвиговом регистре shift (Сдвиг), либо отказаться от нее. В этом случае данные из второго регистра-источника Rm будут поступать в порт АЛУ напрямую, минуя сдвиговый регистр, в обход него, через *байпас* – (bypass).

7.11.2 Второй операнд – константа

Для того, чтобы в качестве второго операнда команды задать *константу*, необходимо в программе на языке Ассемблер указать ее числовое значение после символа «#», являющегося признаком *непосредственного операнда*, то есть операнда, значение которого будет автоматически помещено транслятором с Ассемблера в формат команды и будет доступно процессору на стадии выполнения по содержимому кода команды.

Теоретически Вы можете задать любую 32-разрядную константу. Практически это невозможно, так как в этом случае для ее представления пришлось бы выделить отдельное 32-разрядное слово в команде и формат команды превысил бы 32 бита:

Теоретически возможный формат команды с непосредственной адресацией	
Первое слово	**Код операции**
Второе слово	**32-разрядные данные (константа)**

Разработчики системы команд пошли по другому пути: они ограничились некоторыми, наиболее употребительными на практике константами, упаковав (зашифровав) их так, чтобы они могли разместиться в 32-разрядном формате команды вместе с другими полями, в том числе, с полем кода операции.

Мы не будем подробно изучать механизмы кодирования непосредственных данных в процессорах ARM, которые могут отличаться от команды к команде. Для любознательных читателей рассмотрим два наиболее часто используемых варианта *компрессии*:

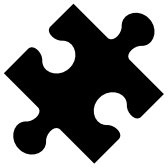
1) Задается любая 8-разрядная константа (от 0 до 255) и число бит сдвига этой константы влево для размещения внутри 32-разрядного слова. Максимальное число разрядов сдвига для размещения байта в старших разрядах слова – 24. Для его представления достаточно пяти бит. Следовательно, вместе с байтовой константой потребуется выделить в формате команды всего 8+5=13 бит.

2) Задаются две 4-битовые константы X и Y (X и Y – любые шестнадцатеричные цифры), для хранения которых достаточно одного байта и двухбитовый признак типа кодирования. В этом случае в формате команды для представления константы потребуется только 8+2=10 бит:

Зашифрованная константа		**Константа (Расшифрованное значение, поступающее на обработку в АЛУ в качестве операнда)**
4 бита: 0xX	**4 бита: 0xY**	
Тип кодирования (2 бита):		
0	**0**	**0x00XY00XY**
0	**1**	**0xXY00XY00**
1	**0**	**0xXYXYXYXY**

В некоторых командах диапазон представления констант шире (см. приложение 1). Для обычного пользователя нужно знать следующее:

- Транслятор с Ассемблера *автоматически* выбирает оптимальный способ кодирования указанной Вами константы;
- Если размещение константы в формате самой команды невозможно, транслятор сгенерирует соответствующее предупреждение. Что делать в этом случае? Вариантов несколько: 1) попробовать задать близкую по значению, но другую константу; 2) разместить константу в регистре общего назначения Rm и в качестве гибкого операнда Op2 использовать регистр Rm с дополнительным сдвигом; 3) воспользоваться для загрузки константы (любой 32-разрядной) специальной псевдокомандой Ассемблера, которая предварительно размещает константу в ПЗУ в виде так называемого «литерального пула», а затем обеспечивает ее загрузку с использованием адресации по содержимому счетчика команд (относительной адресации) – подробно в 9.1.



Транслятор с языка Ассемблера является *высоко интеллектуальным продуктом*, который за Вас может автоматически принять более эффективное решение. Это справедливо и в том случае, если Вы задали константу в неудачном формате, например, записали команду сравнения с константой в виде:

```
CMP Rn, #0xFFFFFFFE ; Сравнить содержимое регистра Rn с (-2)
```

Транслятор автоматически заменит эту команду *командой сравнения с отрицательным значением операнда* (Compare negative):

```
CMN Rn, #0x2        ; После отрицания получим те же (-2)
```

На самом деле, конечно, никакой инверсии знака не делается – эта команда вместо вычитания просто выполняет сложение с указанным операндом.

7.11.3 Аппаратная поддержка операций предварительной обработки данных в сдвиговом регистре

Выделим из программной модели процессора, изображенной на рис. 7.1, часть, касающуюся обработки данных в АЛУ – рис. 7.6. Одним из операндов обязательно будет регистр общего назначения процессора. Второй (универсальный операнд) Op2 может быть: 1) константой (непосредственным операндом), считываемой из регистра команд; 2) регистром общего назначения; 3) регистром общего назначения, содержимое которого проходит предварительную обработку в сдвиговом регистре.

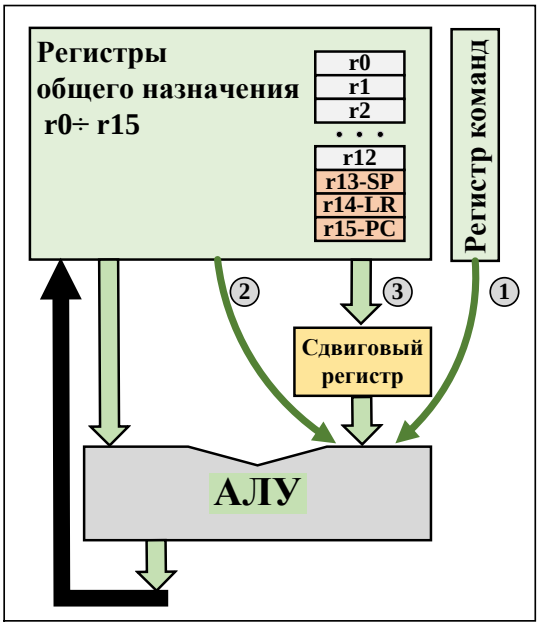


Рис. 7.6 Аппаратная поддержка попутной обработки операнда в кольцевом сдвиговом регистре

Такая архитектура операционного блока ЦПУ характерна для высокопроизводительных сигнальных процессоров, которые обрабатывают числа в формате с фиксированной точкой.

Действительно, если один из операндов арифметической операции имеет формат (m.n), а второй формат (i.q), причем числа разрядов дробной части n и q в них разные, то приходится приводить операнды к одинаковому формату, сдвигая один из них влево или вправо так, чтобы двоичные точки в операндах располагались одинаково. Только в этом случае арифметические операции над числами с фиксированной точкой будут корректными.

Имея в архитектуре процессора кольцевой сдвиговый регистр, операцию приведения чисел к одинаковому формату можно сделать попутной, выполняемой еще до загрузки операнда в порт АЛУ. Это дает существенный выигрыш в производительности процессора. В Cortex-M попутные операции *не увеличивают* время выполнения основной арифметической или логической операции, которая, как и подавляющее большинство операций в RISC-процессорах, выполняется всего за один цик.

Опция попутной обработки позволяет также масштабировать один из операндов, участвующих в операции, на коэффициент 2n, где n – число разрядов сдвига (имеется в виду арифметический сдвиг, ниже - подробнее).

7.11.4 Типы попутных операций

В таблице 7.7 приведены возможные варианты попутных операций *сдвига* (shift), которые могут быть выполнены с данными, расположенными в регистре-источнике Rm:

Таблица 7.7 Варианты попутных операций сдвига

Shift	Выполняемые действия
ASR #n	**Арифметический сдвиг вправо на n бит, 1 ≤ n ≤ 32**
LSR #n	**Логический сдвиг вправо на n бит, 1 ≤ n ≤ 32**
LSL #n	**Логический сдвиг влево на n бит, 1 ≤ n ≤ 31**
ROR #n	**Циклический сдвиг вправо на n бит, 1 ≤ n ≤ 31**
RRX	**Циклический сдвиг вправо на один бит с расширением, – через флаг C (Carry)**
–	**Если опция <shift> опущена, то сдвиг не выполняется. Эквивалентно LSL #0**

Графическая иллюстрация возможных вариантов сдвига представлена на рис. 7.7. Рассмотрим каждый из типов сдвига более подробно:

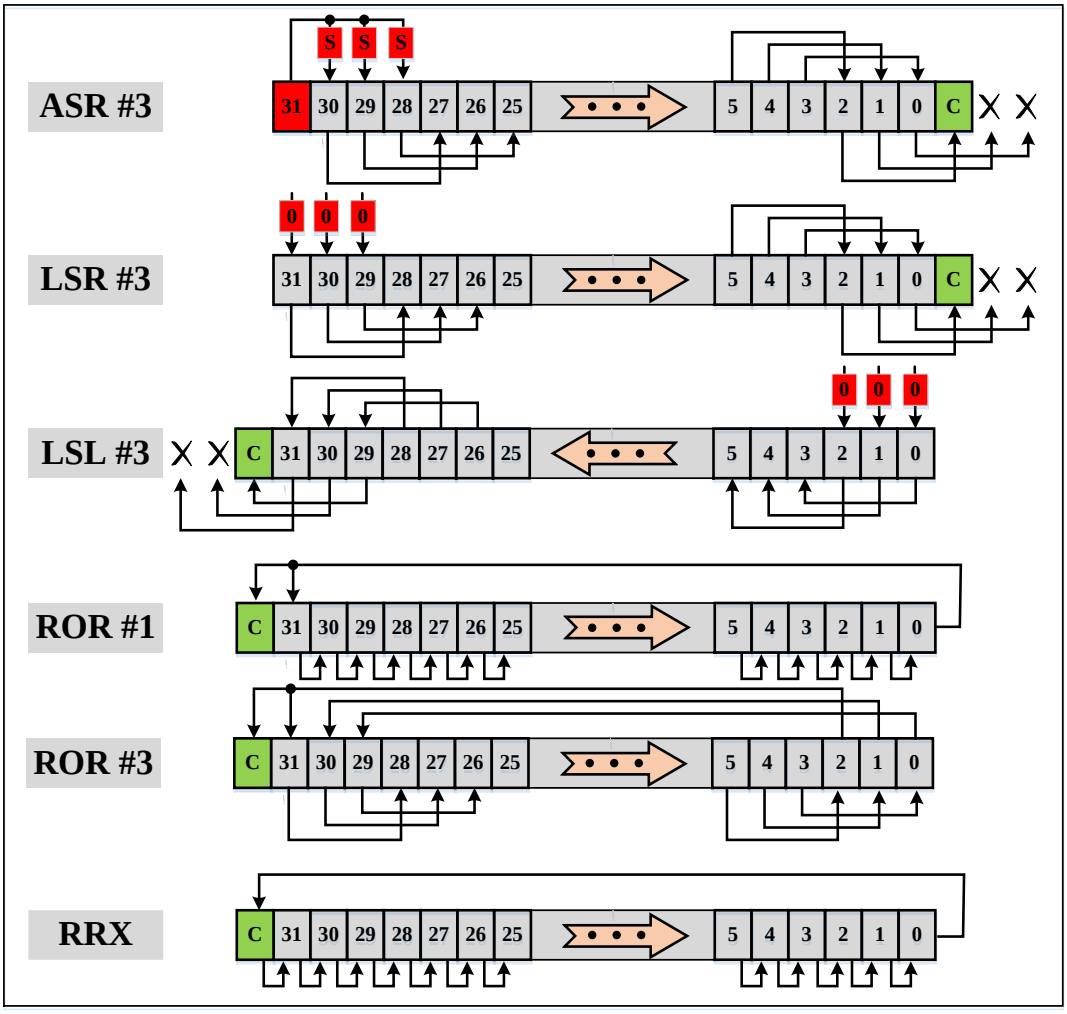


Рис. 7.7 Варианты попутного сдвига второго операнда Op2

7.11.4.1 Арифметический сдвиг вправо – ASR #n

Арифметический сдвиг вправо (**A**rithmetic **S**hift **R**ight – **ASR**) на n бит перемещает все 32 бита исходного операнда в регистре Rm вправо на n бит. При этом вытесняемые вправо за пределы разрядной сетки биты сначала попадают в битовый флаг C (Carry), а при дальнейшем сдвиге теряются. Дополнительно выполняется копирование исходного 31-го бита операнда (знакового разряда числа в дополнительном коде) в n старших бит результата. Операция «*автоматического размножения*» *знакового разряда* исходного числа выделена на рис. 7.7 красным цветом. Такой сдвиг эквивалентен последовательному *делению знакового операнда* на 2 при каждом очередном сдвиге, отсюда и название операции – «арифметический сдвиг вправо».

Если сдвиг выполняется на n разрядов, то исходное число со знаком в регистре Rm делится на 2^n. Если попутная операция используется для «выравнивания» положения двоичной точки в вещественном числе, то самые *младшие разряды* (обычно разряды дробной части числа) *теряются*. При операциях с вещественными числами – это, обычно, незначимые разряды.

Если операция, в которой используется предварительный сдвиг второго операнда, содержит в мнемокоде суффикс «S» (Set – установить флаги), то флаг переполнения C

(Carry) в регистре статуса APSR будет обновляться последним вытесненным за пределы разрядной сетки битом исходного регистра bit[n-1]. Например, при n=1 – нулевым битом. В противном случае, флаг C не будет обновляться. Если выполняемая операция сама меняет содержимое флага C (например, в результате переполнения), то флаг C установится в соответствие с результатом основной операцией.

Примеры:

```
MOV r1, r2, ASR #3        ; Загрузить в регистр r1 содержимое
                          ; регистра r2, предварительно сдвинутое
                          ; арифметически на 3 разряда вправо
                          ;(знаковое деление на 2^3=8)
ADDS r0, r1, r2, ASR #1   ; Предварительно разделить операнд
                          ; в регистре r2 на два и сложить
                          ; с операндом в регистре r1.
                          ; Сохранить сумму в регистре r0
                          ; Установить флаги результата
                          ; Арифметической операции
```

Каждой операции попутного сдвига, которая рассматривается в этой главе, соответствует аналогичная логическая операция сдвига содержимого регистра-источника на заданное число разрядов (глава 11). Их проще использовать, если необходим последовательный анализ битовых переменных в исходном регистре по состоянию флага переноса C. Коды условного выполнения: CS – «флаг C установлен», CC – «флаг C сброшен»).

7.11.4.2 Логический сдвиг вправо LSR #n

Операция *логического сдвига вправо* (**L**ogical **S**hift **R**ight –**LSR**) отличается от операции арифметического сдвига вправо тем, что вместо «авто-размножения знакового разряда числа» выполняется операция *авто-заполнения* освобождающихся при сдвиге старших разрядов нулями (выделена красным цветом на рис. 7.7). При этом все разряды исходного числа по-прежнему сдвигаются вправо на заданное число разрядов. Вытесняемые вправо разряды сначала попадают во флаг переноса, а при дальнейшем сдвиге – теряются.

Операцию логического сдвига вправо часто используют для *деления числа без знака* (целого или вещественного) на 2^n. При этом младшие разряды частного, не помещающиеся в 32-разрядную сетку АЛУ, отбрасываются.

Если в команде, использующей попутную операцию LSR #n, имеется суффикс формирования флагов результатов операции «S» (Set), то флаг C (Carry) окажется обновлен последним, вытесненным вправо битом bit[n-1]. Например, при сдвиге на n=3 разряда, битом bit[2], что и показано на рис. 7.7. Если основная команда сама воздействует на флаг C, то его содержимое будет определяться результатом выполнения основной команды.

7.11.4.3 Логический сдвиг влево LSL #n

Логический сдвиг влево (**L**ogical **S**hift **L**eft - **LSL**) на заданное число разрядов n перемещает содержимое исходного регистра Rm влево на требуемое число разрядов с автоматическим заполнением освобождающихся справа разрядов нулями 0 и «вытеснением» старших разрядов во флаг переноса C с последующей их потерей при дальнейшем сдвиге.

Эта операция эквивалентна *умножению исходного числа без знака* на 2^n, при условии, что переполнения не возникло. Если в результате такой операции переполнение

имеет место (теряются один или более ненулевых старших разрядов, вытесненных за пределы разрядной сетки), то (внимание!) – нет флага, свидетельствующего об этом.

Если в команде, использующей попутную операцию LSL #n, имеется суффикс установки флагов результатов операции «S» (Set), то флаг C (Carry) окажется обновлен последним, вытесненным влево битом bit[32-n]. Например, при сдвиге на n=3 разряда, – битом bit[29], что и показано на рис. 7.7. Если основная операция работает с флагом C, то она и определяет состояние этого флага.

7.11.4.4 Циклический сдвиг вправо ROR #n

Циклический сдвиг вправо (**RO**tate **R**ight – **ROR**) на заданное число разрядов n выполняется над содержимым регистра Rm так, как будто бы это *кольцевой сдвиговый 32-разрядный регистр*. Вытесняемый за пределы разрядной сетки вправо разряд попадает в самый старший разряд регистра и одновременно во флаг переноса C.

На рис. 7.7 эта операция вначале иллюстрируется сдвигом только на один разряд ROR #1, а затем на 3 разряда ROR #3. Если в команде, использующей попутную операцию ROR #n, имеется суффикс установки флагов результатов операции «S» (Set), то флаг C (Carry) окажется обновлен последним, вытесненным вправо битом bit[n-1]. В первом случае – битом bit[0], а во втором – битом bit[2]. Если основная операция воздействует на флаг C, то она и определяет состояние этого флага.

7.11.4.5 Циклический сдвиг вправо с расширением RRX

Циклический сдвиг вправо с расширением (**R**otate **R**ight with e**X**tend- **RRX**) представляет собой циклический сдвиг вправо на один разряд в 33-разрядном регистре, расширенном на один разряд флагом переноса. Регистр Rm и флаг переноса C (Carry) рассматриваются *как один 33-разрядный кольцевой сдвиговый регистр*.

Все разряды такого расширенного регистра сдвигаются вправо на один разряд, в результате чего старший разряд обновляется содержимым флага C, а флаг C – содержимым младшего бита исходного операнда bit[0]. Он установится в регистре статуса APSR только в том случае, если команда, использующая попутную операцию RRX, имеет суффикс «S» (Set) в своем мнемокоде.

Операция RRX используется обычно для сдвига вправо на нужное число разрядов длинных чисел (64-разрядных и более). При этом «выдвинутый» из старшего слова младший бит сначала попадает в флаг C, а при втором сдвиге автоматически «вдвигается» в младшее слово.

1) Чему должно быть равно n в операции попутного арифметического сдвига вправо ASR #n, чтобы все биты исходного операнда заполнились знаковым разрядом числа?
2) При каком значении n в операции попутного логического сдвига вправо LSR #n все биты исходного операнда окажутся обнуленными?
3) В регистре Rm находится число 2^{32}=0x80000000. Выполнятся операция попутного логического сдвига влево на число разрядов n≥1. Чему окажется равен операнд, загружаемый во второй порт АЛУ?
4) В регистре r3 находится битовая маска, в которой все разряды младшего байта установлены в 1 – 0x000000FF. Требуется установить в 1 все разряды старшего байта числа в регистре r2 с помощью операции «Логического ИЛИ» (ORR) со вторым гибким операндом, попутно преобразующим исходную маску в маску 0xFF000000.
5) Маска в регистре r3 содержит нули во всех 16-и младших разрядах (в младшем полуслове) – 0xFFFF0000. Какую попутную операцию нужно выполнить, чтобы получить маску с нулями в старшем полуслове 0x0000FFFF? Как обнулить с ее

помощью старшее слово регистра r2, используя команду «Логического И» (AND) содержимого r2 с маской?

1) n ≥ 31.
2) При n=32.
3) Нулю. Более того, никакого флага о возникшем переполнении (выходе за пределы разрядной сетки) сформировано не будет.
4) Нужная операция преобразования маски может быть выполнена с использованием циклического сдвига вправо на 8 разрядов: ROR #8. Если результат операции «маскирования» должен быть сохранен в том же регистре r2, то соответствующая команда будет двух-операндной, в которой первый регистр-источник будет и регистром-приемником:

```
ORR r2, r3, ROR #8  ; Логическое побитовое ИЛИ: (r2)+ 0xFF000000
```

5) Достаточно выполнить попутный циклический сдвиг на 16 разрядов ROR #16, чтобы исходная маска с нулями в младшем полуслове стала маской с нулями в старшем полуслове: 0xFFFF0000 → 0x0000FFFF. Операция побитового логического умножения на такую маску очистит все разряды старшего полуслова в регистре r2.

```
AND r2, r3, ROR #16 ; Логическое побитовое И: (r2)·(0x0000FFFF)
```

В данном примере вместо попутной операции циклического сдвига вправо можно использовать и операцию логического сдвига влево на те же 16 разрядов LSR #16.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Козаченко В.Ф. Микроконтроллеры: руководство по применению 16-разрядных микроконтроллеров Intel MCS-196/296 во встроенных системах управления. – М.: ЭКОМ, 1997. – 688 с.
3) ARM DDI 0403C_errata_v3, ARMv7-M Architecture Reference Manual, Arm Limited, 2010.
4) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
5) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
6) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

8 ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ КРОСС-СРЕДСТВ

Оглавление

8.1 Общие положения

Программное обеспечение микропроцессорных систем сегодня разрабатывается исключительно на персональных компьютерах с использованием *кросс-средств*, то есть таких программных продуктов, которые устанавливаются и работают на универсальных компьютерах, создавая исполняемые файлы, предназначенные для загрузки и выполнения в конкретном процессоре или микроконтроллере (*целевом устройстве*), используемом разработчиком в своей системе.

Современные кросс-средства являются *интегрированными системами*, которые включают в себя все необходимые пользователю *инструменты*, объединенные общей *графической средой*, позволяющей практически полностью совместить процесс разработки ПО с процессом отладки. При этом пользователю предоставляются несколько возможностей отладки: 1) в *симуляторах* – программно-логических моделях целевого процессора, работающих исключительно на ПК и не требующих какой-либо дополнительной аппаратуры; 2) в созданных производителями микропроцессоров и микроконтроллеров *оценочных платах*, подключенных к ПК по универсальному интерфейсу USB; 3) в *конечных устройствах*, аппаратная часть которых создана разработчиками конкретных приложений (контроллерах) и имеющих необходимые интерфейсы для подключения отладчиков (обычно JTAG). При этом обеспечивается *единый интерфейс* программы *отладчика* с любым из трех устройств, в котором выполняется программа, и *унификация функций отладки* – облегчается взаимодействие специалистов-аппаратчиков и программистов при создании любых микропроцессорных систем.

Современные интегрированные среды поддерживают разработку как минимум на Ассемблере и одном из языков высокого уровня (обычно C/C++). При этом допускается комбинированная разработка, когда часть программных модулей создается на языке высокого уровня, а часть – на Ассемблере. Допускаются и ассемблерные вставки в программы, написанные на языке высокого уровня. Интересно, что по мере развития микропроцессорной техники произошло обогащение языка Ассемблер средствами, которые раньше применялись исключительно в языках высокого уровня. Это касается как команд собственно процессора (например, условного выполнения операций), так и специальных директив Ассемблера, псевдокоманд, которые интегрируют в Ассемблер целый рад инструментов, присущих ранее только языкам высокого уровня. Таким образом, синтаксис современного языка Ассемблер существенно расширен. Это не только синтаксис команд процессора, но и ряд вспомогательных операций – *директив Ассемблера*, которые позволяют не только создавать исполнительный код программы, но и секционировать программу, управлять размещением переменных и констант в памяти, выполнять удобную *символьную отладку* программного продукта.

В этой главе мы выделим важнейшие принципы, которым по умолчанию следуют все разработчики программных продуктов и которые сегодня стали «стандартом де-факто». Основное внимание уделим технологии разработки ПО на Ассемблере. В последующих главах книги используем эти знания для практического изучения эффективных способов решения типовых задач управления на основе системы команд процессоров Cortex-M3/M4/M4F, попутно изучая ее возможности. В качестве основы для дальнейшего изложения выберем интегрированную среду разработки Keil µVision.

8.2 Интегрированная среда разработки Keil µVision

В этой книге будет использоваться одна из наиболее распространенных *интегрированных сред разработки и отладки программного обеспечения* для процессоров ARM (ARM Development Kit) – Keil µVision IDE. Причины выбора этой среды следующие:

1) Полный набор инструментов (специальных программ), интегрированных в единую графическую среду и предназначенных для создания и отладки программного обеспечения широкого класса микропроцессорных устройств на базе процессоров фирмы ARM, в том числе на базе процессорных ядер Cortex-M3/M4/M4F.
2) Наличие общедоступной версии (упрощенной) с некоторыми ограничениями по функциональности, предназначенной для работы на любых компьютерах под операционной системой Windows.
3) Ограничение состоит только в том, что проекты, генерирующие код объемом больше чем 32 К байта нельзя скомпилировать и скомпоновать. Ограничение не является сколько-нибудь существенным для студентов, изучающих микропроцессорную технику, и специалистов, повышающих свою квалификацию. Объема кода 32 К байта достаточно для отладки даже весьма сложных проектов.
4) Ядро среды содержит все *средства разработки* (development tools) и дополнительные *пакеты программного обеспечения* (software packs) в стандарте программного интерфейса для микроконтроллеров ARM-Cortex (Cortex Microcontroller Software Interface standard -CMSIS), включая *промежуточные библиотеки для поддержки микроконтроллеров большинства фирм-производителей*, которые могут использоваться в качестве «целевых» устройств (target devices), то есть устройств, на которых планируется прикладная разработка.
5) Фирма Keil давно специализируется на разработке программного обеспечения для процессоров ARM, а в 2005 г. была куплена фирмой ARM и стала по сути ее подразделением. Это гарантия того, что в программном обеспечении фирмы Keil тщательно учтены все имеющиеся аппаратные и программные возможности ARM-процессоров.
6) Среда содержит встроенный симулятор, максимально полно отражающий особенности архитектуры и системы команд процессорных ядер ARM, удобный как для новичков, так и для специалистов в области микропроцессорной техники.

Для того, чтобы скачать открытую среду разработки через Internet, нужно зайти на сайт фирмы ARM. Фирма Keil является ее подразделением (http://www.keil.com/company/). Перейти в раздел продуктов MDK Microcontroller Development Kit» (http://www2.keil.com/mdk5). Зарегистрироваться. После регистрации инсталляция продукта выполняется автоматически.

В конце инсталляции могут устанавливаться библиотеки для конкретного устройства (микроконтроллера одного из известных производителей). Вы можете указать тип этого микроконтроллера или название оценочной платы, используемой в Вашей разработке. Для них могут быть установлены также примеры практического использования.

После того, как Вы установите комплект разработчика µVision5 (на момент написания книги версия 5 – последняя) на рабочем столе компьютера появится пиктограмма пакета, позволяющая быстро вызывать его с рабочего стола компьютера.

На панели инструментов пакета имеется значок установщика, который позволяет, при необходимости, обновить пакет до последней версии.

В среде µVision имеется встроенное руководство пользователя, доступ к которому возможен через пункт меню Help – Справка. Мы будем раскрывать богатые возможности среды µVision по мере необходимости. Каждый раз соответствующую информацию будет предварять приведенная выше пиктограмма пакета.

Встроенная справочная система обширна и позволяет быстро найти практически любую нужную информацию, в том числе по любой команде процессора (для этого, правда, нужно знать мнемокод команды) или директиве Ассемблера. Информация выдается на языке разработчика – английском.

8.3 Поддерживаемые языки программирования

Интегрированная среда разработки Keil µVision предназначена для создания и отладки программного обеспечения на языке Ассемблер фирмы ARM, имеющем название ARM ASM, а также на языках высокого уровня C или C++ (далее C/C++), наиболее востребованных прикладными программистами.

Преимущество использования языка Ассемблер фирмы разработчика процессоров очевидно: он максимально полно отражает все особенности архитектуры процессоров ARM и их набора команд. Относительно выбора в качестве языка высокого уровня, языка C/C++, можно сказать следующее: на сегодня это самый распространенный язык высокого уровня, используемый большинством высококвалифицированных программистов во всем мире, особенно теми, которым часто приходится программировать на «низком уровне», управляя большим количеством встроенных в микроконтроллеры периферийных устройств. Пожалуй, это самый удобный язык высокого уровня, создающий предельно компактный и быстро исполняемый код, который может конкурировать с кодом, созданным на Ассемблере. Он отличается универсальностью, удобством доступа к памяти и регистрам периферийных устройств. Именно поэтому он широко используется при разработке ПО встраиваемых в оборудование цифровых систем управления.

Особенность любых интегрированных сред разработки – возможность создания программного обеспечения из модулей, написанных как на языке высокого уровня C/C++, так и на Ассемблере. При этом Ассемблер используется там, где нужна предельная оптимизация кода по объему памяти и быстродействию. Его можно использовать также для создания эффективных вставок в программах на языке C/C++.

Среда µVision поддерживает не только написание *кода программы на двух языках, но и отладку программы в исходных кодах* (как на C/C++, так и на Ассемблере).

В книге мы будем изучать систему команд процессоров Cortex-M и приемы эффективного решения типовых задач управления оборудованием на Ассемблере. Такие возможности языка ARM ASM, как директивы секционирования программ, резервирования памяти, инициализации констант, псевдокоманды Ассемблера, упрощающие и облегчающие программирование, макросредства и др., будут рассмотрены по ходу изложения основного материала. Эти сведения всегда будет предварять пиктограмма Ассемблера фирмы ARM.

Специальных глав по языку Ассемблера в книге нет.

8.4 Состав пакета Keil µVision. Этапы разработки ПО

8.4.1 Компоненты интегрированной среды Keil µVision

Интегрированная среда разработки имеет в своем составе все инструменты, необходимые для эффективной работы программиста:

1) *Текстовый редактор* с полными возможностями качественного ввода и редактирования текста программы и дополнительными возможностями цветового выделения синтаксических конструкций языков Ассемблер (мнемокодов команд, директив Ассемблера, регистров и т.д.) и C/C++, что обеспечивает более удобное восприятия текста программы как в процессе ее редактирования, так и в процессе отладки. Конечно, для создания программ можно использовать и любые другие, удобные для программиста текстовые редакторы, главное – они не должны включать в текст программы специальные символы форматирования. Эти символы не распознаются транслятором.
2) *Транслятор с языка Ассемблер*, воспринимающий на входе файл на Ассемблере с расширением «.s» или «.asm» и генерирующий *выходной файл в перемещаемом объектном коде* с расширением «.o» (или «.obj»), а также *файл листинга* с расширением «.lst», содержащий результаты трансляции, в том числе, с сообщениями об ошибках. Дополнительно в файл листинга может включаться таблица символов, используемых в программе.
3) *Компилятор с языка C/C++*, воспринимающий на входе файл на языке Си или Си++ с расширением «.c» или «.cpp» и генерирующий на выходе файл в перемещаемом объектном коде «.o».
4) *Компоновщик или Линкер* – программа, объединяющая несколько исходных файлов в перемещаемом объектном коде в *один файл в неперемещаемом объектном коде*, в котором относительные адреса заменены абсолютными – все взаимные ссылки программных модулей друг на друга и на библиотечные функции «разрешены» (в том смысле, что символическим именам присвоены конкретные физические адреса). Все кодовые секции программных модулей объединены в одну, как и секции данных (инициализированных и неинициализированных).
5) *Библиотекарь* – программа, которая может объединить несколько оттранслированных/ откомпилированных и отлаженных программных модулей в перемещаемом объектном коде в один файл с расширением «.lib» – *специализированную пользовательскую библиотеку,* для последующего многократного использования, но уже без отладки, как готового продукта.
6) *Симулятор* – программно-логическая модель целевого процессора, реализованная на обычном компьютере и полностью имитирующая архитектуру и систему команд целевого процессора. Симулятор не только имитирует процессор, но и некоторые из периферийных устройств – системную периферию. Он позволяет отлаживать программу исключительно на компьютере, без необходимости подключения к нему целевой платы разрабатываемой микропроцессорной системы. Это особенно важно для студентов и инженеров, самостоятельно повышающих свою квалификацию.
7) *Загрузчик* – загружает (размещает) программу в абсолютном объектном коде на выполнение либо в память симулятора, либо в память реальной процессорной платы. Это может быть *оценочная плата*, созданная производителем микроконтроллера для оценки возможности решения прикладной задачи на данном микроконтроллере, или

плата, разработанная самим конечным пользователем, при условии, что в ней реализован один или несколько необходимых для отладки интерфейсов (глава 5).

8) *Отладчик* – позволяет выполнять код программы в режиме прогона, по шагам, с точками останова, при полном контроле текущего содержимого регистров процессора и переменных в памяти. Позволяет оценить быстродействие программы в целом и отдельных ее фрагментов (подпрограмм).
9) *Утилиты преобразования формата* выходного файла в другой формат, например, для загрузки кода в программатор внешних постоянных запоминающих устройств (ПЗУ) или внешней флэш-памяти.

8.4.2 Основные этапы разработки и отладки программного обеспечения

Для всех интегрированных сред разработки порядок разработки и отладки ПО примерно одинаков и иллюстрируется на рис. 8.1.

1) *Создается новый проект*, для которого выбирается *целевой микропроцессор* или *микроконтроллер*, то есть – кристалл, на базе которого будет реализована микропроцессорная система пользователя. *Файл проекта* – это специальный файл, который содержит информацию обо всех исходных файлах, написанных пользователем/пользователями и подключенных к проекту. Имена исходных файлов и места их расположения запоминаются в файле проекта вместе с дополнительной информацией о настройках среды для работы с текущем проектом. Изделия (процессоры и микроконтроллеры) большинства ведущих мировых производителей включены в электронную базу фирмы Keil и могут быть просто выбраны из нее. Если целевой микроконтроллер отсутствует в этой базе (как в случае с отечественными производителями микроконтроллеров), достаточно выбрать из базы тип процессорного ядра (в нашем случае Cortex-M3 или M4 или M4F). В этом случае будут поддерживаться базовые средства отладки ПО в симуляторе. Для эффективной работы с оригинальными периферийными устройствами заказывайте у производителей микроконтроллеров дополнения к стандартному пакету Keil, расширяющие его возможности.
2) Для выбранного целевого процессора, если желаете, можно включить в состав проекта *стартовый файл инициализации процессора* StartUp, который любезно предоставляется фирмой Keil в качестве помощи разработчикам. Как уже упоминалось ранее, он представляет собой некий «шаблон» – основу для создания Вашего собственного стартового файла. Если Вы уже работали с процессором и создали свой собственный стартовый файл – включите его в проект. Основное назначение этого файла – инициализация процессора и встроенных периферийных устройств.
3) *Конфигурируется интегрированная среда* – для нее устанавливаются нужные параметры. В большинстве случаев для начала работы достаточно «параметров по умолчанию». Только иногда потребуется коррекция (будем информировать читателя, в каких конкретно случаях).
4) С использованием встроенного текстового редактора пакета создаются *исходные файлы проекта* либо на языке Ассемблер, либо на C/C++. Это может быть всего один файл, например, MyProg.s на Ассемблере, или несколько, в том числе на языке высокого уровня C/C++. Как минимум, в состав любого проекта должен входить стартовый файл SturtUp.s и файл с программой пользователя (например, MyProg.s).

5) Каждый из исходных файлов запускается на *трансляцию*, если он написан на Ассемблере, или на *компиляцию*, если – на C/C++. Как Вы увидите, для этого достаточно одного щелчка кнопкой мыши на панели команд интегрированной среды. Если в исходных файлах будут обнаружены синтаксические ошибки, Вы получите соответствующие уведомления. Более того, Вы сможете сразу перейти в нужную строку исходного файла, чтобы исправить ошибку. Процесс повторной трансляции или компиляции выполняется для каждого отдельного файла до тех пор, пока ошибок в исходных текстах программных модулей не будет. В результате создается группа файлов проекта в перемещаемом объектном коде.

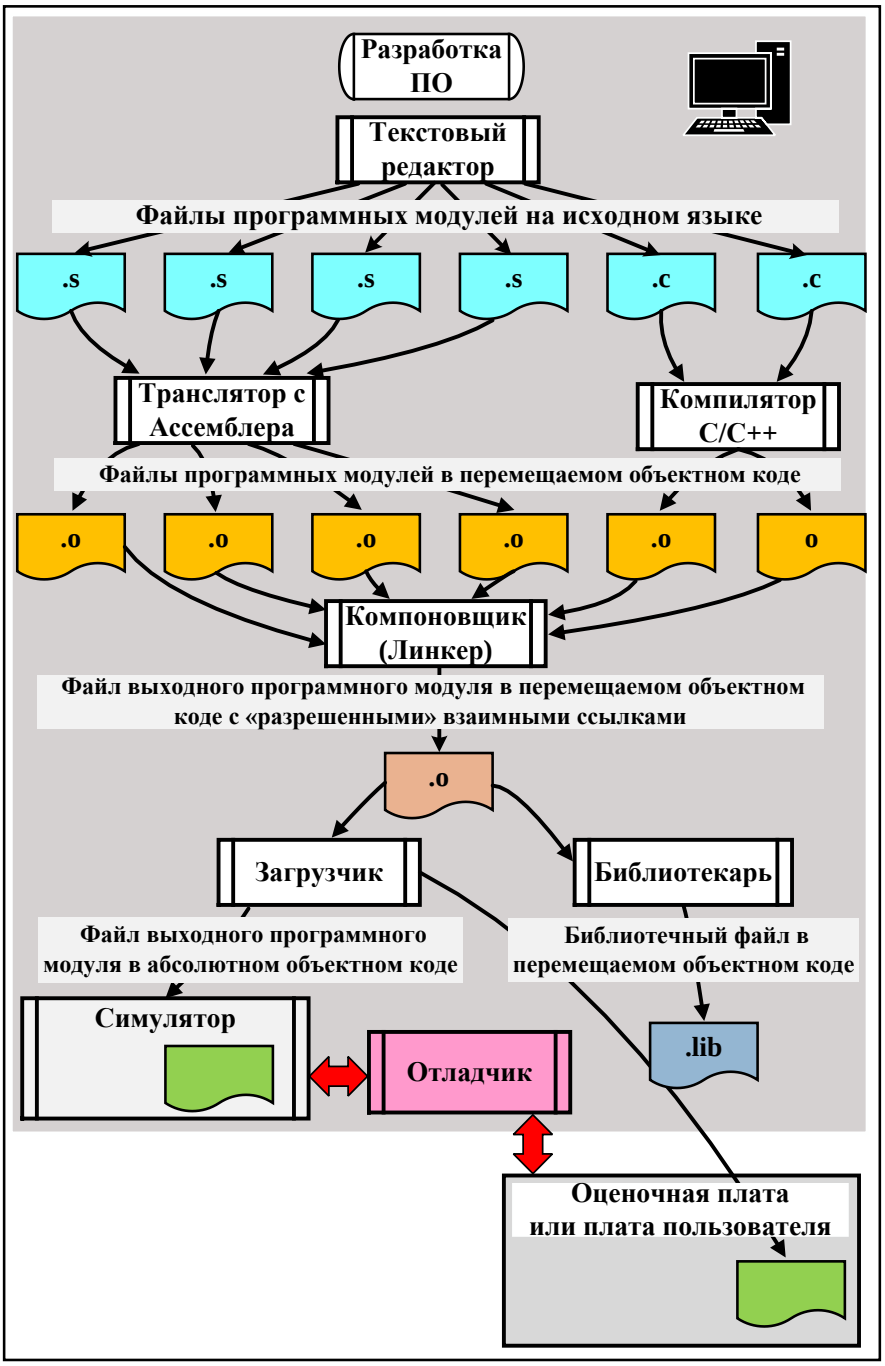
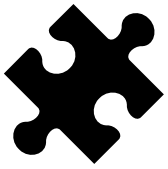


Рис. 8.1 Порядок разработки программного обеспечения в интегрированной среде

6) Выполняется *конфигурирование памяти целевого устройства*. Параметры интегрированной среды настраиваются в соответствии с реальными объемами кодовой памяти и памяти данных целевого микропроцессора, чтобы программа-компоновщик «знала», какими ресурсами она располагает – куда можно разместить код программы, куда – данные. При отладке в симуляторе можно обойтись установками «по умолчанию». Их достаточно для отладки простых учебных проектов.
7) Выполняется *компоновка* проекта, то есть все кодовые секции исходных файлов объединяются в одну кодовую секцию, которая размещается по адресам доступной памяти программ. Все инициализированные секции объединяются в одну общую секцию инициализированных данных, а секции неинициализированных данных – в одну общую секцию неинициализированных данных. Она размещается по адресам доступной памяти данных.
8) Проект, допускающий отладку в *симуляторе процессора*, загружается в симулятор и выполняется под управлением *отладчика*. В этом случае периферийные устройства, которые имеются в составе целевого микроконтроллера, не должны использоваться. Так отлаживаются все модули, выполняющие преимущественно обработку данных.
9) Полностью отлаженные модули в перемещаемом объектном коде могут быть объединены в *пользовательскую библиотеку* как ряд наиболее употребительных *функций*. В последующем Вы сможете автоматически включать код любой библиотечной функции в выходной файл проекта. Для этого достаточно только подключить файл библиотеки к проекту. Нужная функция будет автоматически извлечена из библиотеки компоновщиком.
10) *Последовательная трансляция* программных модулей позволяет постепенно создавать и отлаживать проекты. В начале работы в состав проекта может входить лишь небольшое число файлов, которое далее, по мере отладки проекта, будет увеличиваться. Разные исходные файлы могут создаваться и отлаживаться разными программистами. Только после полноценной отладки отдельного программного модуля программистом он передается *менеджеру всего проекта* и подключается к проекту. Менеджер проекта (обычно руководитель отдела программного обеспечения) отвечает за сборку всех исходных файлов и файлов библиотек в один общий проект и его отладку.
11) Теперь наступает самый ответственный момент – исполнительный код проекта может быть загружен в реальную микропроцессорную систему (оценочную плату или плату разработанного Вами контроллера) и выполнен под управлением отладчика. Проследите, чтобы необходимые для этого интерфейсы имелись на вашей плате (глава 5).
12) Обратите внимание на то, что любой останов программы вызовет останов в работе и встроенных в контроллер периферийных устройств. Поэтому, такая отладка будет полноценной отладкой в реальном времени только в режиме выполнения программы целиком (в режиме прогона).
13) Если в процессе отладки ПО в изделии возникли ошибки, то выполняется процесс редактирования (модификации) отдельных программных модулей.
14) На самом последнем этапе целевой контроллер встраивается в изделие (источник питания, преобразователь и т.д.) и его аппаратная и программная части тестируются в условиях реальной эксплуатации.

Представленная на рис. 8.1. схема разработки программного обеспечения может несколько отличаться для разных сред разработки. Так, компоновщик, а также программа-библиотекарь могут получать на входе несколько файлов в перемещаемом объектном коде «.o» и генерировать в первом случае сразу выходной файл для загрузки в микроконтроллер, а во втором случае – выходной библиотечный файл.

В среде Keil выходной файл имеет особый формат и расширение «.axf». Он не удобен для исследования в текстовых редакторах и используется только загрузчиком при вызове отладчика.



1. В результате работы компоновщика создается выходной файл в абсолютном объектном коде, который при вызове отладчика загружается либо в программу-симулятор, либо в целевую плату (оценочную или контроллер конечного пользователя) на выполнение под управлением программы - отладчика.
2. Так как оценочная плата обычно имеет интерфейс JTAG для подключения отладчика, а в обычных компьютерах такого интерфейса нет, то, кроме оценочной платы, придется приобрести *переходник* со стандартного интерфейса компьютера в интерфейс JTAG – переходник USB/JTAG и установить на компьютер драйвер этого переходника.

Современная интегрированная среда разработки программ для микроконтроллеров IDE, такая как µVision, содержит все описанные выше программные модули в одном пакете, объединенные удобной графической средой, когда вызов нужной функции выполняется одним «щелчком» мыши. Среда содержит и удобные средства настройки всех входящих в нее программ (опций). Более того, в одной среде Вы можете разрабатывать программное обеспечение для микроконтроллеров разных производителей, с разными периферийными устройствами, пользуясь как Ассемблером, так и языком высокого уровня C/C++, используя функции из большого количества специализированных библиотек, созданных либо разработчиками процессоров, либо Вами самими, либо другими пользователями, если они сделали их общедоступными.

Часто весь необходимый программисту комплекс программных средств (утилит) называется одним словом toolchain (набор инструментов) или даже compiler (компилятор) несмотря на то, что он фактически состоит из множества независимых программ, объединенных графической средой в единую систему.

8.5 Окно интегрированной среды µVision5

Общий вид окна интегрированной среды µVision5 представлен на рис. 8.2. Мы будем постепенно знакомиться с назначением входящих в пакет инструментов. Отметим, что Вы можете задавать нужные команды из расположенного в верхней строке меню, открывая конкретные меню: **File** (Файл) – для работы с файлами; **Edit** (Редактор) – для работы с исходными файлами в текстовом редакторе; **View** (Просмотр) – для вывода на экран окна с нужной в данный момент информацией или для временного сокрытия этого окна; **Project** (Проект) – для работы с проектами; **Flash** (Флэш) – для программирования флэш-памяти целевого устройства; **Debug** (Отладчик) – для управления процессом отладки программы, в том числе в симуляторе; **Peripherals** (Периферия) – для работы со встроенной периферией; **Tools** (Средства) – для работы с дополнительными инструментами пакета; **SVCS** (Software Version Control System) – для доступа к системе контроля версий; **Window** (Окна) – для

управления окнами среды, их перемещения, упорядочивания и др.; **Help** (Помощь) – для получения справки по возможностям пакета, Ассемблеру, системе команд и др. При щелчке кнопкой мыши соответствующее меню раскрывается, и появляется возможность выбора нужной команды.

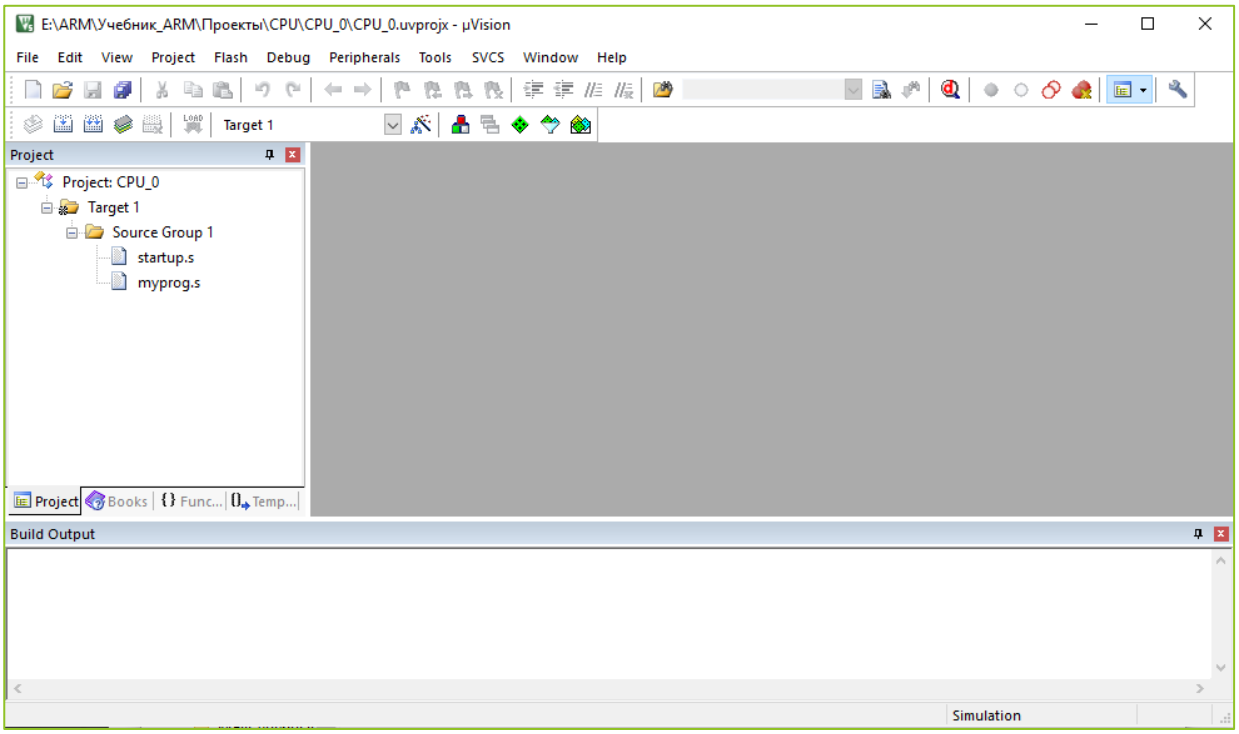


Рис. 8.2 Общий вид окна интегрированной среды µVision

Две расположенные ниже *панели инструментов* содержат ряд наиболее употребительных команд в виде пиктограмм. Так, в первой из них Вы наверняка узнали пиктограммы «Создать файл», «Открыть файл», «Сохранить», а также ряд пиктограмм, которые являются общепринятыми во всех текстовых редакторах «Вырезать», «Скопировать», «Вставить» и др. Некоторые пиктограммы в этой строке будут широко использоваться при отладке, например, для установки и снятия точек останова.

Во второй панели инструментов расположены команды, которые управляют процессом трансляции (компиляции) и сборки файлов проекта.

На всем остальном поле могут располагаться окна с различной информацией (программами в исходных кодах, содержимым нужных при отладке областей памяти и т.д.). Пока Вы видите два из них – *окно проекта* Project и окно результата *создания (построения) выходного файла* Build Output. В последнее окно будут выводиться результаты трансляции файлов Вашего проекта и компоновки. Здесь же отображаются сообщения об ошибках и предупреждения.

В самой верхней строке всегда присутствует имя файла текущего проекта, которое имеет расширение «.uvprojx» с указанием полного пути доступа к проекту. Этот файл имеет специальный формат. Он содержит информацию обо всех программных модулях проекта, целевом процессоре, опциях (настройках) среды и так называемых *параметрах окружения* среды. Это означает, что при повторном открытии проекта для продолжения работы с ним, Вы увидите на экране те же окна, которые Вы открывали ранее, в самом последнем сеансе работы. *Авто-возвращение* к последней используемой конфигурации среды существенно экономит время программиста на настройку среды при отладке ПО.

В пакете µVision используются общепринятые в компьютерах технологии работы с окнами и командами, как в виде так называемых «выпадающих» меню, так и в виде панелей инструментов с пиктограммами наиболее важных команд. Для выполнения нужной команды достаточно просто «щелкнуть мышью» на соответствующей пиктограмме. Команда будет немедленно выполнена.

8.6 Назначение псевдокоманд – директив Ассемблера

Ассемблер – это язык мнемокодов команд процессора. Программа состоит из последовательности команд, каждая из которых записывается в отдельной строке. Однако, наверняка у Вас уже возник ряд вопросов: где будет находиться написанный код в целевом процессоре после трансляции программы на Ассемблере? Можно ли разместить в кодовой памяти константу или таблицу констант, например, значения опорных точек некоторой функции? Как выделить оперативную память под переменную или массив переменных? Как обратиться к таким переменным? Как подключить к проекту важную функцию, которую написал коллега-программист и отладил ее? Как воспользоваться приобретенной дополнительно библиотекой функций? И таких вопросов – множество.

Поэтому, кроме основной задачи по автоматическому переводу ассемблерных команд в машинные коды, язык Ассемблер предоставляет целый ряд возможностей по управлению процессом трансляции и компоновки программы с помощью так называемых *директив* или *псевдокоманд* Ассемблера. Директивы записываются в поле мнемокодов команд и могут иметь, как и команды процессора, один или несколько операндов, а также необязательные опции, специфицирующие их выполнение. Но, в отличие от команд процессора, директивы не транслируются в машинный код. Это лишь указания транслятору или компоновщику на выполнение ими нужных для программиста действий.

Как и любой язык программирования, Ассемблер имеет свой синтаксис. В его основе – синтаксис команд процессора и синтаксис псевдокоманд. Основное назначение псевдокоманд: структурирование программных модулей; резервирование памяти под переменные и константы; помощь программисту в генерации сложных команд, когда псевдокоманда более понятна и автоматически заменяется транслятором на фактически имеющуюся команду, синтаксис которой сложен. С их помощью можно выполнить условное ассемблирование программы в зависимости от требуемой в данный момент версии ПО, расширить систему команд процессора с использованием макросредств. Кроме того, Ассемблер предоставляет дополнительные возможности по автоматическому вычислению арифметических и логических выражений в любой системе счисления непосредственно на стадии трансляции программы, освобождая программиста от рутинной работы с калькулятором.

Еще раз отметим, что в этой книге используется Ассемблер ARM ASM, который может несколько отличаться от языков Ассемблер других производителей (GCC, IAR). Эти отличия касаются, главным образом, набора поддерживаемых языком псевдокоманд и их синтаксиса. Мнемокоды собственно команд процессоров ARM во всех версиях языка Ассемблер должны быть одинаковыми.

8.7 Использование символических имен в программе

К символическим именам, используемым в программе на Ассемблере, относятся:

1) *Метки определенных фрагментов кода*, на которые может быть передано управление, например, имена точек входа в подпрограммы. Значение такого имени – фактический адрес расположения команды, помеченной меткой. Метка записывается с самой первой позиции ассемблерной строки и отделяется от поля команд/директив пробелом или группой пробелов. В ассемблере ARM ASM двоеточие после имени метки (как это делается в ряде Ассемблеров других фирм) не ставится.
2) *Символические имена* (идентификаторы) *констант или переменных*, значения которых соответствуют адресу их фактического расположения в памяти МПС;
3) *Символические имена констант, значения которых только сообщаются Ассемблеру, но память под них не выделяется*.

ARM ASM Вы можете определить символическое имя константы, сообщаемой Ассемблеру с помощью директивы EQU (Эквивалентно). Эта директива подобна широко используемой в языке C/C++ директиве определения констант #define. Ее синтаксис в наиболее употребительной форме:

```
Name EQU expr
```

Здесь:

Name – символическое имя константы;

expr – арифметическое выражение, результатом вычисления которого может быть 32-разрядное целое число, 32-разрядный абсолютный адрес или величина смещения (offset) адреса относительно текущего значения в регистре-указателе (register-relative value) или в счетчике команд (PC-relative value);

Примеры:

```
a    EQU  2    ; Определить константу a (a=2)
b    EQU  0x34 ; Определить константу b (b=0x34=52)
c    EQU  a+b  ; Определить константу c=a+b=54
;…
MOV r0, #a     ; Загрузить константу a в регистр r0
```

Как только константа определена, ее можно использовать в любом месте программы в качестве непосредственного операнда, как показано выше на примере загрузки непосредственными данными регистра ЦПУ r0.

Символические имена никогда не включаются в машинный код программы, но позволяют повысить ее «читабельность», упростить и облегчить дальнейшее сопровождение и модификацию. Поэтому, для обозначения переменных рекомендуется использовать символы, которые имеют смысл для разработчика, например, Speed (скорость), а для обозначения названий подпрограмм (точек входа в них) – символические имена, отражающие выполняемые подпрограммой действия, например, CALC_y1 (вычисление булевой функции y1). Они будут дополнительным комментарием к программе. При выборе имен следует соблюдать следующие правила:

- Имя внутри текущего модуля должно быть уникальным. Двойное определение одинаковых имен не допускается;
- Имя может быть записано буквами в верхнем или нижнем регистре, или и в том, и в другом;
- Ассемблер чувствителен к значению регистра, поэтому переменные «Var» и «var» с точки зрения транслятора являются различными;

- Рекомендуется использование составных имен, разделителем в которых является символ подчеркивания, например, Var_low (младшее слово переменной Var);
- По умолчанию нет никаких ограничений по длине имени, но слишком длинные имена нежелательны;
- Первый символ в имени обязательно должен быть буквой, далее можно использовать цифры, например, a_1;
- Нельзя использовать зарезервированные в Ассемблере символы (мнемокоды команд, директив ассемблера и т.д.);
- Точка «.» может использоваться в именах, в том числе в смысле разделителя составного имени, например, a.high (старшая часть переменной «a»);
- Для выделения *важнейших,* с точки зрения разработчика, имен, допускается заключать их в своеобразные скобки с помощью вертикальных линий «|» слева и справа от имени, например, | .text |. При этом транслятор игнорирует такие «скобки» и заносит в свою таблицу символов просто имя «.text».

Использование символических имен обеспечивает *символьную отладку* программы, когда пользователь обращается к нужным областям памяти в процессе тестирования программы не по их фактическим адресам, а по именам переменных, которые в них содержатся.

8.8 Секционирование программных модулей

Любой проект обычно состоит из *нескольких программных модулей.* Каждый программный модуль может содержать различную информацию: код программы; данные, расположенные в кодовой памяти, например, табличные значения функций; переменные, расположенные в оперативной памяти; специальную область памяти для временного хранения данных, в том числе – область стека. Разные по назначению области памяти программных модулей называются *секциями*. В программном модуле может быть любое число секций.

8.8.1 Директива объявления секции

ARM
ASM

Прежде, чем использовать секцию в программном модуле, ее необходимо объявить. Для этой цели предназначена специальная директива Ассемблера AREA (Область, Секция), имеющая следующий синтаксис:

```
AREA Section_Name {,type } {,attr } …
```

В поле операндов директивы AREA указывается имя объявляемой секции Section_Name, через запятую – ее тип type и далее необязательные опции, атрибуты attr, также перечисляемые через запятую.

Существуют всего два типа секций:

- CODE – кодовая, которая может быть расположена *только в памяти программ*;
- DATA – данных, которая может быть расположена *только в памяти данных*.

Таким образом, *секции* – это поименованные программистом и *полностью независимые друг от друга* области кодовой памяти (тип CODE) или памяти данных (тип DATA). Ими может манипулировать программа-компоновщик, объединяя однотипные секции (с одним и тем же именем) в одну общую секцию с тем же именем. Более того,

компоновщик размещает все секции, имеющие тип CODE, в кодовой памяти, а все секции, имеющие тип DATA – в памяти данных. Естественно, что для этого ему требуется информация о фактической карте памяти используемого микроконтроллера, которую обязательно должен сообщить среде разработки программист. Эта информация будет использоваться и для контроля заполнения памяти программ и памяти данных по мере разработки проекта (включении в его состав новых программных модулей по мере их написания и отладки). В случае переполнения имеющегося в микроконтроллере объема памяти будут выдано сообщение об ошибке.

Все команды процессора или директивы Ассемблера, написанные после директивы AREA, будут принадлежностью текущей объявленной секции до тех пор, пока не будет объявлена новая секция. Никакого маркера конца текущей секции нет. Как только объявляется новая секция, старая секция заканчивается.

В кодовых секциях может располагаться код программы, константы и таблицы констант. В секциях данных могут располагаться только переменные. Для их резервирования с указанием имен переменных и размера используются специальные директивы Ассемблера (см. ниже).

В отличие от языков программирования высокого уровня, где объявления переменных могут чередоваться с командами (операторами языка), в ассемблере требуется четкое разделение этих областей программы – инструкции могут находиться только в кодовых секциях, а описания переменных – в секциях данных. С точки зрения языка Ассемблер, секция представляет собой однородную область памяти, в которой может находиться только однотипная информация: коды команд или переменные (в том числе – стек).

В директиве объявления секции AREA можно использовать дополнительные атрибуты, являющиеся необязательными опциями:

- ***readonly*** (только для чтения) – секция, предназначенная исключительно для чтения данных (это атрибут кодовой секции CODE по умолчанию);
- ***readwrite*** (для чтения и записи) – это атрибут секции данных по умолчанию;
- ***noinit*** (без инициализации) – используется для секции данных типа DATA и означает, что область памяти не инициализируется или инициализируется нулем (0). Эта секция может содержать только директивы резервирования памяти без ее инициализации или директивы резервирования и одновременного обнуления памяти;
- ***align=n*** со значением n от 0 до 31. Опция выравнивания адреса начала секции по модулю 2^n. Показывает, по какому адресу секция должна быть выровнена в памяти. Так, при n равном 0, 1, 2 выравнивание будет выполняться соответственно: по байту (нет выравнивания), по полуслову (двум байтам), по слову (четырем байтам). Фактически n – это число младших бит начального адреса секции, которые должны быть нулевыми. Выравнивание означает пропуск определенного числа байт в памяти до начального адреса расположения секции.

Один из часто используемых атрибутов – атрибут выравнивания ***align=n***. Если он отсутствует, то по умолчанию секции выравниваются по границе 32-разрядного слова – по 4-м байтам. Фактический список атрибутов директивы AREA значительно шире. С ним можно ознакомиться, обратившись в справочную систему пакета µVision5.

Для каждой секции программного модуля в трансляторе с Ассемблера предусматривается свой собственный *счетчик адреса размещения данных*. С началом

секции этот счетчик автоматически обнуляется. По мере записи в текущей кодовой секции новых команд его содержимое автоматически инкрементируется на длину команды в байтах (на 2 или 4 байта). При резервировании и инициализации в кодовой памяти констант, значение счетчика увеличивается на длину константы. Если текущая секция является секцией данных, то с каждой командой резервирования в ней переменных значение счетчика адреса размещения автоматически увеличивается на длину зарезервированных данных в байтах.

Используйте директиву AREA для разделения любого исходного файла с программой на Ассемблере на отдельные секции. Секции с одинаковыми именами могут объявляться много раз как в одном и том же программном модуле, так и в разных. Все они будут объединены компоновщиком в одну общую секцию с тем же именем. Результирующая секция получит атрибуты первой секции, объявленной директивой AREA.

Обязательно разделяйте кодовые секции и секции данных. Помните, что в кодовой секции можно разместить не только текст программы, но и константы, таблицы констант. При большом объеме однотипных данных выделите для них специальную кодовую секцию, например, с именем MyTable. Аналогично поступайте с секциями для размещения переменных. Делите их на функциональные группы и выделяйте для них отдельные секции. Так, специальную секцию данных желательно иметь для системного стека.

1) Допустима ли программа, в которой не объявлено ни одной секции?
2) Можно ли в секции данных разместить мнемокод команды?
3) Какая область памяти данных называется не инициализированной?
4) С какой целью используется выравнивание памяти?
5) По какой границе будет выполнено выравнивание памяти для очередной кодовой секции, если для нее задан атрибут align=10?
6) Зачем может понадобиться такое выравнивание, как выше?

1) Нет. В любом программном модуле должна быть объявлена, по крайней мере, одна секция.
2) Нет.
3) Такая, в которой для переменных просто отводится определенное место в памяти, но запись начальных значений не производится.
4) Процессоры ARM поддерживают доступ и к не выровненным данным (байтам, полусловам), но расположенным в памяти данных. При этом за счет исключения «пропусков» обеспечивается более рациональное использование имеющейся памяти данных конкретного микроконтроллера. При доступе к кодам команд атрибут выравнивания align=0 (по байту) или align=1 (по полуслову) использовать нельзя. Оптимальная скорость конвейера команд обеспечивается при выравнивании памяти по слову (4-м байтам). В этом случае на конвейер команд считывается сразу 32-разрядное слово – одна команда набора ARM или сразу две команды набора Thumb.
5) По $2^{10} = 1024$ байтам, т.е. по границе 1 К байта.
6) Например, для того, чтобы зарезервировать некоторый объем имеющейся памяти в системе для последующего возможного расширения.

Для названий секций можно использовать любые имена. Однако, если это название начинается не с буквы (например, с цифры), то оно должно быть ограничено символами вертикальной черты «|», например, |1_Data|. Некоторые имена секций являются зарезервированными. Так, секция |.text| считается кодовой секцией, которая создается СИ-компилятором. В ней могут располагаться также библиотечные функции языка СИ.

8.8.2 Директива DCI размещения констант в кодовой секции

ARM ASM

В кодовой памяти вместе с кодами операций (собственно программой) могут размещаться и предварительно проинициализированные программистом данные – отдельные константы или таблицы констант, к которым по ходу выполнения программы могут следовать обращения. Это удобно, так как обращения возможны по символическим именам, а не по адресам фактического размещения констант в памяти.

Для резервирования констант в памяти программ (в секциях типа CODE) предусмотрена директива Ассемблера:

```
{label} DCI{.W} expr{,expr}
```

Она может начинаться с необязательной метки, за которой следует мнемокод директивы DCI (от англ. Define Code, Initialization – Определение кода и инициализация), а далее – одно или несколько арифметических выражений expr, отделенных друг от друга запятой. Значения этих выражений, вычисленных на стадии трансляции программы, и будут константами, размещаемыми в памяти. Сколько констант будут представлены в строке операндов директивы, столько и будет размещено в памяти (ограничений нет). Вы можете последовательно использовать несколько таких директив для размещения в кодовой памяти целой таблицы опорных точек некоторой функции. Первая константа автоматически получает символический адрес, соответствующий метке label. Данные в памяти будут выровнены по границе полуслова или полного слова, как это и должно быть в кодовой памяти. Эта директива используется для:

1) Размещения в кодовой памяти отдельных констант и таблиц констант;
2) Для создания таблиц точек входа в процедуры пользователя (начальных адресов этих процедур);
3) Для создания в кодовой памяти зон с начальными значениями переменных, которые должны быть в последующем установлены в ОЗУ (массива копий значений инициализируемых переменных в ОЗУ). В процессе выполнения стартового файла эти значения должны автоматически копироваться в нужные области ОЗУ и, тем самым, выполняться инициализация переменных.
4) Директива DCI может использоваться также для ввода кодов команд, которые не поддерживаются текущей версией языка Ассемблер. Это одно из ее преимущественных областей применения.

Что означает опция команды {.W}? Она определяет формат инициализируемых данных. Если она отсутствует, то данные представляют собой 16-разрядные полуслова, в том числе коды команд набора Thumb. Если опция указана, то резервируются данные в формате полного 32-разрядного слова (в том числе коды команд набора ARM).

Примеры:

```
Const_1   DCI 0x44             ; Резервирование константы 0x44
Tabl_y1   DCI 0,1,2,3,4,5,6,7 ; Резервирование таблицы констант
Const_2   DCI.W 0x44           ; Резервирование константы 0x44
                               ; в формате 32-разрядного слова
                               ; (принудительно)
```

8.8.3 Директива резервирования переменных в секциях данных SPACE

ARM ASM

Имеется специальная директива Ассемблера, которая предназначена для выделения места в секциях данных (типа DATA) под переменные текущего программного модуля – *резервирования* места под переменные проекта. Она выделяет необходимое место в памяти для переменной без ее предварительной инициализации. Если инициализация необходима – ее выполняет программист в своей программе.

Директива SPACE (от англ. место, пространство) имеет следующий синтаксис:

```
{label} SPACE expr
```

Метка label в начале директивы задает символическое имя резервируемой переменной, выражение expr – число байт в памяти, выделяемой для нее (вычисляется на этапе трансляции). Директива может использоваться также для выделения места под буфер данных заданной длины или под какую-либо структуру данных. В этом случае первая ячейка буфера будет иметь символический адрес label. Если выражение expr не задано, то метка просто получает значение текущего адреса в памяти данных (резервируется как бы нулевая область памяти).

Примеры:

```
N          EQU 20     ; Число слов в буфере данных
Var1       SPACE 4    ; Выделить место в ОЗУ под переменную Var1
                      ; (32-разрядное слово)
Var2       SPACE 4    ; Под переменную Var2
Buffer_32 SPACE 4*N ; Под буфер из 20-и 32-разрядных слов
HW_1       SPACE 2    ; Под полуслово HW_1
HW_2       SPACE 2    ; Под полуслово HW_2
B_1        SPACE 1    ; Под байтовую переменную B_1
B_2        SPACE 1    ; Под байтовую переменную B_1
```

Как видите, в выражениях допускается использование ранее определенных символических имен (в нашем случае – N). Вычисления значений арифметических выражений выполняется автоматически на стадии трансляции программного модуля.

Директива SPACE просто выделяет место в оперативной памяти под переменные. Это так называемые *неинициализированные* переменные. Обычно они автоматически обнуляются при сбросе процессора. Их инициализацию, если это необходимо, должна выполнить программа пользователя, что и делается в большинстве программ на Ассемблере.

Ниже мы рассмотрим группу директив, которые могут проинициализировать переменные начальными значениями не только в ПЗУ, но и в ОЗУ. Последнее возможно только в том случае, *когда в стартовый файл проекта включены специальные процедуры копирования «констант инициализации» из области ПЗУ в область ОЗУ.*

Технология такова: сначала необходимо создать «образ» инициализируемых переменных в ПЗУ, а затем, перед тем как запустить программу пользователя на выполнение, автоматически скопировать этот образ из ПЗУ в определенные области ОЗУ. Первую часть задачи выполняет транслятор, размещая копии инициализированных переменных в служебную область ПЗУ вместе со служебной информацией (начальные адреса областей памяти в ОЗУ, объемы данных). Вторая часть задачи – стартовый файл проекта. В нем выполняется анализ числа инициализируемых переменных и места их размещения. «Образ» инициализируемых переменных, находящийся в кодовой памяти, копируется в ОЗУ по нужным адресам. Только после этого управление передается программе пользователя, которая будет иметь дело уже с проинициализированными переменными. Реализация такой технологии возможна только при наличии *специального стартового файла*.

На начальной стадии освоения курса программирования на Ассемблере мы не будем использовать эту технологию, чтобы не усложнять стартовый файл и сделать его понятным для новичков. Будем применять директивы Ассемблера, описанные ниже, исключительно для размещения констант и таблиц констант в *кодовой памяти*. При необходимости инициализации переменных в ОЗУ, нужные константы будем копировать из ПЗУ в ОЗУ соответствующими командами процессора непосредственно в теле пользовательских программ.

Отметим тем не менее, что предварительная инициализация переменных в ОЗУ широко применяется в программах на языке высокого уровня C/C++. В этом случае прежде, чем передать управление основной программе пользователя main(), в стартовом файле проекта выполняются необходимые предварительные действия по копированию «образа» констант из ПЗУ в ОЗУ. Стартовый файл разрабатывается на Ассемблере квалифицированными специалистами и предоставляется программистам на языке C/C++ в готовом виде фирмами – разработчиками компиляторов.

8.8.4 Директива заполнения области памяти константой FILL

ARM ASM

Эта директива не только резервирует определенный промежуток области памяти (кодовой или данных), но и заполняет его заданной константой (выполняет инициализацию переменных):

```
{label} FILL expr {,value {,valuesize}}
```

Она эквивалентна директиве резервирования места в памяти SPACE, но с загрузкой в указанную область памяти длиной в байтах, заданной выражением expr, серии одинаковых значений value («шаблона»). При этом размер «шаблона» заполнения памяти определяется параметром valuesize (число байт в «шаблоне», 1, 2 или 4). Если опции в фигурных скобках не указаны, то заполнение памяти выполняется нулями (память принудительно обнуляется).

Размер константы заполнения («шаблона») может отсутствовать. Но, если он указан, то общий объем резервируемой памяти в байтах, заданный выражением expr, должен быть кратен размеру «шаблона» valuesize. Если размер «шаблона» не указан, Вы можете записать перед значением value лидирующий ноль, чтобы все константы стали 32-разрядными.

Примеры:

```
Var_A     FILL 4, 25, 4        ; Инициализация одного слова Var_A
                               ; значением 25
Buffer_8  FILL 16, 0xFF, 1     ; Заполнить 16 байт в памяти
                               ; константой 0xFF
Buffer_32 FILL (10*4),0xFF     ; Заполнить 10 слов в памяти
                               ; константой 0x000000FF
```

Напомним, что заполнение секции данных (DATA) константами требует стартового файла определенной структуры.

8.8.5 Директивы размещения в памяти констант в заданном формате

Директивы DCB, DCW, DCD, DCQ (от англ. Define Code – определить код) обеспечивают резервирование памяти под константы в ПЗУ и переменные в ОЗУ типа байт (суффикс «B»), полуслово (суффикс «W»), полное 32-разрядное слово (суффикс «D»), двойное 64-разрядное слово (суффикс «Q»). Они могут применяться как в кодовой секции для резервирования констант и таблиц констант, так и в секции данных для инициализации переменных. В последнем случае необходим специальный стартовый файл, поддерживающий инициализацию переменных в ОЗУ (см. выше). Директивы имеют следующий синтаксис:

```
{label} DCB expr1 {,expr2}{,expr2}…
{label} DCD{U} expr1 {,expr2}{,expr3}…
{label} DCW{U} expr1 {,expr2}{,expr3}…
{label} DCQ{U} expr1 {,expr2}{,expr3}…
```

Директива резервирования *байт* DCB обеспечивает их последовательное размещение в памяти без какого-либо выравнивания адресов, директива резервирования *полуслов* DCW выполняет автоматическое выравнивание по четным адресам, а директива резервирования *слов* DCD – автоматическое выравнивание по адресам, кратным числу 4 (дважды четным). Такое же выравнивание адресов памяти (по 4-м байтам) выполняется и при резервировании *двойных слов* директивой DCQ.

Суффикс «U» используется для указания опции отсутствия выравнивания. В этом случае данные располагаются в текущей секции подряд, без каких-либо пропусков.

В поле операндов каждой из этих директив можно указать одно или несколько выражений, значения которых будут вычислены на стадии трансляции и записаны в соответствующие ячейки памяти в качестве начальных значений переменных.

Метка label, как обычно, является символическим именем первой переменной.

Примеры:

```
; Объявить секцию данных по имени MyData
AREA    MyData, DATA, READWRITE
; Зарезервировать три переменные типа слово с инициализацией
Var_1     DCD   0xff33, 37, 0xffaa55bb
; Зарезервировать 200 байт под буфер BUF_0
BUF_0     SPACE   200    ; Все ячейки будут обнулены
; Зарезервировать еще 50 байт
; заполнив все байты константой 0x5A
; Будут проинициализированы числом 0x5A
BUF_1     FILL    50, 0x5A, 1
```

В этой книге директивы DCB, DCW, DCD будут использоваться только для резервирования констант в кодовой памяти. Это позволит использовать более простую и понятную для начинающих структуру стартового файла (глава 9).

8.9 Автоматическое вычисление выражений транслятором

ARM ASM

В ряде приведенных выше примеров использовались символические имена переменных, и транслятору с Ассемблера поручалась *роль калькулятора* для расчета нужных программисту значений. Автоматическое выполнение расчетов на стадии ассемблирования программы существенно облегчает программирование и позволяет избежать множества описок, свойственных человеку, что повышает надежность программного обеспечения.

В выражениях можно использовать не только символические имена переменных и констант, но и значения меток. В последнем случае выражения будут не *абсолютными*, а *относительными* – их окончательные значения будут определены не на этапе ассемблирования конкретного программного модуля, а на этапе компоновки проекта.

В выражениях можно использовать *операторы*, связывающие между собой числа и ранее определенные символические имена. Таблица основных операторов в Ассемблере и их аналогов в языке высокого уровня C/C++ приведена ниже. Операторы указаны в порядке приоритета их выполнения, если он не изменяется круглыми скобками. Самый высокий приоритет имеют операторы в верхней части таблицы. При равенстве приоритетов операции выполняются в порядке слева направо.

Таблица 8.1 Основные операторы языка Ассемблер и их аналоги в языке C/C++

Операторы языка Ассемблер			**Эквивалентные операторы в C/C++**
1	**Унарный минус**	**-**	**-**
2	**Умножение** **Деление нацело** **Остаток от деления**	***** **/** **:MOD:**	***** **/** **%**
3	**Сдвиг влево логический** **Сдвиг вправо логический** **Сдвиг влево циклический** **Сдвиг вправо циклический**	**:SHL:** **:SHR:** **:ROR:** **:ROL:**	<< >>
4	***Числовые операции над двумя операндами*:** **Сложение** **Вычитание** **Логическое побитовое «И»** **Логическое побитовое «ИЛИ»** **Логическое побитовое «Исключающее ИЛИ»**	 **+** **-** **:AND:** **:OR:** **:EOR:**	 **+** **-** **&** **|** **^**
5	***Операции сравнения*:** **Равно** **Больше** **Больше или равно** **Меньше** **Меньше или равно** **Не равно**	 **=** **>** **>=** **<** **<=** **/=,<>**	 **==** **>** **>=** **<** **<=** **!=**
6	***Логические (булевы) операции с двумя операндами:*** **Логическое И** **Логическое ИЛИ** **Логическое Исключающее ИЛИ**	 **:LAND:** **:LOR:** **:LEOR:**	 **&&** **||**

Обратите внимание на то, что имеются два типа логических операций:

1) Побитовые логические операции над числами (п. 4 таблицы 8.1);
2) Логические операции над двумя булевыми операндами A и B, которые могут принимать только два возможных значения {TRUE} или {FALSE} (п. 6).

Число логических операций в Ассемблере несколько шире, чем в Си, за счет циклических сдвигов влево и вправо и операции «Исключающее ИЛИ» с двумя булевыми операндами.

1) Какое значение будет загружено в регистр r0 в следующем фрагменте программы:

```
n     EQU  5
m     EQU  0xF0
MOV r0, #(2*n)
```

2) При выполнении команды `MOV r0, #(m+n)` ?
3) При выполнении команды `MOV r0, #(m :mod: n)` ?
4) При выполнении команды `MOV r0, #(m :or: n)` ?
5) При выполнении команды `MOV r0, #(n :SHL: #2)` ?
6) При выполнении команды `MOV r0, #(m > n)` ?
7) При выполнении команды `MOV r0, #(m = n)` ?

1) 10
2) 0xF5
3) 0
4) 0xF5
5) 0x14
6) 1 (True)
7) 0 (False)

8.10 Внутренняя и внешняя модульность программы

8.10.1 Внутренняя модульность

Мы уже рассматривали ранее понятие *внутренней модульности* программы, (см. 7.7), состоящей из основной программы и некоторого числа подпрограмм (функций), которые могут многократно вызываться по мере выполнения основной программы. Под модулем понимается отдельный файл с программой на Ассемблере, который может транслироваться независимо от остальных файлов. Если отдельные фрагменты программы или подпрограммы находятся внутри текущего модуля, то доступ к ним возможен без каких-либо ограничений. Достаточно, например, пометить начала фрагментов программ или подпрограмм символическими именами и передавать им управление командами B label (brach) – перехода на метку или командой BL label (branch with link) – перехода на начальный адрес подпрограммы с сохранением адреса возврата в регистре связи LR.

Все метки и переменные, объявленные в текущем модуле, являются *локальными метками* - local, доступными исключительно изнутри и недоступными из других модулей. Это предотвращает любое несанкционированное программистом вмешательство внутрь текущего программного модуля извне, в том числе, вызов имеющейся в этом модуле подпрограммы.

Такой модуль представляет собой своеобразный «черный ящик», в котором может быть любое число различных секций, структурирующих программу, и любое число подпрограмм. Под *внутренней модульностью* далее будем понимать: 1) структурирование программы кодовыми секциями и секциями данных; 2) наличие в модуле основной программы из любого числа фрагментов, которым может передаваться управление, и любого числа подпрограмм. В частном случае программный модуль может содержать только основную программу или подпрограмму (или несколько подпрограмм).

В общем случае каждый программный модуль должен иметь некоторый интерфейс с другими модулями (внешними по отношению к нему), если программист предполагает их совместную работу. Об этом – далее.

8.10.2 Директивы объявления функций PROC и FUNCTION

В отличие от языков высокого уровня, в Ассемблере понятие функции не отличается от понятия процедуры. И то, и другое – *подпрограмма*, которая получает на входе некоторый набор аргументов, обрабатывает его и выдает (возвращает) результат в виде значений одной или нескольких переменных (возможно даже – массива данных). Подпрограммы могут вызываться многократно с разными наборами аргументов, генерируя различные результаты. С точки зрения Ассемблера, функция – это та же подпрограмма, но возвращающая лишь один результат.

Для идентификации начала и конца подпрограммы в Ассемблере предусмотрены специальные директивы: PROC (объявление процедуры) и ENDP (конец текущей процедуры) или FUNCTION (объявление функции) и ENDF (конец функции). Эти директивы равноценны и представляют собой своеобразные необязательные операторные скобки, в которые может быть заключено тело любой процедуры:

```
Label     PROC
; Тело подпрограммы
;    …
          ENDP
```

или

```
Label     FUNCTION
; Тело функции
;    …
          ENDFUNC
```

Вы можете не использовать при оформлении процедур специально предназначенные для этого директивы. Однако лучше все же воспользоваться этой возможностью. В этом случае Ваш программный модуль будет более читабельным. Кроме того, по мере роста Вашей квалификации, Вы сможете задавать в директивах PROC/FUNCTION дополнительные опции для контроля правильности передачи параметров в подпрограммы на этапе отладки. Технологии передачи параметров в подпрограммы и вложенным подпрограммам будет посвящен отдельный раздел книги (глава 15).

8.10.3 Внешняя модульность

Ситуация модуля – «черного ящика» – существенно меняется, когда подпрограмма или переменная, к которой следует обращение, находится в другом программном модуле или в библиотечном файле. В этом случае программист должен «предупредить» транслятор, что указанное им имя является по отношению к текущему модулю *внешним*. С другой стороны, в том программном модуле, где находится переменная или метка, к которой может последовать обращение извне, необходимо сообщить транслятору, что этот символ должен быть «*общедоступным*» (*глобальным*).

Только при таком подходе при объединении файлов проекта компоновщиком не возникнет конфликта имен: при трансляции исходных файлов эти имена будут помечены как внешние или глобальные (без присвоения им конкретных адресов), а при компоновке и размещении выходного исполнительного модуля в памяти они получат конкретные значения. Программисты говорят, что «*разрешение*» всех внешних ссылок выполняется не на стадии трансляции программных модулей, а на стадии их компоновки. При этом термин «разрешение» внешних ссылок используется в смысле расчета и присвоения символическим именам конкретных физических адресов в соответствии с окончательным реальным размещением кодовых секций и секций данных в памяти микроконтроллера.

ARM ASM Имеются две директивы Ассемблера, которые управляют *видимостью переменных*. Их можно располагать в любом месте программы, однако правилом хорошего тона при разработке ПО считается размещение таких директив в начале каждого программного модуля. В этом случае они будут отражать *интерфейс текущего программного модуля с внешним миром* (другими программными модулями).

8.10.3.1 Директива объявления глобального (общедоступного) имени

Если разработчик считает, что какие-либо символы должны быть доступны из других модулей (разрешает внешний доступ к ним), то он должен пометить эти символы

как *глобальные*, доступные отовсюду с помощью директивы Ассемблера EXPORT («экспортировать имя» вовне). Ее упрощенный синтаксис, используемый наиболее часто:

```
EXPORT symbol [WEAK, type]
```

Имя директивы EXPORT размещается, как обычно, в поле мнемокода команды. За ним, после пробела, указывается имя глобального символа symbol, который должен быть доступен извне. В квадратных скобках могут быть указаны приведенные ниже опции:

Опция WEAK («Слабый»): используется преимущественно в описании точек входа в подпрограммы обслуживания прерываний/исключений на начальной стадии разработки ПО, когда вместо реальной процедуры обслуживания прерывания/исключения, которая еще не создана, используется *процедура шаблон* – как бы «пустышка», которая, ничего не делая, просто возвращает управление обратно или реализует бесконечный цикл (лишь фиксирует факт попадания в процедуру обслуживания прерывания без выполнения каких-либо конкретных действий). Это позволяет выполнить начальную инициализацию таблицы векторов прерываний/исключений, которая должна обязательно присутствовать в начальной области кодовой памяти для работы любой программы (см. 9.1). В дальнейшем, возможно уже в самом конце разработки, Вы напишете программный модуль, в котором определите настоящую процедуру обслуживания прерывания/исключения с точно таким же именем, но без опции «слабого» символа WEAK. В этом случае произойдет следующее – компоновщик поймет, что появилось более «сильное» глобальное имя и заменит имя процедуры-шаблона на имя реальной процедуры. В таблицу векторов прерываний/исключений будет включен другой адрес. Использование этой опции позволяет не удалять из программного модуля процедуру-шаблон, заменяя ее на новую автоматически, без каких-либо конфликтов многократного определения имен.

Опция type задает тип секции, к которой относится данная глобальная метка. Возможны лишь два варианта, в соответствии с двумя возможными типами секций:

- DATA – метка будет расположена в секции данных (глобальная переменная);
- CODE – метка будет расположена в кодовой секции (точка входа в общедоступную подпрограмму или метка константы/таблицы констант);

8.10.3.2 Директива объявления внешнего имени

Если в текущем программном модуле имеется обращение к процедуре, расположенной в другом программном модуле, выполняется передача управления в другой модуль или выполняется доступ к переменной, расположенной в другом модуле по записи или чтению, то соответствующее имя должно быть помечено как *внешнее*. Для этой цели используется директива IMPORT («импортирование» имени извне) с синтаксисом:

```
IMPORT symbol [WEAK, type]
```

Если опция «WEAK» не специфицирована (как это бывает в большинстве случаев) и указанный в директиве внешний символ при компоновке модулей проекта не найден, то компоновщик генерирует сообщение об ошибке.

Если же при определении внешнего имени указана опция «WEAK», а символ в процессе компоновки не найден, то возможны следующие варианты:

1) Команда перехода (B) или вызова подпрограммы (BL) автоматически заменяется командой NOP (нет операции) и фактически выполняется переход на или в? следующую команду программы;

2) В других командах (например, загрузки данных) адрес внешней метки автоматически заменяется нулем. Этот механизм может, в частности, использоваться для условного ветвления программы в зависимости от того, подключена ли к проекту библиотека с нужной подпрограммой или нет.

Тип переменной type, так же, как и в директиве EXPORT, определяется типом секции, в которой дальше будет объявляться переменная (DATA, CODE). Он указывается лишь тогда, когда описание внешних интерфейсов модуля выполняется в самом его начале, еще до объявления конкретной секции.

В большинстве применений директивы EXPORT и IMPORT могут не содержать опции «WEAK». Эта опция в директиве EXPORT обязательно потребуется при создании шаблонов процедур обработки прерываний/исключений для того, чтобы можно было с самого начала работы над любым проектом сгенерировать таблицу векторов перехода на подпрограммы прерываний/исключений.

8.10.4 Реализация внешней модульности в программах

Приведем структуру двух программных модулей. Из первого модуля вызывается подпрограмма суммирования двух 64-битных чисел SUM_64, расположенная во втором модуле, а сами исходные переменные в виде старшего и младшего слов операндов находятся в памяти данных первого модуля.

Если глобальное имя объявляется после директивы объявления секции (данных или кодовой) директивой AREA, то тип глобальной метки по умолчанию является типом секции, в которой она объявлена и опция типа в директиве EXPORT не нужна. Другими словами – эта опция может потребоваться только в самом начале программного модуля, когда ни одна из секций еще не объявлена, а интерфейс модуля с внешним миром декларируется. Такая же технология распространяется на объявление внешних имен директивой IMPORT. Если это делается внутри уже объявленной секции, то внешнее имя автоматически получает тип текущей секции.

Символы в двух программных модулях, объявленные директивой EXPORT в одном (глобальное, общедоступное имя) и директивой IMPORT в другом (внешнее имя), должны быть абсолютно идентичны. В процессе компоновки они будут иметь одинаковые адреса.

```
; Модуль №1
; Интерфейс с другими модулями:
; Объявление внешней подпрограммы
      IMPORT SUM_64, CODE
; Объявление глобальных переменных
      EXPORT W1_L, DATA
      EXPORT W1_H, DATA
      EXPORT W2_L, DATA
      EXPORT W2_H, DATA
;     …
; Объявление кодовой секции
      AREA MyCode, CODE, ReadOnly
;     …
; Передача параметров в подпр.-му
;     …
; Вызов подпрограммы
      BL SUM_64
;     …
; Объявление секции данных
      AREA MyData, DATA, ReadWrite
W1_L  SPACE 4
W1_H  SPACE 4
W2_L  SPACE 4
W2_H  SPACE 4
;     …
; Конец программного модуля №1
      END
```

```
; Модуль №2
; Интерфейс с другими модулями:
; Объявление общедоступной подпрограммы
      EXPORT SUB_64, CODE
; Объявление внешних переменных
      IMPORT W1_L, DATA
      IMPORT W1_H, DATA
      IMPORT W2_L, DATA
      IMPORT W2_H, DATA
;     …
; Объявление кодовой секции
      AREA MyCode, CODE, ReadOnly
;     …
; Подпрограмма суммирования SUM_64
SUM_64      PROC
; Тело подпрограммы
;     …
            ENDP
; Объявление секции данных
      AREA MyVar, DATA, ReadWrite
BUF   SPACE 4*20
;     …


; Конец программного модуля №2
      END
```

Ассемблер допускает и второй способ определения внешних имен – директивой EXTERN (внешнее имя). Между этими двумя директивами имеется некоторое различие (см. табл. 8.2).

Таблица 8.2 Описание директив

Директива	Особенности работы
IMPORT	**Внешнее имя всегда включается в таблицу символов. Если внешний символ не был найден во время компоновки (например, файл с такой меткой или именем переменной еще не написан или не подключен к проекту) – генерируется сообщение об ошибке.**
EXTERN	**Внешнее имя включается в таблицу символов только тогда, когда к нему имеется обращение из текущего программного модуля. В противном случае – не включается, и сообщение об ошибке не генерируется. Это удобно, когда программные модули отлаживаются отдельно. Модуль, на который делается внешняя ссылка, может быть создан потом.**

1. Как изменится начало кодовой секции в модуле № 1 примера, если объявления общедоступных и внешних имен будут делаться не в начале файла, а по ходу программы?
2. А как при этом будет выглядеть начало кодовой секции в модуле № 2?
3. Как Вы думаете, какую из двух директив нужно использовать, если модуль с подпрограммой FUNC1 еще не написан, а модуль, из которого вызывается функция FUNC1, уже имеется и основную программу этого модуля без вызова функции FUNC1 нужно отладить?
4. Допустимо ли в разных программных модулях иметь кодовые секции с одинаковыми именами (в примере секции MyCode)?

5. А могут ли секции данных в разных программных модулях иметь разные имена?
6. Какой объем буфера данных резервируется в секции данных модуля № 2?
7. В каких случаях Вы будете пользоваться опцией WEAK («Слабый символ»)?
8. А можно ли обойтись без шаблонов подпрограмм обслуживания прерываний/исключений?

1. После объявления кодовой секции, перед использованием символа SUM-64 (вызовом подпрограммы) нужно объявить его внешним символом:

```
; Объявление кодовой секции
     AREA MyCode, CODE, ReadOnly
; Объявление внешнего символа
      IMPORT SUM_64
```

2. Естественно, что описанная в этом модуле подпрограмма должна быть объявлена общедоступной. При этом, как и в предыдущем случае, тип имени будет автоматически определяться типом секции, в которой находится директива:

```
; Объявление кодовой секции
     AREA MyCode, CODE, ReadOnly
; Объявление общедоступного символа – начала подпрограммы
     EXPORT SUM_64
```

3. Директиву EXTERN.
4. Да, в этом случае обе секции при компоновке будут объединены в одну с общим именем.
5. Да, как в нашем примере. При компоновке они будут объединяться в одну по правилам, указанным компоновщику (например, последовательно друг за другом).
6. Арифметическое выражение (4*20) будет вычислено на стадии трансляции с результатом 80 байт.
7. Например, для описания начальных адресов подпрограмм обслуживания прерываний/исключений директивой EXPORT, которые пока еще окончательно не? разработаны, а являются подпрограммами-шаблонами (первыми приближениями).
8. Нет, так как, по крайней мере, начальная часть таблицы векторов прерываний/исключений должна обязательно присутствовать в проекте. Вы не должны выполнять ни одну программу без начальной инициализации процессора и стека – об этом в следующей главе подробно.

Список рекомендуемой литературы

1) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
2) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.
3) Электронный ресурс http://www.keil.com

9 ПЕРВЫЕ ШАГИ В ПРОГРАММИРОВАНИИ НА АССЕМБЛЕРЕ

Оглавление

9.1 Структура стартового файла проекта

Процессор – сложное устройство, не только программно-управляемое с помощью пользовательских программ, но *программно-настраиваемое*, с помощью дополнительных специальных программ. Проект не может состоять из одной программы пользователя, в нем обязательно должна присутствовать *программа инициализации* процессора и встроенных в него периферийных устройств (в том числе сопроцессора). Функцию начальной инициализации берет на себя так называемый стартовый файл SrartUp.s, который пишется не на языке высокого уровня, а на Ассемблере, так как должен иметь доступ ко всем системным ресурсам процессора без каких-либо ограничений.

Производители микроконтроллеров, микропроцессоров, а также интегрированных сред разработки предлагают пользователям свои версии стартовых файлов, адаптированные не только к типу центрального процессора, но и к встроенной в процессор и микроконтроллер периферии. Начинающим программистам очень сложно самостоятельно выбрать тип стартового файла по ряду причин:

1) Низкая начальная квалификация, которая не позволяет понять, что и с какой целью включено в стартовый файл, что можно добавить или безболезненно исключить из него.
2) Стартовые файлы используются как своеобразные «черные ящики», особенно программистами на языках высокого уровня – в них можно «заглядывать», но трогать не рекомендуется, во избежание возможных проблем.
3) От типа и структуры стартового файла зависит и спектр возможностей, которые будут поддерживаться при программировании на Ассемблере и на C/C++.

Перед авторами книги стоял выбор: 1) ничего не объясняя, дать какой-то стартовый файл в качестве «черного ящика» и приступить к изучению системы команд процессора и технологии программирования на Ассемблере; 2) попытаться предварительно объяснить читателю, по крайней мере, назначение отдельных блоков стартового файла, чтобы в последующем можно было, при необходимости, самостоятельно модифицировать его.

Мы выбрали второй путь, рассчитывая на *заинтересованного читателя*, который до всего хочет «докопаться» сам, которого не устраивает принцип: «возьми и пользуйся, за тебя уже все продумали более квалифицированные специалисты». Именно поэтому мы рассмотрели раньше такие, казалось бы, очень специфические вопросы, как секционирование программных модулей, принцип построения таблицы векторов прерываний/исключений, использование директив процессора для управления распределением памяти и др. (изложенные, в основном, в предыдущей главе). Теперь, знакомясь со структурой стартового файла проекта, Вы будете «вооружены», – многое Вам, если не все в этом файле, должно быть понятно. Если это не так, перечитайте, пожалуйста, предыдущую главу.

Итак, предлагается версия стартового файла StartUp_1.s, подготовленная авторами книги специально для начинающих изучать систему команд процессоров ARM Cortex-M и программирование на Ассемблере в среде µVision. Она содержит подробные комментарии, описывающие назначение отдельных фрагментов этого файла:

```
; Программный модуль стартового файла StartUp_1
; Определить переменную "Размер стека" (1 К байт)
Stack_Size      EQU     0x00000400
; Объявить секцию данных для размещения стека системы
; без инициализации памяти, с атрибутом выравнивания
; по 8 байтам
                AREA    STACK, NOINIT, READWRITE, ALIGN=3
; Зарезервировать область памяти под стек
; с числом байт Stack_Size
Stack_Mem       SPACE   Stack_Size    ; (1 К байт)
; Метка вершины стека (авто-декрементный стек)
__initial_sp
```

При работе с подпрограммами (функциями), а также для временного сохранения и восстановления данных обязательно нужен стек. Здесь задается размер стека и объявляется специальная секция STACK в памяти данных, которая будет выделена для него директивой SPACE. Так как стек заполняется в сторону убывающих адресов, вершиной стека является адрес следующей, после директивы резервирования памяти, ячейки памяти – _initial_sp.

```
; Vector Table
; Объявить секцию для размещения таблицы векторов
; прерываний/исключений
; Для компановщика определяется как область памяти
; данных RESET. Будет автоматически размещена компоновщиком
; в начале памяти программ
                AREA    RESET, DATA, READONLY
; Объявить параметры таблицы векторов - глобальными именами
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size
; Инициализация векторов обработчиков
; прерываний/исключений
__Vectors       DCD     __initial_sp  ; Вершина стека - Top of Stack
                DCD     Reset_Handler ; Точка выхода в обработчик исключения
                                      ; по сбросу процессора Reset Handler
                DCD     NMI_Handler   ; Точка входа в обработчик
                                      ; немаскируемого прерывания NMI
;               ...
; Далее по аналогии могут быть объявлены и остальные
; вектора обработчиков прерываний/исключений
__Vectors_End
__Vectors_Size  EQU     __Vectors_End - __Vectors
```

Один из важнейших разделов стартового файла – определение таблицы векторов прерываний/исключений и размещение ее в начале кодовой памяти. Как видите, первый вектор – это адрес вершины стека в системе, второй – точка входа в обработчик прерывания по сбросу процессора, а третий – в обработчик прерывания по немаскируемому запросу прерывания. Мы ограничились только этими тремя векторами, при необходимости Вы сможете определить и все остальные векторы прерываний/исключений. Заметьте, что сначала векторы размещаются в секции памяти данных по имени Reset и только затем, автоматически, компоновщик переносит их в начальную область кодовой памяти, как это и требуется в соответствии с архитектурой ARM-процессоров. Заметьте также, что для размещения адресов точек входа в подпрограммы (векторов) применяется псевдокоманда резервирования 32-разрядных слов

в памяти данных DCD. В качестве параметров для нее используются символические имена точек входа в подпрограммы-обработчики, которые определяются ниже.

```
38  ; Объявление кодовой секции для размещения
39  ; подпрограмм обработчиков прерываний/исключений
40                  AREA    |.text|, CODE, READONLY
41
42  ; Обработчик прерывания по сбросу процессора Reset
43  Reset_Handler   PROC
44  ; Объявить процедуру Reset_Handler общедоступной
45  ; Она может быть перепрограммирована в последующем
46  ; Поэтому - используется опция WEAK ("Слабая метка")
47                  EXPORT  Reset_Handler [WEAK]
48  ; Инициализация процессора (через регистр CONTROL)
49  ;               ...
50  ; В данной версии стартового файла не выполняется
51  ; Процессор будет работать по умолчанию
52  ; в режиме потока (Thread Mode)
53  ; с привилегированным доступом ко всем ресурсам (Privilegied)
54  ; с пока выключенным сопроцессором FPU
55
56  ; Передать управление пользовательской программе MyProg
57  ; Объявление точки входа в программу пользователя
58  ; (внешняя метка)
59                  IMPORT  MyProg
60  ; Псевдо-команда: загрузить адрес MyProg в регистр r0
61                  LDR     r0, =MyProg   ; r0 <- адрес точки входа
62  ; Косвенная передача управления программе пользователя
63                  BX      r0            ; PC <- (r0)
64                  ENDP      ; Конец процедуры Reset_Handler
```

Подпрограммы обработчиков прерываний/ исключений должны быть размещены в кодовой секции, которая здесь и объявляется. Это должны быть общедоступные процедуры, более того, такие, которые в последующем можно будет переопределить – опция WEAK в объявлении глобальной метки.

В процедуре обработчика по сбросу процессора должна быть выполнена его инициализация. В данной версии файла она не делается, процессор будет работать в режиме «по умолчанию». Единственная операция, которая выполняется в обработчике Reset_Halter в данном случае, – это передача управления программе пользователя по внешней метке MyProg (это может быть любое имя, например, main, соответствующее имени Вашей программы). Для этой цели используется специальная *псевдокоманда* Ассемблера LDR Rn, =Label, с особенностями работы которой мы познакомимся уже в этой главе. Фактически, в регистр r0 загружается адрес точки входа в программу пользователя, а следующей командой косвенной передачи управления по содержимому регистра r0 (BX r0) этот адрес записывается в счетчик команд процессора PC, в результате чего и выполняется переход на программу пользователя.

Именно с команды LDR r0, =MyProg (расположенной в стартовом файле) будет выполняться любая программа пользователя после сброса процессора в среде µVision, когда Вы будете отлаживать ее в симуляторе. Обязательно объявляйте точку входа в свою программу MyProg общедоступной с помощью директивы Ассемблера EXPORT.

```
65
66  ; "Пустой" обработчик немаскируемого прерывания NMI ("шаблон")
67  NMI_Handler     PROC
68  ; Объявить процедуру NMI_Handler общедоступной
69  ; В дальнейшем может быть перепрограммирована,
70  ; использована опция WEAK ("Слабая метка")
71                  EXPORT  NMI_Handler [WEAK]
72  ; Зациклить программу обработчика
73                  B       .
74                  ENDP    ; Конец процедуры NMI_Handler
```

Теперь покажем, на примере обработчика немаскируемого прерывания NMI, как программируется «шаблон» любой процедуры обслуживания прерывания/исключения. Он содержит всего лишь команду зацикливания программы (передачи управления на себя – «B .»). Точка «.» в Ассемблере означает текущее значение счетчика команд PC в начале выполнения команды, адрес размещения текущей команды в памяти. Такой обработчик ничего не делает, но ничего и не портит. Если прерывание NMI возникнет (и разрешено), мы попадем в эту процедуру. Сам факт попадания важен, так как будет свидетельствовать о том, что запрос прерывания воспринимается системой. Далее Вы сможете заменить «слабого» обработчика реальным «сильным», в котором опция WEAK отсутствует.

Осталось только выровнять текущую кодовую секцию по 32-разрядному слову,

```
75  ; Выровнять кодовую секцию по 4-х байтовому слову
76                  ALIGN
77  ; Конец ассемблерного текста стартового модуля
78                  END
```

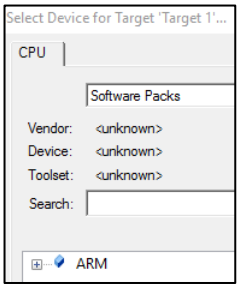
чтобы последующие коды программы пользователя размещались в памяти

оптимально с точки зрения считывания их на конвейер команд, и завершить текст программного модуля директивой END – конца ассемблерного текста.

Стартовый файл исключительно важен. В нем, как минимум, резервируется место под стек системы и передается управление программе пользователя. Как максимум, выполняется инициализация процессора и периферии. Если Вы не все поняли в структуре стартового файла, не огорчайтесь! Мы будем постепенно знакомиться с основами программирования на Ассемблере и, в том числе, дадим дополнительные объяснения по работе этого файла. Вы можете найти файл в каталоге проекта CPU_0.

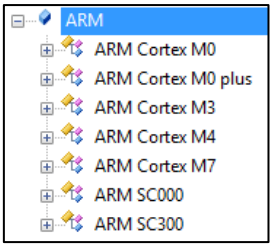
9.2 Как создать новый проект?

Для того, чтобы создать *новый проект*, обратитесь в меню Project и выберите из него команду «**New µVision Project**». Откроется окно создания проекта «**Create New Project**», в котором можно «путешествовать» по дереву каталогов Вашего компьютера с целью выбора нужного каталога. Если каталог для проекта еще не создан, среда µVision позволяет сделать это. Не выходя из окна создания проекта, нужно обычными средствами операционной системы перейти в подходящую директорию и создать в ней новую папку командой «**Создать папку**». Естественно, для этого необходимо указать символическое имя каталога, допустим «Проба» (имена на русском языке допускаются операционной системой). После этого нужно открыть папку и создать в ней файл нового проекта. Для этого достаточно ввести только имя проекта, а стандартное расширение будет создано автоматически. Вы можете использовать для имени проекта то же имя, которое имеет и папка, его содержащая. Это удобно и исключает путаницу. Осталось выполнить в текущем окне команду «**Сохранить**» проект.



После этой команды среда µVision предложит Вам определиться с *целевым устройством*, для которого Вы будете писать и отлаживать программное обеспечение – откроется окно «**Select Device for Target 'Target 1'**» («**Выберите целевое устройство, 'Цель 1'**»).

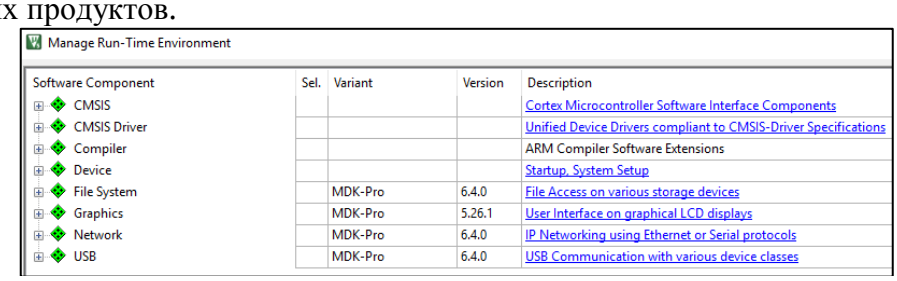
Имеется в виду, что в последующем Вы сможете проверить свою программу не на одном устройстве, а на нескольких, например, сначала в симуляторе, затем – на оценочной плате, потом на созданном Вами контроллере. Мы привели фрагмент этого окна, чтобы показать дальнейшую последовательность действий.



Поддерживаемые средой µVision устройства (процессоры ARM и микроконтроллеры на их основе) скрыты в списке ARM за «крестиком» «+». Щелкните на нем мышкой и список раскроется. Таким же образом выберете целевое устройство, для которого планируется разработка. Мы изучаем семейство процессорных ядер на базе ARM Cortex M4. Откройте соответствующую позицию списка **«ARM Cortex M4»**. Вам предоставляются две возможности – выбрать процессоры Cortex M4 без интегрированного на кристалл сопроцессора поддержки вычислений с плавающей точкой **(«ARMCM4»)** или с ним **(«ARMCM4_FP»)**. Выбирайте последний вариант **(«ARMCM4_FP»)**, так как мы обязательно будем изучать и работу сопроцессора FPU. Как только выбор сделан, появится дополнительная информация в окне краткого описания возможностей выбранного процессора. По предыдущим главам книги они нам уже знакомы: полноценный 32-разрядный процессор, предназначенный для встраиваемых в оборудование приложений; простая программная модель; высокая производительность и малое потребление энергии; высокая плотность

кода; эффективный встроенный контроллер прерываний; совместимость вниз (без команд использования сопроцессора) с ядром Cortex-M3.

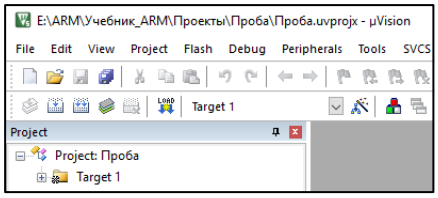
После выбора процессорного ядра Вам будет предложено дополнительно выбрать **окружение реального времени выполнения**. Окно менеджера выбора соответствующих программных продуктов.



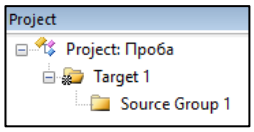
Первая позиция содержит компоненты программного интерфейса микроконтроллеров Cortex, которые заметно ускоряют и облегчают программирование на языке высокого уровня C/C++. Среди них: заголовочные файлы с определениями регистров периферийных устройств; средства абстрактного доступа к ним; примеры.

Вторая позиция позволяет подключить унифицированные драйверы интерфейсов, такие как драйвер Ethernet, флэш-памяти и др.

Остальные позиции также предоставляют дополнительные опции программистам на языке C/C++, начиная от стартовых файлов и кончая поддержкой графики, USB-интерфейсов и прочее. С точки зрения программиста на Ассемблере, тем более, начинающего, *можно опустить все дополнительные возможности*, нажав сразу (внизу



окна) клавишу «OK». Наш новый проект будет создан и отображен в окне проектов с именем «Проба» (показана верхняя часть окна). Для данного проекта указано одно целевое устройство на базе селектированного нами процессора Cortex-M4F.



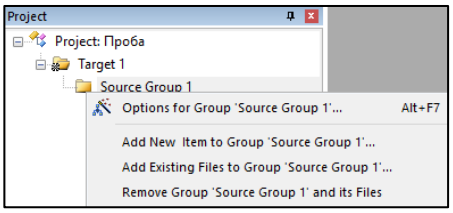
Обратите внимание на то, что пока в нашем проекте есть только одно целевое устройство Target 1 и – никаких файлов, входящих в проект. Щелкните мышкой на «крестике» перед именем Target 1. В окно проектов будет выведено имя «**Первой группы исходных файлов проекта**» – «**Source Group 1**». Обычно в проекте достаточно одной группы исходных файлов. Однако, в ряде случаев, например, если Вы разрабатываете программное обеспечение на конкурентной основе несколькими группами программистов, исходных групп файлов может быть несколько. Кроме того, для каждой из групп файлов можно устанавливать свои опции (настройки) среды.

9.3 Как создать файл с программой на Ассемблере и включить его в проект?

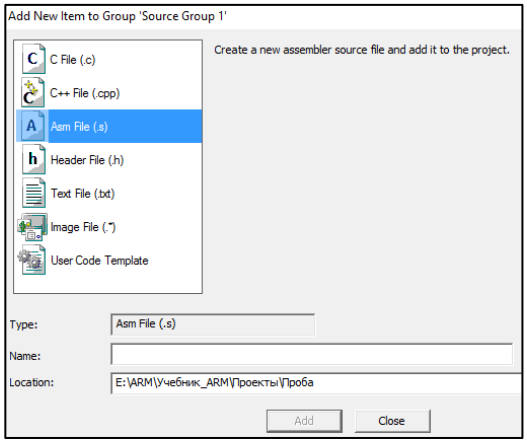
Открыть новый файл для написания программы можно несколькими способами:

1) Из меню «**Файл**» («**File**»), выбрав команду «**New**» («**Новый**»).
2) Щелкнув клавишей мыши на стандартную пиктограмму открытия нового файла – первая в строке инструментов среды  . Работа с файлами в µVision аналогична работе с файлами в любой компьютерной программе и поэтому не рассматривается. Как видите, используются и стандартные пиктограммы основных действий при работе с файлами: создать новый файл, открыть имеющийся, сохранить, сохранить информацию во всех открытых исходных файлах проекта.

3) Щелкните *правой клавишей мыши* на пиктограмме группы файлов в исходном коде (Source Group 1). В большинстве программных продуктов правая клавиша мыши используется для вызова так называемого *контекстно-зависимого меню*. Здесь именно такой случай: Во второй позиции этого меню имеется команда «**Add New Item to Group 'Source Group 1**» – «**Добавить новый объект в группу 'Группа исходных файлов 1**'». Новый объект – это и есть новый исходный файл – либо на Ассемблере, либо на СИ. Попутно заметим, что следующая позиция этого меню «**Add Existing Files to Group 'Source Group 1**'» позволяет добавить (включить) уже существующий файл (который Вы создали ранее или просто скопировали из другого каталога) в данную группу, т.е. *подключить уже имеющийся файл к проекту.*

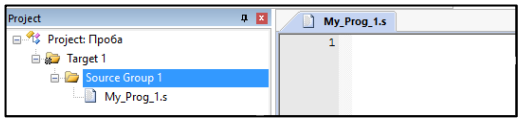


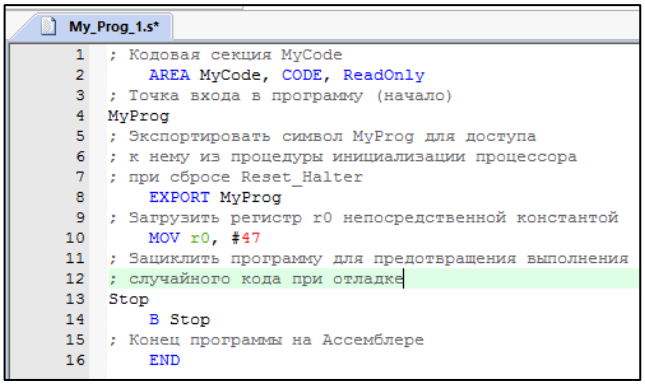
Таким образом, если Вы воспользовались для создания нового файла обычными средствами (п.1, 2), то после его сохранения просто подключите файл к проекту. В последнем случае (п. 3) создание файла выполняется одновременно с его подключением.



При этом на экран выводится окно, в котором прежде всего нужно выбрать тип создаваемого и подключаемого файла. Выбор большой: это могут быть файлы на языке высокого уровня C или C++, на Ассемблере, заголовочные и т.д. Мы выбираем исходный файл на Ассемблере. Для этого достаточно щелкнуть мышью на соответствующей позиции списка, после чего она будет выделена подсветкой, и задать имя исходного файла. Ассемблерные файлы по умолчанию имеют расширение «.s». После ввода имени файла и нажатия клавиши «Add» файл сразу будет подключен к проекту (еще «пустой») и откроется окно редактирования, в которое можно ввести текст программы.

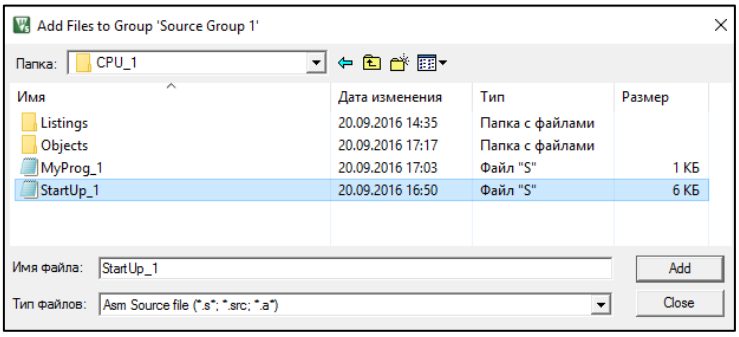
Как только Вы нажмете на «+» в окне проектов в позиции «**Группа исходных файлов 1**» «**Source Group 1**», содержимое этой группы раскроется, и Вы увидите, что в ней уже находится файл My_Prog_1.s.



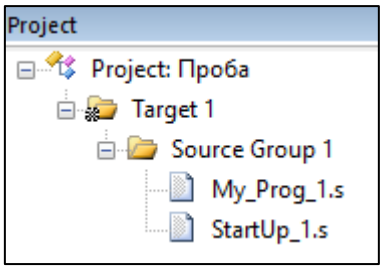


Теперь можно попробовать написать простую программу, ведь мы уже находимся во встроенном редакторе среды. В программе объявляется кодовая секция MyCode, точка входа программу MyProg, которая должна быть доступна из других файлов проекта, и выполняется лишь одно действие – инициализация регистра ЦПУ кодом #47. Во избежание выполнения в процессе отладки программы «неопределенного» кода программа зацикливается (управление передается на ту же самую команду «B Stop»). Обратите внимание на то, что в окне редакирования имя программы помечено звездочкой «*». Это означает, что она еще не прошла этап ассемблирования (трансляции). Как только это случится, «звездочка» автоматически исчезнет.

9.4 Как подключить к проекту стартовый файл?



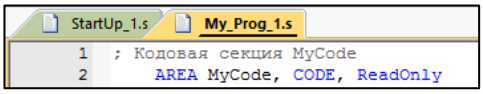
Мы уже не раз упоминали, что в любой проект должен быть включен стартовый файл. Для этого вызовем еще раз контекстно-зависимое меню из окна проектов (см. выше) и выберем команду Добавить файлы в группу «**Исходная группа 1**» (**Add Files to Group 'Source Group 1'**). Нужно перейти в каталог, который содержит требуемый файл, указать его тип (в данном случае исходный файл на языке Ассемблер – Asm Source file (*.s*; *src*; *.a*). Все имеющиеся в каталоге файлы данного типа будут показаны в окне. Нужно выделить искомый файл мышью (после этого он будет подсвечен) и нажать клавишу «**Добавить**» («**Add**»). Файл будет подключен к проекту.



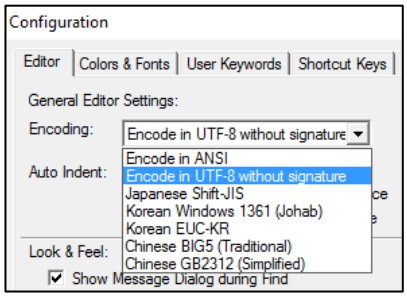
Окно проектов изменится: теперь в проекте уже два файла – с нашей первой программой My_Prog_1.s и стартовый файл StartUp_1.s. Обратите внимание на то, что к текущему проекту мы подключили стартовый файл, расположенный в каталоге CPU_1. Следовательно, *к проекту могут подключаться любые файлы, расположенные в любых каталогах*, их не обязательно копировать в каталог текущего проекта. Это удобно, так как позволяет постепенно подключать к проекту те файлы, которые уже прошли стадию отдельного тестирования.

9.5 Как перейти к редактированию исходного файла?

Как только нужные исходные файлы подключены к проекту, их можно редактировать (модернизировать). Для этого достаточно дважды щелкнуть мышью на имени файла в окне проектов. В основном окне µVision появится вкладка с текстом исходного файла.



Так как исходных файлов может быть несколько, между вкладками (исходными программами) можно переключаться. Для этого достаточно щелкнуть мышью на имени вкладки, она станет активной – ее имя будет подчеркнуто, а вкладка подсвечена желтым цветом. Можете приступать к редактированию. Технология редактирования исходных файлов похожа на работу в любом качественном текстовом редакторе и не требует особых пояснений. Единственное отличие состоит в том, что все синтаксические конструкции языка Ассемблер или C/C++ выделяются цветом для удобства пользователей.

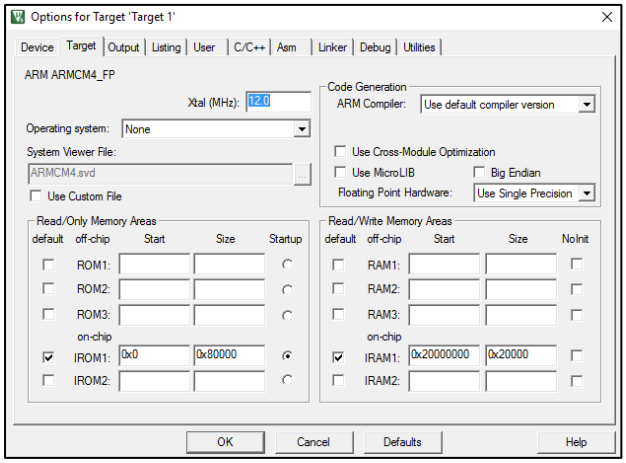


При необходимости, Вы можете перенастроить редактор в соответствии со своими предпочтениями. В этой книге используются установки «по умолчанию». Впрочем есть одно «но». Если Вы хотите, чтобы комментарий в Ваших программах на русском языке правильно отображался в исходных файлах и в файлах листинга (результатах трансляции) войдите в меню «**Edit**» («**Редактор**») и выберите команду «**Configuration**» («**Конфигурация**»). Среди множества возможных настроек выберите опцию «**Encoding**» («**Кодирование**») и установите кодировку *UTF-8*, как

показано во фрагменте окна опций редактора. Вы сможете видеть комментарии на русском языке везде, за исключением окна «Дизассемблера» и некоторых других окон среды µVision, где эти символы не отображаются.

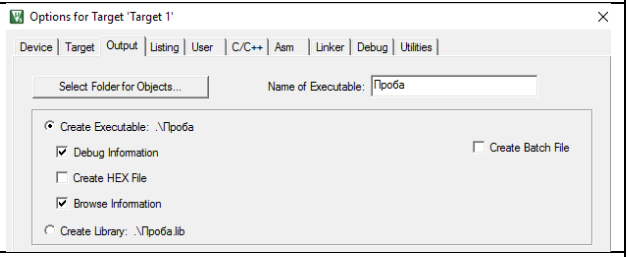
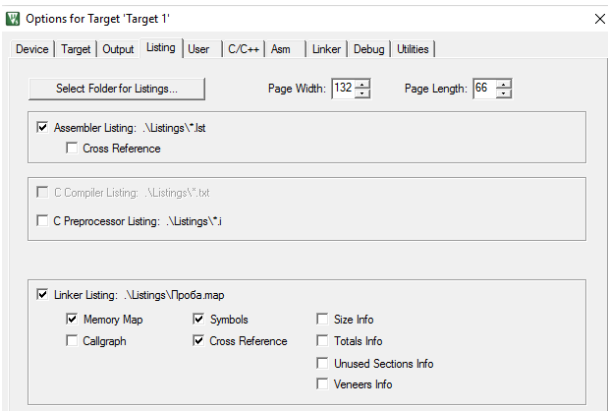
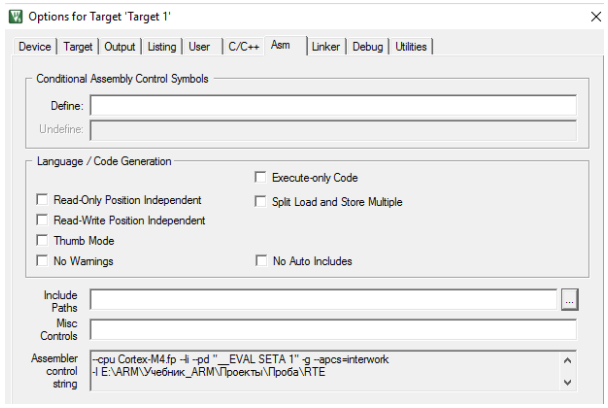
9.6 Как настроить параметры среды µVision?

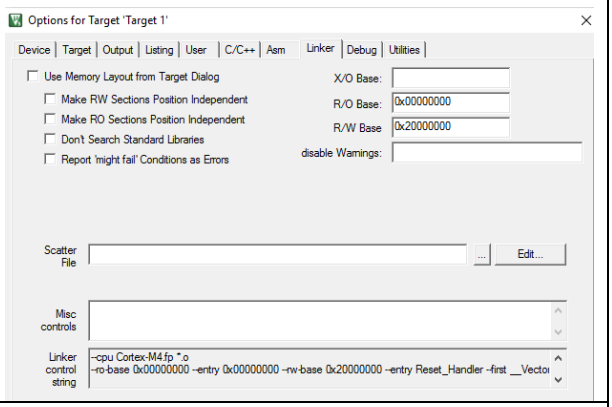
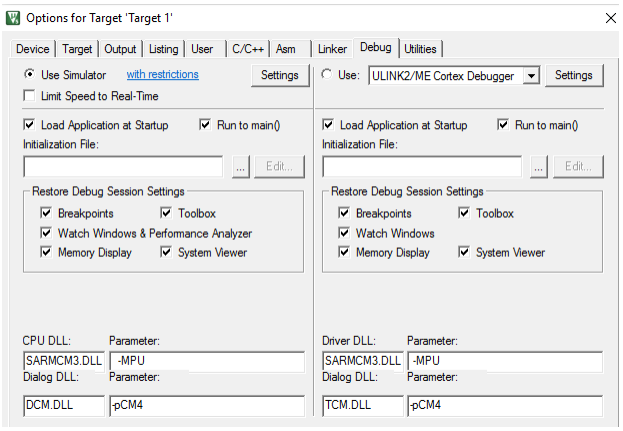
Прежде чем запустить файлы проекта на трансляцию/компиляцию и последующую отладку требуется *настройка* среды. В общем случае – это тонкая работа, которая позволяет настроить все инструменты среды (транслятор, компоновщик, отладчик и др.) на нужный пользователю режим работы. При программировании на Ассемблере настройка заметно упрощается, и по большей части достаточно будет опций «по умолчанию». По мере приобретения навыков работы со средой µVision Вы сможете самостоятельно менять настройки, адаптируя их к особенностям своих проектов. Поэтому, поступим так: дадим лишь краткие рекомендации по настройке для начинающих.



Установить опции для целевого устройства можно из меню Проекта: «**Project» – «Options for Target 'Target 1'…»** или просто щелкнув мышью на пиктограмме этой команды в строке инструментов. Откроется окно, в котором имеются несколько «вкладок» для настройки опций типа процессора **Device** (мы это уже делали при создании нового проекта), целевой платы **Target**, выходных файлов **Output**, файла листинга **Listing**, пользовательских настроек **User**, опций компилятора **C/C++**, Ассемблера **ASM**, компоновщика **Linker**, отладчика **Debug**, дополнительных утилит **Utilities**. Перед отладкой любого проекта настройка среды uVision – *обязательна.* Дальше, по мере работы над проектом, добавления в него новых файлов и их отладки, значения установленных Вами опций будут сохранены – их повторного ввода не потребуется. Советуем на первом этапе установить все опции «**по умолчанию**» «**Defaults**» и сверить с приведенными ниже краткими советами и рекомендациями.

Таблица 9.1 Рекомендации по настройке опций

Вкладка	Рекомендации по настройке для программистов на Ассемблере
Device	**Оставьте тип устройства, который Вы уже выбрали, создавая новый проект.**
Target	**«По умолчанию». При отладке в симуляторе точной конфигурации памяти конкретного устройства не требуется. Можно принять диапазон встроенной памяти программ IROM от 0x до 0x80000 (0.5 Гб), так как в открытой версии среды код все равно ограничивается объемом 32 Кб. Объема встроенной памяти данных IRAM 0x20000 (131 Кб) начиная с адреса 0x20000000 также вполне достаточно. Обе области кодовой памяти и памяти данных соответствуют унифицированной карте памяти процессоров ARM. Остальные опции нужны только при использовании реальной аппаратуры или при программировании на C/C++.**
Output	**Достаточно разрешить создание выходного исполняемого файла с именем текущего проекта «Create Executable», генерацию отладочной информации «Debug Information» и информации для просмотрщиков «Browse Information»:** 
Listing	**Разрешите формирование файла листинга «Assemgler Listing» с расширением «.lst», установив максимальную длину строки «Page Width» в количестве 132-х символов, чтобы комментарии могли разместиться в строке, а также формирование файла результатов компоновки «Linker Listing» с расширением «.map», который будет содержать карту памяти «Memory Map», информацию о символах «Symbols» и перекрестных ссылках «Cross Reference»** 
User	**«По умолчанию». Используются только при программировании на C/C++. Можно дополнительно включить опцию выдачи звукового сигнала при завершении трансляции или построения проекта «Beep When Complete». Если включить опцию старта отладки «Start Debugging», то сразу после построения или перестроения проекта автоматически запускается режим отладки.** 
C/C++	**«По умолчанию». Использовать при программировании на C/C++.**
ASM	**«По умолчанию». Не устанавливать опцию Thumb, так как в процессорах Cortex-M3/M4 используется единый набор команд Thumb-2, и выбор между генерацией 16- или 32-разрядных команд делается транслятором автоматически. Опции раздельных областей памяти («Independent») и памяти только для выполнения («Execute-only Code») не устанавливать, так как кодовая память в процессорах Cortex-M3/M4 будет хранить как коды, так и данные.** 

Linker	**«По умолчанию». Начальные адреса областей памяти «Только для чтения» R/O Base (0x00000000) и «Для чтения и записи» R/W Base (0x20000000) соответствуют расположению ПЗУ и ОЗУ в унифицированной карте памяти процессоров ARM. Коррекция не требуется.** 
Debug	**Выберите отладку программы в симуляторе «Use Simulator» и разрешите устанавливать точки останова «Breakpoints», работу с окнами переменных «Watch Windows», вывод на дисплей содержимого памяти «Memory Display», работу с инструментами «Toolbox» и просмотр состояния системы «System Viewer». Разрешите автоматическую загрузку приложения при старте «Load Application at Startup» и выполнение программы main() «Run to main()».** 
Utilites	**«По умолчанию». На начальном этапе утилиты не используются.**

По мере приобретения навыков работы со средой µVision Вы узнаете, что дополнительно можно настраивать и опции группы файлов и даже отдельных файлов в группе. Если этого не делать, то опции файлов автоматически наследуются от опций групп, а опции групп – от опций целевого устройства, рассмотренных выше. Пока это нас полностью устраивает.

9.7 Как выполнить трансляцию исходных файлов проекта и сборку проекта?

Для этого можно воспользоваться меню «**Проект**» («**Project**») или (что значительно проще) пиктограммами на панели инструментов среды, которые доступны простым «щелчком» мыши:  . Первая пиктограмма запускает процесс «**ассемблирования», «трансляции» («Translate»)** текущего исходного файла на языке Ассемблер или «**компилирования» («Compilate»)** исходного файла на языке C/C++ (выбор делается автоматически по расширению файла). Вторая пиктограмма запускает процесс «**сборки» («компоновки»)** уже оттранслированных/откомпилированных файлов проекта – создает выходные файлы проекта для загрузки и отладки: «**Build**» («**Построить проект**»). Третья – выполняет последовательно оба действия: трансляцию/компилирование всех исходных файлов и новую сборку проекта – «**Rebuild**» («**Перестроить проект**»).

На первом этапе (трансляция) создаются выходные файлы в перемещаемом объектном коде, а на втором (сборка) – в исполняемом объектном коде, которые могут загружаться на выполнение и отладку в симулятор или в целую плату. Четвертая пикторгамма предназначена для пакетной обработки исходных файлов (если проект

большой и сложный), а последняя, пятая, – для остановки (прерывания) запущенного процесса трансляции/компоновки.

Переключаясь между исходными файлами проекта и запуская каждый из них на трансляцию, Вы можете убедиться в том, что, с точки зрения синтаксиса языка программирования, Ваши программы правильные. Пример вывода результата трансляции программы My_Prog_1.s в окно «**Build Output**» («**Окно вывода результатов построения проекта**») показан ниже. Он свидетельствует о том, что синтаксических ошибок в нашей первой программе нет.

Если будет обнаружена какая-либо ошибка, то сообщение о ней будет обязательно выведено в окно результатов трансляции.

```
Build Output
*** Using Compiler 'V5.05 update 2 (build 169)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
assembling My_Prog_1.s...
"My_Prog_1.s" - 0 Error(s), 0 Warning(s).
```

1. Измените в исходной программе имя регистра r0 на имя r20. Оттранслируйте программу. В чем ошибка?
2. Щелкните мышью на коде ошибки. Вы автоматичеки попадете на строку в исходной программе, в которой она обнаружена. Голубой треугольник являтся маркером строки с синтаксической ошибкой. Можете приступать к редактированию исходного файла для устранения ошибки. Выполните повторную трансляцию. Убедитесь, что ошибка исчезла. А теперь замените команду MOV на команду LDR r0, #47. Опять ошибка. Что она означает?

```
 9  ; Загрузить регистр r0 непосредственной константой
10      MOV r20, #47
11  ; Зациклить программу для предотвращения выполнения
12  ; случайного кода при отладке
```

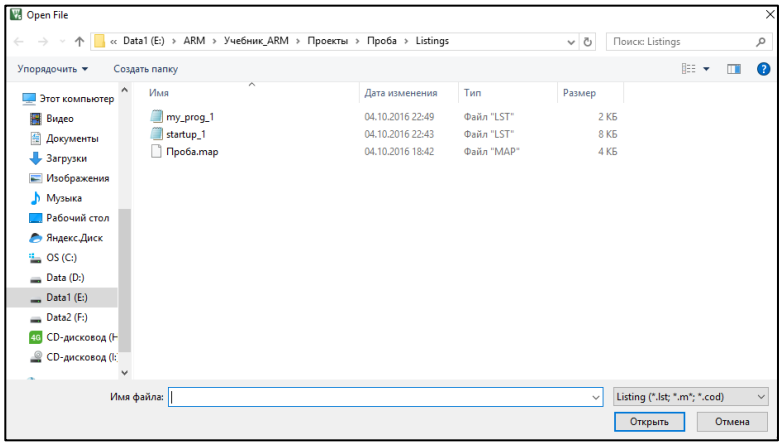
3. Убедитесь, что при трансляции стартового файла StartUp_1 ошибок нет.

1. Сообщение об ошибке: свидетельствует о неправильном использовании имени регистра. В действительности регистра с именем r20 нет среди регистров ЦПУ (только r0-r15).

```
My_Prog_1.s(10): error: A1647E: Bad register name symbol
"My_Prog_1.s" - 1 Error(s), 0 Warning(s).
```

2. Сообщение об ошибке `My_Prog_1.s(10): error: A1152E: Unexpected operator` говорит о том, что мы использовали невыполнимый оператор, то есть команду, которой нет в системе команд процессора. Это действительно так. Дальше мы покажем, как правильно выполнять загрузку регистров процессора данными.
3. Да. Это так.

Теперь можно изучить файлы с результатами трансляции. Для этого прямо из среды µVision вызовем команду открыть файл (можно щелкнуть на пиктограмме  клавишей мыши) и перейдем в подкаталог текущего проекта, который содержит файлы листингов «Listings», укажем типы интересующих нас файлов (*.lst), и все файлы с таким расширением будут выведены в окно. Выберем нужный (он будет автоматически подсвечен) и нажмем кнопку «**Открыть**». В окне среды µVision появится дополнительная вкладка с содержимым файла листинга. Он содержит с правой стороны копию программы на языке



Ассемблера, а с левой – результаты ее трансляции в машинный код: номер строки, адрес размещения данных (в HEX-коде) и код сгенерированной машинной инструкции.

```
ARM Macro Assembler    Page 1


   1 00000000          ; Кодовая секция MyCode
   2 00000000                  AREA            MyCode, CODE, ReadOnly
   3 00000000          ; Точка входа в программу (начало)
   4 00000000                  ENTRY
   5 00000000          MyProg
   6 00000000          ; Экспортировать символ MyProg для доступа
   7 00000000          ; к нему из процедуры инициализации процессора
   8 00000000          ; при сбросе Reset_Halter
   9 00000000                  EXPORT          MyProg
  10 00000000          ; Загрузить регистр r0 непосредственной константой
  11 00000000 F04F 002F       MOV             r0, #47
  12 00000004          ; Зациклить программу для предотвращения выполнения
  13 00000004          ; случайного кода при отладке
  14 00000004          Stop
  15 00000004 E7FE            B               Stop
  16 00000006          ; Конец программы на Ассемблере
  17 00000006                 END
```

В нашей программе всего две команды: первая (MOV) является 32-разрядной, а вторая (B) – 16-разрядной. Транслятор автоматически делает оптимальный выбор между командами ARM/THUMB унифицированного единого набора команд Thumb-2, создавая команду наименьшей возможной длины.

Обратите внимание на то, что директивы Ассемблера в машинные коды не транслируются, так как являются командами для транслятора, а не для процессора. Все адреса в файлах листинга являются *относительными* – отсчитываются от начального (нулевого) адреса текущей секции.

```
  2 00000000          ; Определить переменную "Размер стека" (1 К байт)
  3 00000000 00000400
                      Stack_Size
                              EQU             0x00000400
```

Исследуйте файл листинга стартового файла. Вначале с помощью директивы EQU (Эквивалентно) мы определили символическое имя переменной Stack_Size, задающей требуемый нам размер стека. Так как эта переменная объявлена только для транслятора, под нее не выделена память, хотя введенное нами значение отобразилось в столбце машинных кодов (3-я строка). Это свидетельствут о том, что транслятор воспринял эту переменную как константу и присвоил ей указанное нами значение.

```
 10 00000000          Stack_Mem
                              SPACE           Stack_Size  ; (1 К байт)
 11 00000400          ; Метка вершины стека (авто-декрементный стек)
 12 00000400          __initial_sp
```

Дальше мы объявили секцию данных и зарезервировали в ней 1Кбайт памяти с помощью директивы SPACE и преварительно определенного символического имени Stack_Size. Как видите, счетчик текущего адреса в секции данных автоматически получил нужное приращение (0x400), а следующая за ней метка вершины стека _initial_sp – относительное значение 0x400.

Так как выполнена только трансляция исходных файлов, а компоновка еще не выполнена, то фактические адреса размещения подпрограмм в файле листинга пока заменяются нулями, но место под них обязательно выделяется.

```
27 00000000 00000000
                    __Vectors
                              DCD             __initial_sp ; Вершина стека - Top of Stack
28 00000004 00000000          DCD             Reset_Handler ; Точка выхода в обработчик исключения

29 00000008         ; по сбросу процессора Reset Handler
30 00000008 00000000          DCD             NMI_Handler ; Точка входа в обработчик
31 0000000C         ; немаскируемого прерывания NMI
32 0000000C         ;       ...
33 0000000C         ; Далее по аналогии могут быть объявлены и остальные
34 0000000C         ; вектора обработчиков прерываний/исключений
35 0000000C         __Vectors_End
36 0000000C 0000000C
                    __Vectors_Size
                              EQU             __Vectors_End - __Vectors
```

С каждой директивой DCD резервирования в секции сброса (RESET) очередного вектора, счетчик адреса расположения данных в текущей секции инкрементируется на 4. Это позволяет транслятору точно вычислить размер таблицы векторов прерываний/исключений – 0x0C (переменная _Vectors_ Size). Напомним, что все содержимое этой секции будет автоматически размещено компоновщиком в начальной области кодовой памяти (с нулевого адреса). Дальше в стартовом файле определяется кодовая секция |.Text|, в которой располагаются процедуры обработчиков прерываний/исключений.

```
56 00000000          ; Передать управление пользовательской программе MyProg
57 00000000          ; Объявление точки входа в программу пользователя
58 00000000          ; (внешняя метка)
59 00000000                  IMPORT          MyProg
60 00000000          ; Псевдо-команда: загрузить адрес MyProg в регистр r0
61 00000000 4801             LDR             r0, =MyProg ; r0 <- адрес точки входа
62 00000002          ; Косвенная передача управления программе пользователя

63 00000002 4700             BX              r0          ; PC <- (r0)
64 00000004                  ENDP                        ; Конец процедуры Reset_Handler
```

Первая процедура – обработчик сброса процессора только передает управление программе пользователя. Для этого использована специальная псевдокоманда Ассемблера «`LDR r0, =MyProg`», которая загрузит внешний адрес начала пользовательской программы в регистр r0. С помощью команд косвенной передачи управления по содержимому этого регистра (BX r0) мы окажемся в начальной точке нашего приложения. Именно с этого места будет выполняться программа после компоновки приложения и запуска отладчика.

Итак, оба исходных файла проекта прошли этап трансляции. Можно выполнить компоновку проекта – построить проект. Нажмите клавишу  **«Build»** («**Построить проект**» (на панели инструментов и обратите внимание на сообщение в окне вывода «**Build Output**», которое говорит о том, что при создании выходного объектного файла ошибок и предупреждений не было.

```
Build target 'Target 1'
linking...
Program Size: Code=20 RO-data=12 RW-data=0 ZI-data=1024
".\Проба.axf" - 0 Error(s), 0 Warning(s).
```

В окне выводится также краткая информация об использованной в проекте памяти. Она означает следующее: объем в байтах: всего кода проекта Code=20; памяти только для чтения RO – data=12, неинициализированной памяти данных для чтения и записи RW-data=0; инициализированной нулями памяти данных ZI-data=1024.

В результате компоновки создается не только выходной объектный файл, который можно загрузить на выполнение или отладку, но и файл карты загрузки «.map» проекта. Можете, для начала, пропустить специфическую информацию о том, что находится в этом файле и вернуться к ней при необходимости. Прежде всего в файле «.map» Вы сможете найти информацию о *локальных и глобальных символах* Вашего проекта, их значении (Value), типе (Type), занимаемом объеме памяти в байтах (Size) и об объекте (секции), в котором они определены. Рассмотрим, для примера, таблицу глобальных символов нашего проекта. Среди них символ _Vectors, имеющий значение 0x0, тип Data и размер 4 байта (1 слово), расположенный в секции RESET.

Действительно, таблица векторов прерываний/исключений и начального адреса стека должна начинаться с нулевого адреса кодовой памяти.

```
  Global Symbols

  Symbol Name                              Value     Ov Type        Size  Object(Section)
  __Vectors                                0x00000000   Data           4  startup_1.o(RESET)
  Reset_Handler                            0x0000000d   Thumb Code     4  startup_1.o(.text)
  __Vectors_End                            0x0000000c   Data           0  startup_1.o(RESET)
  __Vectors_Size                           0x0000000c   Number         0  startup_1.o ABSOLUTE
  NMI_Handler                              0x00000011   Thumb Code     2  startup_1.o(.text)
  MyProg                                   0x00000019   Thumb Code     0  my_prog_1.o(MyCode)
```

Символы *Reset_Handler* и *NMI Handler* – начальные адреса (0xd и 0x11) соответствующих подпрограмм-обработчиков прерываний/исключений, занимающих в памяти 4 и 2 байта соответственно. Оба символа относятся к программному коду (тип – Thumb Code) и расположены в кодовой секции |.text|. Последний символ MyProg – начальный адрес приложения пользователя 0x19.

В этом же файле находится *карта памяти проекта*. Прежде всего отметим, что в любом проекте имеется *точка входа в проект* (Image Entry point), с которой начинается отладка проекта. В нашем случае – это процедура обслуживания прерывания по сбросу процессора (расположена по адресу Reset_Handler = 0xd), которая должна выполнить его инициализацию и передать управление приложению пользователя.

В карте памяти используются следующие обозначения:

- Load Region LR_1 – область загрузки;
- Execution Region ER_RO – область выполнения «Только для чтения»;
- Execution Region ER_RW – область выполнения «Для чтения и записи»;
- Execution Region ER_ZI – область выполнения «Для чтения и записи», инициализированная нулями.

```
Memory Map of the image

  Image Entry point : 0x0000000d

  Load Region LR_1 (Base: 0x00000000, Size: 0x00000020, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_RO (Base: 0x00000000, Size: 0x00000020, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x00000000   0x0000000c   Data   RO            7    RESET               startup_1.o
    0x0000000c   0x0000000c   Code   RO            8  * .text               startup_1.o
    0x00000018   0x00000006   Code   RO            1  * MyCode              my_prog_1.o


    Execution Region ER_RW (Base: 0x20000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)

    **** No section assigned to this execution region ****


    Execution Region ER_ZI (Base: 0x20000000, Size: 0x00000400, Max: 0xffffffff, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x20000000   0x00000400   Zero   RW            6    STACK               startup_1.o
```

Первые две области относятся к кодовой памяти (ПЗУ), а остальные – к памяти данных (ОЗУ). Область загрузки может содержать не только код программы, но и дополнительные данные, например таблицы констант, которыми должна быть проинициализирована определенная часть памяти данных (область инициализированных переменных). В нашем случае весь код попадает в область выполнения: это таблица векторов прерываний/исключений; код обработчиков прерываний/исключений; собственно программа пользователя. Область памяти данных для чтения и записи ER_RW пока нами не использована, а область памяти для чтения и записи, инициализированная нулями ER_IZ, – использована для размещения стека.

9.8 Как запустить программу на отладку? Приемы отладки.

Проект создан: все исходные программы прошли стадию трансляции и компоновки. Можно запустить процесс отладки: из меню «**Debug**» («**Отладка**») или щелкнув мышью в строке инструментов на пиктограмме  «**Start/Stop Debug Session**» «**Старт/Стоп отладочной сессии**».

Ранее мы показали, как установить опции целевого устройства, чтобы процесс отладки выполнялся в симуляторе. В этом случае, после запуска отладочной сессии код проекта будет загружен в программу-симулятор, и управление кодом будет передано отладчику (автоматически появится и строка инструментов отладчика).

В процессе загрузки кода будет автоматически сформирована таблица векторов прерываний/исключений (в нашем случае ее начальная часть), к которой процессор обращается в самом начале своей работы, в частности для инициализации указателя стека SP. После этого будет вызван обработчик прерывания по сбросу процессора Reset_Handler. Его начальный адрес и является *точкой входа в проект*.

```
60  ; Псевдо-команда: загрузить адрес MyProg в регистр r0
61                  LDR     r0, =MyProg   ; r0 <- адрес точки входа
62  ; Косвенная передача управления программе пользователя
63                  BX      r0            ; PC <- (r0)
```

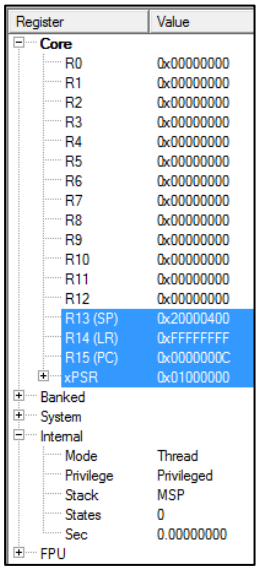
Так как обработчик состояния сброса процессора находится в стартовом файле, он будет активизирован (открыт), и курсор текущей команды, подлежащей выполнению (синий и желтый треугольнички), будет установлен на начальной команде обработчика.

Можно открыть и окно Дизассемблера из меню **View (Просмотр)**. В этом окне курсор (желтая стрелка) также будет показывать на команду, подлежащую выполнению.

Однако, в отличие от окна программы в исходных кодах, это будет уже не псевдокоманда, а реальная, созданная транслятором, команда процессора:

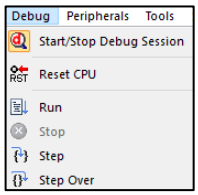
```
0x0000000C 4801      LDR          r0,[pc,#4]  ; @0x00000014
```

Таким образом, *точка входа в процесс отладки* – это начало процедуры обработки прерывания по сбросу процессора Reset_Handler (0xC). Обязательно откройте окно текущего содержимого регистров центрального процессора из меню **View** (**Просмотр**) или щелкнув на пиктограмме этой команды в строке инструментов отладчика  .

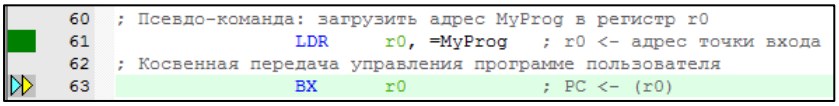


Именно в этом окне мы будем отслеживать ход выполнения программы. Отметим, что текущее содержимое регистров ЦПУ автоматически отображается *в шестнадцатеричной системе счисления*. Для раскрытия/сокрытия содержимого определенной группы регистров достаточно шелкнуть мышью на соответствующем прямоугольничке (символ «+» – регистры скрыты; «-» – открыты. В нашем примере открыто содержимое регистров процессорного ядра и внутренних регистров процессора, отвечающих за его режим работы. Вы видите, что указатель стека SP уже проинициализирован значением вершины стека 0x20000400, счетчик команд PC – начальным адресом обработчика Reset_Handler, а сам процессор находится в потоковом режиме работы Thread с привилегированным доступом ко всем ресурсам Privileged, что и должно быть по умолчанию. Бит, установленный в регистре состояния программы PSR – это тот, который отвечает за текущий набор команд (ARM или Thumb), с которым в данный момент работает процессор. Так как все процессоры Cortex-M всегда работают в одном и том же режиме Thumb-2 с автоматической генераций наиболее компактного кода, то по умолчанию установлен именно бит Thumb.

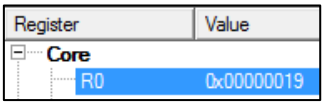
Теперь можно обратиться к командам управления отладкой, которые находятся в соответствующем меню, начальный фрагмент которого показан ниже, или к панели инструментов среды µVision  , которые более удобны, так как позволяют запустить нужную команду отладчика простым щелчком мыши. Первая команда меню **запускает/останавливает отладочную сессию**, вторая – вызывает **сброс процессора**, то есть переход на обработчик прерывания Reset_Handler, третья – позволяет запустить программу в **режиме непрерывного выполнения** (**прогона - RUN**), четвертая – **остановить** процесс непрерывного выполнения, пятая, который Вы будете пользоваться наиболее часто на этапе изучения команд процессора, – **выполнить одну текущую команду** – сделать шаг. Это так называемый *режим пошагового выполнения программы*, который наиболее удобен для начинающих. С остальными возможностями отладчика и его командами будем знакомиться постепенно.



Выполним нашу программу в пошаговом режиме работы. Сделаем один шаг. Курсор текущей команды, подлежащей выполнению, в окне с исходным кодом программы переместится, указывая на следующую команду, а уже выполненная команда будет отмечена слева зеленым прямоугольником (признаком выполнения).



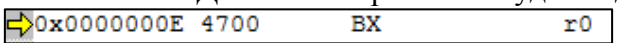
Так как эта псевдокоманда должна загрузить в регистр r0 адрес точки входа в программу приложения, обратим внимание на изменение содержимого регистра r0 в окне регистров. Оно изменилось (0x19), а факт изменения отмечен в окне синим фоном. Точно так

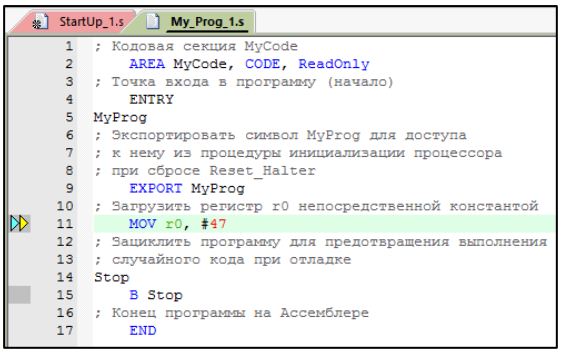


же изменилось и состояние счетчика команд PC, который теперь указывает на адрес очередной команды, подлежащей выполнению 

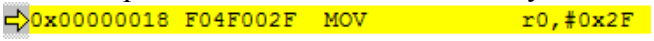
.

В окне Дизассемблера также будет виден адрес расположения этой команды.

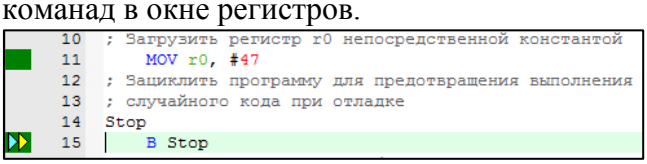




Сделаем еще один шаг. Мы должны перейти к программе пользователя. Среда µVision автоматически активизирует окно этой программы (имя файла выделяется зеленым фоном и подчеркиванием) и устанавливает курсор слева от команды, подлежащей выполнению. Это первая команда нашей простой программы MOV r0,#47. Если Вы обратитесь к окну Дизассемблера, то увидите, что курсор подлежащей выполнению команды также переместился и показывает на ту же команду.



Однако, обратите внимание на некоторое несоответствие. Адрес этой команды –не число, загруженное ранее в регистр r0 (0x19), а на единицу меньше. Дело в том, что раньше в процессорах ARM младшим битом в команде перехода устанавливался нужный набор команд процессора (0- ARM, 1- Thumb), в котором процессор должен работать после завершения перехода. Так как в процессорах Cortex-M переключений из одного набора команд в другой нет и процессор всегда работает в режиме Thumb-2, то для сохранения совместимости с предыдущими версиями ARM-процессоров адрес перехода всегда устанавливается на 1 больше, чем фактический адрес. В процессе выполнения команды перехода (в данном случае команды косвенной передачи управления BX r0 по адресу в регистре r0) «лишний» младший бит автоматически обнуляется, и переход выполняется на правильный адрес, о чем свидетельствует и новое содержимое счетчика команад в окне регистров.



Сделаем очередной шаг и увидим результат выполнения нашей программы – содержимое регистра r0 изменится и будет равно 0x2F=47d, как и требовалось. При очередном шаге выполняется команда передачи управления на эту же команду. Процесс выполнения программы зацикливается. Это сделано для того, чтобы в режиме прогона вы не попали в ситуацию выполнения «мусорного» (случайного) кода.

Содержимое счетчика команд не меняется, и процессор все время выполняет одну и ту же команду перехода на саму себя. Обратите внимание на то, что обе команды нашей программы помечены зеленым прямоугольником – значит выполнены.

9.9 Непосредственная и регистровая адресация операндов

Вычислительная мощность любого процессора зависит от имеющейся в его распоряжении системы команд, а она строится на базе принятых и реализованных разработчиками процессора *способов адресации операндов*. Под способом адресации операнда понимается *механизм кодирования операнда* в формате команды. Самым простым способом адресации является *непосредственная адресация*, когда в формате команды часть разрядов отводится для представления кода операции, а часть – для представления собственно *операнда-константы*:

Команда	
Поле кода операции	**Поле непосредственного операнда-константы**

Этот метод имеет естественные ограничения. В ARM-процессорах любая команда должна иметь фиксированный формат – 32 разряда. Процессоры Cortex-M допускают как 32-разрядный, так и 16-разрядный формат команд. Следовательно, непосредственный операнд не может занимать всю разрядную сетку команды и должен размещаться в поле меньшей разрядности (например, в младшем полуслове для 32-разрядной команды).

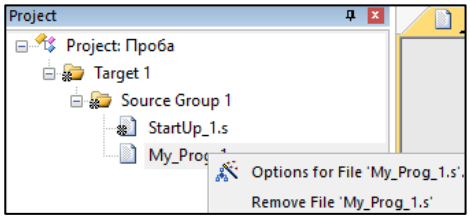
Так как большинство операций в ARM-процессорах выполняются над содержимым регистров, естественной является *регистровая адресация* операндов, когда в формате команды указывается не сам операнд, а адрес (внутренний код) регистра ЦПУ, в котором находится операнд:

Команда			
Поле кода операции	**Код регистра Rd**	**Код регистра Rn**	**Код регистра Rm**

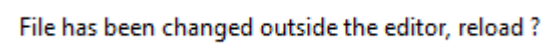
Мы имеем 16 регистров общего назначения ЦПУ r0-r15. Следовательно, для кодирования всех регистров ЦПУ (младшего и старшего банков) достаточно всего 4-х разрядов, а для кодирования регистров только младшего банка r0-r7 – всего 3-х разрядов. Напомним, что в общем случае команды процессора могут быть 3-операндные и 2-операндные. В первом случае поле в формате команды, используемое для кодирования двух регистров-источников и одного регистра приемника, имеет длину 3*4=12 бит, а во втором – 2*4=8 бит. Эти цифры уменьшаются при работе только с регистрами младшего банка (3*3=9 бит и 2*3=6 бит). Система команд процессоров ARM Cortex-M устроена так, что при работе с регистрами младшего банка в основном генерируются более компактные 16-разрядные команды набора Thumb, а при использовании хотя бы одного регистра из старшего регистрового банка, – 32-разрядные команды набора ARM.

Рекомендации по работе с исходной программой при ее модернизации:

1) Вы сколько угодно раз можете редактировать исходную программу, ассемблировать ее, компоновать, строить проект и запускать его на отладку. Имейте только в виду, что на диске при сохранении файлов останется только последняя версия Вашей программы, а все предыдущие будут потеряны.
2) Если Вы желаете сохранить исходную версию программы, поступите так: находясь в окне редактирования программы, выберите из меню «Файл» («File») команду «**Сохранить как**» («**Save as**») и сохраните текущий файл под новым именем, например, My_Prog_2.s. Вкладка с содержимым нового файла автоматически появится на экране. Можете приступать к редактированию.
3) Однако, к текущему проекту будет все еще подключен исходный файл My_Prog_1. Именно он будет транслироваться и компоноваться с остальными файлами проекта до тех пор, пока Вы не *отсоедините* его от проекта. Сделать это можно так: в окне проекта выделите файл, подлежащий отключению, щелкните правой клавишей мыши на его имени и из выпадающего контекстного меню выберите команду «**Отключить файл**» («**Remove File 'My_Prog_1.s**»). Он будет отключен от текущего проекта (увидите в окне проекта), но останется на диске в каталоге проекта и может быть при необходиомости подключен вновь.



4) После завершения редактирования нового файла и сохранения на диске, подключите его к проекту из контекстного меню исходной группы файлов «Source Group 1». Теперь можно выполнять трансляцию нового файла, просмотр листинга, построение и отладку всего проекта с новым исходным файлом.
5) Любой уже ненужный файл можно закрыть, щелкнув на его имени правой клавишей мыши. В появившемся контекстно-зависимом меню нужно выбрать команду «**Закрыть**» («**Close**»).
6) Если Вы просматривали файл листинга, а затем модернизировали исходный текст и перетранслировали программу, среда µVision уведомит Вас о изменении в этом файле: Файл был изменен извне. Перевывести? В случае Вашего согласия новая версия файла будет выведена на экран.



В задании ниже попробуйте разные варианты работы с исходными файлами. Для вышего удобства мы сохранили все модифицированные версии файлов (отличаются цифрами в конце имени файла). Если не справитесь сами, – можете подключить к проекту готовые файлы и поработать с ними.

1) Модернизируйте программу My_Prog_1 так, чтобы выполнялась последовательная загрузка в один и тот же регистр r3 нескольких непосредственных констант: 47, 48, 49, 50. Оттранслируйте ее. Откройте файл листинга, проанализируйте коды сгенерированных команд и выделите в них битовые поля, которые отводятся для размещения непосредственных операндов. Создайте выходные файлы проекта (постройте проект) и выполните его отладку.
2) Теперь сделайте последовательную загрузку этих же непосредственных констант в разные регистры ЦПУ r0, r1, r2, r3. Выполните трансляцию. Найдите в кодах команд поля, которые содержат адреса регистров-приемников данных.
3) Загрузите те же операнды в регистры старшего регистрового банка r9, r10, r11, r12. Изменилать ли технология адресации регистров-приемников?
4) Еще раз измените программу: загрузите регистры r0, r1 кодами 47, 48. Рассчитайте сумму содержимого этих регистров с записью результата в регистр r0 (ADD r0,r1), затем – сумму текущего содержимого r0 и r1 с записью результата в регистр r7 (ADD r7, r0, r1). Просмотрите файл листинга. Почему эти две команды ADD имеют разный формат?
5) Загрузите программу на отладку. Результат сложения соответствует ожидаемому?

```
10 00000000                     ; Загрузить регистр r3 последовательно несколькими
11 00000000                     ; непосредственными константой
12 00000000 F04F 032F           MOV             r3, #47
13 00000004 F04F 0330           MOV             r3, #48
14 00000008 F04F 0331           MOV             r3, #49
15 0000000C F04F 0332           MOV             r3, #50
```

1) Фрагмент файла листинга My_Prog_2.lst показывает, что генерируются 32-разрядные команды, в которых в старшем полуслове находится код операции, а в младшем – непосредственные данные. Они занимают младший байт младшего полуслова (разряды 7-0). Старший байт младшего полуслова (пока без детальных комментариев) содержит поле адреса регистра-назначения (младший нибл) – число 3.

2) Результат трансляции файла My_Prog_3.s выглядит так: старшее полуслово не изменилось (код операции), а младшее по-прежнему содержит в младшем байте непосредственные операнды, а в старшем байте – коды (адреса) регистров-приемников, соответствующие их номеру. Таким образом, в команде MOV одновременно используются два способа адресации – непосредственная операнда-источника и регистровая – операнда-приемника.

```
10 00000000                 ; Загрузить несколько регистров ЦПУ
11 00000000                 ; непосредственными данными
12 00000000 F04F 002F       MOV             r0, #47
13 00000004 F04F 0130       MOV             r1, #48
14 00000008 F04F 0231       MOV             r2, #49
15 0000000C F04F 0332       MOV             r3, #50
```

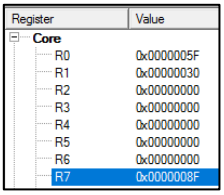
3) Файл листинга My_Prog_4.lst подтверждает, что в 4-битовом поле команды (младший нибл второго байта) действительно находится адрес регистра-приемника данных, соответствующий его номеру. Следовательно, технология регистровой адресации операндов распространяется на весь регистровый банк процессора.

```
10 00000000                 ; Загрузить несколько регистров ЦПУ
11 00000000                 ; непосредственными данными
12 00000000 F04F 092F       MOV             r9, #47
13 00000004 F04F 0A30       MOV             r10, #48
14 00000008 F04F 0B31       MOV             r11, #49
15 0000000C F04F 0C32       MOV             r12, #50
```

4) Из файла листинга My_Prog_5.lst следует, что первую двухоперандную команду сложения транслятору удалось закодировать более компактным 16-разрядным кодом Thumb – под поле адресов регистра-источника и приемника отведено только 2*3=6 бит. Вторая 3-операндная команда сложения является уже 32-разрядной командой набора ARM и в ней имеются три 4-битовых поля адресов первого и второго операнда-источника и операнда-приемника. Можете поэкспериментировать самостоятельно, чтобы определить, где в команде реально находятся эти поля.

```
10 00000000                 ; Инициализация регистров r0, r1
11 00000000 F04F 002F       MOV             r0, #47
12 00000004 F04F 0130       MOV             r1, #48
13 00000008                 ; Найти сумму этих двух констант. Сохранить в r0
14 00000008 4408            ADD             r0,r1
15 0000000A                 ; Найти сумму нового значения r0 и r1
16 0000000A                 ; Сохранить в регистре r7
17 0000000A EB00 0701       ADD             r7,r0,r1
```

5) Да. Первое сложение должно дать результат 0x2F+0x30=5F, что соответствует содержимому регистра r0. Второй командой к этому значению добавляется число в регистре r1 (0x30) и результат (0x5F+0x30=0x8F) сохраняется в регистре r7:



Мы познакомились с двумя типовыми способами адресации операндов: непосредственной и регистровой. Поняли, что транслятор в зависимости от типа операции и используемых регистров выбирает наиболее экономичный способ кодирования команд, используя при этом как 16-разрядные, так и 32-разрядные команды *унифицированного набора команд Thumb-2*. При этом никаких дополнительных директив переключения между наборами команд Ассемблеру давать не нужно.

Даже из простых примеров видно, что технология кодирования команд процессоров Cortex-M достаточно сложна, особенно в том случае, когда выполняются попутные операции сдвига и требуется закодировать не только адрес регистра-источника, но и тип сдвига операнда и число разрядов сдвига. Поэтому, далее в книге мы будем *полностью доверять транслятору* и проверять работу программ «по факту», т.е. по результату, не вдаваясь детально в особенности размещения данных в отдельных полях команды.

Очень надеемся, что столь подробное изложение основ работы в интегрированной среде разработки µVision в этой главе не охладит пыл читателя в изучении системы команд процессора и методов эффективного программирования практических задач. В следующих главах будем полностью опираться на приобретенный здесь опыт.

9.10 Как заменить шаблон обработчика прерывания/исключения реальной процедурой?

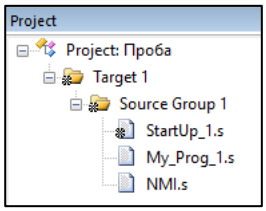
В этой главе мы рассмотрели структуру стартового файла, пригодную для разработчиков программного обеспечения на Ассемблере. В его основе лежит таблица векторов прерываний/исключений, которая должна быть размещена в начале кодовой секции перед выполнением любой программы пользователя. Осталось неясно, как процедуру «шаблон» заменить реальной процедурой. Покажем это на примере процедуры обслуживания немаскируемого прерывания NMI.

ARM ASM

Прежде всего на основе исходного файла StartUp1.s создадим новый файл с нужной нам процедурой обслуживания прерывания NMI.s. Проще всего это сделать с использованием команды «Сохранить как» с последующим редактированием копии файла. Оставим в файле только кодовую секцию |.text|, которая предназначена как раз для размещения процедур обработчиков прерываний/исключений. Добавим в текст (для иллюстрации изменений) две команды NOP (пустые операции). В отличие от процедуры шаблона в директиве объявления глобальной метки NMI_Handler, *уберем опцию* [WEAK], то есть сделаем эту метку и помеченную ею процедуру «**сильной**», чтобы на стадии сборки всего проекта она могла бы автоматически заменить «процедуру-шаблон».

```
1  ; Пример обработчика прерывания NMI для замены шаблона
2  ; Объявление кодовой секции для размещения
3  ; подпрограмм обработчиков прерываний/исключений
4                  AREA    |.text|, CODE, READONLY
5  ; Обработчик немаскируемого прерывания NMI
6  NMI_Handler     PROC
7  ; Объявить процедуру NMI_Handler общедоступной
8  ; Опция "Слабая метка" [WEAK] отсутствует
9                  EXPORT  NMI_Handler
10 ; Для примера две "пустые операции"
11                 NOP
12                 NOP
13 ; Зациклить программу обработчика
14                 B       .
15                 ENDP    ; Конец процедуры NMI_Handler
16 ; Конец ассемблерного текста стартового модуля
17                 END
```

Прежде, чем выполнить трансляцию нового файла, его нужно подключить к проекту. Сделаем это. Теперь в составе проекта «Проба» уже три файла: заимствованный нами из каталога CPU_1 стартовый файл, который, в том числе, содержит «процедуру-шаблон», файл с программой пользователя и только что подключенный к проекту файл NMI.s с реальной процедурой обслуживания немаскируемого прерывания.




```
 1 00000000         ; Пример обработчика прерывания NMI для замены шаблона
 2 00000000         ; Объявление кодовой секции для размещения
 3 00000000         ; подпрограмм обработчиков прерываний/исключений
 4 00000000                 AREA            |.text|, CODE, READONLY
 5 00000000         ; Обработчик немаскируемого прерывания NMI
 6 00000000         NMI_Handler
                            PROC
 7 00000000         ; Объявить процедуру NMI_Handler общедоступной
 8 00000000         ; Опция "Слабая метка" [WEAK] отсутствует
 9 00000000                 EXPORT          NMI_Handler
10 00000000         ; Для примера две "пустые операции"
11 00000000 BF00            NOP
12 00000002 BF00            NOP
13 00000004         ; Зациклить программу обработчика
14 00000004 E7FE            B               .
15 00000006                 ENDP                        ; Конец процедуры NMI_Handler
16 00000006         ; Конец ассемблерного текста стартового модуля
17 00000006                 END
```

Выполним трансляцию нового файла и просмотрим файл листинга. Действительно, процедура обслуживания прерывания NMI изменилась (в нее включены две команды NOP).

При этом результаты трансляции двух остальных файлов, естественно, не меняются. Изменения можно будет увидеть только после компоновки проекта и загрузки его на отладку. В процессе компоновки «процедура-шаблон» будет автоматически заменена реальной процедурой.

Но увидеть это можно будет только в файле карты загрузки проекта проба.map, из которого следует, что новая процедура имеет адрес загрузки 0x19 и занимает шесть байт в памяти (как и должно быть в соответствии с файлом листинга).

```
NMI_Handler                             0x00000019   Thumb Code     6  nmi.o(.text)
```

Мы уже отмечали, что адрес перехода, загружаемый в счетчик команд, должен быть на 1 больше фактического адреса размещения процедуры. Это бит, подтверждающий, что после перехода режим выполнения процессором набора команд Thumb-2 должен быть сохранен. Он будет автоматически сброшен при выполнении перехода. Именно поэтому в таблице символов точка входа в процедуру NMI имеет значение 0x19, а не 0x18.

Таким образом, Ассемблер ARMASM допускает отладку любых программ с процедурами-шаблонами обработчиков прерываний/исключений. Как только реальные обработчики понадобятся – их можно будет легко подключить к проекту.

Список рекомендуемой литературы

1) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
2) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.
3) Электронный ресурс http://www.keil.com

10 ДОСТУП К ДАННЫМ В РЕГИСТАХ ЦПУ И ПАМЯТИ

Оглавление

10.1 Загрузка регистров ЦПУ. Краткая справка

В большинстве случаев программисты стараются полностью использовать разрядность процессора – работают с 32-разрядными словами, загружая их в 32-разрядные регистры ЦПУ на обработку. Возможна работа и с более короткими данными – полусловами и байтами. В последнем случае процессор будет автоматически выполнять преобразование более коротких данных в «длинные» 32-разрядные слова. В зависимости от типа команды и использованных в ней операндов транслятор может генерировать либо 16-, либо 32-разрядные команды унифицированного набора команд Thumb-2.

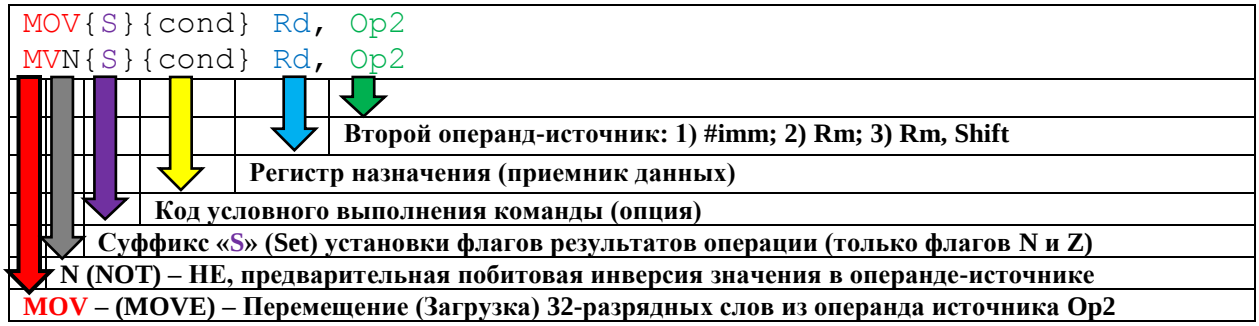
Краткий список наиболее употребительных команд и псевдокоманд загрузки регистров ЦПУ представлен ниже – табл. 10.1. Все они рассматриваются далее более подробно.

Таблица 10.1 Наиболее употребительные команды и псевдокоманды загрузки регистров

Синтаксис	Способ адресации операнда-источника	Описание
MOV Rd, Op2 **Op2 – Rn;** **Op2 - #Const** **Op2 – Rn, Shift**	***Второй универсальный операнд:*** ***Регистровая*** ***Непосредственная*** ***Регистровая с предварительным сдвигом операнда на заданное число разрядов***	**Пересылка из регистра в регистр** **Загрузка константы** **Пересылка из регистра с предварительной обработкой в сдвиговом регистре** **(Вырабатывает флаги N, Z!)**
Псевдокоманда: **загрузить адрес по метке** **ADR Rd, Label**	***Непосредственная – по содержимому счетчика команд PC и непосредственному смещению*** **Ассемблер автоматически рассчитывает величину смещения до метки и генерирует команду** **ADR Rd, (PC, #offset)**	**Непосредственная адресация, с расчетом адреса в памяти по текущему содержимому счетчика команд и непосредственному смещению до метки Label** **Rd ← ((PC) + #offset)**
LDR Rd, [Rn]	***Косвенная*** **Фактически генерируется команда базовой адресации с нулевым непосредственным смещением:** **LDR Rd, [Rn, #0]**	**Косвенная адресация по текущему содержимому регистра-указателя Rn (базового регистра)** **Rd ← [(Rn)]**
LDR Rd, [Rn, #offset]	***Базовая с непосредственным смещением указателя***	**Базовая адресация с непосредственным смещением указателя относительно базы** **Rd ← [(Rn) + #offset]**
Использование в качестве базового регистра счетчика команд PC или указателя стека SP		
LDR Rd, [PC, #offset]	***Относительная – по содержимому счетчика команд PC и непосредственному смещению***	**Базовая с непосредственным смещением относительно содержимого счетчика команд PC** **Rd ← [(PC) + #offset]**
LDR Rd, [SP, #offset]	***Стековая – по текущему содержимому указателю стека SP и непосредственному смещению***	**Базовая с непосредственным смещением относительно указателя стека SP** **Rd ← [(SP) + #offset]**
LDR Rd, [Rn, Rm]	***Базово-индексная (Rn – базовый, Rm – индексный регистры)***	**Базовая со смещением, заданным содержимым индексного регистра** **Rd ← [(Rn) + (Rm)]**
Псевдокоманда: считать код по метке **LDR Rd, Label**	***Относительная – по содержимому счетчика команд PC и непосредственному смещению*** **Ассемблер автоматически рассчитывает смещение до метки относительно PC и генерирует команду:** **LDR Rd, [PC, #offset]**	**Косвенная с предварительным смещением относительно текущего содержимого счетчика команд PC** **Rd ← [(PC) + #offset]**
Псевдокоманда: **загрузить 32-разрядное слово** **LDR Rd, =Value**	***Загрузка данных с возможным созданием литерального пула в кодовой памяти*** **В зависимости от величины значения Value генерируется либо команда непосредственной загрузки MOV Rd, #Value,** **либо команда косвенной загрузки с предсмещением относительно PC** **LDR Rd, [PC, #offset]**	**Загрузка непосредственных данных или данных из предварительно автоматически созданного на этапе трансляции литерального пула в кодовой памяти (из области констант)**

10.2 Команды пересылки данных

Для инициализации регистров ЦПУ начальными значениями часто применяется команда MOV с непосредственным операндом #imm. Она является частным случаем загрузки регистра содержимым универсального второго операнда Op2, который представляется в трех вариантах:



Здесь и далее при изучении системы команд мы будем выделять красным цветом корень в мнемокоде команды, синим цветом – операнд/ы-приемники, зеленым цветом – операнды-источники данных. Дополнительными цветами (черным, фиолетовым и др.) будем отмечать префиксы и суффиксы, которые модифицируют основную операцию. Если соответствующая опция является необязательной, то она помещается в фигурные скобки "{}". Так, в нашем случае опция «S» позволяет дополнительно установить флаги N и Z в регистре статуса программы пользователя, а опция {cond} может быть одним из стандартных условий выполнения данной команды. Напомним, что если указанное Вами условие ложно, то операция не выполняется, и процессор автоматически переходит к выполнению следующей команды. Обратите внимание на то, что в отличие от многих других процессоров, в которых операции пересылки данных не могут формировать флаги, в процессорах ARM *это возможно* (могут модифицироваться два флага – отрицательного результата N и нуля – Z).

Второй операнд Op2 в этой команде универсальный: непосредственный операнд #imm, заданный любым арифметическим выражением; регистр-источник данных или регистр-источник, содержимое которого перед загрузкой подвергается дополнительной обработке в кольцевом сдвиговом регистре ЦПУ. Суффикс «**N**» позволяет выполнить загрузку данных с еще одной попутной операцией – побитовом инвертированием значения второго операнда. По сути – это будет уже *логическая операция побитового «НЕ»* с загрузкой результата в регистр назначения.

Мы уже упоминали о том, что не все 32-разрядные константы могут быть заданы в качестве непосредственных операндов, а только такие, значения которых могут быть «сжаты» для размещения в формате 32-разрядной команды (см. 9.9).

Интересно, что транслятор с Ассемблера автоматически распознает среди всех констант 16-разрядные константы и генерирует специальную версию команды пересылки, которая обеспечивает загрузку данных только в младшее полуслово регистра назначения с автоматической очисткой старшего полуслова:

```
MOVW{cond} Rd, #imm16
```

Дополнительный суффикс «W» (может быть опущен), указывает на то, что загружается 16-разрядная константа (полуслово). Вместе с представленной ниже командой загрузки 16-разрядной константы в старшее полуслово регистра назначения программист получает возможность последовательной инициализации любого регистра ЦПУ любой 32-разрядной константой независимо от ее значения. Единственное, что нужно сделать – это разбить константу на две составляющие, младшее и старшее

полуслово, и последовательно загрузить в регистр назначения сначала младшее полуслово (командой MOVW), а затем старшее (командой MOVT). Синтаксис команд загрузки 16-разрядных констант:

```
MOVW{cond} Rd, #imm16
MOVT{cond} Rd, #imm16
```

16-разрядный операнд-источник данных (полуслово)

Регистр назначения (приемник данных)

Код условного выполнения команды (опция)

W - Младшего полуслова с автоматической очисткой старшего

T - Старшего полуслова с сохранением значения младшего полуслова. Флаги не модифицируются

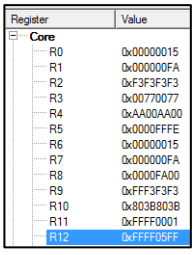
MOV – (MOVE) – Перемещение (Загрузка) 16-разрядного полуслова в регистр назначения

```
1  ; Объявить кодовую секцию MyCode
2        AREA MyCode, CODE, ReadOnly
3  ; Объявить точку входа в программу
4        ENTRY
5  ; Объвить точку входа глобальной переменной
6        EXPORT MyProg
7  MyProg
8  ; Загрузка регистров общего назначения
9  ; непосредственными константами
10       MOV r0, #21
11       MOV r1, #0xfa
12       MOV r2, #0xf3f3f3f3
13       MOV r3, #0x00770077
14       MOV r4, #0xAA00AA00
15       MOV r5, #65534
16 ; Загрузка из других регистров ЦПУ
17       MOV r6,r0
18       MOV r7,r1
19 ; Загрузка из регистров с предварительным сдвигом
20       MOV r8,r1,LSL #8
21       MOV r9,r2,ASR #8
22       MOV r10,r3,ROR #1
23 ; Загрузка побитово-проинвертированных значений
24       MVN r11,r5
25       MVN r12,r1,LSL #8
26 ; Последовательная загрузка 32-разрядных слов
27   ; Загрузка только младших полуслов
28       MOV r2,#2_10101010
29       MOVW r3,#65535
30   ; Загрузка только старших полуслов
31       MOVT r2, #2_11101110
32       MOVT r3, #0xFF
33 ; Ограничение дальнейшего выполнения программы
34 Stop  B Stop
35 ; Конец ассемблерного текста
36       END
```

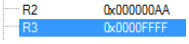
Проверим, как работают команды загрузки регистров ЦПУ (проект CPU_1 с ранее описанным стартовым файлом StartUP_1.s). Программа пользователя MyProg_1.s сначала выполняет загрузку регистров различными непосредственными константами, затем загрузку из других регистров процессора, уже содержащих данные и, наконец, загрузку из регистров процессора с предварительной обработкой операндов в кольцевом сдвиговом регистре. Первая команда из этой группы выполняет предварительный логический сдвиг влево на 8 разрядов, вторая – арифметический сдвиг вправо на столько же разрядов, а последняя – циклический сдвиг на 1 разряд вправо.

Дальше демонстрируется выполнение операции логического побитового «НЕ» второго операнда с загрузкой результата по месту назначения. В конце программы выполняется последовательная загрузка 32-разрядных констант с помощью двух команд загрузки сначала младшего, а затем старшего полуслов. В программе демонстрируются также способы ввода непосредственных констант в разных системах счисления.

Выполним трансляцию файлов проекта и запустим программу на выполнение в пошаговом режиме. Проследим за тем, как меняются значения в регистрах назначения.

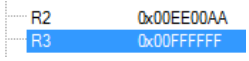


Убедимся, что загрузка регистров (r0-r12) 32-разрядными операндами выполняется правильно. Для этого обратимся к «Регистровому окну», в котором по умолчанию все значения отображаются в шестнадцатеричной системе счисления, или проверим текущее содержимое каждого регистра непосредственно в окне исходной программы, наводя курсор на символическое имя нужного регистра – его текущее содержимое будет тут же выдано в окно исходной программы.

Продолжим пошаговое выполнение программы и проследим за тем, как выполняется последовательная загрузка регистров (r2, r3) двумя полусловами. Вначале загружаются младшие полуслова, а старшие автоматически обнуляются: 

.

Затем выполняется «дозагрузка» старших полуслов при сохранении значений младших:



Как видите, последовательная загрузка 32-разрядного слова работает.

1) Докажите, что команда в строке 21 программы выполняется правильно?

2) Если в команде загрузки есть две попутные операции – сдвига и побитового инвертирования, то какое действие выполняется первым? Докажите на примере команды в строка 25 программы.

3) Какая технология кодирования 32-разрядных операндов использована в командах в строках 10-14? Проверьте Ваши предположения, изучив файл листинга.
4) Теперь попробуйте задать произвольную 32-разрядную константу, которая не может быть закодирована методами выше. Оттранслируйте программу. Что означает полученное сообщение об ошибке в окне вывода?
5) Как поступить в этом случае? Измените программу и выполните ее отладку.
6) Какую команду нужно использовать, чтобы побитово проинвертировать содержимое регистра r1, т.е. фактически выполнить команду «Логическое НЕ»? Проверьте в отладчике.

1) В регистр r2 было загружено число 0xF3F3F3F3, которое процессор рассматривает как 32-разрядное отрицательное (старший бит равен 1). Следовательно, при арифметическом сдвиге вправо на 8 разрядов младший байт 0xF3 будет полностью вытеснен за пределы разрядной сетки и потерян, а в старшие 8 разрядов будет «расширен» знаковый разряд исходного числа, т.е. «1». Получим 0xFFF3F3F3.

2) Вначале выполняется операция извлечения исходных данных из регистра-источника (0xFA) и обработка в регистре сдвига с получением числа 0xFA00. После этого результат побитово инвертируется и сохраняется по месту назначения (0xFFFF05FF).

3) Фрагмент файла листинга свидетельствует о том, что константы могут быть заданы двумя шестнадцатеричными цифрами «XY» и величиной сдвига этого кода влево, а также комбинациями двух HEX-цифр: 0xXYXYXYXY, 0x00XY00XY или 0xXY00XY00. Шифрация констант выполняется транслятором автоматически.

```
 8 00000000          ; Загрузка регистров общего назначения
 9 00000000          ; непосредственными константами
10 00000000 F04F 0015        MOV             r0, #21
11 00000004 F04F 01FA        MOV             r1, #0xFA
12 00000008 F04F 32F3        MOV             r2, #0xF3F3F3F3
13 0000000C F04F 1377        MOV             r3, #0x00770077
14 00000010 F04F 24AA        MOV             r4, #0xAA00AA00
15 00000014 F64F 75FE        MOV             r5, #65534
```

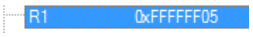
4) Изменим строку 10 программы: `10  MOV r0, #0xAB3477CD` . При трансляции исходного файла получим в окне вывода сообщение, которое говорит о том, что

```
Build Output
*** Using Compiler 'V5.05 update 2 (build 169)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
assembling MyProg_1.s...
MyProg_1.s(10): error: A1871E: Immediate 0xAB3477CD cannot be represented by 0-255 shifted left by 0-23 or duplicated in all, odd or even bytes
"MyProg_1.s" - 1 Error(s), 0 Warning(s).
```

введенная нами константа 0xAB3477CD не может быть закодирована с использованием возможных для этой команды методов кодирования: байта 0-255 и дополнительного сдвига влево на (0-23) разряда; байта, который «размножается» на все, четные или нечетные байты.

5) Можно разбить константу на два полуслова и загрузить полуслова по очереди. Дальше мы покажем, как эту операцию можно выполнить другим способом.

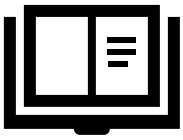
```
; Загрузка регистров общего назначения
; непосредственными константами
      MOV r0, #0x77CD
      MOVT r0, #0xAB34
```

- Команду MOVN r1, r1. Например если добавить ее в программу, то результат загрузки регистра r1 будет 

. Мы убедились, что система команд *допускает использование одного и того же регистра в качестве источника и приемника операции*, если предыдущее значение в нем больше не требуется для продолжения расчетов.

```
11      MOV r1, #0xFA
12      MVN r1,r1
```

10.3 Загрузка 32-разрядных констант. Относительная адресация по содержимому счетчика команд. Литеральные пулы.



Последовательная загрузка 32-разрядных констант младшим и старшим полусловами не удобна для программистов. Однако, в 32-разрядном процессоре нужна именно такая инициализация. При попытке создать команду непосредственной адресации 32-разрядных операндов мы сталкиваемся с принципиальным ограничением архитектуры RISC-процессоров Reduced Instruction Set Computer (компьютеров с сокращенным набором команд), в строгом соответствии с которой должно работать процессорное ядро Cortex-M3/M4. По этой идеологии, все команды должны иметь одинаковый формат (32 разряда) и выполняться одинаково быстро – за один такт. Однако, при использовании 32-разрядного непосредственного операнда в формате 32-разрядной команды не остается ни одного свободного разряда для кодирования операции. Выход из этого «тупика» разработчики ARM нашли оригинальный: предложили использовать загрузку таких «длинных» констант с использованием *относительной адресации по текущему состоянию счетчика команд PC и дополнительному непосредственному смещению* до места расположения константы в кодовой памяти. Проблема выполнения такой команды за один такт была решена аппаратно за счет введения в архитектуру процессора сразу двух шин, подключенных к кодовой памяти - ICode (шины считывания инструкций) и DCode (шины считывания данных), которые при выполнении команд загрузки «длинных» констант *работают параллельно*. В результате чего константа может «считываться» из памяти программ на фоне выполнения кода команды загрузки, не замедляя работу конвейера команд (см. 5.2.4). Потребовалось усовершенствовать и транслятор с Ассемблера, чтобы он мог вместе с командами «длинной» загрузки автоматически генерировать и сами константы в кодовой памяти. Для реализации этой технологии в язык Ассемблер ARMASM включена специальная **псевдокоманда «длинной» загрузки регистров** ЦПУ:

```
LDR Rd, =Const
```

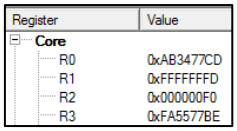
Попробуем использовать ее на практике (проект CPU_2): В состав проекта входит тот же стартовый файл StartUp_1.s и файл пользователя MyProg_2.s. В нем выполняется инициализация первых четырех регистров процессора начальными значениями. Выполните трансляцию исходного файла и изучите файл листинга (показан его фрагмент): все псевдокоманды преобразованы в машинные коды и могут быть выполнены.

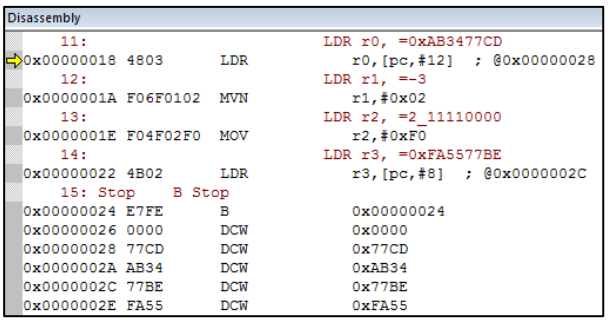
StartUp_1.s | MyProg_2.s

```
1  ; Объявить кодовую секцию MyCode
2        AREA MyCode, CODE, ReadOnly
3  ; Объявить точку входа в программу
4        ENTRY
5  ; Объвить точку входа глобальной переменной
6        EXPORT MyProg
7  MyProg
8  ; Загрузка регистров общего назначения
9  ; константами с использованием
10 ; псевдо-команд загрузки
11       LDR r0, =0xAB3477CD
12       LDR r1, =-3
13       LDR r2, =2_11110000
14       LDR r3, =0xFA5577BE
15 Stop  B Stop
16       ALIGN
17 ; Конец ассемблерного текста
18       END
```

```
 8 00000000          ; Загрузка регистров общего назначения
 9 00000000          ; константами с использованием
10 00000000          ; псевдо-команд загрузки
11 00000000 4803            LDR             r0, =0xAB3477CD
12 00000002 F06F 0102       LDR             r1, =-3
13 00000006 F04F 02F0       LDR             r2, =2_11110000
14 0000000A 4B02            LDR             r3, =0xFA5577BE
```

Проверим это. Создадим выходной объектный файл проекта и запустим процесс отладки в симуляторе. Выполним программу в пошаговом режиме и проанализируем результат в окне регистров: загрузка выполнена абсолютно правильно. Но почему же тогда в файле листинга команды имеют не только разную кодировку, но и разный формат (16 и 32 разряда)?





Ответ на этот вопрос становится понятным, если открыть окно дизассемблера и просмотреть реально создаваемый транслятором код. Красным цветом в этом окне выделен исходный код программы, написанный с использованием псевдокоманд, а черным – реально сгенерированный код, который будет на самом деле выполнять процессор. Так, вместо псевдокоманды LDR r0, =0xAB3477CD процессор будет выполнять команду, LDR r0, [pc, #12], входящую в его систему команд. Это команда косвенной загрузки регистра r0 из ячейки памяти, адрес которой определяется как текущее содержимое счетчика команд PC плюс указанное после запятой смещение: (PC)+#12 (указывается в десятичной системе счисления). После точки с запятой дизассемблер подсказывает, по какому реальному адресу последует обращение за данными: @0x00000028 (этот адрес, как и адрес размещения кодов команд в самом левом столбце окна, указывается в шестнадцатеричной системе счисления).

Такой способ адресации данных называется *относительной адресацией по содержимому счетчика команд PC*. Возникает естественный вопрос: а что считается текущим содержимым счетчика команд PC? Это - *начальный адрес возможного размещения в памяти следующей команды, но обязательно кратный 4* (двукратно четный). Проверим это на примере первой команды `LDR r0, [pc, #12]`: она является 16-разрядной и расположена в памяти по адресу 0x18. Следующая команда будет находиться по адресу 0x18+2= 0x1A, который не выровнен по границе полного слова (не кратен 4). В качестве текущего значения PC принимается очередной дважды четный адрес 0x18+4=0x1C. Добавим указанное в команде смещение 12d=0x0C, получим адрес 0x28, который Дизассемблер и указал в комментарии к этой команде. Что за данные располагаются по этому адресу? Мы видим, что с помощью директив размещения в кодовой памяти константы-полуслова DCW транслятор разместил здесь сначала код 0x77CD, а затем код 0xAB34, то есть заданную нами 32-разрядную константу 0xAB3477CD.

При использовании псевдокоманды LDR Rd,=literal загрузки данных в регистр ЦПУ транслятор *автоматически* создает в кодовой памяти так называемый *литеральный пул*, то есть область памяти, содержащую значение требуемой константы. В процессе выполнения команды относительной загрузки данных по содержимому PC выполняются два действия: 1) по кодовой шине ICode, подключенной к памяти программ, считывается код операции, определяется смещение и рассчитывается эффективный адрес доступа к данным в программной памяти (PC+offset); 2) эффективный адрес доступа выставляется на шину данных DCode и 32-разрядный код данных считывается в процессор и загружается в регистр-приемник ЦПУ.

Обе эти операции выполняются параллельно, следовательно, вся команда загрузки регистра выполняется за один такт, как и положено в процессорах с RISC-архитектурой.

Цена за решение проблемы загрузки 32-разрядных кодов – это вставки в программный код – литеральные пулы, являющиеся по существу константами, автоматически размещенными в кодовой памяти транслятором и используемыми в командах загрузки слов с относительной адресацией по текущему содержимому PC.

В псевдокоманде загрузки регистра ЦПУ 32-разрядным словом LDR Rd,=literal после знака равенства *символ непосредственного операнда «#» не ставится*. Константа может быть задана в любой системе счисления или арифметическим выражением, компоненты которого определены заранее.

Продолжим исследование нашей программы. Обратим внимание на то, что две последующие команды загрузки превратились в обычные команды MVN и MOV, рассмотренные нами ранее. Таким образом, транслятор с Ассемблера является *интеллектуальным*: он автоматически генерирует код, который может быть выполнен быстрее и не требует размещения в кодовой памяти дополнительных констант в виде литеральных пулов, экономя также память системы. Последняя псевдокоманда заменяется командой относительной загрузки по содержимому PC с генерацией еще одной константы. Она размещается вслед за уже созданной константой, то есть литеральный пул по мере трансляции программы автоматически расширяется (см. содержимое окна дизассемблера).

Заметьте, что между последней командой программы и литеральным пулом транслятор вставил нулевое *полуслово-шаблон* (директива размещения полуслова в кодовой памяти DCW 0x0000). Это происходит автоматически для того, чтобы все данные в литеральном пуле *были выровнены по границе полного 32-разрядного слова*. Только в этом случае доступ к константам в кодовой памяти по шине DCode будет оптимизирован, и любая команда загрузки регистра будет выполняться за один такт. О том, что транслятор выполнил вставку такого шаблона, Вы получите предупреждение в окне вывода при трансляции исходного файла: «Добавлено 2 байта шаблона по адресу 0xe».

```
Build Output
*** Using Compiler 'V5.05 update 2 (build 169)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
assembling MyProg_2.s...
MyProg_2.s(17): warning: A1581W: Added 2 bytes of padding at address 0xe
"MyProg_2.s" - 0 Error(s), 1 Warning(s).
```

Предупреждение не является ошибкой, и с проектом можно работать дальше. Впрочем, если хотите избавиться от предупреждения, используйте в конце своей программы директиву выравнивания последующего кода по границе полного слова ALIGN (это и будет адрес размещения литерального пула).

1) Действительно ли команда MVN r1, #02 эквивалентна команде LDR r1,=-3?

2) В чем преимущество генерации команды MOV r2, #0xF0 по сравнению с генерацией команды LDR r2,= literal? Ведь она является 32-разрядной командой из набора ARM, то есть более длинной?

3) Правильно ли транслятор сгенерировал смещение в последней команде?
4) Уберите признак комментария перед директивой ALIGN и выполните повторную трансляцию программы. Что изменилось в файлах листинга?
5) Попробуйте задать константу арифметическим выражением, компоненты которого определены. Программа работает?

1) Да, команда MVN побитово инвертирует значение в операнде-источнике 0x2 и загружает результат в регистр-приемник. Получаем число 0xFFFFFFFD, что в точности соответствует числу (-3) в дополнительном коде.

2) Команда относительной загрузки по содержимому счетчика команд действительно короче – 2 байта. Но не забывайте о том, что в кодовой памяти потребуется разместить литеральный пул с константой – 32-разрядное слово. Общие затраты кодовой памяти будут на 2 байта больше. Именно поэтому транслятор автоматически выбирает более эффективный вариант.

3) Да. Адрес размещения текущей команды 0x22, а следующей – 0x24 (так как текущая – 16-разрядная). Он удовлетворяет условию «дважды четный» и принимается за текущее содержимое PC. Добавляя смещение #8, получим 0x24+0x08=0x2C. Именно по этому адресу и размещена в кодовой памяти константа 0xFA5577BE.

```
15 0000000C E7FE    Stop    B                Stop
16 0000000E         ;   ALIGN
17 0000000E         ; Конец ассемблерного текста
18 0000000E                 END
           00 00 AB3477CD
           FA5577BE
```

```
15 0000000C E7FE    Stop    B                Stop
16 0000000E 00 00           ALIGN
17 00000010         ; Конец ассемблерного текста
18 00000010                 END
           AB3477CD
           FA5577BE
```

4) В первом случае транслятор выполняет работу за программиста автоматически, размещая нулевое полуслово-шаблон после последнего кода программы для того, чтобы литеральный пул был выровнен по дважды четному адресу. Во втором случае эту же операцию выполняет сам программист, зная, что он использовал псевдокоманды загрузки регистров ЦПУ, которые будут сопровождаться авто-генерацией литеральных пулов. В последнем случае транслятор не выдает предупреждений.

5) Да. Например, введите в качестве константы выражение: 2_1100+37. Сумма двоичного и десятичного чисел будет равна 12+37=49=0x31. Транслятор вычислит выражение сам и сгенерирует правильную команду

```
   17:                          LDR r4, =2_1100+37
0x00000024 F04F0431  MOV             r4,#0x31
```

10.4 Директива принудительного размещения литеральных пулов

ARM ASM

По умолчанию транслятор размещает «литеральные пулы» в конце текущей кодовой секции. Предполагается, что зона констант находится недалеко от места размещения собственно программного кода, поэтому обратиться к ней можно с использованием относительной адресации – по базовому адресу в счетчике команд PC и непосредственному смещению #offset, величина которого находится в диапазоне (-255÷4095). Такое 12-разрядное смещение можно разместить в 32-разрядном поле команды Thumb-2.

```
;*****************************************************
; Пример принудительного размещения литеральных пулов
; директивой Ассемблера LTORG
;*****************************************************
; Объявление кодовой секции
  AREA MyCode, CODE, ReadOnly
; Основная программа
; Объявление основной программы общедоступной
        EXPORT MyProg
MyProg                          ; Начало основной процедуры
        LDR R5,=0x1133FFAA      ; Инициализация регистра R5
        BL FUNC_1               ; Вызов подпрограммы FUNC_1
Stop    B Stop                  ; Бесконечный цикл

; Разместить литеральный пул перед подпрограммой
        LTORG
; Подпрограмма
FUNC_1
        NOP             ; «Пустая процедура»
        BX LR           ; Возврат в основную программу

; Зарезервировать 5000 байт для будущих расширений
Reserv
        SPACE 5000
;*****************************************************
; Конец ассемблерного текста
        END
```

В ряде случаев диапазона возможных смещений может не хватить. Например, если программист зарезервировал в конце текущей кодовой секции большой объем памяти для последующих расширений (проект CPU_3). Выйти из положения можно, *принудительно* указав место расположения литеральных пулов с помощью директивы Ассемблера **LTORG**.

Она не требует никаких дополнительных операндов и своим расположением лишь указывает место, куда транслятор может разместить литеральный пул. Это может быть промежуток между отдельными подпрограммами или между основной программой и подпрограммами, как показано в этом примере. Из фрагмента листинга видно, что константа действительно размещается там, где указал программист (см. строку 16).

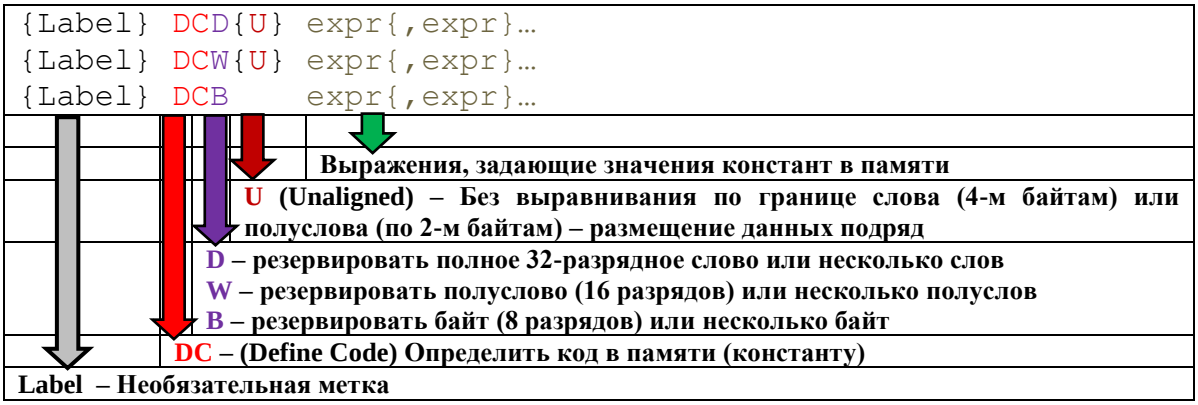
```
10 00000000          MyProg                              ; Начало основной процедуры
11 00000000 4D01             LDR             R5,=0x1133FFAA ; Инициализация регистра R5
12 00000002 F000 F803        BL              FUNC_1      ; Вызов подпрограммы FUNC_1
13 00000006 E7FE     Stop    B               Stop        ; Бесконечный цикл
14 00000008
15 00000008          ; Разместить литеральный пул перед подпрограммой
16 00000008 1133FFAA         LTORG
17 0000000C          ; Подпрограмма
18 0000000C          FUNC_1
19 0000000C BF00             NOP                         ; «Пустая процедура»
20 0000000E 4770             BX              LR          ; Возврат в основную программу
21 00000010
22 00000010          ; Зарезервировать 5000 байт для будущих расширений
23 00000010          Reserv
24 00000010 00 00 00
           00 00 00
```

Напомним, что директива Ассемблера SPACE резервирует определенный объем памяти (в том числе кодовой). Если она применяется в кодовой секции, то зарезервированная область памяти обнуляется (см. строку 24). Мы выделили 5000 резервных байт, поэтому размещение литерального пула в конце текущей кодовой секции стало невозможным. Пришлось размещать его между основной программой и подпрограммой.

10.5 Загрузка данных из кодовой памяти по адресам размещения данных

ARM ASM

Ассемблер позволяет не только резервировать определенные области в кодовой памяти, как это было показано выше, но и инициализировать эти области, размещая в них нужные программисту константы. Константы могут помечаться метками, значения которых будут соответствовать текущим адресам расположения констант. Напомним, что для резервирования констант в памяти используются следующие директивы Ассемблера:



Без суффикса «U» размещение слов в памяти выполняется с автовыравниванием по границе слова (4-м байтам), полуслов – с автовыравниванием по границе полуслова (двум байтам). Байты размещаются подряд, без выравнивания. Если опция «U» в первых двух директивах присутствует, то слова и полуслова размещаются в памяти без выравнивания.

При автовыравнивании транслятор автоматически вставляет перед резервируемой константой нужное число «нулевых» байтов - шаблонов.

Выражения для задания полуслов должны быть в диапазоне целых 16-разрядных чисел без знака или со знаком, то есть от -32768 до 65535. Выражения для задания байт должны быть в диапазоне целых байт со знаком или без знака, то есть от -128 до 255. Байты можно задать и последовательностью символов, каждый из которых заключается в одиночные кавычки, или строкой символов, заключенной в двойные кавычки.

Пример использования директив резервирования констант в памяти можно найти в проекте CPU_4. Программа приложения содержит четыре псевдокоманды загрузки регистров ЦПУ по фактическим адресам размещения данных в памяти: LDR Rd, Label, а в конце программы расположены директивы резервирования и инициализации памяти двумя 32-разрядными словами, двумя полусловами и четырьмя байтами. Обратите внимание на то, что в каждой из директив, через запятую, можно задать любое требуемое число констант. Всего в памяти проинициализировано четыре слова, каждое из которых получило свое символическое имя Const_0, Const_1,… Именно эти имена использованы в псевдокомандах загрузки данных по их меткам.

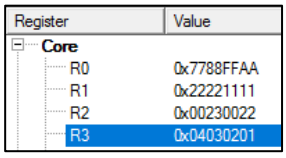
```
 7  MyProg
 8  ; Загрузка регистров ЦПУ из предварительно
 9  ; проинициализированной кодовой памяти
10  ; Псевдокоманды загрузки слов по известным
11  ; адресам их расположения
12          LDR r0,Const_0
13          LDR r1,Const_1
14          LDR r2,Const_2
15          LDR r3,Const_3
16  Stop    B Stop
17  ; Инициализация констант в кодовой памяти
18  Const_0 DCD 0x7788FFAA
19  Const_1 DCD 0x22221111
20  Const_2 DCW 34,35
21  Const_3 DCB 1,2,3,4
```

```
10 00000000          ; Псевдокоманды загрузки слов по известным
11 00000000          ; адресам их расположения
12 00000000 4802            LDR             r0,Const_0
13 00000002 4903            LDR             r1,Const_1
14 00000004 4A03            LDR             r2,Const_2
15 00000006 4B04            LDR             r3,Const_3
16 00000008 E7FE    Stop    B               Stop
17 0000000A         ; Инициализация констант в кодовой памяти
18 0000000A 00 00 7788FFAA
                    Const_0 DCD             0x7788FFAA
19 00000010 22221111
                    Const_1 DCD             0x22221111
20 00000014 22 00 23
           00       Const_2 DCW             34,35
21 00000018 01 02 03
           04       Const_3 DCB             1,2,3,4
```

Просмотрите листинг этой программы. Обратите внимание на то, что транслятор перед директивой DCD резервирования слова выполнил автовыравнивание памяти по границе слова (добавил два «нулевых» байта), выдав соответствующее предупреждение в окно вывода результатов.

Обратите также внимание на порядок расположения полуслов и байт в памяти. Они расположены, начиная с младшего полуслова или младшего байта. Если теперь выполнить сгенерированный код, то увидим, что загрузка данных выполнена правильно.

```
   16: Stop       B Stop
0x00000020 E7FE      B             0x00000020
0x00000022 0000      DCW           0x0000
0x00000024 FFAA      DCW           0xFFAA
0x00000026 7788      DCW           0x7788
0x00000028 1111      DCW           0x1111
0x0000002A 2222      DCW           0x2222
0x0000002C 0022      DCW           0x0022
0x0000002E 0023      DCW           0x0023
0x00000030 0201      DCW           0x0201
0x00000032 0403      DCW           0x0403
```



Как же работают псевдокоманды загрузки данных по адресам? Обратимся к окну дизассемблера и посмотрим, какие команды на самом деле выполняет процессор вместо псевдокоманд. Вы видите знакомые команды относительной адресации памяти по содержимому счетчика команд PC и непосредственно заданному в команде смещению. Давайте проследим, к каким областям памяти обращается при этом процессор. В первом случае – к данным по адресу 0x24, где находится первая записанная в память константа, во втором случае – по адресу 0x28, месту расположения второй константы и т.д.

```
   12:                                  LDR r0,Const_0
0x00000018 4802      LDR           r0,[pc,#8]  ; @0x00000024
   13:                                  LDR r1,Const_1
0x0000001A 4903      LDR           r1,[pc,#12]  ; @0x00000028
   14:                                  LDR r2,Const_2
0x0000001C 4A03      LDR           r2,[pc,#12]  ; @0x0000002C
   15:                                  LDR r3,Const_3
0x0000001E 4B04      LDR           r3,[pc,#16]  ; @0x00000030
   16: Stop       B Stop
0x00000020 E7FE      B             0x00000020
```

Итак, для доступа к данным по их символическим адресам можно использовать директивы загрузки регистров по меткам, которые присвоены константам при их размещении в памяти. Единственное отличие этих псевдокоманд от псевдокоманд с автоматическим созданием в памяти литеральных пулов состоит в том, что данные в памяти резервирует сам программист. Иногда это бывает полезно, так как делает программу более понятной.

10.6 Косвенная адресация операндов в памяти

10.6.1 Общие положения

Один из основных способов адресации операндов в памяти, применяемый в процессорной технике – *косвенная адресация*. Ее признаком является использование для обозначения операнда квадратных скобок […], внутри которых указывается символическое имя *регистра-указателя* – регистра, содержащего адрес ячейки памяти, к которой производится доступ. В общем случае после имени регистра-указателя, через запятую, внутри квадратных скобок можно задать смещение относительно текущего содержимого регистра-указателя [Rn,#offset]. Такое смещение называется *непосредственным*. Его значение добавляется к содержимому базового регистра перед доступом к данным. При этом исходное содержимое базового регистра не модифицируется. Задавая разные величины смещений можно получить доступ к любым элементам массива, начальный адрес которого находится в регистре-указателе. В системе команд ARM-процессоров регистр-указатель называют *базовым регистром*. Если смещение указано, то такая адресация называется *базовой с непосредственным смещением.*

Если величина смещения указывается через запятую после квадратных скобок [Rn],#offset, то такая адресация называется *косвенной адресацией с пост-смещением указателя*: сначала производится доступ к данным по текущему содержимому указателя, а затем к нему добавляется смещение.

Как видите, способ адресации в псевдокоманде загрузки LDR Rd,=Value (относительная адресация по текущему содержимому счетчика команд) по существу является косвенной базовой адресацией с непосредственным смещением, в которой *функцию базового регистра выполняет счетчик команд* [PC,#offset]. Единственное отличие состоит в том, что после доступа к данным по эффективному адресу, рассчитанному как (PC)+#offset, счетчик команд PC автоматически перезагружается адресом следующей, подлежащей выполнению команды (получает приращение +2 или +4 в зависимости от длины текущей команды).

Итак, относительная адресация по содержимому счетчика команд – это частный случай косвенной базовой адресации с непосредственным смещением, заданным в формате самой команды. Какой может быть величина смещения? Для набора команд Thumb-2 смещение кодируется 12-разрядным числом и имеет диапазон -255÷4095. Это означает, что обращение за данными, расположенными в литеральном пуле выполняется в основном вперед.

10.6.2 Псевдокоманда загрузки адреса данных в регистр указатель

Любой из регистров ЦПУ может выполнять функцию *регистра-указателя* данных. Для его начальной *инициализации* можно применять обычные команды загрузки данных, описанные выше. Однако, если данные в памяти уже помечены меткой, проще воспользоваться

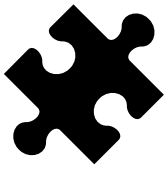
специальной псевдокомандой ассемблера «*Загрузить в регистр назначения адрес метки*». Она работает в том случае, когда данные расположены в той же секции на ограниченном удалении от команды и имеет следующий синтаксис:

```
ADR{cond}{.W} Rd,label
```

Компонент псевдокоманды	Назначение
cond	**опциональный суффикс условного выполнения команды**
.W	**опциональный спецификатор ширины генерируемой команды. Если указан, то длина команды составляет 32 разряда, в противном случае – 16 разрядов.**
Rd	**имя регистра-приемника данных (регистра-указателя)**
label	**метка, символическое имя данных в памяти**

Псевдокоманда генерирует реальную команду загрузки в регистр назначения адреса, который рассчитывается как текущий адрес в счетчике команд плюс заданное в команде непосредственное смещение **ADR Rd, (PC)+#offset**. Величина нужного смещения рассчитывается транслятором автоматически. В режиме 32-разрядного кодирования значение смещения не должно превышать ±4095, а в режиме 16-разрядного кодирования должно находиться в диапазоне 0÷1020. В этом случае ссылка на данные возможна только вперед, в направлении возрастающих адресов. Если генерация смещения в допустимом диапазоне невозможна, то транслятор выдает сообщение об ошибке. Первое, что нужно сделать в этом случае – использовать опцию «.W» с более широким диапазоном смещений. Если это также не даст результата – воспользоваться уже рассмотренной нами псевдокомандой загрузки LDR Rd,=Value, которая за счет создания литерального пула позволяет обращаться к данным по любым адресам в памяти объемом 4 Гбайта.

Создаваемый командой ADR код является позиционно независимым. Это позволяет размещать вместе с кодом программы данные и таблицы данных, к которым обеспечивается простой доступ из той же текущей кодовой секции командами с косвенной адресацией.



1) Команду ADR можно использовать и для последующей косвенной передачи управления командами BX или BLX. При этом Вы должны сами установить признак набора команд Thumb, в котором процессор должен продолжать работать после завершения перехода (установить бит 0 адреса в «1»). В процессе выполнения команды косвенной передачи управления он будет автоматически очищен и переход состоится в нужную точку программы.
2) Не рекомендуется использовать в команде ADR счетчик команд PC и указатель стека SP в качестве регистра приемника.

10.6.3 Доступ к данным с использованием косвенной адресации

Откроем проект CPU_5 с демонстрацией основ косвенной адресации операндов. В программе приложения MyProg_5.s вначале выполняется инициализация регистра-указателя r0 адресом расположения в памяти константы Const_0 с использованием псевдокоманды ADR. Далее выполняется команда косвенной загрузки регистра r5 данными по текущему адресу в регистре-указателе r0. После чего указатель r0 трижды переинициализируется для доступа к

```
 7  MyProg
 8  ; Демонстрация основ косвенной адресации данных
 9  ; Загрузить регистр-указатель r0 адресом
10  ; расположения данных Const_0
11          ADR r0,Const_0
12  ; Загрузить слово по этому адресу в регистр r5
13          LDR r5, [r0]
14  ; Выполнить косвенную загрузку еще трех слов
15  ; в регистры r6,r7,r8
16          ADR r0,Const_1
17          LDR r6, [r0]
18          ADR r0,Const_2
19          LDR r7, [r0]
20          ADR r0,Const_3
21          LDR r8, [r0]
22  Stop    B Stop
23  ; Инициализация констант в кодовой памяти
24  Const_0 DCD 0x7788FFAA
25  Const_1 DCD 0x22221111
26  Const_2 DCW 34,35
27  Const_3 DCB 1,2,3,4
```

остальным константам, и эти константы загружаются в регистры r6, r7, r8. Как и в предыдущем проекте, в конце программы в текущей кодовой секции размещаются четыре константы-слова.

Рассматривая содержимое файла листинга, убеждаемся в том, что транслятор смог сгенерировать короткие 16-разрядные команды ADR, так как данные расположены на небольшом удалении от команд. В листинге указаны псевдокоманды. Обратитесь к окну дизассемблера, чтобы посмотреть, что они представляют собой в реальности: псевдокоманды заменяются командами ADR Rd, {PC}+#offset.

```
    17:                                 ADR r0,Const_1
0x0000001C A004      ADR            r0,{pc}+4  ; @0x00000030
    18:                                 LDR r6, [r0]
0x0000001E 6806      LDR            r6,[r0,#0x00]
    19:                                 ADR r0,Const_2
0x00000020 A004      ADR            r0,{pc}+4  ; @0x00000034
    20:                                 LDR r7, [r0]
```

```
0x0000002A E7FE      B              0x0000002A
0x0000002C FFAA      DCW            0xFFAA
0x0000002E 7788      DCW            0x7788
0x00000030 1111      DCW            0x1111
0x00000032 2222      DCW            0x2222
0x00000034 0022      DCW            0x0022
0x00000036 0023      DCW            0x0023
0x00000038 0201      DCW            0x0201
0x0000003A 0403      DCW            0x0403
```

```
 8 00000000          MyProg
 9 00000000          ; Демонстрация основ косвенной
                      адресации данных
10 00000000          ; Загрузить регистр-указатель r
                     0 адресом
11 00000000          ; расположения данных Const_0
12 00000000 A004             ADR              r0,Const_0
13 00000002          ; Загрузить слово по этому адре
                     су в регистр r5
14 00000002 6805             LDR              r5, [r0]
15 00000004          ; Выполнить косвенную загрузку
                      еще трех слов
16 00000004          ; в регистры r6,r7,r8
17 00000004 A004             ADR              r0,Const_1
18 00000006 6806             LDR              r6, [r0]
19 00000008 A004             ADR              r0,Const_2
20 0000000A 6807             LDR              r7, [r0]
21 0000000C A004             ADR              r0,Const_3
22 0000000E F8D0 8000        LDR              r8, [r0]
23 00000012 E7FE     Stop    B                Stop
24 00000014          ; Инициализация констант
25 00000014          ; в текущей кодовой секции
26 00000014 7788FFAA
                     Const_0 DCD              0x7788FFAA
27 00000018 22221111
                     Const_1 DCD              0x22221111
28 0000001C 22 00 23
           00        Const_2 DCW              34,35
29 00000020 01 02 03
           04        Const_3 DCB              1,2,3,4
```

Реальные адреса обращения к данным указаны в комментариях к созданным командам. Они точно соответствуют месту расположения констант в памяти. Так, константа Const_1 действительно расположена по адресу 0x30. Обозначение {PC} говорит о том, что это не истинный адрес расположения текущей команды в памяти, а дополнительно модернизированный кодом операции. В нашем случае увеличенный на 0x10 (0x1C+0x10+4=0x30). Будем доверять транслятору в формировании кодов операций, не вдаваясь в детали кодирования команд.

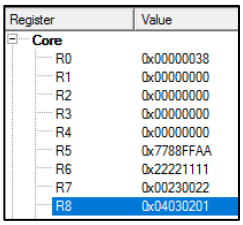
1) Постройте проект, загрузите в отладчик и выполните в пошаговом режиме. Проследите за тем, как последовательно меняется содержимое регистра-указателя r0 и как работает команда косвенной загрузки данных по его текущему содержимому LDR Rd, [r0].

2) Вам не кажется, что каждый раз выполнять новую перезагрузку регистра-указателя, если данные расположены в памяти последовательно, не имеет смысла? Проще добавить к значению в регистре-указателе нужное смещение и считать очередные данные. Попробуйте сделать это с использованием команды инкрементирования содержимого регистра указателя r0 на 4 (ADD r0, #4). Проверьте работу программы.

3) До сих пор мы пользовались обычными командами косвенной загрузки. Но, если данные расположены последовательно, целесообразно применить команду с *пост-авто-инкрементированием указателя*. В этом случае вслед за именем регистра-указателя в квадратных скобках через запятую нужно задать величину смещения. Сделайте это. Выполните отладку.

4) Перед Вами задача. Извлечь из массива слов в кодовой памяти вторую и четвертую константы и найти их сумму с записью результата в регистр r5. Попробуйте сделать это.

1) Загрузка данных выполнена правильно. Регистр r0 содержит адрес последней константы Const_3 – 0x38, которая после выполнения программы скопирована в регистр R8. Обратите внимание на то, что все команды косвенной адресации в действительности генерируются с *нулевым непосредственным смещением* [r0, #0x00]. В ассемблерной программе смещение может быть опущено (при этом оно по умолчанию считается нулевым).



2) Фрагмент модифицированной программы MyProg_5_1.s представлен ниже. Директивы резервирования констант остались без изменения в конце программы (не показаны). Перед каждой загрузкой содержимое регистра указателя увеличивается на +4. Заметьте, что команда сложения – двух-операндная: регистр r0 является одновременно как источником, так и приемником данных.

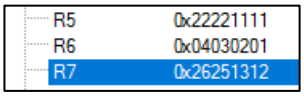
```
MyProg
; Демонстрация основ косвенной адресации данных
; Загрузить регистр-указатель r0 адресом
; расположения данных Const_0
        ADR r0,Const_0
; Загрузить слово по этому адресу в регистр r5
        LDR r5, [r0]
; Инкрементировать указатель на 4-е байта
        ADD r0,#4
; Загрузить следующую константу
        LDR r6, [r0]
; Повторить операцию для следующих двух констант
        ADD r0,#4
        LDR r7, [r0]
        ADD r0,#4
        LDR r7, [r0]
Stop    B Stop
```

3) Фрагмент модифицированной программы My_Prog_5_2 показан ниже. *Косвенная адресация с пост-авто-смещением* указателя позволяет сначала выполнить доступ к данным, а затем, той же командой модифицировать содержимое указателя (+4). При исследовании файла листинга обратите внимание на то, что эти команды уже 32-разрядные.

```
; Демонстрация основ косвенной адресации данных
; Загрузить регистр-указатель r0 адресом
; расположения данных Const_0
        ADR r0,Const_0
; Загрузить слово по этому адресу в регистр r5
; с пост- авто- смещением указателя на +4
        LDR r5, [r0],#4
; Загрузить следующие три константы
        LDR r6, [r0],#4
        LDR r7, [r0],#4
        LDR r8, [r0]
Stop    B Stop
; Инициализация констант
; в текущей кодовой секции
Const_0 DCD 0x7788FFAA
Const_1 DCD 0x22221111
Const_2 DCW 34,35
Const_3 DCB 1,2,3,4
```

4) Для решения задачи необходима косвенная адресация памяти с непосредственным смещением [r0, #offset]. При этом содержимое указателя r0 не модифицируется, а доступ к данным выполняется по эффективному адресу, равному сумме содержимого регистра-указателя и заданного в команде непосредственного смещения. Если оставить в r0 начальный адрес нашего массива, то для доступа ко второй константе смещение должно быть равно +4, а к четвертой +12. Возможный фрагмент такой программы (MyProg_5_3.s) показан ниже: Результат ее работы доказывает, что и загрузка элементов массива, и их обработка, выполнены правильно. Используйте этот метод для выборки нужных элементов массива, задавая нужные величины смещений.

```
MyProg
; Демонстрация основ косвенной адресации данных
; Сложить две константы Const_1 и Const_3
; Сохранить результат в регистре r7
        ADR r0,Const_0
; Загрузить первую константу в регистр r5
; с заданным пре-смещением указателя
        LDR r5, [r0,#4]
; Загрузить вторую константу в регистр r6
; со своим  пре-смещением указателя
        LDR r6, [r0,#12]
; Рассчитать сумму двух констант
        ADD r7,r6,r5
```



Мы познакомились с рядом псевдокоманд Ассемблера, которые позволяют загружать регистры ЦПУ данными из кодовой памяти по их символическим адресам LDR Rd, label, загружать данные с одновременным размещением их в кодовой памяти (в литеральных пулах) LDR Rd,=literal, загружать в регистры процессора адреса расположения данных ADR Rd, label для последующего использования этих регистров в качестве регистров-указателей в командах косвенной адресации памяти. Во всех этих псевдокомандах фактически используются специальные команды процессора с относительной адресацией по содержимому счетчика команд PC и автоматически рассчитанному транслятором непосредственному смещению.

В ряде случаев интеллектуальный транслятор может сгенерировать более простую команду, в частности, команду загрузки в регистр непосредственной константы.

При использовании регистров младшего регистрового банка (R0-R7) вероятность получения более компактного 16-разрядного кода существенно возрастает. Пользуйтесь, по возможности, этими регистрами.

10.7 Команды работы с памятью

Архитектура процессоров ARM является архитектурой Загрузки/Сохранения в том смысле, что все вычислительные операции выполняются исключительно над данными, находящимися в регистрах процессора – сверхоперативной памяти процессора. Поэтому, прежде чем обрабатывать данные, их необходимо загрузить в регистры из памяти или регистров периферийных устройств, а после обработки – сохранить в памяти или в регистрах периферии (выдать результат обработки).

10.7.1 Какие данные используются в операциях загрузки/сохранения?

Память в ARM-процессорах байтовая, каждый байт имеет свой персональный адрес, как и каждое 16-разрядное полуслово (четный адрес) и каждое 32-разрядное слово (дважды четный адрес, кратный 4). Все регистры процессора 32-разрядные. Данные в них можно интерпретировать как 32-разрядное слово, два 16-разрядных полуслова (младшее и старшее) или четыре 8-разрядных байта – младший и старший в каждом из двух полуслов.

Как процессор выполняет загрузку данных из памяти?

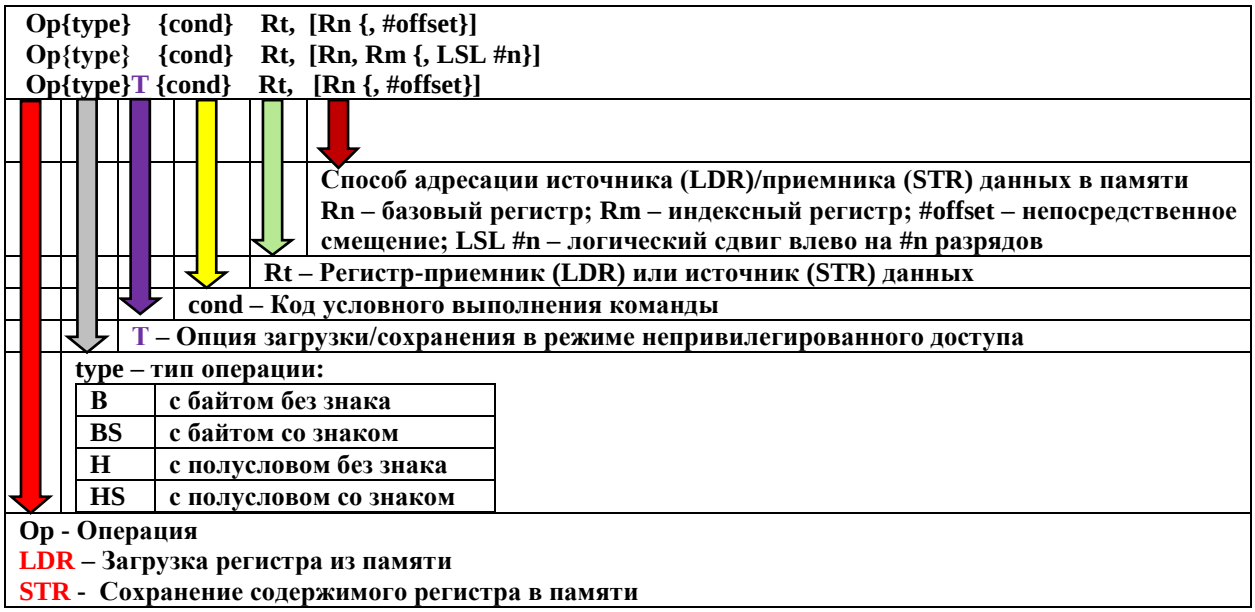
- 32-разрядное слово загружается в регистр процессора целиком без каких-либо изменений;
- 16-разрядное полуслово загружается в младшее полуслово регистра «как есть». Далее все зависит от типа полуслова, специфицированного в команде загрузки дополнительным суффиксом:
  - **H** (полуслово без знака) – во все старшие биты регистра записываются нули;
  - **SH** (полуслово со знаком) – во все старшие биты записывается знаковый разряд числа.
- 8-разрядный байт загружается в младший значащий байт регистра «как есть». Далее все зависит от типа байта, заданного дополнительным суффиксом в команде загрузки:
  - **B** (байт без знака) – во все старшие биты регистра записываются нули;
  - **SB** (байт со знаком) – во все старшие биты записывается знаковый разряд числа.

Таким образом, операция загрузки слова из памяти в регистр процессора представляет собой обычную операцию пересылки данных, а операция загрузки полуслова или байта – *автоматического преобразования формата числа* из 16-разрядного или 8-разрядного в 32-разрядный формат. Для этого преобразования процессору необходимо «знать», является ли исходное число числом без знака или числом со знаком в дополнительном коде. Если исходное число является беззнаковым (в команде указан суффикс H или B), то выполнятся операция *беззнакового расширения*, в противном случае (указан суффикс SH или SB) – операция *знакового расширения*. Это одно из преимуществ системы команд ARM-процессоров. Данные могут храниться в памяти в укороченных форматах (полусловах, байтах), а их обработка будет выполняться в расширенном 32-разрядном формате. При этом загрузка «укороченных» данных сопровождается автопреобразованием формата, и никаких дополнительных операций для этого не требуется.

Операции сохранения содержимого регистров в памяти выполняются без дополнительных преобразований данных – данные сохраняются «как есть»: полное слово, полуслово или байт, причем команды с суффиксами (B и SB), (H и SH) просто сохраняют в памяти байт или полуслово соответственно. Впрочем, один из пары суффиксов, Вы обязательно должны указать, так как при его отсутствии в памяти будет сохранено полное 4-байтовое слово. По умолчанию, если дополнительный суффикс не указан, то команда работает с полным 32-разрядным словом.

10.7.2 Форматы команд загрузки/сохранения данных

Форматы команд загрузки регистров ЦПУ из памяти (LDR) и сохранения значений регистров в памяти (STR) с возможными опциями показаны ниже:



В отличие от всех других команд процессора, в которых имя регистра-приемника данных всегда указывается на первом месте, сразу после кода операции, в командах загрузки/сохранения на первом месте всегда находится имя регистра ЦПУ, участвующего в операции (независимо от того, является ли он приемником или источником данных). На втором месте указывается используемый в команде способ адресации памяти. Ниже мы рассмотрим возможные способы адресации подробно. Здесь лишь отметим, что синтаксис команд в первой и третьей строке соответствует *базовой адресации с непосредственным смещением* содержимого указателя, а во второй строке – *базово-индексной адресации*.

Значение спецификатора типа операции «type» было описано выше: оно определяет формат данных, с которыми имеет дело процессор, и, следовательно, необходимость знакового или беззнакового расширения в область старших разрядов (только при загрузке данных в регистры).

Суффикс «T» – особый. Он заставляет процессор работать при выполнении команды в режиме *непривилегированного доступа*, то есть в режиме, когда доступ к регистрам специального назначения процессора *заблокирован*, во избежание случайного непреднамеренного вреда (например, случайной переинициализации регистра управления процессора).

10.7.3 Базовая адресация с непосредственно заданным смещением указателя

Возможности базовой адресации с непосредственным смещением очень широки. Они раскрыты в табл. 10.2 с указанием выполняемых действий и преимущественной области применения этого метода адресации.

Таблица 10.2 Варианты базовой адресации с непосредственным смещением указателя

Мнемоника	Способ адресации	Основное применение
[Rn]	***Косвенная*** **по содержимому базового регистра Rn** **Rt ← [(Rn)]**	**Единичный доступ к данным, адрес которых известен и находится в базовом регистре Rn.**
[Rn, #offset]	***Базовая с непосредственно заданным смещением*** **#offset** **Rt ← [(Rn)+#offset)]**	**Единичный доступ к элементу массива с известным смещением адреса относительно начала массива #offset.**
[Rn, #offset]!	***Базововая с пре-смещением указателя на константу*** **#offset** **Rn ← (Rn)+#offset;** **Rt ← [(Rn)].**	**Последовательная обработка элементов массива (в цикле) с пре-смещением указателя перед каждым доступом на величину #offset.**
[Rn], #offset	***Базовая с пост-смещением указателя на константу*** **#offset** **Rt ← [(Rn)]** **Rn ← (Rn)+#offset;**	**Последовательная обработка элементов массива (в цикле) с пост-смещением указателя после каждого доступа на величину #offset.**

Красным цветом выделены признаки дополнительных возможностей этого типа адресации, которые позволяют выполнить либо *пре-смещение*, либо *пост-смещение* содержимого указателя, что позволяет автоматизировать процесс обработки данных в циклах.

10.7.3.1 Косвенная адресация (базовая с нулевым смещением)

Графическая иллюстрация классической косвенной адресации памяти, когда доступ к данным производится по текущему адресу, находящемуся в базовом регистре (регистре-указателе), показана на рис. 10.1. В этом случае содержимое регистра-указателя не модифицируется и для доступа к новым данным его нужно каждый раз пере-инициализировать (как мы и делали в примере MyProg_5.s). Содержимое базового регистра всегда указывает на младший адрес размещения данных в памяти независимо от формата передаваемых данных (байт, слово, полуслово).

После извлечения данных из памяти содержимое этой ячейки памяти остается прежним. При записи – меняется, но только в тех байтах, которые соответствуют типу пересылаемых данных (для байта – только в младшем байте, для полуслова – в младшем полуслове, для слова – во всех 4-х байтах). Напомним, что доступ к памяти в процессорах ARM – побайтовый.

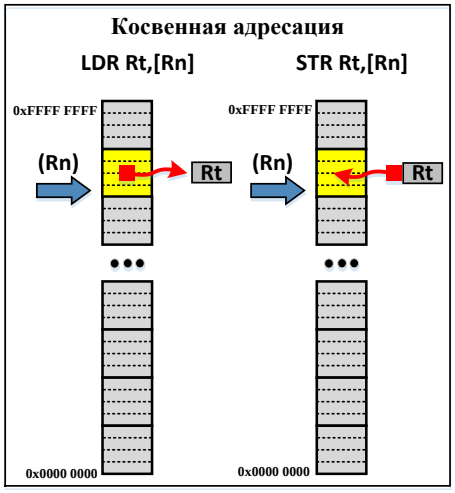


Рис. 10.1 Косвенная адресация памяти при загрузке и сохранении данных

10.7.3.2 Базовая адресация с непосредственным смещением

Для этого типа адресации программист после имени базового регистра, через запятую, указывает непосредственное смещение, которое перед доступом к данным прибавляется к содержимому базового регистра. После доступа к данным значение указателя остается прежним. Именно смещение будет определять так называемый *эффективный адрес* доступа к данным Adr (красная стрелка на рис. 10.2). В системе команд ARM-процессоров поддерживается именно косвенная адресация с непосредственным смещением. Если смещение опущено (как для косвенной адресации), то транслятор автоматически генерирует нулевое смещение. Последовательность операций, выполняемых при записи и чтении данных, показана на рис. 10.2 цифрами.

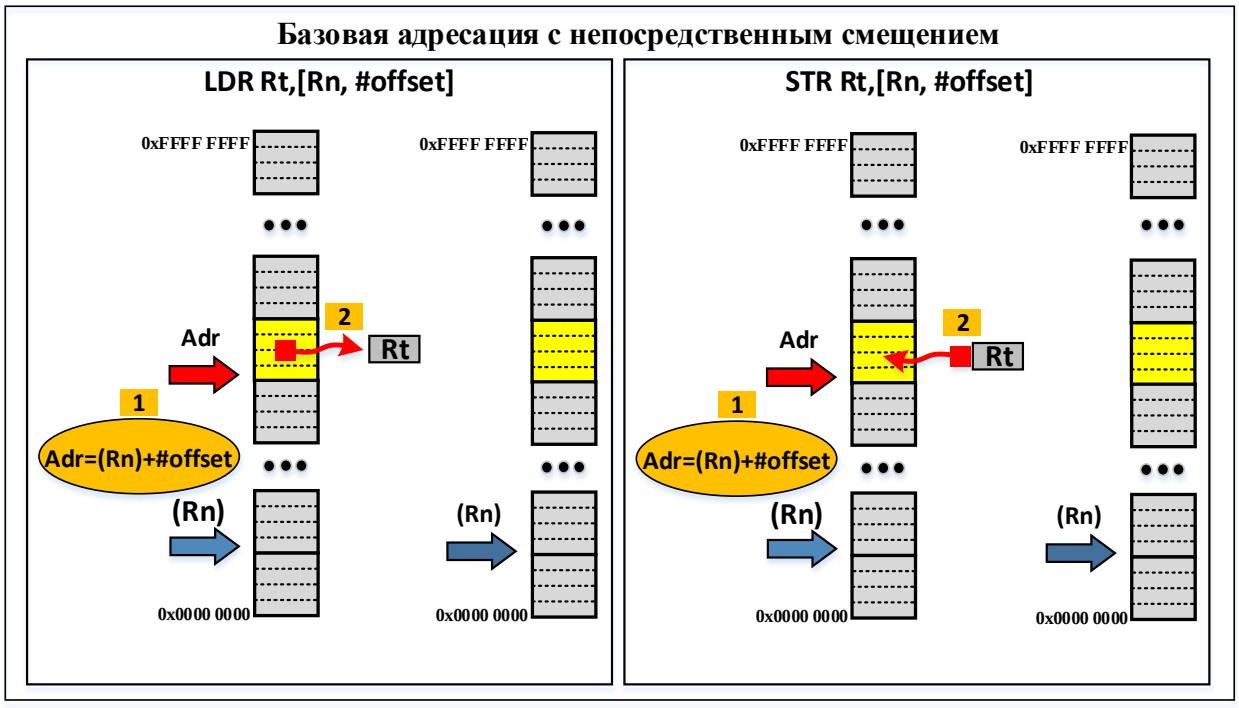


Рис. 10.2 Базовая адресация с непосредственным смещением

10.7.3.3 Базовая адресация с пре-смещением

Адресация с *пре-смещением* отличается тем, что перед каждым доступом к данным содержимое базового регистра (указателя) предварительно модифицируется: к нему добавляется заданное в команде смещение – рис. 10.3.

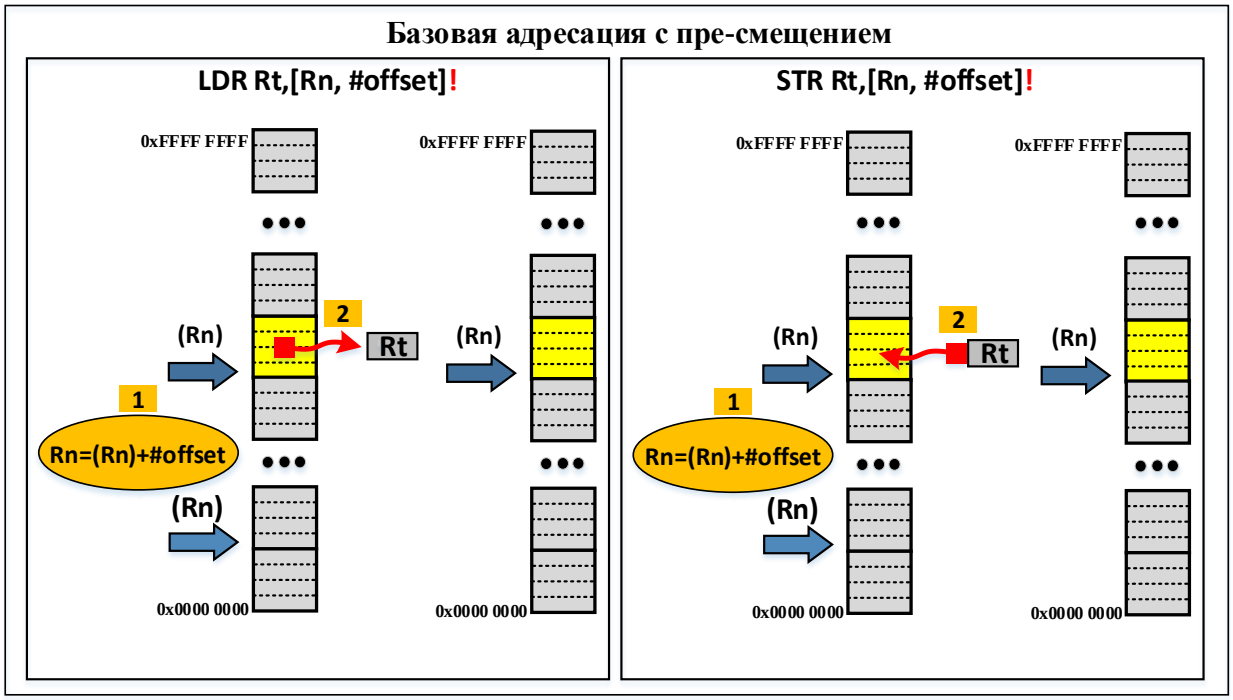


Рис. 10.3 Базовая адресация с пре-смещением

10.7.3.4 Базовая адресация с пост-смещением

Адресация с *пост-смещением* модифицирует значение базового регистра (указателя) после выполнения доступа к данным – рис. 10.4.

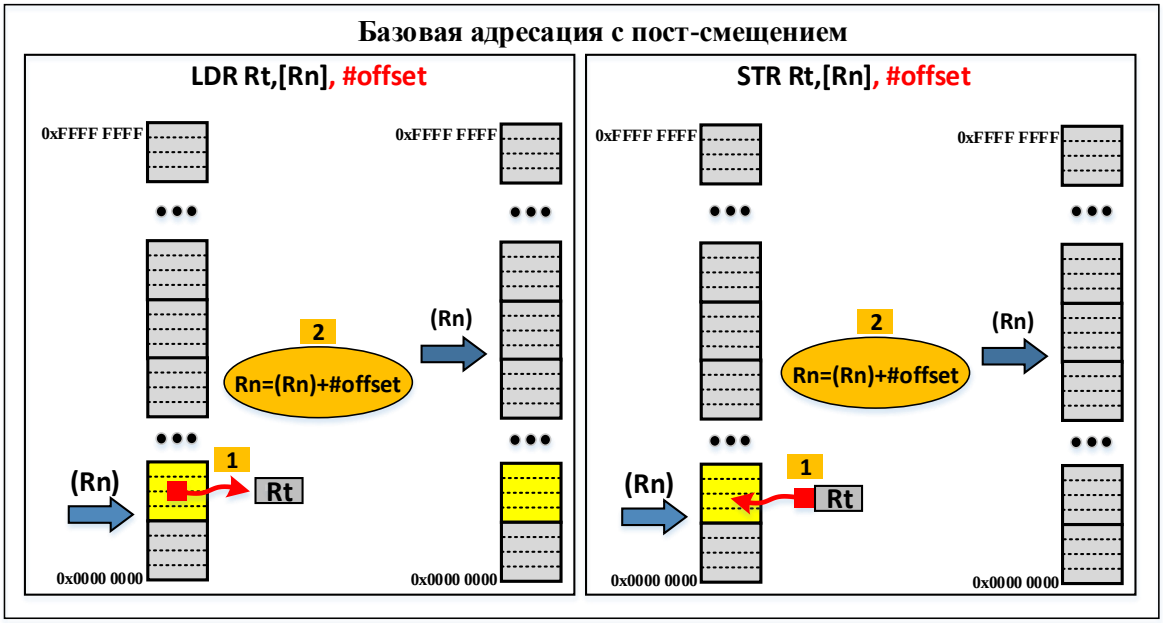


Рис. 10.4 Базовая адресация с пост-смещением

10.7.4 Базовая адресация со смещением, заданным содержимым регистра, и базово-индексная адресация

Величина смещения #offset может содержаться в одном из регистров общего назначения процессора, который в этом случае называется *регистром смещения* Rm. Это удобно, когда смещение объекта данных относительно базового адреса в регистре Rn заранее не известно и должно быть вычислено. Мнемоника этого способа адресации представлена в табл. 10.3.

Таблица 10.3 Мнемоника базовой адресации со смещением

Мнемоника	Способ адресации	Основное применение
[Rn, Rm]	**Базовая со смещением в регистре смещения Rm** **Rt ← [(Rn)+(Rm)]**	**Доступ к данным с предварительным вычислением смещения и записью его в регистр смещения Rm.**

На рис. 10.5 дана графическая иллюстрация этого метода адресации.

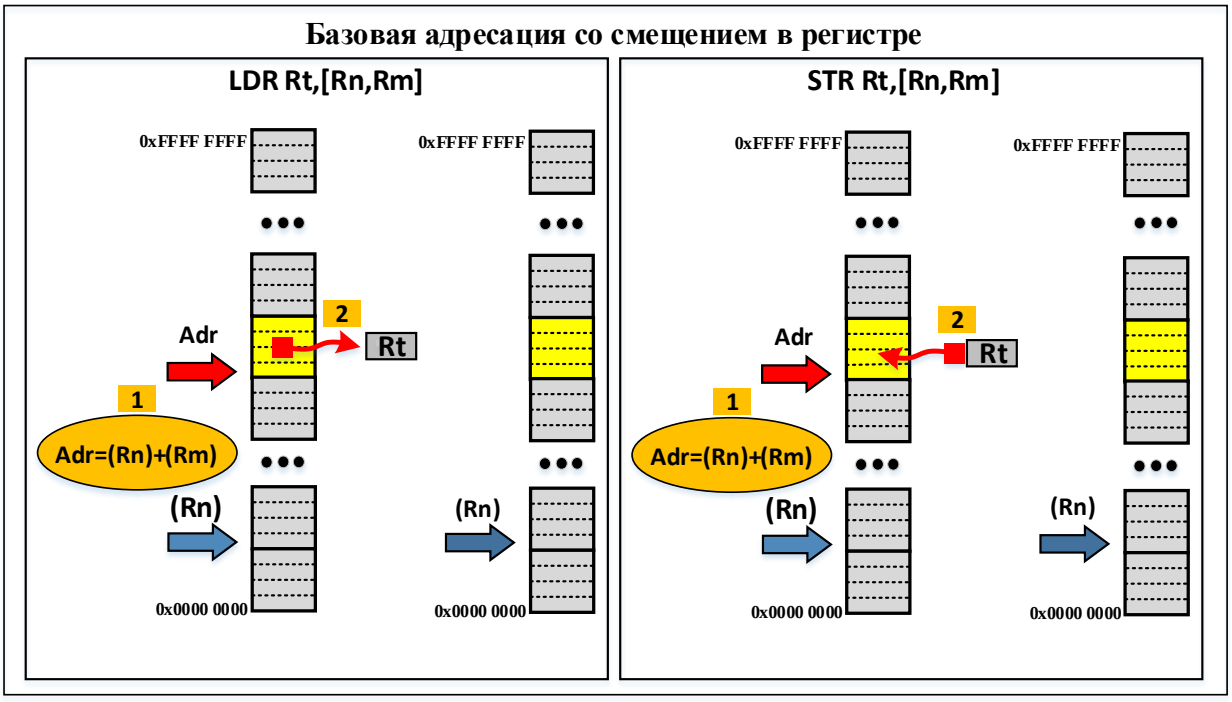


Рис. 10.5 Базовая адресация со смещением, заданным содержимым регистра

Данные в памяти часто представляют собой однородный массив (байт, полуслов, слов, длинных слов), отдельный элемент которого a[i] определяется именем массива «a» и индексом «i». Самый удобный метод адресации однородных массивов – это задание начального адреса массива в базовом регистре Rn и индекса в *индексном регистре* Rm. В зависимости от длины данных в массиве индекс должен масштабироваться (см.табл. 10.4).

Таблица 10.4 Масштабирование индекса в зависимости от формата данных

Формат данных в массиве	Offset
Байт	**i**
Полуслово	**2 x i**
Слово	**4 x i**
Длинное слово	**8 x i**

Для этого в архитектуре процессора предусмотрено использование специального кольцевого сдвигового регистра, который может увеличивать значение в регистре Rm в 1, 2, 4, или в 8 раз.

Мнемоника настоящей базово-индексной адресации *с авто-масштабированием индекса* выглядит как показано в табл. 10.5.

Таблица 10.5 Мнемоника базово-индексной адресации

Мнемоника	Способ адресации	Основное применение
[Rn, Rm LSL #n] **n=0,1,2 или 3**	**Базово-индексная по содержимому базового регистра Rn и индексу в регистре Rm с авто-масштабированием индекса числом разрядов сдвига влево: offset = i x 2^n = (Rm) x 2^n;** **Rt ← [((Rn)+offset)]**	**Доступ к данным по начальному адресу массива и индексу с автоматическим расчетом смещения в зависимости от формата данных в массиве.**

Рис. 10.5 можно использовать для иллюстрации этого метода адресации, заменив выражение для вычисления эффективного адреса доступа к данным на следующее:

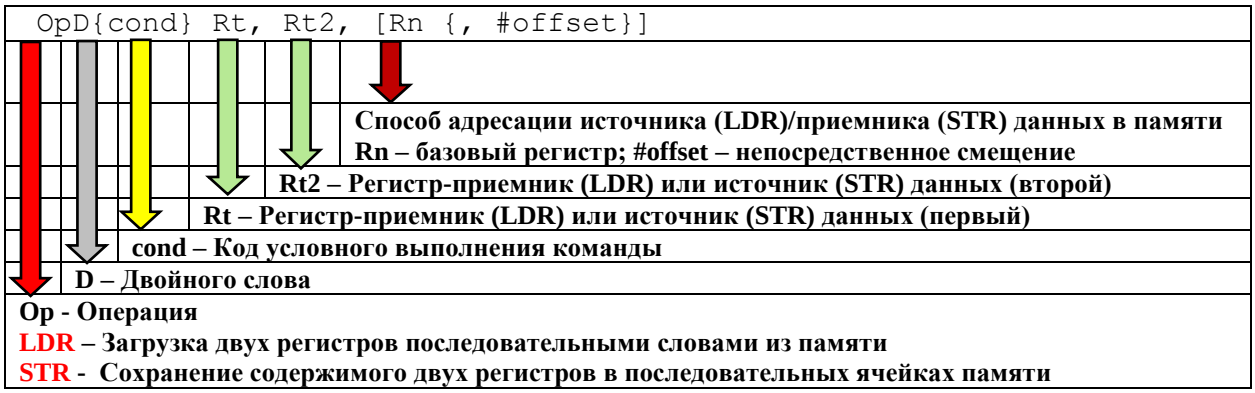
Adr = (Rn)+(Rm) x 2^n.

Способ базово-индексной адресации элементов массива оптимален для автоматизированной обработки данных в циклах – Вы просто должны инкрементировать или декрементировать значение в индексном регистре при очередном доступе к данным, смещение будет определено автоматически.

1) Первый операнд после мнемокода любой команды загрузки/сохранения – это регистр назначения Rt, приемник в команде LDR или передатчик в команде STR.
2) Квадратные скобки «[ ]» являются признаком косвенной адресации.
3) Если внутри них содержится имя только базового регистра Rn, то он является регистром-указателем адреса расположения данных (это соответствует классической косвенной адресации).
4) Если после имени базового регистра Rn через запятую внутри квадратных скобок указано смещение #offset, то адрес доступа к данным определяется суммой содержимое базового регистра и величины смещения. Исходное содержимое регистра-указателя не модифицируется.
5) Если смещение указано через запятую после квадратных скобок – то это признак адресации с пост-смещением указателя. В этом случае содержимое базового регистра модифицируется сразу после доступа к данным.
6) Восклицательный знак «!» после квадратных скобок является признаком адресации с пре-смещением указателя. В этом случае содержимое базового регистра модифицируется на величину смещения перед доступом к данным.
7) Для обработки массивов данных одного формата в циклах используйте базово-индексную адресацию с авторасчетом величины смещения по значению индекса и формату данных в массиве.

10.8 Загрузка и сохранение двойных слов

Для ускорения операций загрузки/сохранения данных используются операции с двойными словами (64-бита), синтаксис которых представлен ниже:



На месте первого операнда через запятую указываются оба 32-разрядных регистра-приемника или источника данных, которые образуют один 64-битовый регистр Rt, Rt2. К основному мнемокоду команды добавляется суффикс «D». В командах двойной загрузки/сохранения применяется только базовая адресация с непосредственным смещением. Другие способы адресации недоступны.

1) Являются ли термины «косвенная адресация» и «базовая адресация» синонимами?

2) Какой метод адресации лучше использовать при копировании массива данных из ПЗУ в ОЗУ?

3) Какой формат данных содержится в массиве, если для его обработки используется способ адресации [R0, R1 LSL #1]?

1) Термин «базовая» адресация несколько шире: базовая с непосредственным смещением; с пре- или пост-смещением; базово-индексная, в том числе с автомасштабированием индекса.

2) Наиболее удобен метод базовой адресации с пост-смещением содержимого указателей [Rn], #offset, когда их содержимое будет автоматически обновляться после каждой операции копирования единицы информации на длину данных.

3) Массив содержит полуслова, так как индекс в регистре R1 автоматически удваивается перед доступом к данным.

10.9 Технология организации циклов при работе с массивами данных

Большие объемы данных (массивы, структуры и т.д.), как правило, обрабатываются автоматически с использованием *циклов*. Прежде чем привести примеры подпрограмм работы с массивами данных в памяти, рассмотрим наиболее общие *принципы организации циклов* на языке Ассемблер. В простейшем цикле заданное программистом число данных обрабатывается *единообразно* (по одному и тому же алгоритму). В цикле всегда имеется:

- *Заголовок цикла*, в котором задается *число проходов цикла* путем инициализации переменной, называемой *счетчиком числа циклов*, устанавливаются начальные значения других переменных, используемых в теле цикла, в том числе регистров-указателей, адресующих память.
- *Тело цикла* - последовательность команд, которая повторяется при каждом проходе цикла, обеспечивая, в том числе, авто-модификацию указателей для обращения к очередным данным в памяти и дополнительные команды *модернизации значения в счетчике числа циклов* (инкрементирования или декрементирования);
- *Окончание цикла* – команда или последовательность команд, в которой проверяется *условие выхода из цикла*. Если оно выполняется – организуется выход из цикла. Если нет – то тело цикла выполняется вновь. При этом все подготовительные операции по доступу к очередным данным должны быть выполнены заранее, в теле цикла во время предыдущего прохода или в заголовке цикла.

Возможны две технологии организации циклов – рис. 10.6.

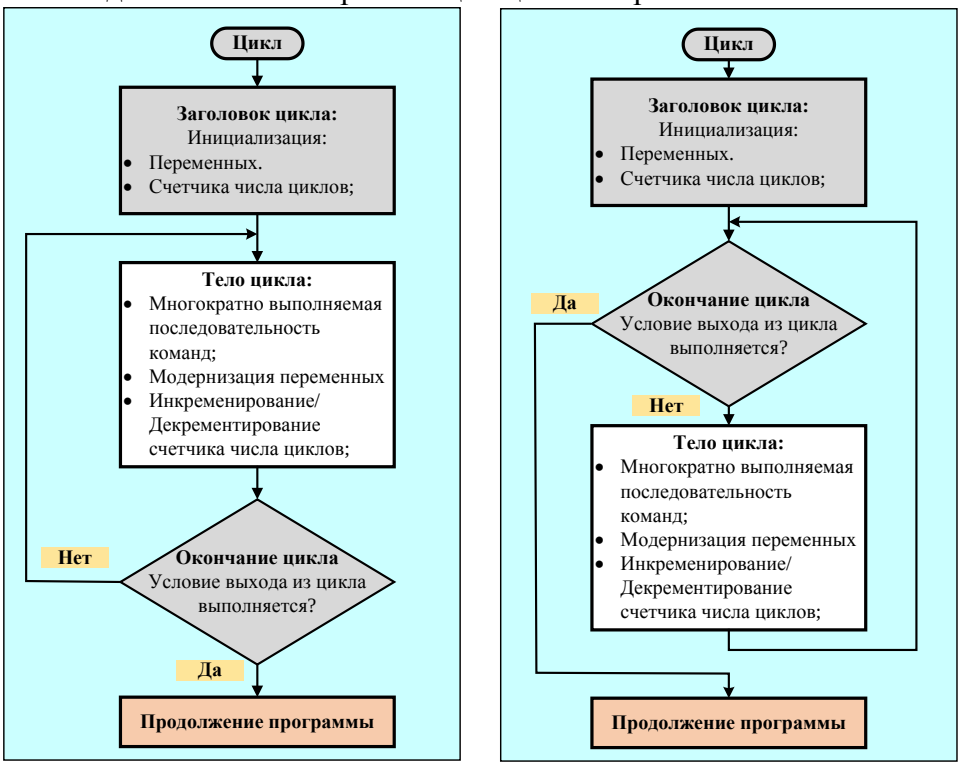


Рис. 10.6 Технологии организации циклов на Ассемблере

Они отличаются порядком расположения *блока проверки условия выхода из цикла* по отношению к блоку тела цикла. В первом случае этот блок располагается после тела цикла, поэтому цикл выполняется, по крайней мере, один раз, после чего следует проверка условия выхода из цикла. Во втором случае условие выхода из цикла проверяется непосредственно перед телом цикла, и поэтому цикл может вообще ни разу не выполниться. В первом случае требуется только один условный переход, причем в начало цикла, если условие выхода из цикла ложное, а во втором – два: один условный переход вперед в обход тела цикла, если условие выхода из цикла истинно, и второй, безусловный переход в начало цикла для очередной проверки условия выхода из цикла.

10.9.1.1 Цикл с пост-проверкой условия выхода

Счетчик числа циклов в теле цикла может обрабатываться по-разному: или декрементироваться или инкрементироваться. В первом случае в блоке проверки условия выхода из цикла значение счетчика проверяется на ноль, а во втором – на достижение заданного числа проходов. Операция декрементирования может сопровождаться попутной установкой *флагов результатов операции* и после нее можно сразу использовать команду условной передачи управления по «не нулю» в начало цикла, без применения дополнительной команды сравнения. Операция инкрементирования потребует дополнительной команды сравнения текущего числа проходов с заданным. Сравните приведенные ниже два варианта структур цикла:

Вариант № 1 – Декрементирование счетчика числа циклов и использование для перехода в начало цикла команды условной передачи управления BNE

```
; Определить переменную – «Число проходов цикла»
N    EQU  10
; Инициализировать счетчик числа циклов
     MOV R0, #N
Loop
; Тело цикла
;    …
; Декрементировать счетчик числа циклов
     SUBS R0, #1          ; «S» - Установить флаги
; Повторить цикл, если значение счетчика циклов не равно нулю
     BNZ  Loop
; В противном случае, выйти из цикла и продолжить выполнение
; программы
Next
;    …
```

Вариант № 2 – Инкрементирование счетчика числа циклов и использование дополнительной команды сравнения CMP для проверки условия выхода из цикла

```
; Определить переменную «Число проходов цикла»
N    EQU  10
; Обнулить счетчик числа выполненных циклов
     MOV R0, #0
Loop
; Тело цикла
;    …
; Инкрементировать счетчик числа выполненных циклов
     ADD R0, #1
; Сравнить число проходов с заданным
     CMP R0, #N
; Повторить цикл, если заданное число проходов не достигнуто
     BNE  Loop
; Выйти из цикла и продолжить выполнение программы
Next
;    …
```

1) Сколько раз будет выполняться цикл в Вариантах № 1 и № 2, если N=10?
2) Сколько раз будет выполняться цикл в Вариантах № 1 и № 2, если N=0?
3) Можно ли заменить в окончании цикла для варианта № 2 условие «Не эквивалентно» (NE) на условие «Строго меньше для сравнения чисел без знака» (LO), на условие «Меньше или равно для сравнения чисел без знака» (LS)?

1) И в том и в другом случае – 10 раз.
2) Первая операция декрементирования счетчика числа циклов в варианте № 1 приведет к установке в регистре R0 значения 4 294 907 295 (максимально возможного целого 32-разрядного числа). Следовательно, всего будет выполнено до обнуления счетчика 4 294 907 296 проходов. Это эквивалентно фактически «бесконечному циклу». В варианте № 2 с каждым проходом тела цикла значение счетчика числа циклов будет увеличиваться на 1 и при выполнении 10 проходов условие выхода из цикла станет истинным. Будьте внимательны с заданием начальных значений в счетчике числа циклов!
3) На условие «LO» – да, на условие «LS» – нет. В последнем случае число выполненных проходов цикла увеличится на 1 по сравнению с заданным.

Принципиальной разницы в этих двух вариантах организации циклов нет, однако предпочтение все же следует отдавать первому варианту, так как при прочих равных условиях он чуть быстрее и проще для понимания.

10.9.1.2 Цикл с пре-проверкой условия выхода

Как уже отмечалось, в этом случае тело цикла может и не быть выполненным. В системе команд процессоров Cortex-M имеются две специальные команды, облегчающие реализацию циклов с пре-проверкой условия выхода из цикла:

```
CBZ Rn, label  - Compare and branch if zero
                 Сравнение и переход, если ноль
CBNZ Rn, label -  Compare and branch if non-zero
                 Сравнение и переход, если не ноль
```

Они выполняют два действия одновременно: проверяют текущее значение в регистре-источнике Rn на ноль и в случае выполнения (Z) или не выполнения (NZ) этого условия обеспечивают передачу управления *вперед* на указанную в команде метку. Обратите внимание на подчеркнутое выше слово – «вперед». Передача управления назад, в сторону уменьшающихся адресов, т.е. в сторону начала цикла – *не поддерживается*. Следовательно, эти команды можно использовать для организации досрочного выхода из цикла или для условных переходов вперед внутри тела цикла. Они оптимальны и для организации циклов с пре-проверкой условия выхода из цикла, как показано ниже:

Вариант № 3 – Декрементирование счетчика числа циклов при каждом проходе и проверка условия выхода из цикла перед телом цикла командой CBZ

```
; Инициализация счетчика числа циклов
     MOV R0, #N
;    …
; Проверить условие выхода из цикла
Cange_Exit
     CBZ R0, Next
; Тело цикла
;    …
; Декрементировать счетчик числа циклов
     SUB R0, #1
; Возврат управления в блок проверки условия выхода из цикла
     B Cange_Exit
Next
;    …
```

Заметьте, что команды CBZ, CBNZ сами тестируют содержимое регистра-источника на ноль. Поэтому специально выставлять флаги с помощью суффикса «S», как в команде SUB не нужно.

Мы рассмотрели реализацию простых условий выхода из цикла по текущему состоянию специальной переменной-счетчика числа циклов. В общем случае выход из цикла выполняется по результату сравнения любой переменной с другой переменной или с заданным значением. Условием выхода из цикла может быть также проверка на истинность любого логического выражения. Циклы с пост-проверкой условия выхода аналогичны операторам языков высокого уровня типа Repeat … Until <Cond> – «Повторять, пока логическое условие не будет истинным», а циклы с пре-проверкой условия операторам Wihle <Cond> Do… – «Пока логическое условие истинно, выполнять» (более подробно в главе 15).

10.10 Разработка подпрограмм копирования данных в памяти

Одной из наиболее часто решаемых на практике задач является задача *копирования массива констант* из ПЗУ в ОЗУ микропроцессорной системы. Она может возникнуть, например, при инициализации значений переменных в ОЗУ. При загрузке рабочей программы начальные значения *инициализируемых переменных* сначала загружаются в ПЗУ вместе с кодом программы, а затем, перед выполнением программы, копируются в нужные области оперативной памяти. При разработке программ на C/C++ эту функцию выполняют специальные подпрограммы в стартовом файле проекта.

Возьмем за основу один из предыдущих проектов, в которых в конце рабочей программы была выполнена инициализация таблицы констант в кодовой секции, и изменим основную программу так, чтобы она вызывала подпрограмму, выполняющую копирование массива слов заданной длины из ПЗУ в ОЗУ – проект CPU_6 (MyProg_6.s). Прежде всего договоримся, какие параметры будем передавать в подпрограмму по имени Copy_Rom_Ram. Очевидно, что это должны быть три параметра: начальный адрес массива констант в ПЗУ (A_Rom); начальный адрес массива переменных в ОЗУ (A_Ram) и длина массива в количестве 32-разрядных слов (N_32).

Последний параметр зависит исключительно от приложения и может быть установлен в начале программы директивой определения символической переменной EQU.

```
 7  MyProg
 8  ; Обявить и присвоить значение переменной N_32
 9  ; "Число слов в массиве"
10  N_32    EQU 4
```

```
42  ; Инициализация констант в текущей кодовой секции
43  A_ROM
44  Const_0 DCD 0x7788FFAA
45  Const_1 DCD 0x22221111
46  Const_2 DCW 34,35
47  Const_3 DCB 1,2,3,4
```

Значение первого параметра (A_ROM) будет автоматически определено при резервировании и инициализации констант в конце нашей программы – это адрес фактического размещения таблицы констант в памяти.

```
50  ; Объявить секцию данных в оперативной памяти
51          AREA MyData, Data, ReadWrite
52  ; Зарезервировать в ней 100 слов
53  A_RAM   SPACE 4*N_32
54  ; Конец ассемблерного текста
55          END
```

Значение второго параметра (A_RAM) будет определено при резервировании заданного объема ОЗУ под переменные проекта, что нужно сделать, например, сразу после кодовой секции, объявив секцию данных MyData. Директива SPACE резервирует нужное число слов в памяти данных, начиная с адреса A_RAM, фактическое значение которого будет известно только после компоновки проекта. Заметьте, что для определения объема резервируемой области памяти мы использовали арифметическое выражение (4*N_32), в котором символ N_32 уже определен нами в начале программы.

```
19  ; ******************************************
20  ; Подпрограмма копирования слов из ПЗУ в ОЗУ
21  ; Входные параметры:
22  ; r0 - начальный адрес массива слов в ПЗУ
23  ; r1 - начальный адрес массива слов в ОЗУ
24  ; r2 - число элементов массива
25  ; Используемые регистры:
26  ; r3 - для временного хранения слов
27  ; ******************************************
```

Итак, создадим заголовок подпрограммы. Это комментарий, который содержит название подпрограммы, перечень входных параметров, передаваемых ей из основной программы содержимым регистров ЦПУ с указанием их имен, перечень используемых в подпрограмме регистров (значения которых могут «портиться» при работе программы). Если подпрограмма возвращает значение в основную программу – отмечается, каким образом, например, содержимым одного из регистров. В нашем случае подпрограмма лишь выполняет копирование данных из одного места памяти в другое.

Существуют разные *способы передачи параметров* в подпрограммы и возврата из них результата, среди которых один из самых простых и поэтому чаще используемых – *значениями регистров ЦПУ*. Для удобства пользователей фирма ARM разработала стандарт передачи параметров в подпрограммы, в котором, в частности, рекомендуется использовать для этой цели регистры младшего регистрового банка (желательно r0-r3). В заголовке подпрограммы может указываться и другая информация, вплоть до фамилии разработчика, даты создания подпрограммы и даты последнего обновления.

```
28  ; Точка входа в подпрограмму
29  Copy_Rom_Ram
30  ; Считать текущее слово из ПЗУ и сохранить в ОЗУ
31  ; с использованием базовой адресации с пост-смещением
32          LDR r3,[r0],#4
33          STR r3,[r1],#4
34  ; Декрементировать счетчик числа элементов массива
35          SUB r2,#1
36  ; Сравнить с нулем (все ли слова скопированы?)
37          CMP r2,#0
38  ; Если не все - повторить операцию копирования
39          BNE Copy_Rom_Ram
40  ; Все слова скопированы, возврат в основную программу
41          BX lr
```

Где размещается подпрограмма? Обычно в конце основной программы, в текущей кодовой секции, перед константами и таблицами констант, если они есть. Она представляет собой цикл, в котором слово считывается из массива-источника в регистр временного хранения данных r3 и тут же сохраняется в массиве-приемнике. Для этого применяется *базовая адресация с пост-смещением указателя*.

Оба регистра указателя r0 и r1 автоматически получают приращение +4 для доступа к очередной ячейке памяти массива-источника и приемника в следующем

«*проходе тела*» цикла. Счетчик числа циклов (регистр r2) декрементируется, и результат проверяется на ноль командой сравнения. Для выработки флагов можно использовать также команду SUB с установленной опцией «S» – SUBS. В последнем случае команда сравнения не нужна. Если не все слова скопированы – цикл повторяется. Для этого используется команда условной передачи управления по признаку «Не эквивалентно» (NE). В противном случае выполняется возврат в основную программу (содержимое регистра связи LR загружается в счетчик команд PC).

```
11  ; Загрузить параметры и вызвать подпрограмму
12  ; копирования массива слов
13          ADR r0,A_ROM
14          LDR r1,=A_RAM
15          MOV r2,#N_32
16          BL Copy_Rom_Ram
17  ; Зациклить программу
18  Stop    B Stop
```

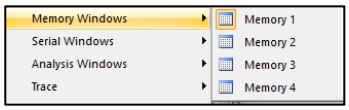
Таким образом, подпрограмма может скопировать нужное число слов из одного места памяти в другое (не обязательно из ПЗУ в ОЗУ, но и из одной области ОЗУ в другую). Вернемся к основной программе и покажем, как выполняется передача параметров в подпрограмму. Для передачи первого параметра мы применили псевдокоманду Ассемблера «Загрузить адрес данных по их метке» ADR, для второго параметра – псевдо-команду загрузки регистра 32-разрядным числом с одновременным созданием литерального пула, содержащего этот адрес, а для третьего параметра – обычную команду MOV с непосредственной адресацией операнда. Дело в том, что данные в ПЗУ расположены «близко» и команда относительной адресации по текущему содержимому счетчика команд ADR может быть сгенерирована. Область ОЗУ начинается в соответствии с картой памяти с адреса 0x20000000. Это «длинный адрес», использование для загрузки которого командой ADR вызовет ошибку. Приходится генерировать команду, которая будет использовать копию «длинного адреса» в литеральном пуле. Байтовые константы, напротив, можно загрузить и простой командой пересылки.

Далее следует вызов подпрограммы. После возврата основная программа зацикливается для удобства отладки. Естественно, что эту подпрограмму можно вызвать из основной программы произвольное число раз с нужными параметрами.

Выполните компиляцию и компоновку проекта CPU_6. Загрузите программу в отладчик.

10.11 Как открыть окно содержимого памяти?

При работе с памятью целесообразно окрыть окно содержимого памяти (дампа памяти) для наблюдения за результатом работы программы. Сделать это можно, находясь в отладчике, в меню «View» («Просмотр»), выбрав одну из четырех возможных областей памяти, например, Memory 1. Сложность может быть в том, что Вы не знаете реального начального адреса области ОЗУ.




```
MyData                          0x20000000   Section       16  myprog_6.o(MyData)
A_RAM                           0x20000000   Data          16  myprog_6.o(MyData)
STACK                           0x20000010   Section     1024  startup_1.o(STACK)
__initial_sp                    0x20000410   Data           0  startup_1.o(STACK)
```

Загляните в карту загрузки CPU_6.map, созданную компоновщиком, и найдите в ней значение символа A_RAM. В фрагменте этого файла Вы видите, что секция данных MyData начинается с адреса 0x20000000 и занимает в памяти всего 4 слова. Дальше зарезервировано место для стека (1 Кбайт), который пока не используется. Следовательно, для наблюдения за процессом копированием данных в ОЗУ нужно открыть окно памяти с начальным адресом 0x20000000.

Проследите за выполнением программы в регистровом окне и в окне дампа памяти в пошаговом режиме работы и в режиме прогона. Обратите внимание, что адрес возврата при обращении к подпрограмме сохраняется в регистре связи LR, а при возврате из нее восстанавливается из регистра LR в счетчик команд PC. Содержимое окна памяти доказывает, что программа работает. Цветом выделены байты памяти, содержимое которых при выполнении программы изменилось. Все четыре слова скопированы в ОЗУ, что и требовалось.

1) Модифицируйте программу и подпрограмму так, чтобы данные, расположенные в ПЗУ, копировались по полуслову. Выполните отладку. Результат копирования такой же или отличается?

2) Сделайте то же самое, но для копирования массива по байтам. Выполните отладку.

3) Так какая же из трех подпрограмм лучше?
4) В нашей подпрограмме выполняются «как бы лишние» операции пост-смещения указателей. Можно ли написать подпрограмму так, чтобы значения указателей не выходили за пределы зарезервированной для массивов памяти. Попробуйте.
5) Оставьте те же директивы резервирования констант в кодовой памяти. Проверьте, как работает команда двойной загрузки слов в регистры ЦПУ, сложите первые две константы Const_0 и Const_1 после загрузки. Оцените результат.
6) Зарезервируйте в кодовой секции строку символов “Privet ARM Cortex-M4F CPU” с использованием директивы DCB. В качестве признака конца строки (прерывателя строки) используйте нулевой байт 0. Напишите подпрограмму копирования в ОЗУ строки символов любой длины, кончающейся «прерывателем». Проверьте ее работу в отладчике.

1) В подпрограмме необходимо заменить команды загрузки/сохранения слов на команды загрузки/сохранения полуслов с использованием базовой адресации с пост- смещением содержимого указателя. Естественно, число проходов цикла необходимо увеличить вдвое:

```
28  Copy_H_Rom_Ram
29  ; Считать текущее полуслово из ПЗУ и сохранить в ОЗУ
30  ; с использованием базовой адресации с пост-смещением
31          LDRH r3,[r0],#2
32          STRH r3,[r1],#2
```

```
MOV r2, #(N_32*2),
```

чтобы скопировать все данные в ПЗУ. Результат будет точно таким же. Модифицированный файл MyProg_6_1.s.

2) Изменения в подпрограмме касаются использования команд загрузки/сохранения байт с базовой адресацией и пост-смещением, а также с увеличением числа проходов цикла в основной программе (`MOV r2, #(N_32*4)`). Результат копирования тот же. (см. MyProg_6_2.s).

```
28  Copy_B_Rom_Ram
29  ; Считать текущий байт из ПЗУ и сохранить в ОЗУ
30  ; с использованием базовой адресации с пост-смещением
31          LDRB r3,[r0],#2
32          STRB r3,[r1],#2
```

3) Ответ зависит от того, что за данные находятся в исходном массиве. Если они однородные и их число кратно 4-м, то разумнее использовать самую быструю процедуру копирования слов. Если данные – байты или полуслова, число которых кратно 2-м, то процедуру копирования полуслов. В общем случае, при неизвестных заранее объемах пересылки используется побайтовое копирование.

4) Да. Придется изменить логику организации цикла: тестировать счетчик числа циклов не в конце, а перед телом цикла – использовать технологию *пре-проверки условия выхода из цикла*. (MyProg_6_3.s). Если счетчик будет обнулен, достаточно просто выйти из подпрограммы, используя для этого команду условного возврата в основную программу по нулю EQ («Эквивалентно») – BXEQ lr. Для проверки состояния счетчика можно выполнить пересылку данных из него (r2) в регистр временного хранения информации r3 с опцией установки флагов результатов операции MOVS. Как видите, если начальное значение счетчика циклов равно нулю, то тело цикла не выполняется ни разу и сразу следует возврат в основную программу.

```
25  ; Точка входа в подпрограмму
26  Copy_B_Rom_Ram
27  ; Загрузить в регистр временного хранения r3
28  ; текущее число нескопированных элементов массива
29  ; с установкой флагов (суффикс "S")
30          MOVS r3,r2
31  ; Если все элементы скопированы, выйти из подпрограммы
32          BXEQ lr
33  ; Считать текущий байт из ПЗУ и сохранить в ОЗУ
34  ; с использованием базовой адресации с пост-смещением
35          LDRB r3,[r0],#1
36          STRB r3,[r1],#1
37  ; Декрементировать счетчик числа элементов массива
38          SUB r2,#1
39  ; Переход в начало подпрограммы
40          B Copy_B_Rom_Ram
```

5) Обратитесь к файлу MyProg_6_4.s, содержащему пример решения задачи. Сначала выполняется инициализация базового регистра r0 начальным адресом области ПЗУ, содержащей таблицу констант. Далее командой двойной загрузки слов LDRD в регистры назначения r1, r2 выполняется считывание сразу двух констант Const_0 и Const_1, которые расположены в памяти по последовательным адресам. Теперь достаточно выполнить трех-операндную команду сложения с сохранением результата в регистре r3. Результат работы программы показывает, что загрузка данных и их обработка выполнены правильно.

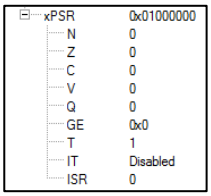
```
7   MyProg
8   ; Демонстрация работы команд двойной загрузки слов
9           ADR r0,ROM_0
10          LDRD r1,r2,[r0,#0]
11  ; Сложить оба слова в регистрах r1, r2
12          ADD r3, r1,r2
```

R1	0x7788FFAA
R2	0x22221111
R3	0x99AB10BB

6) Каждый символ имеет свой код ASCII (американский стандарт представления символьной информации в компьютерах) и хранится в памяти в виде байта. Строка символов выглядит в памяти как последовательность байт – ASCII-кодов соответствующих символов. Закодировать отдельный символ можно, поместив его в *одиночные кавычки*, а строку символов – в *двойные кавычки*. Для этой цели используется директива резервирования и инициализации байт DCB. Обратите внимание на то, что в конце последовательности кодов символов мы разместили байт-признак конца строки. Это сделано для того, чтобы в процессе посимвольного копирования строки, прекратить копирование после обнаружения кода «конца строки». В подпрограмме используется пост-проверка условия выхода из цикла. Для этого код текущего символа сравнивается с кодом конца строки (0). Если конец строки достигнут – выполняется возврат в основную программу. В процессе отладки подпрограммы проследите за формированием флага Z (нулевого результата). Для этого раскройте содержимое регистра статуса программы пользователя xPSR в окне регистров. Пример программы приложения –в файле MyProg_6_5.s.

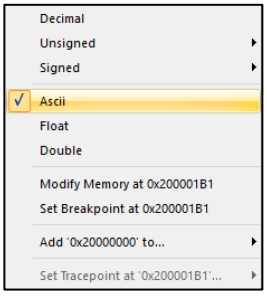
```
; Инициализация строки в кодовой памяти
ROM_0
String_1 DCB "Privet ARM Cortex-M4F CPU", 0
```

```
16  ; ****************************************
17  ; Подпрограмма копирования строки из ПЗУ в ОЗУ
18  ; Входные параметры:
19  ; r0 - начальный адрес строки символов в ПЗУ
20  ; r1 - начальный адрес копиии строки в ОЗУ
21  ; ; Используемые регистры:
22  ; r3 - для временного хранения символа (байта)
23  ; Последним символом строки должен быть 0 -
24  ; конца строки (терминатор строки)
25  ; ****************************************
26  ; Точка входа в подпрограмму
27  Copy_String
28  ; Загрузить в регистр временного хранения r3
29  ; ASCII-код очередного символа строки - источника
30  ; Скопировать символ в ОЗУ. Использовать
31  ; базовую адресацию с пост-смещением указателей
32          LDRB r3,[r0],#1
33          STRB r3,[r1],#1
34  ; Сравнить код символа с признаком конца строки
35          CMP r3,#0
36  ; Если не все символы скопированы, повторить
37          BNE Copy_String
38  ; Получен символ конца строки.
39  ; Возврат в основную программу
40          BX lr
```

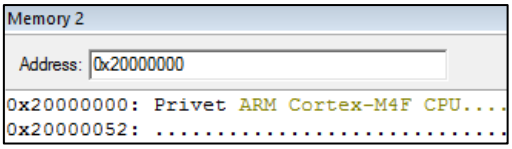


10.12 Как управлять форматом выдачи данных в окно памяти при отладке?

Находясь в окне дампа памяти, Вы можете воспользоваться *контекстно-зависимым меню* и изменить представление данных в памяти для удобства анализа расположенных в ней данных. Так, при выполнении последнего задания целесообразно открыть окно памяти, начиная с адреса 0x20000000 (начала секции данных в ОЗУ) и вызвать это меню правой клавишей мыши. Теперь можно изменить формат представления данных в окне, выбрав обычное десятичное представление (если опция Decimal установлена), или шестнадцатеричное (если сброшена – нет «галочки» в позиции Decimal), Знаковое (Signed) или беззнаковое (Unsigned) представление. В средней части меню можно выбрать особые форматы представления данных: в виде ASCII-символов (Ascii) - текстового сообщения; в виде 32-разрядных чисел в формате с плавающей точкой однократной (Float) или двойной точности (Double).



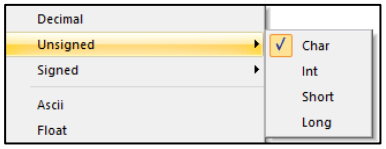
Выберите для приложения MyProg_6_5.s представление информации в памяти в виде символов (Ascii). Тогда, при завершении программы Вы получите в окне дампа памяти соответствующее текстовое сообщение, как при выводе текста на экран компьютера. Оно свидетельствует о том, что заданная в ПЗУ строка символов скопирована правильно. К сожалению, в это окно возможен вывод только латинских символов (кириллица не поддерживается). Вывод строковых сообщений в окно дампа памяти помогает в отладке сложных программ, визуализируя этот процесс.



Обозначения форматов данных в Ассемблере и в C/C++ несколько отличаются. Среда разработки µVision поддерживает обозначения, принятые в СИ: символ, короткое целое, целое, длинное целое. В таблице 10.6 показано соответствие этих форматов в СИ и Ассемблере (*в скобках*), а также взаимное расположение данных: каждое короткое целое Short (Полуслово) состоит из двух символов Char (Байт); каждое целое Int (Слово) из двух коротких целых (полуслов); каждое длинное целое Long (Длинное слово) из двух обычных 32-разрядных слов:

Таблица 10.6 Возможные форматы вывода данных в окно дампа памяти

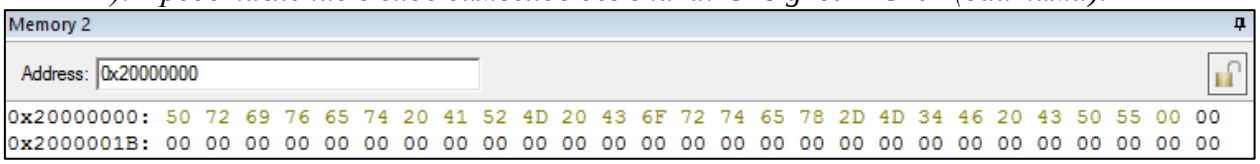
Char (Байт)	**Char (Байт)**	**Char (Байт)**	**Char (Байт)**	**Char (Байт)**	**Char (Байт)**	**Char (Байт)**	**Char (Байт)**
Short (Полуслово -16 р.)		**Short (Полуслово -16 р.)**		**Short (Полуслово -16 р.)**		**Short (Полуслово -16 р.)**	
Int (Слово – 32 р.)				**Int (Слово – 32 р.)**			
Long (Длинное слово – 64 р.)							



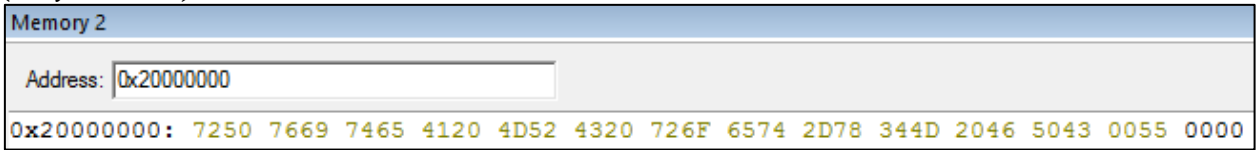
Среда µVision позволяет выбрать для дампа памяти наиболее удобный формат представления данных. Для этого нужно всего лишь поставить «галочку» рядом с обозначением соответствующего формата. Данные могут быть представлены как числа без знака, так и со знаком.

Рассмотрим варианты вывода данных для нашей конкретной программы. Используем Hex-формат:

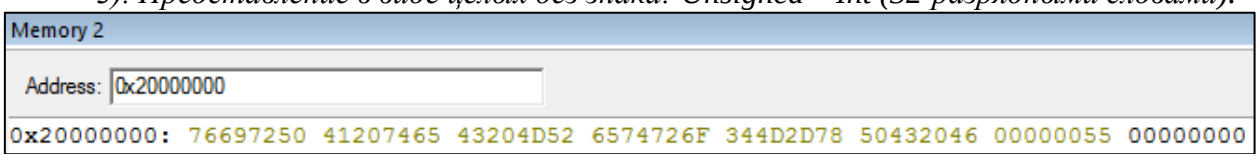
1). Представление в виде символов без знака: Unsigned – Char (байтами):



2). Представление в виде коротких целых без знака: Unsigned – Short (полусловами):



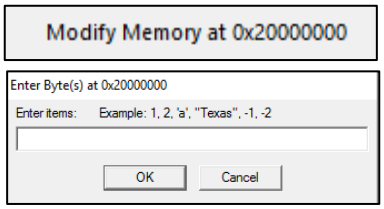
3). Представление в виде целых без знака: Unsigned – Int (32-разрядными словами):



Первое представление позволяет проанализировать ASCII-коды использованных в строке символов. Например, символу пробела соответствует код 0x20, символу ‘e’ – код 0x65, символу ‘M’ – код 0x4D и т.д. Во втором представлении каждая пара байтов

объединяется в полуслово, и значения полуслов выводятся в удобном для программиста виде – сначала старший байт, затем младший (реально данные располагаются в памяти, начиная с младшего байта). В третьем варианте каждые четыре байта объединяются в одно 32-разрядное слово, которое также представляется в удобном для анализа виде.

Если Вам не удобен Hex-код, выберите десятичный формат представления данных. Таким образом, среда µVision может выполнять *функции встроенного калькулятора*, быстро преобразуя данные в памяти из одного формата в другой (Hex ↔ Dec), *в том числе из целочисленного формата в формат с плавающей точкой и обратно*, что часто требуется в процессе отладки. Контекстное меню окна памяти позволяет также оперативно *модифицировать значения в ячейках памяти* и устанавливать точки останова. Для изменения содержимого ячейки памяти достаточно щелкнуть правой клавишей мыши на содержимом этой ячейки (она будет выделена пунктирной рамкой) вызвать контекстно-зависимое меню правой клавишей мыши, в котором выбрать команду модификации содержимого этой ячейки. Появится окно со строкой ввода и примерами ввода данных. Данные можно вводить цифрами, символами, строкой символов или последовательностью цифр, разделенных запятой. При множественном вводе данные будут вводиться в память последовательно, начиная с указанного адреса.



Более простой метод ввода значений в память состоит в том, чтобы дважды щелкнуть на содержимом ячейки памяти левой клавишей мыши. Ее содержимое будет выделено фоном, и Вы сможете модифицировать его, вводя нужную последовательность цифр, заканчивая ввод клавишей «Enter». Ограничение метода состоит в том, что ввод возможен только в текущем формате, например, Hex-цифрами, если соответствующее представление данных установлено. Таким образом корректируется значение только в одной ячейке памяти.

1) Поэкспериментируйте с выводом информации в окно дампа памяти в различных форматах, например, в формате Char (байта) в десятичном коде.
2) Попробуйте модифицировать одну ячейку памяти и целую область, начиная с определенного адреса.
3) Проверьте, правильно ли вводятся отрицательные числа? Убедитесь в этом, изменив формат ввода на Hex-формат.
4) Проверьте, как работает технология модификации текущего значения в одной ячейке памяти по двойному щелчку мыши.

Среда µVision предоставляет удобные инструменты для начальной инициализации значений переменных или массивов в любых областях памяти, а также для их модификации в процессе отладки. При этом данные могут вводиться и отображаться в наиболее удобной для программиста форме.

11 АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ

Оглавление

Арифметические операции являются основными, определяющими вычислительную производительность процессора. В процессорах ARM, кроме традиционных для микроконтроллеров команд сложения и вычитания, имеются также команды умножения и деления. Этот набор перекрывает весь спектр операций, необходимых для эффективных вычислений. Так как ядро процессора 32-разрядное, большинство операций выполняется над словами, хотя, как Вы уже заметили в предыдущей главе, процессор без труда может обрабатывать и данные меньшего формата – полуслова и байты. При этом загрузка полуслов и байт на обработку из памяти сопровождается автоматическим преобразованием их формата к формату слова. Это преобразование выполняется по-разному для чисел без знака и со знаком в дополнительном коде. Так как разрядная сетка ЦПУ 32-разрядная, параллельно в АЛУ могут обрабатываться и два полуслова и 4 байта, входящих в слово. При этом, правда, необходимо соблюдать определенную осторожность, так как обычные флаги результатов операций не формируются, хотя и имеются дополнительные возможности определения промежуточных переполнений (см. приложение 1).

11.1 Команды сложения и вычитания слов

Синтаксис 32-разрядных операций сложения и вычитания:

```
ADD{S}{cond} {Rd}, Rn, Op2
ADC{S}{cond} {Rd}, Rn, Op2
SUB{S}{cond} {Rd}, Rn, Op2
SBC{S}{cond} {Rd}, Rn, Op2
RSB{S}{cond} {Rd}, Rn, Op2
```

Второй операнд-источник: 1) #imm; 2) Rm; 3) Rm, Shift

Первый операнд-источник

Регистр – приемник (опция)

Код условного выполнения команды (опция)

S (SET) – Опция установки флагов результатов операции: N, Z, C, V

ADD – (ADD) - Сложить 32-разрядные операнды:	**Rd ← Rn + Op2**
ADC – (ADD with Carry) - Сложить с учетом переноса:	**Rd ← Rn + Op2 + Carry**
SUB – (SUB) - Вычесть 32-разрядне операнды:	**Rd ← Rn - Op2**
SBC – (SUB with Carry) - Вычесть с учетом флага C (Заема):	**Rd ← Rn + Op2 - /Carry**
RSB – (Reverse Subtrucnt) - Реверсивное вычитание:	**Rd ← Op2 - Rn**

Все команды могут быть трехоперандными, причем в качестве второго операнда - источника можно использовать универсальный второй операнд: непосредственную константу, регистр ЦПУ или регистр ЦПУ, содержимое которого подвергается предварительной обработке в кольцевом сдвиговом регистре процессора, например, масштабированию. Имя регистра-приемника является опциональным. Если оно опущено, то функцию приемника выполняет первый операнд-источник Rn – команда становится двухоперандной.

Для расширения эффективной разрядности процессора при обработке данных (в операциях многословной арифметики) предусмотрены команды сложения с учетом ранее возникшего переноса C (Carry) и вычитания с учетом ранее возникшего «заема». Его функцию в процессорах ARM также выполняет флаг C: он сбрасывается, если при вычитании был «заем» и выставляется – если «заема» не было. Можно считать, что флаг Carry является инверсией флага «заема» Borrow при вычитании.

В общем случае флаги результатов операций не модифицируются. Для того, чтобы они были выработаны, необходимо использовать команды с дополнительным суффиксом

«S» - установки флагов. При этом будут сформированы флаги N, Z, C, а также флаг «знакового переполнения» V.

Интерес представляет команда реверсивного (обратного) вычитания, когда из значения второго операнда вычитается значение первого. Она находит широкое применение, например, для инвертирования знака числа: RSB r1, #0. Из нуля вычитается содержимое регистра r1 и сохраняется в том же регистре. Следовательно, это команда *инвертирования знака числа*, расположенного в регистре r1.

В конце предыдущей главы мы рассмотрели основные приемы наблюдения за содержимым памяти в среде µVision, которые позволяют исследовать это содержимое, рассматривая его как данные в любом формате (числа со знаком или без знака). Содержимое памяти может быть представлено в десятичной или шестнадцатеричной системе счисления. Кроме того, при отладке программист может быстро ввести в память любые исходные данные и на этом наборе переменных проверить работу программы. Фактически мы имеем встроенный «преобразователь» типовых форматов данных в памяти. Рекомендуем активно пользоваться этой возможностью, в том числе для отладки программ, содержащих вычислительные блоки.

```
33          ALIGN
34  ; Объявить секцию данных в оперативной памяти
35          AREA MyData, Data, ReadWrite
36          EXPORT Array_W
37          EXPORT Sum_W
38          EXPORT Sub_W
39  ; Зарезервировать в ней 4-е 32-разрядных слова
40  ; источников данных (без инициализации)
41  Array_W SPACE 4*4
42  ; Зарезервировать в ОЗУ место для размещения
43  ; результов сложения и вычитания слов
44  Sum_W   SPACE 4*2
45  Sub_W   SPACE 4*2
```

Покажем, как это делать, на примере проекта CPU_7. В конце программы приложения MyProg_7.s объявляется секция данных и в ней резервируется массив из 4-х слов Array_W. Он будет содержать исходные данные, которые мы можем ввести, используя окно дампа памяти, непосредственно перед выполнением программы в отладчике.

```
 7  MyProg
 8  ; Операции сложения/вычитания слов
 9  ; Выполнить попарное сложение/вычитание
10  ; 32-разрядных слов из массива Array_W.
11  ; Сохранить суммы в памяти по адресу Sum_W
12  ; Сохранить разности в памяти по адресу Sub_W
13    ; Инициализация указателей
14          LDR r0,=Array_W
15          LDR r4,=Sum_W
16          LDR r5,=Sub_W
17    ; Первая сумма и первая разность
18          LDR r1,[r0],#4
19          LDR r2,[r0],#4
20          ADD r3,r1,r2
21          STR r3,[r4],#4
22          SUB r3,r1,r2
23          STR r3,[r5],#4
24
25    ; Вторая сумма и вторая разность
26          LDR r1,[r0],#4
27          LDR r2,[r0]
28          ADD r3,r1,r2
29          STR r3,[r4]
30          SUB r3,r1,r2
31          STR r3,[r5]
```

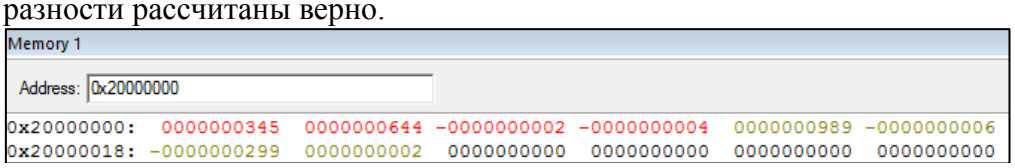
Будем исследовать операции сложения и вычитания слов, поэтому зарезервируем еще два слова для сохранения в памяти результатов попарного суммирования исходных слов массива – Sum_W и Sub_W – для сохранения результатов попарного вычитания исходных слов. Сама программа выполняет установку указателей в регистрах r0, r4 и r5 на начальные адреса этих областей памяти, а затем, с использованием команд загрузки и сохранения данных в памяти (базовая адресация с пост-смещением и обычная косвенная адресация) выполняет попарный расчет сумм и разностей слов, находящихся в массиве Array_W. При этом слова сначала загружаются в регистры r1, r2, а затем выполняется расчет сумм и разностей с сохранением результатов в памяти.

Обратите внимание на то, что начальные адреса областей ОЗУ объявлены глобальными переменными (директива EXPORT). Это сделано для того, чтобы они попали в таблицу символов проекта CPU_7.map, и мы узнали фактическое размещение переменных в памяти. После «сборки» проекта нужно открыть окно памяти, начиная с адреса 0x20000000 (см. файл CPU_7.map).

```
Array_W                          0x20000000   Data          16  myprog_7.o(MyData)
Sum_W                            0x20000010   Data           8  myprog_7.o(MyData)
Sub_W                            0x20000018   Data           8  myprog_7.o(MyData)
```

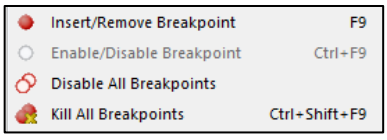
Первые четыре слова – массив исходных данных, два следующих слова – результаты попарного суммирования, следующие два слова – результаты попарного вычитания исходных операндов.

Представим исходные данные и результаты расчетов в удобной для наблюдения десятичной системе счисления в формате знаковых длинных целых чисел (Decimal, Signed, Int) слов со знаком. Загрузим программу в отладчик, откроем окно дампа памяти и заполним первые четыре слова произвольными десятичными кодами. Выполним программу в пошаговом режиме и проследим за тем, как исходные данные последовательно считываются из памяти: при каждом доступе к очередной ячейке памяти ее содержимое выделяется красным цветом. При обращении к памяти по записи результат записи выделяется в окне памяти салатовым цветом. Убедитесь, что обе суммы и обе разности рассчитаны верно.



11.2 Технология отладки программ с точками останова

Интегрированная среда разработки µVision предоставляет широкие возможности по отладке программ. Наиболее часто используется выполнение программы в режиме «прогона» (RUN) и пошаговое выполнение. Одним из очень эффективных средств является также отладка программы с *точками останова*. Точкой останова в простейшем случае является метка какой-то команды (адрес ее расположения), при достижении которой в режиме прогона следует останов программы, чтобы программист смог исследовать промежуточные результаты в регистрах процессора и памяти и убедиться в правильности выполнения определенного фрагмента программы. Если все правильно, можно выполнить программу в режиме прогона до следующей точки останова и убедиться в правильности работы очередного фрагмента. Таким образом, отладка с точками останова – это поблочная отладка программы с промежуточным контролем результатов. В общем случае точки останова могут быть и более сложными, например, по доступу процессора к указанному адресу в памяти. В этом случае прерывание наступит тогда, когда процессор обратится к памяти по заданному адресу.

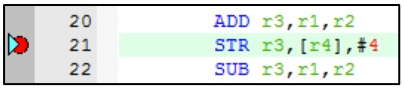


Установка и удаление простых точек останова выполняется командами из меню Debug (Отладка) или соответствующими пиктограммами на панели инструментов. Первая команда позволяет установить/удалить точку останова, вторая – разрешить/запретить ее использование (активировать/деактивировать), третья – запретить все точки останова, четвертая – удалить все точки останова.

Технология установки точки останова:

1) находясь в режиме отладки, щелкните мышью в окне программы пользователя в соответствующей строке программы – эта строка будет выделена голубым треугольником в начале строки;

2) выполните команду установить точку останова – рядом с треугольником появится красный «кружок» (символ точки останова). Можете выполнить программу в режиме прогона. При достижении точки останова произойдет автоматический останов.



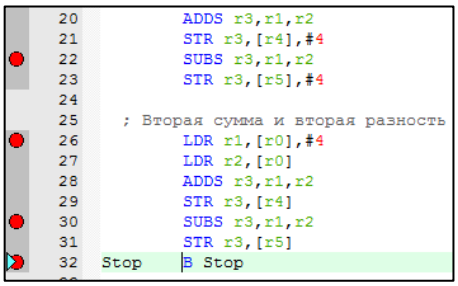
Допускается устанавливать любое число точек останова. Точку останова *по доступу к конкретной ячейке памяти* можно установить из контекстного меню окна памяти. Для этого достаточно щелкнуть левой клавишей мыши на содержимом нужной ячейки (она будет выделена пунктирным прямоугольником) и вызвать контекстное меню правой клавишей мыши. В этом меню имеется команда *«Set Breakpoint at ...»* «Установить точку останова» по текущему адресу.

1) Модернизируйте программу так, чтобы по результатам операций сложения и вычитания вырабатывались флаги N, Z, C, V.

2) Введите те же исходные данные в память, что и в примере выше. Установите точки останова после выполнения каждой операции сохранения результата арифметической операции в памяти. Раскройте в регистровом окне содержимое регистра статуса программы PSR для наблюдения за состоянием флагов. Выполните отладку программы по точкам останова с контролем состояния флагов после каждой операции. Объясните принцип формирования флагов для каждой арифметической операции.

3) Измените набор исходных данных таким образом, чтобы при первой операции возник флаг знакового переполнения V. Проверьте в отладчике.

1) Необходимо добавить к мнемонике команд сложения и вычитания суффикс «S» – установки флагов (MyProg_7_1.s).



2) Вид исходной программы после установки точек останова изменится. Точку останова желательно ставить на следующей команде, после завершения очередного вычисления и сохранения результата в памяти. При этом можно будет одновременно просмотреть и состояние флагов в окне регистров и результат расчета в памяти. Для тех же исходных данных получим: флаг N вырабатывается только при получении отрицательного результата; флаг Z всегда сброшен, так как все результаты – ненулевые; флаг C возник как флаг переноса при сложении двух чисел (0xFFFFFFFE и 0xFFFFFFFC) и как флаг отсутствия «заема» при вычитании; флаг переполнения не вырабатывается, так как все результаты находятся в допустимом диапазоне чисел со знаком в дополнительном коде.

№	Операция	Состояние флагов			
		N	Z	C	V
1	345+644=989	0	0	0	0
2	345-644=-299	1	0	0	0
3	-2+(-4)=-6	1	0	1	0
4	-2- (-4)=+2	0	0	1	0

3) Добавим к максимально возможному положительному числу (0x7FFFFFFF) еще одно положительное (0x2) – результат выйдет за диапазон допустимых чисел со знаком в

дополнительном коде (станет как бы отрицательным) – будут установлены и флаг N и флаг V (переполнения). Убедитесь в этом.

11.3 Использование командного окна среды µVision и окна наблюдаемых переменных при отладке программ

11.3.1 Командное окно

Интегрированная среда µVision имеет широкие возможности по отладке программ, с которыми мы уже частично познакомились. Вы знаете, как просматривать и модифицировать содержимое регистров ЦПУ и содержимое памяти, выбирать в окне памяти наиболее удобную форму представления данных. Сейчас мы рассмотрим некоторые, часто используемые при отладке, возможности **Командного окна** (**Command Window**), которое предоставляет программисту дополнительный набор команд по управлению отладкой. Открыть это окно, как обычно, можно из меню **View** (**Просмотр**).

В нижней части окна Command имеется строка ввода *команд управления отладкой*, начинающаяся с «промпта» >, как в операционной системе DOS. Это символ приглашения пользователя к вводу команд управления отладкой. Среди множества команд выделим те, которые целесообразно использовать начинающим программистам.

11.3.1.1 Вывод на дисплей текущего содержимого регистров и переменных. Функция встроенного калькулятора.

Команда вывода на дисплей командного окна текущего значения регистра или переменной – одна из самых простых. Все, что нужно сделать – это ввести в командной строке имя регистра или переменной и нажать клавишу «Enter» (Ввод). Текущее содержимое регистра или переменной будет выведено на дисплей командного окна *в системе счисления по умолчанию* (возможны два варианта – в десятичной или в шестнадцатеричной системе счисления).

В общем случае, можно ввести любое арифметическое выражение, в состав которого могут входить также символические имена регистров процессора, и заставить µVision вычислить это выражение – *функция встроенного калькулятора*. В примере ниже (рис. 11.1) мы запросили вывод на дисплей окна «Command» текущих значений регистров r0, r1, а затем двух арифметических выражений (r1+5) и (r1+0xFF).

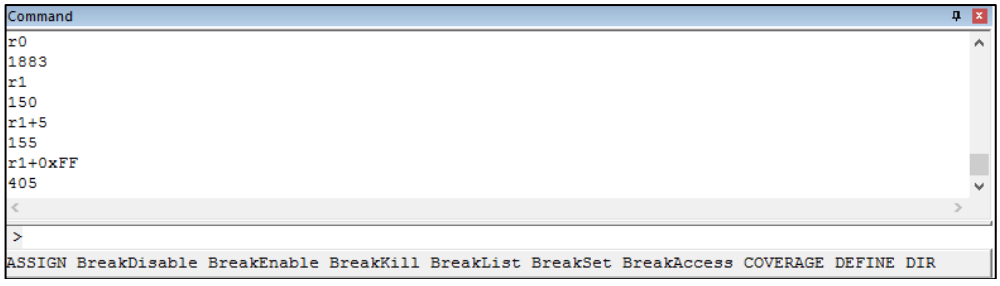


Рис. 11.1 Окно программы µVision

В нижней части командного окна выводятся списки-подсказки команд управления отладкой. Так, имеются команды: запретить/разрешить точку останова, удалить все точки останова, вывести на дисплей листинг имеющихся точек останова и т.д. После того, как Вы введете первую букву предполагаемой команды, строка подсказок изменится: в ней автоматически будет показан список только тех команд, которые начинаются с этой

буквы. Используйте меню **Help** («**Помощь**») для того, чтобы познакомиться более подробно с назначением и особенностями применения команд управления отладкой.

11.3.1.2 Как модифицировать значение регистра или переменной?

Для этого достаточно ввести в командной строке имя регистра/переменной, знак равенства «=», а за ним требуемое значение. Примеры:

r1=78 <Enter>; r1=0xAB <Enter>; r1=55+0x45+r0 <Enter>

Ввод заканчивается нажатием клавиши «Enter». При вводе допустимо использование арифметических выражений, то есть функция встроенного калькулятора работает, как показано в третьем примере.

11.3.1.3 Как изменить систему счисления для выдачи информации на дисплей?

Среда µVision имеет *встроенную переменную* с символическим именем **Radix** («**Основание системы счисления**»), которая отвечает за *систему счисления по умолчанию*. Именно она используется для вывода данных на дисплей командного окна, а также в некоторые другие окна среды, например, в **окно наблюдаемых переменных** (**Watch Window**).

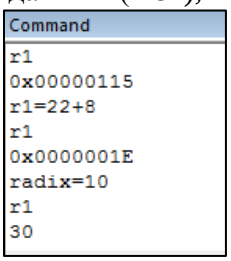
Заметим, что это не касается вывода данных в регистровое окно, в котором данные всегда представляются в Hex-формате, а также в окно дампа памяти, в котором, как мы уже видели, они могут представляться во множестве форматов, задаваемых пользователем в контекстно-зависимом меню этого окна.

Значение системной переменной Radix может быть либо 10 (десятичная система), либо 16 (шестнадцатеричная система). Чтобы установить требуемый формат вывода, нужно выполнить команду модификации значения системной переменной:

Radix=10 <Enter> – установить десятичное представление данных (Dec);

Radix=16 <Enter> – установить шестнадцатеричное представление данных (Hex);

Приведем пример вывода на дисплей и модификации содержимого регистра r1: 1) вывод в формате по умолчанию (Hex); 2) модификация с использованием встроенного калькулятора; 3) проверка результата модификации; 4) смена системы счисления; 5) вывод содержимого r1 в десятичной системе счисления. Все введенные команды отображаются на дисплее и могут быть просмотрены с использованием механизма прокрутки.



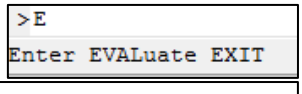
11.3.1.4 Как использовать командное окно для модификации содержимого памяти?

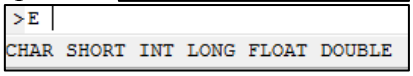
Установка начальных значений в определенных областях памяти – одна из наиболее часто встречающихся задач при отладке программ. Это можно сделать двумя путями:

1) Открыть окно памяти и выполнить инициализацию содержимого памяти с использованием контекстно-зависимого меню окна. Это удобно, так как одновременно позволяет установить нужный формат представления данных (см. 10.12).
2) Выполнить в командном окне команду «**Enter**» (**Ввод** данных).

Для второго варианта наберите сразу после «промта» в строке ввода первую букву команды Enter «**E**». Подсказка в нижней строке будет содержать список возможных команд, начинающихся с этой буквы.

В верхнем регистре набраны буквы для ввода соответствующей команды. Из подсказки следует, что первого символа «E» достаточно – нажимаем клавишу пробела для продолжения диалога, строка подсказок изменяется:





Теперь нам предлагается указать тип данных, которые будут вводиться в память: CHAR (символы) – байтовый ввод; SHORT – короткие 16-разрядные числа со знаком или

без знака (полуслова); INT – целые числа со знаком или без знака (слова); LONG – длинные целые числа со знаком или без знака (двойные слова); FLOAT – числа в формате с плавающей точкой однократной точности; DOUBLE – то же, но двойной точности.

Достаточно ввести первую букву типа вводимых данных, например, «S» и нажать клавишу пробела для продолжения диалога. Следующая подсказка говорит о том, что нужно ввести начальный адрес области памяти, знак равенства и после него список выражений, разделенных запятыми, значения которых будут записаны в нужную область памяти.

```
>E Short
<address> = expr [,...]
```

Начальный адрес вводится исключительно в шестнадцатеричной системе счисления, а далее, после знака равенства, вводится последовательность данных (уже в удобной для Вас системе счисления). Ввод данных завершается нажатием клавиши «Enter», о чем свидетельствует соответствующая подсказка.

```
>E Short 0x20000000=57,58,59,60,61,62,63,64
, expr | <cr>
```

Поэкспериментируйте с просмотром содержимого регистров ЦПУ, их модификацией из командной строки, а также с командами ввода данных в память. Не забывайте устанавливать в окне дампа памяти нужный формат вывода информации. Только при этом Вы получите соответствие выполненных Вами действий и результата, отображенного в окне памяти.

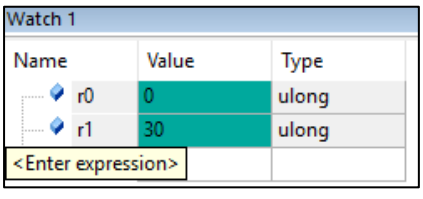
11.3.2 Окна наблюдаемых переменных. Использование в процессе отладки.

Вы наверняка заметили, что в регистровое окно (Registers Window) информация *всегда выдается в шестнадцатеричной системе счисления*. Для просмотра содержимого регистров в десятичной системе, конечно, можно пользоваться командным окном, как описано выше. Однако, это не очень удобно, так как приходится выполнять несколько команд просмотра содержимого регистров последовательно.

Есть и другой путь. Среда µVision позволяет открывать окна **наблюдаемых переменных проекта** (**Watch Window**), в которых может отображаться основная информация, интересующая программиста в процессе отладки. Это может быть информация не только о текущем состоянии переменных в памяти системы, но и о текущем состоянии регистров процессора. Следовательно, *регистры ЦПУ могут быть наблюдаемыми переменными.*

Одним из преимуществ метода отладки программ с использованием окон наблюдаемых переменныхWatch является возможность представления в них данных в десятичной системе счисления, если она выбрана как система по умолчанию. Это очень удобно, так как в окне регистров данные будут в Hex-формате, а в окнах Watch – в Dec-формате, то есть информация будет представлена *в двух системах счисления одновременно.*

Открыть такое окно можно из меню **View** (**Просмотр**), выбрав команду **Watch Windows** (**Окна наблюдаемых переменных**) или щелкнув мышью на соответствующей пиктограмме  на панели инструментов. Всего можно открыть два окна: Watch1 и Watch2. В появившемся окне нужно последовательно указать имена наблюдаемых переменных или регистров с помощью ввода соответствующих выражений. Из примера выше видно, что регистры r0 и r1 стали наблюдаемыми переменными. В столбце Value будет отображено текущее значение наблюдаемых переменных (в установленной системе счисления), а в столбце Type – тип переменных (в



данном случае ulong – длинное целое без знака). Значения в окне наблюдаемых переменных будут автоматически обновляться в процессе отладки.

11.4 Арифметические операции с непосредственными операндами

Операции с непосредственными операндами часто используются для:

1) смещения содержимого регистров-указателей на 1, 2, 4, 8 байт для доступа к очередному байту, полуслову, слову или двойному слову в памяти;
2) смещения содержимого регистра-указателя на величину индекса при доступе к отдельным элементам массивов или структур данных;
3) инкрементирования/декрементирования счетчиков числа выполненных циклов.

Во всех перечисленных выше случаях величина смещения невелика – обычно не выходит за пределы байта. При этом можно применять команды сложения/вычитания содержимого регистра-приемника со вторым универсальным операндом Op2, заданным в виде непосредственной константы. Диапазон возможных непосредственных констант может быть расширен с 8 до 12 разрядов (до 0x0FFF) с использованием команд сложения/вычитания с 12-разрядными непосредственными операндами #imm12:

```
ADD  {Rd}, Rn, #imm12
ADDW {Rd}, Rn, #imm12
SUB  {Rd}, Rn, #imm12
SUBW {Rd}, Rn, #imm12
```

Непосредственный 12-разрядный операнд – от 0 до 4095 (до 0x0FFF)

Первый операнд-источник

Регистр – приемник (опция)

W – Опция, указывающая на использование 12-разрядного непосредственного операнда

ADD – (ADD) - Сложить с 12-разрядным операндом: Rd ← Rn + #imm12

SUB – (SUB) - Вычесть 12-разрядный операнд: Rd ← Rn - #imm12

```
 7  MyProg
 8  ; Демонстрация возможностей увеличения/уменьшения
 9  ; текущего содержимого регистра на непосредственную константу
10    ; Инициализация регистра-указателя r0
11          MOV r0, #20
12    ; Инкрементирование указателя  на величину смещения
13          ADD r0,#1
14          ADD r0,#2
15          ADD r0,#4
16    ; Декрементирование указателя на величиу смещения
17          SUB r0,#4
18          SUB r0,#2
19          SUB r0,#1
20  ; Использование в качестве смещения произвольной
21  ; байтовой константы
22          ADD r0, #0xFA
23          SUB r0, #0xF0
24  ; Использование в качестве смещения содержимого
25  ; другого регистра ЦПУ
26          MOV r1, #7
27
28          SUB r0, r1
29          ADD r0, r1, LSL #8  ; Второй универсальный операнд
30  ; Использование 16-разрядного непосредственного операнда
31          ADD r0,#0x0FAA
32          SUB r0,#0x0F88
33  ; Явное задание суффикса "W" в команде для указания
34  ; "длинного" 12-разрядного непосредственного операнда
35          ADDW r0,#0x0FAA
36          SUBW r0,#0x0F88
37  ; Попытка смещения на величину 32-разрядного слова
38  ;       ADD r0,#0xFAFA
```

Для определенности Вы можете использовать суффикс «W» в мнемонике команды. Однако, если суффикс «W» отсутствует, транслятор с Ассемблера все равно сгенерирует правильную команду, если введенная Вами константа на самом деле является 12-разрядной.

Эти команды имеют особенности: 1) не выставляют флагов результатов операций; 2) не допускают использование суффикса «S» принудительной установки флагов; 3) не могут быть выполнены условно. Поэтому они применяются, главным образом, для смещения указателей на величину индекса при

доступе к массивам и различным структурам данных.

Пример использования команд сложения и вычитания с непосредственными данными представлен в проекте CPU_8 (MyProg_8.s). После инициализации регистра r0 выполняется несколько команд увеличения и уменьшения его содержимого на константы, не превышающие байта, в том числе на константы, предварительно обработанные в сдвиговом регистре ЦПУ. Дальше демонстрируются команды сложения/вычитания с 12-разрядными непосредственными операндами, расширяющие диапазон возможных смещений. Последняя закомментированная команда показывает, что константы большей разрядности (больше 12 бит), в том числе 32-разрядные константы, в качестве непосредственных операндов в арифметических операциях недопустимы – транслятор выдаст предупреждение об ошибке.

```
39  ; Использование предварительно определенных констант
40  Const1  EQU 23
41  Const2  EQU 0x7F
42    ; Загрузить первую константу в регистр r1
43          MOV r1, #Const1
44    ; Сложить со второй
45  SUM_32  ADD r1, #Const2
46  ; Зациклить программу
47  Stop    B .
```

В конце программы показано, как в арифметических операциях можно использовать константы, предварительно определенные с помощью директивы EQU.

1) Откройте окно наблюдаемых переменных – регистров ЦПУ r0 и r1. Установите в качестве системы счисления по умолчанию десятичную систему.

2) Выполните программу в пошаговом режиме и наблюдайте одновременно за содержимым регистров r0, r1 в регистровом окне (Hex-формат) и в окне наблюдаемых переменных (Dec-формат). Убедитесь, что итоговый результат полностью соответствует алгоритму.

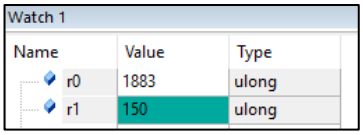
3) Уберите символ комментария в строке 38 программы. Выполните трансляцию программы. О чем говорит выданное транслятором сообщение об ошибке?

4) Просмотрите файл листинга и убедитесь в том, что команды ADD Rd, #imm12 и ADDW Rd, #imm12 генерируют один и тот же код операции.

5) Почему в системе команд процессора недопустимы 32-разрядные непосредственные операнды. Как же быть, если они реально требуются?

1) Используйте команду управления отладкой: Radix=10

2) Последнее содержимое окна Watch1 подтверждает правильность выполненных операций.



3) Сообщение об ошибке информирует программиста о выходе непосредственного операнда 0xFAFA за допустимый диапазон 12-разрядных

```
error: A1492E: Immediate 0x0000FAFA out of range for this operation.  Permitted values are 0x00000000 to 0x00000FFF
```

значений для данной команды. В нем транслятор напоминает о том, что возможные значения не должны превышать 0x0FFF:

4) Это действительно так. Суффикс «W» можно опускать. Транслятор, будучи интеллектуальным, сам сгенерирует нужный код, если обнаружит 12-разрядную константу.

5) В этом случае в формате команды не останется ни одного бита для представления кода операции. Выход есть – выполните инициализацию любого свободного регистра ЦПУ 32-разрядной константой с помощью псевдокоманды LDR Rd, =Const. Используйте этот регистр в нужной арифметической операции.

В арифметических операциях без каких-либо ограничений можно использовать любые байтовые непосредственные константы. Использование 12-разрядных констант возможно, но без формирования флагов результатов операции и условного выполнения команд. При разрядности константы, превышающей 12 бит, потребуется ее предварительная загрузка в один из регистров ЦПУ с использованием псевдокоманды загрузки, создающей в кодовой памяти литеральный пул с численным значением этой константы.

11.5 Арифметические операции с массивами данных в памяти

11.5.1 Обработка массивов 32-разрядных слов

В большинстве вычислительных задач с использованием целочисленной арифметики разрядности АЛУ процессоров ARM (32) более чем достаточно, и контроль возможных переполнений при значительном числе операций может не потребоваться. Рассмотрим в качестве примера процедуру расчета суммы заданного числа 32-разрядных слов, расположенных в ОЗУ (проект CPU_9, программа MyProg_9.s). Она предполагает сложение всех слов массива, начальный адрес расположения которых передается в подпрограмму содержимым регистра r0, а длина массива – содержимым регистра r1. Подпрограмма выполняет расчет суммы и сохраняет ее в следующей после массива свободной ячейке памяти (записывает в «хвост» массива). При этом используются уже известные нам операции загрузки регистров из памяти с использованием базовой адресации с пост-смещением указателя.

```
18  ;************************************************
19  ; Подпрограмма расчета контрольной суммы массива
20  ; 32-разрядных слов в ОЗУ
21  ; Входы: r0 - начальный адрес массива слов
22  ;         r1 - число слов в массиве
23  ; Выход: Сумма всех слов массива, сохраненная
24  ;         в "хвосте" массива
25  ; Используемые регистры:
26  ;         r2 - регистр текущей суммы
27  ;         r3 - регистр временного хранения
28  ;************************************************
29  SUM_W
30  ; Обнулить регистр текущей суммы массива
31          MOV r2,#0
32  Loop
33  ; Считать очередное слово из массива слов в ОЗУ
34  ; с пост-смещением указателя на 4-е байта
35          LDR r3, [r0],#4
36  ; Добавить очередной байт к текущей сумме
37          ADD r2,r3
38  ; Декрементировать счетчик числа циклов. Установить флаги
39          SUBS r1,#1
40  ; Если не все слова массива обработаны - повторить цикл
41          BNE Loop
42  ; Все слова обработаны. Сохранить сумму по следующему
43  ; свободному адресу (в "хвосте" массива)
44          STR r2,[r0]
45  ; Возврат в основную программу
46          BX lr
```

```
48  ; Выровнять память по границе слова
49          ALIGN
50  ; Объявить секцию данных в оперативной памяти
51          AREA MyData, Data, ReadWrite
52          EXPORT RAM_0
53  ; Зарезервировать в ней 16 32-разрядных слов
54  RAM_0
55          SPACE 16*4
56  ; Конец ассемблерного текста
57          END
```

В конце основной программы объявляется секция не инициализируемых данных, значения которых устанавливаются пользователем в окне памяти в процессе отладки программы.

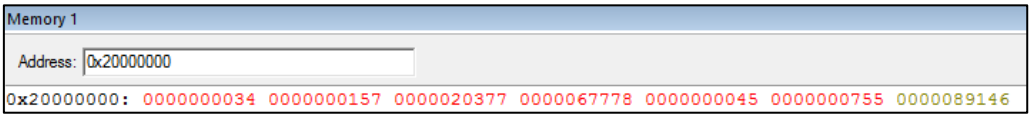
Начальный адрес этой области памяти RAM_0 передается в подпрограмму перед ее вызовом. Вы можете выполнить отладку программы, предварительно проинициализировав память любыми начальными значениями.

```
 7  MyProg
 8  ; Определить переменную "Число слов в массиве"
 9  N_W     EQU 6
10  ; Передать параметры и вызвать подпрограмму
11  ; расчета контрольной суммы слов массива в ОЗУ
12          LDR r0,=RAM_0
13          MOV r1, #N_W
14          BL SUM_W
15  ; Зациклить основную программу
16  Stop    B Stop
```

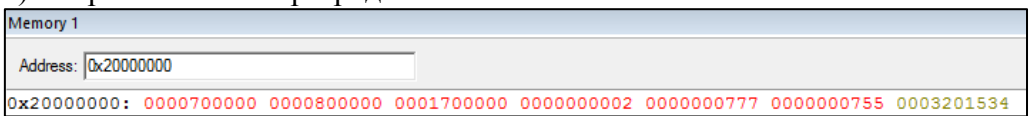
Ниже представлены несколько наборов исходных данных в виде

шести слов без знака и слов со знаком в дополнительном коде с результатами суммирования, сохраненными в конце массива. Естественно, сначала Вы должны установить нужный формат выдачи данных в окно памяти («десятичное представление – слова без знака» или «десятичное представление – слова со знаком»), проинициализировать память и только затем запустить программу на выполнение (можно в режиме RUN с последующим «ручным» остановом).

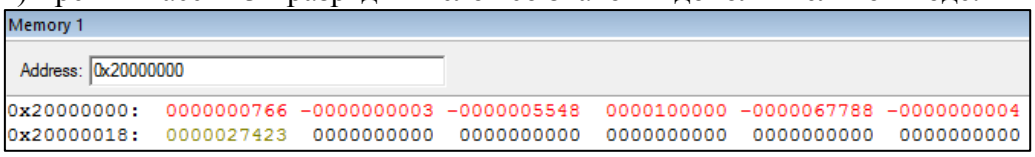
а) Первый массив 32-разрядных слов без знака:



б) Второй массив 32-разрядных слов без знака:



в) Третий массив 32-разрядных слов со знаком в дополнительном коде:



1) Почему в команде сохранения суммы (строка 44) не указана величина пост-смещения?
2) Может ли в первых двух наборах исходных данных при одной из операций сложения возникнуть перенос C (переполнение при работе с числами без знака)? Проверьте!
3) Возникнет ли флаг переноса C при обработке первых слов массива в наборе переменных со знаком? Проверьте!
4) Возникнет ли при суммировании этого массива флаг знакового переполнения V?

1) Это не требуется, так как сохраняется только одно слово.
2) Нет – числа слишком маленькие.
3) Да. Но мы не должны *обращать на него внимания*, так как работаем с числами со знаком. Для них контролируется флаг знакового переполнения V.
4) Нет, так как ни при одной операции сложения мы не выходим за формат допустимых 32-разрядных чисел со знаком.

Сложение и вычитание 32-разрядных слов без знака и со знаком в дополнительном коде выполняется одними и теми же командами процессора. В первом случае переполнение контролируется флагом переноса C, а во втором – флагом знакового переполнения V.

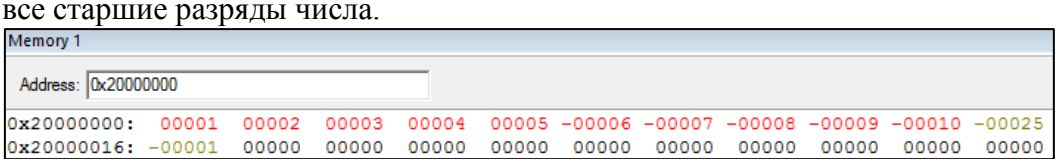
11.5.2 Арифметические операции с массивами байт и полуслов

Мы уже отмечали, что интерфейс микропроцессорной системы с внешними устройствами (дискретным и аналоговыми датчиками) может иметь существенно меньшую разрядность, чем разрядность процессора. При этом внешние данные могут храниться в памяти в виде последовательностей байт или полуслов (со знаком или без). Процессоры ARM обеспечивают обработку таких данных *с авторасширением формата* до размера 32-разрядного слова.

Приведем пример. Данные расположены в ОЗУ в виде последовательности полуслов со знаком. Нужно найти контрольную сумму всех полуслов в этом массиве. В рассмотренной выше подпрограмме необходимо сделать минимальные изменения: вместо команды загрузки полного слова из памяти использовать команду загрузки полуслова со знаком (LDRSH) с авто-расширением до 32-разрядного формата (см. MyProg_9_1.s).

```
33  Loop
34  ; Считать очередное полуслово из массива в ОЗУ
35  ; расширить с учетом знака до 32-разрядного формата
36  ; Выполнить пост-смещение указателя на 2-а байта
37          LDRSH r3, [r0],#2
```

Введем в память произвольную последовательность полуслов со знаком и убедимся в том, что результат расчета контрольной суммы правильный. Обратите внимание, как сумма массива -25 представлена в виде 32- разрядного слова: в младшем полуслове «-25», а в старшем «-1», то есть только один знаковый разряд, расширенный на все старшие разряды числа.

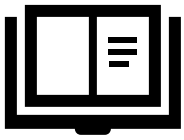


1) Выполните отладку программы MyProg_9_1.s на нескольких наборах исходных данных. Какое максимальное число полуслов может быть обработано без знакового переполнения?
2) Какие изменения необходимо сделать в этой подпрограмме для поиска суммы полуслов без знака?
3) Как найти сумму массива байт со знаком?
4) Сумму массива байт без знака?

1) Без переполнения можно сложить до 65536 полуслов со знаком. При этом сумма не выйдет за пределы 32-разрядной сетки процессора.
2) Достаточно заменить всего одну команду: вместо LDRSH использовать команду LDRH (MyProg_9_2.s).
3) Используйте команду загрузки в регистр байта со знаком с авто-расширением знака на все старшие разряды слова LDRSB (MyProg_9_3.s).
4) Используйте команду загрузки в регистр байта без знака LDRB с расширением влево нулями.

Можно использовать попутные операции авторасширения формата байта или полуслова до формата полного слова при загрузке данных в регистры ЦПУ. При арифметической обработке таких данных вероятность возникновения переполнения (знакового и беззнакового) минимальна.

11.6 Многословная арифметика



Арифметика чисел большой разрядности используется, в частности, при сложении/вычитании двойных слов – 64-битовых. Пусть имеются два таких слова D1 и D2. Требуется найти их сумму и разность.

Каждое двойное слово состоит из младшего и старшего обычного 32-разрядного слова. В таблице слева показаны обозначения этих компонент двойных слов. Операции сложения/вычитания начинаются с младших слов. При этом используются обычные команды сложения/вычитания ADDS и SUBS обязательно с суффиксом «S» установки флагов результатов операции для последующего учета возможного переноса или «заема» в старшие разряды.

	D1_High (r2)	**D1_Low (r1)**
	D2_High (r4)	**D2_Low (r3)**
Команды сложения двойных слов		
ADC #0	**ADCS**	**ADDS**
Команды вычитания двойных слов		
SBC #0	**SBCS**	**SUBS**
Результат сложения/вычитания		
SUM_H (r7) **SUB_H (r7)**	**SUM_W (r6)** **SUB_W (r6)**	**SUM_L (r5)** **SUB_L (r5)**

```
 7  MyProg
 8  ; Вычисление суммы и разности двух двойных слов
 9  ; Инициализация первого двойного слова
10          LDR r2,=0xFF225555
11          LDR r1,=0x4444AAAA
12  ; Инициализация второго двойного слова
13          LDR r4,=0x22113333
14          LDR r3,=0xBBBB2222
15  ; Расчет суммы двойных слов
16  SUM_D
17  ; Сумма младших слов
18          ADDS r5,r1,r3
19  ; Сумма старших слов
20          ADCS r6,r2,r4
21  ; Учет возможного переполнения
22          MOV r7,#0
23          ADC r7, #0
24
25  ; Расчет разности двойных слов
26  SUB_D
27  ; Разность младших слов
28          SUBS r5,r1,r3
29  ; Разность старших слов
30          SBCS r6,r2,r4
31  ; Учет возможного "заема"
32          MOV r7,#0
33          SBC r7, #0
34
35  ; Зациклить основную программу
36  Stop    B Stop
```

Сложение/вычитание старших слов выполняется с учетом ранее возникшего переноса/«заема» и также с учетом суффикса «S» установки флагов сложения/вычитания ADDS/SUBS. Учет возможного переноса за пределы 64-битной разрядной сетки можно выполнить командой сложения нуля с флагом переноса и вычитания из нуля флага «заема».

Естественно, указанные выше операции можно выполнить только с операндами в регистрах процессора. Пусть для размещения данных в ЦПУ выбраны регистры, указанные в таблице в скобках. Напишем простую программу, которая будет использовать команды непосредственной загрузки данных в регистры процессора с вычислением суммы и разности двойных слов MyProg_10.s, (проект CPU_10).

Отладку программы целесообразно выполнить по точкам останова в конце каждого фрагмента: вычисления сначала суммы, а затем разности двойных слов. Содержимое окна регистров, показанное ниже, подтверждает правильность алгоритма. Обратите внимание на то, что при сложении двойных слов учтен возникший флаг переноса и самое старшее слово суммы равно 0x1. При вычитании «заема» нет, поэтому старшее слово разности – нулевое.

R1	0x4444AAAA
R2	0xFF225555
R3	0xBBBB2222
R4	0x22113333
R5	0x88898888
R6	0xDD112221
R7	0x00000000

R1	0x4444AAAA
R2	0xFF225555
R3	0xBBBB2222
R4	0x22113333
R5	0xFFFFCCCC
R6	0x21338888
R7	0x00000001

Команды сложения/вычитания с учетом переноса/«заема» позволяют выполнять арифметические операции с числами неограниченной разрядности, как со знаком, так и без знака.

11.7 Макроопределение. Макровызов. Использование макробиблиотек

11.7.1 Общие положения

ARM ASM

Этот параграф при первом изучении можно просмотреть бегло. Он рассказывает о том, что такое *макросредства* языка Ассемблер и как ими пользоваться при программировании на Ассемблере. Дело в том, что при огромных возможностях системы команд процессора, которые мы еще только начинаем изучать, число регистров процессора ограничено. При разработке алгоритмов решения вычислительных задач очень трудно держать в голове распределение регистров процессора по функциональному признаку. Было бы неплохо разработать алгоритм решения часто встречаемой задачи так, чтобы в нем использовались лишь символические имена переменных, соответствующие их назначению (макроопределение – макрос). А дальше, при вызове макроса, указать транслятору, какие фактические имена регистров процессора или переменных следует использовать вместо символических имен.

Для решения подобных задач применяются специальные директивы языка Ассемблер, которые позволяют описать алгоритм решения задачи с использованием символических переменных, которые будем называть *формальными параметрами*. Описание алгоритма с использованием формальных параметров называется *макроопределением* – это фрагмент текста на языке Ассемблер, заключенный в операторные скобки из двух директив Ассемблера MACRO (начало макроопределения) и MEND (конец макроопределения). Синтаксис макроопределения:

```
          MACRO
{$label}  MacroName{$cond} {$parameter{,$parameter}...}
          ; Тело макроопределения (любая последовательность кода)
          MEND
```

Между операторными скобками в первой строке указывается название макроопределения (*макроса*) MacroName – символическое имя, по которому определенная в макросе последовательность команд может быть вызвана в любом месте программы. Это должно быть любое имя, кроме зарезервированных имен команд процессора или директив ассемблера.

После имени макроса указывается список формальных параметров, которые соответствуют физическому смыслу располагаемых в них данных. Их имена предваряются знаком доллара «$» и отделяются друг от друга запятой. Число формальных параметров не ограничивается. Именно формальные параметры могут использоваться в любых командах процессора, расположенных в теле макроса, вместо имен регистров и имен реальных переменных проекта.

Опционально в качестве параметров могут использоваться и метки начала макроса `$label`, которые при макровызове сопровождаются генерацией оригинальных имен меток с дополнительным цифровым кодом для исключения эффекта многократного определения имен при многократных вызовах макросов.

Опционально в качестве формального параметра можно указать и код условного выполнения создаваемой «новой» команды процессора $cond. При вызове он должен быть заменен фактическим параметром – реальным кодом условного выполнения.

Макровызов представляет собой имя макроса со списком *фактических параметров* (имен реальных регистров процессора, реальных переменных, констант). При *макрорасширении* вместо формальных параметров транслятор генерирует соответствующие им фактические параметры, выполняя своеобразную *автоматическую*

замену переменных. Это позволяет автоматизировать процесс генерации кода и исключает возможные ошибки. Достаточно отладить один раз макрокоманду и затем использовать ее многократно.

11.7.2 Примеры макроопределений суммы и разности двойных слов

Используем введенные выше символические имена компонентов двойных слов – источников данных (D1_H, D1_L), (D2_H, D2_L) и символические имена их суммы или разности (R_H, R_W, R_L) для разработки макроопределений. Расположим макроопределения в самом начале программы пользователя MyProg_10_1.s, так как они представляют собой лишь текстовые описания соответствующих алгоритмов. Как видите, макрос суммирования длинных слов получил название D_ADD, а их вычитания – D_SUB. Всего в каждом макроопределении по 7 формальных параметров, имя каждого из которых предваряется знаком доллара «$».

```
1  ; Макроопределение операции сложния двойных слов
2          MACRO
3          D_ADD $D1_H, $D1_L, $D2_H, $D2_L, $R_H, $R_W, $R_L
4  ; Сумма младших слов
5          ADDS $R_L,$D1_L,$D2_L
6  ; Сумма старших слов
7          ADCS $R_W,$D1_H,$D2_H
8  ; Учет возможного переполнения
9          MOV $R_H,#0
10         ADC $R_H, #0
11         MEND
12
13 ; Макроопределение операции вычитания двойных слов
14         MACRO
15         D_SUB $D1_H, $D1_L, $D2_H, $D2_L, $R_H, $R_W, $R_L
16 ; Разность младших слов
17         SUBS $R_L,$D1_L,$D2_L
18 ; Разность старших слов
19         SBCS $R_W,$D1_H,$D2_H
20 ; Учет возможного "заема"
21         MOV $R_H,#0
22         SBC $R_H, #0
23         MEND
```

В теле макроопределения вместо имен регистров процессора использованы их символические имена, смысл которых отражает расположенные в них данные. Так, $D1_H, $D1_L – формальные параметры, содержащие старшее и младшее 32-разрядные слова первого 64-битового двойного слова. При использовании таких имен смысл выполняемых операций в теле макроса становится более понятным. Так, первая команда ADDS выполняет сложение младших слов исходных двойных слов – $D1_L, $D2_L, с сохранением результата в младшем слове результата $R_L. Макроопределение начинается директивой MACRO и заканчивается директивой MEND.

11.7.3 Как вызываются макрокоманды?

```
39 ; Расчет суммы двойных слов
40 SUM_D
41 ; Макровызов операции сложения двойных слов
42         D_ADD r2,r1,r4,r3,r7,r6,r5
43 ; Расчет разности двойных слов
44 SUB_D
45 ; Макровызов операции вычитания двойных слов
46         D_SUB r2,r1,r4,r3,r7,r6,r5
```

После того, как макроопределения сделаны, то есть нужные алгоритмы описаны на языке формальных параметров, они могут быть использованы в любом месте программы пользователя с помощью *макровызовов* – символических имен новых макрокоманд со списком фактических параметров, которые транслятор должен использовать вместо формальных параметров при генерации кода макроса. Такая генерация кода называется *макрорасширением*.

```
39 00000008          ; Расчет суммы двойных слов
40 00000008          SUM_D
41 00000008          ; Макровызов операции сложения двойных слов
42 00000008                  D_ADD           r2,r1,r4,r3,r7,r6,r5
 4 00000008          ; Сумма младших слов
 5 00000008 18CD             ADDS            r5,r1,r3
 6 0000000A          ; Сумма старших слов
 7 0000000A EB52 0604        ADCS            r6,r2,r4
 8 0000000E          ; Учет возможного переполнения
 9 0000000E F04F 0700        MOV             r7,#0
10 00000012 F147 0700        ADC             r7, #0
```

Выполните трансляцию программы MyProg_10_1.s. Откройте файл листинга. Вы увидите, что макроопределение отображается в нем просто как текст, не требующий преобразования в машинный код, а макровызов – автоматически преобразуется в нужный набор реальных команд процессора, в котором каждый формальный параметр заменен фактическим (в данном

случае – именем используемого регистра процессора). Транслятор не только выполняет эту замену переменных, но и создает машинный код соответствующих инструкций. Одна строка исходной программы с макровызовом (строка 42) автоматически заменяется нужным числом строк с реальными командами процессора.

```
32  ; Вычисление суммы и разности двух двойных слов
33  ; Инициализация первого двойного слова
34          LDR r2,=0xFF225555
35          LDR r1,=0x4444AAAA
36  ; Инициализация второго двойного слова
37          LDR r4,=0x22113333
38          LDR r3,=0xBBBB2222
```

Перед вызовом макросов должны быть установлены исходные значения двойных слов. В нашем примере инициализация выполняется путем последовательной загрузки регистров (r2, r1) и (r4, r3) непосредственными словами-константами, после чего выполняется вызов макросов расчета суммы и разности. Выполните отладку этой программы и убедитесь в правильности полученных результатов.

11.7.4 Как использовать макробиблиотеки?

Все нужные программисту макроопределения можно записать в отдельный файл, который будет называться *библиотекой макроопределений*, например, MyMacro.s. Если функции, находящиеся в этом файле отлажены, то можно просто подключить к текущему файлу приложения библиотечный файл с использованием директивы ассемблера GET («Получить файл») или INCLUDE («Подключить файл»):

```
GET filename или INCLUDE filename
```

При выполнении этой псевдокоманды указанный файл включается в программу «как есть», без какой-либо обработки. Если в тексте программы-приложения встречается макровызов, то он автоматически заменяется соответствующим расширением из библиотеки. Это позволяет не только существенно экономить место в основной программе, но и делать ее более наглядной и удобной для последующей модификации (сопровождения). В имени подключаемого файла можно указывать полный путь доступа, если файл не находится в текущем каталоге. По умолчанию текущим является каталог, в котором находится файл программы приложения.

```
1   ; Подключить библиотеку макроопределений
2           GET MyMacro.s
3   ; Объявить кодовую секцию MyCode
4           AREA MyCode, CODE, ReadOnly
5   ; Объявить точку входа в программу приложения
6           ENTRY
7   ; Объвить точку входа глобальной переменной
8           EXPORT MyProg
9   MyProg
10  ; Вычисление суммы и разности двух двойных слов
11  ; Инициализация первого двойного слова
12          LDR r2,=0xFF225555
13          LDR r1,=0x4444AAAA
14  ; Инициализация второго двойного слова
15          LDR r4,=0x22113333
16          LDR r3,=0xBBBB2222
17  ; Расчет суммы двойных слов
18  SUM_D
19  ; Макровызов операции сложения двойных слов
20          D_ADD r2,r1,r4,r3,r7,r6,r5
21  ; Расчет разности двойных слов
22  SUB_D
23  ; Макровызов операции вычитания двойных слов
24          D_SUB r2,r1,r4,r3,r7,r6,r5
25  ; Зациклить основную программу
26  Stop    B Stop
27  ; Конец ассемблерного текста
28          END
```

В подключаемых файлах могут находиться не только макроопределения, но и определения переменных проекта, сделанные с помощью директивы EQU. Имейте в виду, что эти директивы не применяются для подключения объектных библиотек. Файл с библиотекой макроопределений в конце должен иметь директиву конца Ассемблерного текста END.

Сохраним два наших макроопределения в файле MyMacro.s и подключим этот файл к программе приложения MyProg_10_2.s, в котором в начале файла находится директива GET подключения макробиблиотеки. О том, что библиотека действительно подключена, можно удостовериться в файле листинга. В нем же можно посмотреть, как макрокоманды расширяются в последовательности реальных операций процессора. Выполните отладку этой программы, убедитесь в правильности рассчитанной суммы и разности двойных слов.

Макросредства – эффективный метод представления нужных пользователю алгоритмов с использованием символических имен регистров и переменных, что делает программу более понятной и «читабельной», сокращая ее объем. С их помощью можно даже создавать недостающие для конкретных приложений команды процессора.

1) Что такое формальный параметр?
2) Что такое фактический параметр?
3) Чем макрос отличается от подпрограммы? Преимущества и недостатки макросов по сравнению с подпрограммами.

1) Символическое имя, которое используется в макро-описании вместо имени регистра ЦПУ или адреса переменной в памяти, отражающее смысл расположенных в нем данных.
2) Реальное имя регистра ЦПУ или имя переменной в памяти.
3) При каждом макровызове, в отличие от подпрограммы, в текст программы вставляется фрагмент ассемблерного текста, выполняющий нужную функцию. Если вызовов много, то объем программного кода увеличивается по сравнению с подпрограммой. Однако, время выполнения макроса меньше, чем время выполнения подпрограммы, так как не используются команды для вызова и возврата из подпрограммы, а также команды сохранения контекста в стеке.

1) Разработать макроопределение функции D_LOAD, копирующей из массива двойных слов в ОЗУ два двойных слова с записью их значений в регистры процессора – формальные параметры (D1_H, D1_L), (D2_H, D2_L). Двойные слова расположены в памяти с обычным порядком расположения слов: младшее слово по младшему адресу, старшее – по старшему. Начальный адрес массива находится в регистре-указателе процессора, определенном формальным параметром A_D. Установить указатель адреса A_D на следующий свободный адрес памяти.
2) Разработать макроопределение функции SAVE_3W, сохраняющей тройное слово (из 3-х обычных слов) (R_H, R_W, R_L) в памяти, начиная с адреса, расположенного в регистре-указателе A_D.
3) Включить обе функции в макробиблиотеку MyMacro_1.s и с ее помощью создать подпрограмму сложения D_ADD двух двойных слов из массива в памяти, начальный адрес которого передается содержимым регистра r0.
4) Написать основную программу с передачей параметров в подпрограмму и ее вызовом. Отладить на тестовом наборе двойных слов в памяти.

1) Пример макроопределения содержится в файле MyMacro_1.s.

В качестве имени регистра-указателя используется формальный параметр $A_D (адрес расположения двойных слов в ОЗУ). При загрузке слов из памяти применяется адресация с пост-смещением указателя на 4 байта.

```
; Макроопределение загрузки двух двойных слов из памяти
; в регистры процессора, начиная с адреса в $A_D
        MACRO
        D_LOAD $D1_H, $D1_L, $D2_H, $D2_L, $A_D
; Загрузить последовательно 4-е слова с пост-смещением
; указателя на 4 байта
        LDR $D1_L, [$A_D], #4
        LDR $D1_H, [$A_D], #4
        LDR $D2_L, [$A_D], #4
        LDR $D2_H, [$A_D], #4
        MEND
```

2) Пример второго макроопределения находится в том же файле. Для обозначения регистра-указателя использован тот же формальный параметр $A_D. Младшие слова сохраняются в режиме пост-смещения указателя на 4 байта, старшее слово – без смещения.

```
; Макроопределение операции сохранения тройного слова
; из регистров ЦПУ в память, начиная с адреса $A_D
        MACRO
        SAVE_3W $R_H, $R_W, $R_L, $A_D
        STR $R_L, [$A_D], #4
        STR $R_W, [$A_D], #4
        STR $R_H, [$A_D]
        MEND
```

3) Фактически подпрограмма состоит из трех вызовов макросов. Первый вызов обеспечивает загрузку двух двойных слов в регистры ЦПУ, второй выполняет расчет суммы с учетом возможного переполнения (из 3-х слов), а третий сохраняет результат расчета в «хвосте» массива исходных данных. В конце подпрограммы следует возврат в основную программу.

```
; ************************************************
; Подпрограмма сложения двух длинных слов массива
; с сохранением суммы в памяти
; Входы: r0 - начальный адрес массива
; Выход: Сумма из трех слов в продолжение массива
; Используемые регистры r1,r2,r3,r4,r5,r6,r7
;************************************************
SUM_DOUBLE
; Загрузить пару двойных слов в регистры ЦПУ из памяти
        D_LOAD r2, r1, r4, r3, r0
; Расчет суммы двойных слов
        D_ADD r2,r1,r4,r3,r7,r6,r5
; Сохранение трех слов результата сложения в памяти
        SAVE_3W r7,r6,r5,r0
; Возврат в основную программу
        BX lr
```

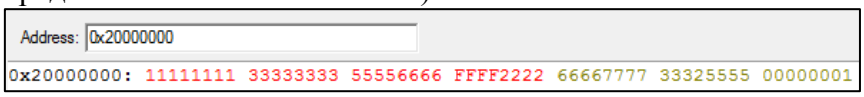
4) В самом начале основной программы (см. MyProg_10_3.s) нужно подключить файл с новой макробиблиотекой, содержащей все макроопределения MyMacro_1.s. Передача параметров в подпрограмму сводится к загрузке начального адреса размещения массива исходных данных RAM_0 в регистр-указатель r0. Дальше следует вызов подпрограммы. В конце основной программы не забудьте открыть секцию данных и зарезервировать в ней нужное число байт под переменные проекта. Выполните трансляцию и сборку файлов проекта. После загрузки программы в симулятор проинициализируйте память начальными значениями двух двойных слов. Помните, что принятый в проекте порядок расположения слов в памяти такой – младшее слово находится по младшему адресу.

```
; Подключить библиотеку макроопределений
        GET MyMacro_1.s
```

```
MyProg
; Передать параметр в подпрограмму
; Начальный адрес массива двойных слов в памяти
        LDR r0,=RAM_0
; Вызвать подпрограмму расчета суммы двойных слов
        BL SUM_DOUBLE
```

```
; Объявить секцию данных в ОЗУ
        AREA MyData, DATA, ReadWrite
; Зарезервировать 10 слов
RAM_0   SPACE 10*4
```

Выполните программу. Один из наборов исходных данных и результаты суммирования двойных слов представлены ниже (в окне памяти используется Hex-представление чисел без знака).



Используйте макросы для формального описания достаточно сложных алгоритмов обработки данных в ЦПУ с помощью формальных параметров. Включайте макрокоманды в подпрограммы. Это позволит объединить преимущества обеих технологий (макрокоманд и подпрограмм). Используйте макросы, в том числе, для передачи параметров в подпрограммы.

1) Имеется ли в процессорах ARM специальный флаг «заема» Borrow?
2) Нужно ли в обычных операциях сложения/вычитания байт/полуслов, загруженных из памяти в регистры ЦПУ с авто-расширением нулями (для чисел без знака) или знаковым разрядом (для чисел со знаком), учитывать флаг переноса/«заема»?
3) Каков будет формат суммы массива из 100 полуслов без знака?

1) Нет. Вместо него применяется инверсное значение флага C.
2) При единичных операциях – нет. Результат всегда будет в пределах разрядной сетки процессора. При обработке массивов – это зависит от длины массива.
3) Максимальное значение суммы 65535*100 = 6553500, что существенно меньше максимального положительного 32-разрядного числа (232-1) = 4 294 967 295. Контроль переполнения не требуется.

11.8 Команды умножения и деления

Процессоры ARM-Cortex M3/M4/M4F, в отличие от простых микроконтроллеров, поддерживают операции умножения и деления 32-разрядных операндов с получением 32-разрядного результата, а также операции «длинного» умножения с получением 64-битного произведения в двух регистрах ЦПУ.

Синтаксис операций первого типа (32) = (32) · (32) следующий:

```
 MUL {Rd}, Rn, Rm
UDIV {Rd}, Rn, Rm
SDIV {Rd}, Rn, Rm
```

Второй операнд-источник

Первый операнд-источник

Регистр – приемник (опция)

MUL - (Multiply) – Умножение 32-разрядных операндов: **Rd ← Rn · Rm**

DIV - (Divide) – Деление 32-разрядных операндов: **Rd ← Rn / Rm**

U – (Unsigned) – Операция с числами без знака

S - (Signed) – Операция с числами со знаком в дополнительном коде

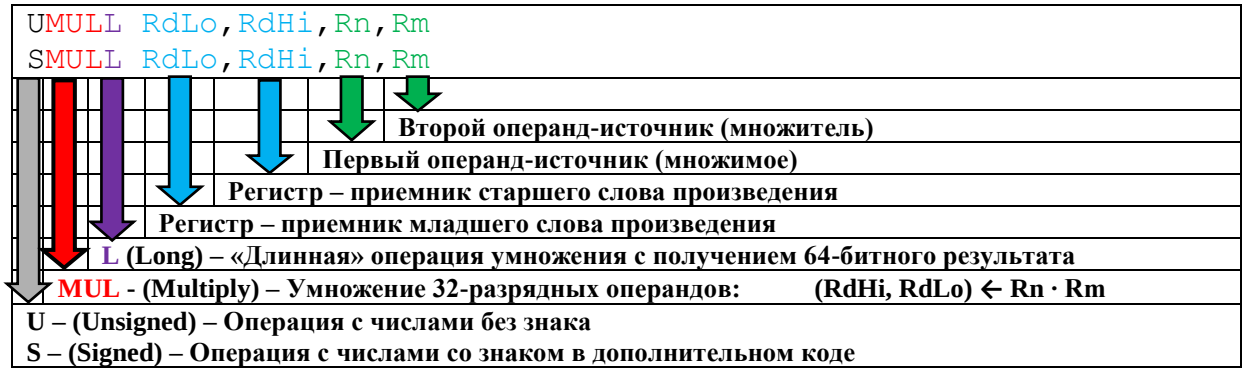
Команда MUL является универсальной. Она выполняет умножение 32-разрядных операндов *без знака или со знаком* в регистрах источниках Rn и Rm. Она рассчитана, главным образом, на умножение операндов, полученных путем предварительной загрузки из памяти 16-разрядных полуслов без знака или со знаком (команды LDRH, LDRSH), а также байт без знака или со знаком (LDRB, LDRSB), с автоматическим расширением влево до формата 32-разрядного слова: (U16→U32, S16→S32) или (U8→U32), (S8→S32).

Как правило, периферия микроконтроллеров является 16-разрядной или даже 8-разрядной. Поэтому исходные переменные в большинстве случаев имеют формат, не превышающий 16-и разрядов. При умножении любых чисел (U16*U16) или (S16*S16) результат никогда не превысит формат U32 или S32, то есть будет абсолютно точным в пределах 32-разрядной сетки процессора. Исходя из этого, команда MUL не контролирует переполнение при умножении. Предполагается, что программист использует входные операнды в форматах, при которых переполнения невозможны.

На самом деле, как будет показано ниже, команда MUL генерирует правильный результат и в том случае, когда исходные операнды являются 32-разрядными, но лишь тогда, когда произведение попадает в допустимый диапазон представления 32-разрядных чисел без знака или со знаком. В противном случае, старшее слово произведения теряется, и результат умножения оказывается недостоверным.

Команды UDIV и SDIV имеют собственные префиксы «U» и «S», которые специфицируют тип выполняемой операции: с числами без знака или со знаком. Целая часть частного сохраняется в регистре назначения Rd, а остаток отбрасывается.

Обратите особое внимание на то, что контроль переполнения при операциях умножения *не предполагается*. Программист должен обеспечить отсутствие возможных переполнений правильным выбором форматов исходных данных. Поэтому, если опасность переполнения имеется, то нужно использовать команды «длинного умножения» с суффиксом «L» для получения 64-битного произведения сразу в двух регистрах процессора (64) = (32) · (32):



Команды «длинного» умножения полностью исключают возможность получения некорректного результата. При этом разрядность АЛУ процессора фактически увеличивается с 32 до 64 разрядов! С учетом рассмотренных выше операций сложения и вычитания длинных слов появляется возможность выполнения любых арифметических операций над 32-разрядными словами с получением 64-битного результата, *при отсутствии опасности переполнения.*

В следующем параграфе будут рассмотрены операции множественной загрузки/сохранения содержимого регистров ЦПУ (из памяти/в память), которые значительно облегчают не только реализацию сложных вычислительных алгоритмов, но и их отладку.

11.9 Команды множественной загрузки/сохранения регистров

11.9.1 Назначение и области применения

Процессоры ARM имеют ряд эффективных команд, которые целесообразно использовать при реализации вычислительных алгоритмов и при их отладке. В первую очередь это касается команд *множественной загрузки* данных из памяти в регистры ЦПУ и *множественного сохранения* содержимого регистров в памяти. Так как вычислительные операции могут выполняться только над данными, расположенными в регистрах процессора, и там же сохраняются результаты расчетов, то наличие команд, позволяющих проинициализировать сразу группу регистров или сохранить результаты расчетов из группы регистров в определенной области памяти, является существенным преимуществом ARM-процессоров.

Эти преимущества особенно наглядно проявляются в процессе отладки – результаты расчетов в окне памяти среды µVision можно представить в любом удобном для программиста формате (знаковых чисел или чисел без знака, в шестнадцатеричной или в десятичной системе счисления). Наглядная визуализация результатов расчетов позволяет быстро протестировать работу сложных алгоритмов, найти возможные ошибки, сокращает время разработки программного обеспечения.

11.9.2 Синтаксис основных команд множественной загрузки/сохранения регистров

Синтаксис двух, наиболее часто используемых команд множественной загрузки LDM и множественного сохранения регистров STM, следующий:

```
LDM  Rn{!}, reglist
STM  Rn{!}, reglist
```

reglist – Листинг регистров ЦПУ, участвующих в операции множественной загрузки или сохранения, заключенный в фигурные скобки {Ri, Rj,…Rm}

! – Опция авто-инкрементирования содержимого базового регистра (+4) после выполнения очередной пересылки данных (increment after)

Rn – Имя базового регистра ЦПУ, содержащего начальный адрес области памяти

M – (Multiple) – Множественная операция пересылки 32-разрядных слов

LD – (Load) - Загрузка нескольких регистров ЦПУ из памяти
ST – (Store) - Сохранение содержимого нескольких регистров в памяти

Особенности команд:

1) Поддерживается косвенная адресация памяти по содержимому базового регистра Rn (одного из регистров ЦПУ). Перед выполнением команды он должен содержать начальный адрес области памяти, к которой последует обращение.
2) Имя базового регистра Rn указывается сразу за мнемокодом команды на месте первого операнда команды. В отличие от описанной ранее мнемоники команд однократной пересылки данных в память, символ косвенной адресации в виде квадратных скобок вокруг имени базового регистра «[ ]» *не используется*.
3) Список (листинг) регистров процессора, которые загружаются данными из памяти или содержимое которых сохраняется в памяти, указывается на месте второго операнда команды. Он обязательно заключается в фигурные скобки {Ri, Rj, …Rm}, которые в

данном случае являются не признаком опции операнда, а обязательным атрибутом списка регистров.

4) После каждого доступа к памяти по чтению или по записи адрес доступа к памяти (точнее, используемое смещение относительно базового адреса) автоматически инкрементируется на 4 байта, указывая на очередную ячейку памяти в направлении *восходящих адресов*: Mem[Rd]; Mem[Rd+4]; Mem[Rd+8], … Данные извлекаются из памяти или записываются в нее последовательно пословно.
5) Если после имени базового регистра Rn указана опция «!» (восклицательный знак), то это признак *пост-автомодификации* содержимого базового регистра после каждого доступа к памяти. Это требуется, в частности, при последовательной загрузке разных групп регистров из двух рядом расположенных областей памяти. При этом можно *без переинициализации базового регистра* выполнить подряд две команды множественной загрузки/сохранения. Без символа «!» пост-инкрементируется только величина смещения относительно текущего содержимого базового регистра, а с символом «!» – и содержимое самого базового регистра.
6) Допускается указывать в листинге имена регистров в любой последовательности, однако они будут сохраняться в последовательности, *начиная с имени самого старшего регистра в направлении самого младшего*. Так, если список регистров имеет вид {r4, r1, r3, r6, r2}, то операция множественного сохранения начинается с регистра с максимальным номером r6 и продолжается в направлении убывающих номеров r4, r3, r2, r1.
7) Порядок множественной загрузки данных из памяти обратный: сначала загружаются данные в регистр с наименьшим номером, а затем – в направлении возрастающих номеров регистров. Этот порядок обеспечивает работоспособность вложенных подпрограмм.
8) В листинге допускается указывать диапазон регистров, участвующих в операции, например, {r1 – r5}. Это эквивалентно списку: {r1, r2, r3, r4, r5}.
9) Команды множественной загрузки/сохранения являются *прерываемыми*. Это означает, что при поступлении запроса прерывания/исключения последовательность операций пересылки данных прерывается и автоматически возобновляется после завершения обслуживания прерывания/исключения. В аппаратной части процессоров ARM имеются специальные средства, которые позволяют сохранить информацию о том, с какого места нужно возобновить множественную пересылку данных.
10) Базовый регистр не может входить в листинг регистров за исключением указателя стека SP. Эта возможность применяется при сохранении/восстановлении контекста программы при обработке прерываний. *Не рекомендуется* включать в список регистров также счетчик команд PC, во избежание «случайных» переходов.

Команды множественной загрузки/сохранения регистров имеют особый, отличающийся от всех других команд синтаксис: 1) нет привычного символа косвенной адресации «[ ]»; 2) опция «!» означает не пре- а пост-модификацию базового регистра; 3) в качестве первого операнда всегда указывается имя базового регистра, а второго – листинг регистров ЦПУ. Будьте внимательны!

11.9.3 Использование операций множественной загрузки/сохранения регистров для отладки арифметических алгоритмов

11.9.3.1 Вычисления с 32-разрядными числами без знака

Рассмотрим пример организации вычислений с 32-разрядными операндами без знака (проект CPU_11, файл MyProg_11.s). Требуется вычислить выражение F=X+(Y·Z)+(V/Q), в котором все исходные операнды предварительно размещены в текущей кодовой секции в виде таблицы констант. Для этого используем директиву ассемблера DCD размещения в памяти констант-слов. Распределим регистры ЦПУ для удобного решения задачи, например так: r0=r1+(r2·r3)+(r4/r5).

```
36  ; Инициализация констант в текущей кодовой секции
37  ; 32-разрядные слова без знака
38  A_ROM
39  X       DCD 30777
40  Y       DCD 200
41  Z       DCD 10
42  W       DCD 7000
43  Q       DCD 20
```

```
 7  MyProg
 8  ; Демонстрация использования команд умножения и деления
 9  ; слов без знака для вычисления выражения
10  ; F=X+(Y*Z)+(W/Q). Распределение регистров ЦПУ:
11  ; r0=r1+(r2*r3)+(r4/r5)
12  ; Загрузка регистров исходными операндами
13    ; Инициализация базового регстра r6 начальным
14    ; адресом таблицы констант в памяти.
15          ADR r6,A_ROM
16    ; Множественная загрузка регистров
17          LDM r6,{r1-r5}
```

Следовательно, в начале алгоритма нужно проинициализировать группу регистров {r1 - r5} исходными значениями, что можно сделать с использованием команды множественной загрузки регистров из кодовой памяти по известному начальному адресу A_ROM.

```
18  ; Основной вычислительный алгоритм
19    ; Вычисление выражения (r4/r5)
20          UDIV r4,r5  ; Частное в r4
21    ; Вычисление выражения (r2*r3)
22          MUL r2,r3   ; Произведение в r2
23    ; Суммирование компонент
24          ADD r0,r1,r2
25          ADD r0,r4
```

Дальше следует реализация собственно вычислительного алгоритма, особенность которого – использование команды умножения MUL (работает и для знаковых и для беззнаковых чисел) и команды деления слов без знака UDIV. Для сложения компонент используем две команды ADD (одну трехоперандную, вторую двухоперандную). Результат расчета сохраним в регистре r0.

Для проверки алгоритма на любых наборах переменных поступим так: зарезервируем в конце программы некоторую область ОЗУ для размещения в ней содержимого всех, используемых при вычислениях, регистров процессора.

Цель – записать исходные данные и результат расчета в память для удобного анализа в окне дампа памяти среды µVision, настроенном на нужный нам формат выдачи информации (в данном случае Decimal, Unsigned, Int – слов без знака в десятичной системе счисления).

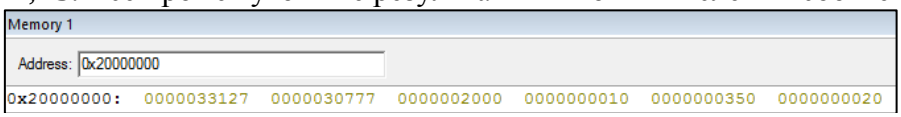
Это позволит задавать любые наборы исходных данных и просматривать как результат расчета, так и промежуточные результаты в удобном для наблюдения формате. Для сохранения результатов в памяти воспользуемся командой множественного сохранения содержимого сразу группы регистров.

```
26  ; Сохранение промежуточных и итогового результата
27  ; в памяти для анализа
28    ; Инициализация базового регистра r6 начальным
29    ; адресом области ОЗУ
30          LDR r6,=A_RAM
31    ; Множественное сохранение содержимого регистров
32          STM r6,{r0-r5}
33  ; Закончить вычисления
34  Stop    B .
```

```
44          ALIGN
45  ; Секция данных в ОЗУ
46          AREA MyData, Data, ReadWrite
47  A_RAM
48          SPACE 100*4
49  ; Конец ассемблерного текста
50          END
```

Выполним трансляцию и сборку проекта. Для указанного в программе набора исходных данных результаты вычислений в окне памяти будут выглядеть так. Напомним,

что в памяти последовательно сохранено содержимое шести регистров ЦПУ: r0, r1, r2, r3, r4, r5. Все промежуточные результаты и итог вычислений соответствуют ожиданиям.



1) Можно ли выполнить загрузку регистров ЦПУ традиционным способом с использованием следующих команд: напомним, что X, Y, Z, … –, это адреса размещения соответствующих операндов в кодовой памяти.

```
; Загрузка регистров исходными операндами
      LDR r1, X
      LDR r2, Y
      LDR r3, Z
      LDR r4, W
      LDR r5, Q
```

2) Приведите пример значений операндов Y и Z, при которых результат произведения (Y·Z) будет некорректным (возникнет переполнение разрядной сетки). Проверьте в отладчике.
3) Приведите пример значений исходных операндов, при которых частное в операции деления будет рассчитано неточно.
4) Возможно ли переполнение разрядной сетки при выполнении операций сложения? Как его можно зафиксировать?
5) Какой флаг будет об этом свидетельствовать?

1) Да, конечно. Так как данные расположены «близко», можно использовать относительную адресацию операнда по текущему состоянию PC и смещению.

2) Для этой команды исходные операнды являются 32-разрядными числами без знака U32*U32. В общем случае результат должен быть U64, но он всегда «усекается» до 32 разрядов, то есть старшее слово произведения, если оно есть, просто отбрасывается. При этом возможное переполнение не контролируется. Только в том случае, когда исходные числа были в формате U16 и перед операцией умножения загружены в регистры ЦПУ с расширением формата до U32, переполнения при умножении полностью исключаются. В противном случае, любая операция умножения исходных операндов, результат которой превышает максимально возможное 32-разрядное число без знака 232-1=4,294,967,295, будет выполнена некорректно (старшее слово произведения будет потеряно). Примеры: (65536*65536); (171798692*25). Убедитесь в этом.

3) При операции деления остаток всегда отбрасывается. Например, при операции (7000/21)=333.333 дробная часть результата будет потеряна.
4) Да. Для контроля используйте команды сложения с дополнительным суффиксом «S» установки флагов.
5) Флаг переноса «C» (так как мы работаем с числами без знака).

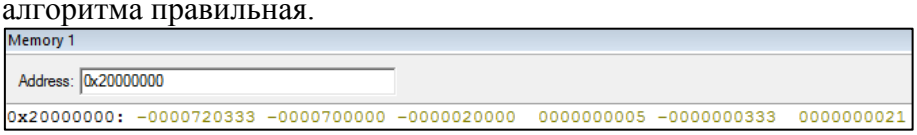
11.9.3.2 Вычисления с 32-разрядными числами со знаком в дополнительном коде

Напишем программу для решения той же вычислительной задачи, но с использованием слов со знаком в дополнительном коде (MyProg_11_1.s). Добавим префикс «S» к команде деления слов SDIV. Команду умножения MUL оставим без изменений, так как она по умолчанию работает и со знаковыми операндами. Заметьте также, что операции сложения чисел со знаком не отличаются от операций сложения чисел без знака. В конце кодовой секции проинициализируем таблицу исходных слов со знаком с помощью директивы Ассемблера DCD.

```
 7  MyProg
 8  ; Демонстрация использования команд умножения и деления
 9  ; слов со знаком для вычисления выражения
10  ; F=X+(Y*Z)+(W/Q). Распределение регистров ЦПУ:
11  ; r0=r1+(r2*r3)+(r4/r5)
12  ; Загрузка регистров исходными операндами
13    ; Инициализация базового регстра r6 начальным
14    ; адресом таблицы констант в памяти.
15          ADR r6,A_ROM
16    ; Множественная загрузка регистров
17          LDM r6,{r1-r5}
18  ; Основной вычислительный алгоритм
19    ; Вычисление выражения (r4/r5)
20          SDIV r4,r5  ; Частное в r4
21    ; Вычисление выражения (r2*r3)
22          MUL r2,r3   ; Произведение в r2
23    ; Суммирование компонент
24          ADD r0,r1,r2
25          ADD r0,r4
26  ; Сохранение промежуточных и итогового результата
27  ; в памяти для анализа
28    ; Инициализация базового регистра r6 начальным
29    ; адресом области ОЗУ
30          LDR r6,=A_RAM
31    ; Множественное сохранение содержимого регистров
32          STM r6,{r0-r5}
33  ; Закончить вычисления
34  Spop    B .
35
36  ; Инициализация констант в текущей кодовой секции
37  ; 32-разрядные слова без знака
38  A_ROM
39  X       DCD -700000
40  Y       DCD -4000
41  Z       DCD +5
42  W       DCD -7000
43  Q       DCD +21
```

Для множественной загрузки регистров процессора данными из кодовой памяти используем тот же базовый регистр r6, содержащий начальный адрес размещения данных. Завершим вычислительный алгоритм операцией сохранения результатов расчета в оперативной памяти для последующего анализа. Для этой цели включим в программу секцию данных и зарезервируем в ней место под переменные проекта (аналогично предыдущей программе).

Выполним трансляцию и сборку файлов проекта. При выполнении отладки не забудем открыть окно дампа памяти и установить в нем режим отображения информации (Dicimal, Signed, Int – знаковых 32-разрядных слов). Результат работы программы на наборе переменных, указанных в тексте программы, показывает, что реализация алгоритма правильная.



1) Почему в программе нельзя использовать команду инициализации базового регистра r6 в виде: ADR r6, A_RAM?
2) Почему в режиме отладки программы пришлось изменить формат выдачи данных в окно памяти?
3) Куда записывается остаток от деления при выполнении операции SDIV?
4) При каких исходных числах возможно переполнение в процессе операции знакового умножения?
5) Возможно ли знаковое переполнение при выполнении операций суммирования. Какой флаг в этом случае будет установлен и при каких условиях?

1) Метка A_RAM является «дальней» 0x20000000, а в командах относительной адресации по счетчику команд PC величина смещения ограничена.
2) Так как в программе используются слова со знаком.
3) При делении целых чисел остаток от деления отбрасывается. В данном случае (-7000)/(+21)= -3333333. В результате получаем частное -333 (см. содержимое регистра r4 в дампе памяти).
4) При любых операндах, когда произведение S32*S32=S64 будет выходить за допустимый диапазон представления 32-разрядных чисел со знаком ($-2^{31} \div +2^{31}-1$), то есть (-2,147,483,648 ÷ +2,147,483,647). При этом старшее слово результата отбрасывается, и никакого флага переполнения не формируется. Примеры некорректных операций: (-1,000,000,000)*(+3); (+2,000,000,000)*(+2). Используйте команду MUL при работе с расширенными до формата S32 16-разрядными исходными операндами S16. В этом случае произведение (S16*S16) = S32 никогда не выйдет за пределы разрядной сетки процессора.
5) Да. Контроль переполнений возможен при анализе состояния флага V, но только в том случае, если команды сложения имеют опцию формирования флагов «S».

11.10 Пример использования команд «длинного» умножения

В заключение этой главы рассмотрим пример использования операций «длинного умножения». В каталоге CPU_11 имеется файл MyProg_11_2.s, содержащий подпрограмму попарного умножения 32-разрядных слов без знака, расположенных в памяти, начиная с заданного адреса в регистре r0, с сохранением 64-битного произведения в оперативной памяти, начиная с заданного адреса в регистре r1.Число пар исходных слов задается содержимым регистра r2.

```
; *************************************************************
; Подпрограмма попарного умножения 32-раздрядных слов массива
; чисел без знака с сохранением 64-разрядных произведений
; в оперативной памяти
; Входы: r0 - начальный адрес массива исходных слов
;        r1 - начальный адрес массива произведений в ОЗУ
;        r2 - число пар исходных слов (счетчик числа циклов)
; Выход: Массив 64-битовых слов-произведений в ОЗУ
; Используемые регистры:
;        (r3,r4) - пара исходных 32-разрядных операндов
;        (r5,r6) - пара регистов для хранения произведения
;*************************************************************
MUL_U32_U32
; Загрузить очередную пару исходных слов в регистры (r3,r4)
; с авто- пост- инкременитрованием указателя r0
        LDM r0!,{r3,r4}
; Попарно умножить текущие слова без знака с сохранением
; произведения в регистрах (r5,r6)
        UMULL r5,r6,r3,r4
; Сохранить текущее произведение в оперативной памяти с
; пост- авто- инкрементированием указателя r1
        STM r1!,{r5,r6}
; Декрементировать счетчик числа циклов, и если не все
; операции умножения выполнены, - повторить
        SUBS r2,#1
        BNE MUL_U32_U32
; Вернуться в основную программу
        BX lr
;-------------------------------------------------------------
```

В подпрограмме используется команда множественной загрузки регистров процессора (r3, r4) из таблицы констант в ПЗУ с пост-автоинкрементированием указателя (r0) для многократного последовательного считывания нескольких пар исходных слов из памяти в регистры ЦПУ.

Результат умножения очередной пары слов записывается в регистры (r5, r6) и, с помощью команды множественного сохранения их содержимого в памяти, сохраняется в ОЗУ. При этом также используется команда с пост- авто-инкрементированием указателя (r1) для последовательного сохранения любого числа произведений в памяти.

Основная программа выполняет передачу параметров в подпрограмму и осуществляет ее вызов.

```
 7  MyProg
 8  ; Пример использования подпрограммы "длинного" умножения
 9  ; слов без знака
10  ; Определение числа исходных 32-разрядных слов в массиве
11  N_32    EQU 6
12  ; Передача параметров в подпрограмму умножения и ее вызов
13    ; Загрузка начального адреса размещения слов в ПЗУ
14          ADR r0,A_ROM
15    ; Загрузка начального адреса размещения произведений в ОЗУ
16          LDR r1,=A_RAM
17    ; Загрузка счетчика числа циклов
18          MOV r2,#(N_32/2)
19    ; Вызов подпрограммы
20          BL MUL_U32_U32
21
22  ; Закончить вычисления
23  Stop    B .
```

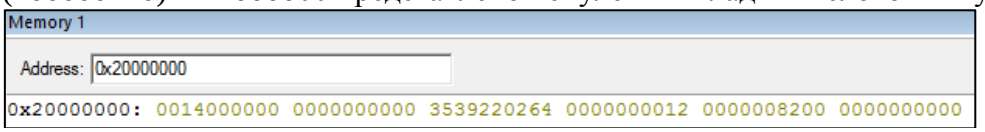
В конце кодовой секции размещается блок инициализации исходных данных – слов без знака.

```
56  A_ROM
57  X       DCD 700000
58  Y       DCD 20
59  Z       DCD 1713396
60  W       DCD 32146
61  Q       DCD 4100
62  F       DCD 2
```

Для размещения результатов вычислений в оперативной памяти, как обычно, объявляется секция данных, в которой резервируется требуемое число слов (в данном случае, с запасом).

```
65  ; Секция данных в ОЗУ
66          AREA MyData, Data, ReadWrite
67  A_RAM
68          SPACE 100*4
69  ; Конец ассемблерного текста
70          END
```

Выполните трансляцию и сборку файлов проекта. Откройте окно памяти и настройте его на выдачу информации в виде 32-разрядных слов без знака в десятичной системе счисления. Выполните программу. Результат показан ниже: первое произведение (700000*20) = 14000000 представлено ненулевым младшим словом и нулевым вторым.



Второе произведение (1,713,396*32,146) = 55,078,827,816 – уже двумя ненулевыми словами. В Hex-формате произведение выглядит так – 0xCD2F43728 (при этом старшее слово 0xC = 12D, а младшее D2F43728 = 3,539,220,264D), что соответствует информации, выданной в окно памяти. Третье произведение представлено ненулевым младшим словом (4100*2) = 8200 и нулевым старшим.

1) Выполните отладку программы с другими наборами исходных данных.
2) Модифицируйте подпрограмму для вычисления «длинного произведения» слов со знаком в дополнительном коде и сохранения результатов в памяти.
3) Что нужно изменить в основной программе для отладки новой подпрограммы? Выполните отладку.

2) Используйте вместо команды UMULL команду SMULL. Измените имя подпрограммы на MUL_S32_S32 (см. MyProg_11_3.s).
3) Кроме нового имени подпрограммы измените также исходные данные в ПЗУ: введите слова со знаком. Не забудьте при отладке сменить формат выдачи информации в окно памяти на знаковый.

При умножении переменных всегда должны использоваться форматы исходных операндов и соответствующие им команды умножения, которые исключают выход произведения за допустимые диапазоны чисел. В противном случае возможно получение неправильных результатов, в частности, усечение результата до одного слова вместо двух. Найти такие ошибки чрезвычайно сложно. В конце книги мы рассмотрим формат чисел с плавающий точкой и операции с такими числами, при которых отмеченные здесь проблемы полностью исключаются.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер..с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) ARM DDI 0403C_errata_v3, ARMv7-M Architecture Reference Manual, Arm Limited, 2010.
3) ARM DDI 0337H, Cortex-M3. Technical Reference Manual. Arm Limited, 2010.
4) ARM DUI 0068B. ARM Developer Suite. Assembler Guide. Arm Limited, 2001.
5) ARM DUI 0553A. Cortex-M4 Devices. Generic User Guide. Arm Limited, 2010.
6) ARM DDI 0439D. ARM Cortex-M4 Processor. Technical Reference Manual. Arm Limited, 2013.
7) ARM DDI 0479B. Cortex-M System Design Kit. Technical Reference Manual. Arm Limited. – 2011.
8) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
9) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
10) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
11) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

12 РАБОТА С БИТОВЫМИ ПЕРЕМЕННЫМИ. РЕАЛИЗАЦИЯ ЛОГИЧЕСКИХ КОНТРОЛЛЕРОВ И ДИСКРЕТНЫХ АВТОМАТОВ

Оглавление

12.1 Логические побитовые операции

Логическими называются команды, которые работают с битовыми переменными. В системе команд имеется пять основных логических команд:

```
AND{S}{cond}{Rd},Rn,Op2
ORR{S}{cond}{Rd},Rn,Op2
EOR{S}{cond}{Rd},Rn,Op2
BIC{S}{cond}{Rd},Rn,Op2
ORN{S}{cond}{Rd},Rn,Op2
```

Второй универсальный операнд: 1) #Const; 2) Rm; 3) Rm, Shift

Первый операнд-источник

Операнд-приемник (опция). Если не указан, то приемником является первый операнд-источник.

Код условного выполнения команды

S – (Set) – Опция установки флагов результатов операции: N, Z, C

AND – Logical AND – побитовое «Логическое И»:	**Rd ← (Rn AND Op2)**
ORR – Logical OR – побитовое «Логическое ИЛИ»:	**Rd ← (Rn OR Op2)**
EOR – Exclusive OR – побитовое «Исключающее ИЛИ»:	**Rd ← (Rn XOR Op2)**
BIC – Bits Clear – побитовое «Логическое И НЕ»:	**Rd ← (Rn AND NOT Op2)**
ORN – OR NOT – побитовое «Логическое ИЛИ НЕ»:	**Rd ← (Rn OR NOT Op2)**

Логические операции выполняются одновременно над всеми 32-мя битами первого операнда-источника Rn и второго универсального операнда Op2, который может быть непосредственной константой, регистром, или регистром, содержимое которого предварительно (попутно) сдвигается на заданное число разрядов в кольцевом сдвиговом регистре процессора. Таким образом, при выполнении логической операции над двумя одноименными битами операндов (имеющих одинаковые номера), параллельно выполняются такие же логические операции над всеми остальными битами 32-разрядных операндов.

Часто содержимое второго операнда рассматривается как *битовая маска*, с помощью которой выполняется *модификация отдельных бит или целых битовых полей* первого регистра-источника. Ниже (табл. 12.1) показан принцип такой модификации на примере одного байтового поля исходных операндов.

Таблица 12.1 Иллюстрация применения «битовых масок»

Логическая операция	Назначение	Пример								
И	**Сброс в «0» бит, для которых в маске установлены нули.**	**Операция**	**Операнды/Результат**							
		AND	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**
			1	**1**	**0**	**0**	**0**	**0**	**1**	**1**
			1	**0**	**0**	**0**	**0**	**0**	**1**	**1**
ИЛИ	**Установка в «1» бит, для которых в маске установлены единицы.**	**Операция**	**Операнды/Результат**							
		OR	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**
			0	**0**	**1**	**1**	**1**	**1**	**0**	**0**
			1	**0**	**1**	**1**	**1**	**1**	**1**	**1**
Исключающее ИЛИ	**Инвертирование бит, для которых в маске установлены единицы.**	**Операция**	**Операнды/Результат**							
		EOR	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**
			0	**0**	**1**	**1**	**1**	**1**	**0**	**0**
			1	**0**	**0**	**0**	**1**	**0**	**1**	**1**

И НЕ	**Сброс в «0» бит, для которых в маске установлены единицы**	**Операция**	**Операнды/Результат**						
		AND NOT	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**
			1	**1**	**0**	**0**	**0**	**0**	**1**	**1**
			0	**0**	**1**	**1**	**0**	**1**	**0**	**0**
ИЛИ НЕ	**Установка в «1» бит, для которых в маске установлены нули.**	**Операция**	**Операнды/Результат**						
		OR NOT	**1**	**0**	**1**	**1**	**0**	**1**	**1**	**1**
			0	**0**	**1**	**1**	**1**	**1**	**0**	**0**
			1	**1**	**1**	**1**	**0**	**1**	**1**	**1**

Вы наверняка заметили, что в представленном выше списке логических команд не хватает операции «Логического НЕ». Эту функцию выполняет уже рассмотренная нами раньше (см. 10.2) команда пересылки операнда с предварительной побитовой инверсией:

```
MVN{S}{cond} Rd, Op2
```

Она, в отличие от предыдущих команд, всегда двухоперандная. Регистр-приемник указывается обязательно (не может быть опцией) и всегда принимает результат побитового инвертирования.

Обратимся к проекту CPU_13, в котором (программа MyProg_13) демонстрируются возможности команд маскирования битовых полей. Регистр r0 загружается начальными данными, которые модифицируются с использованием маски, загруженной в регистр r1. Первая команда AND r0, r1 использует маску «как есть», а вторая – выполняет «попутный» циклический сдвиг маски на 4 разряда вправо (ROR #4). В результате нулевое поле маски перемещается в область самых старших разрядов для очистки старшего нибла регистра r0. В программе показаны возможности ввода непосредственных операндов в разных системах счисления, в том числе в двоичной. Обратите внимание на то, что операция побитового инвертирования выполняется командой MVN, а также на то, что для установки в «1» или сброса в «0» нужного бита или битового поля можно использовать разные команды (ORR, ORN) и (AND, BIC). Предпочтительнее та из них, для которой маска может быть задана более коротким непосредственным операндом.

```
 7  MyProg
 8  ; Демонстрация технологии маскирования бит
 9  ; и битовых полей
10  ; Загрузить в регистр r0 начальные данные
11          LDR r0,=0xFFAABB55
12  ; Загрузить в регистр r1 маску
13          MOV r1,#0xFFFFFFF0
14  ; Очистить младший нибл регистра r0
15          AND r0,r1
16  ; Очистить старший нибл регистра r0
17          AND r0,r1,ROR #4
18  ; Установить в "1" все разряды младшего нибла r0
19          ORR r0, #2_00001111
20  ; Установить в "1" все разряды старшего байта r0
21          LDR r1,=0x0FFFFFFF
22          ORN r0,r1
23  ; Проинвертировать все биты регистра r0
24          MVN r0, r0
25  ; Установить 3-й бит регистра r0
26          ORR r0,#0x00000008
27  ; Очистить 3-й бит регистра r0
28          AND r0,#0xFFFFFFF7
29  ; Установить 3-й бит регистра r0 командой ORN
30          ORN r0,#0xFFFFFFF7
31  ; Очистить 7-й бит регистра r0 командой BIC
32          BIC r0,#2_10000000
```

1) Какие операции сдвига называются «попутными»?
2) В чем их преимущество?
3) Выполните отладку программы. Убедитесь в правильности операций маскирования данных.
4) Просмотрите исполняемый файл в окне Дизассемблера и найдите команду, которую транслятор заменил на более компактную.

1) Операции со вторым универсальным операндом-источником, в котором задается дополнительный сдвиг указанного типа содержимого операнда на заданное число разрядов.
2) Фактически одной командой, за то же время (один такт) выполняются две операции: сдвига и логическая или арифметическая.

3) Выполните программу в пошаговом режиме, проследите за содержимым регистра r0: 0xFFAABB55; 0xFFAABB50; 0x0FAABB50; 0x0FAABB5F; 0xFFAABB5F; 0x005544A0; 0x005544A8; 0x005544A0; 0x005544A8; 0x00554428.
4) Это команда загрузки регистра r1: удалось заменить ее командой загрузки другой константы с предварительной побитовой инверсией. В результате не пришлось создавать в кодовой секции литеральный пул с исходной константой. Этот пример – демонстрация возможностей интеллектуального транслятора с языка Ассемблер.

```
    21:                                 LDR r1,=0x0FFFFFFF
0x0000002A F06F4170  MVN             r1,#0xF0000000
```

12.2 Операции сдвига

Мы рассмотрели ранее (см. 7.11.4) общий принцип выполнения «попутных» операций сдвига данных в кольцевом сдвиговом регистре процессора, если второй операнд-источник задается как «гибкий второй операнд». Попутные операции сдвига можно использовать во многих командах. Фактически одна такая команда выполняет сразу две операции: предварительного сдвига содержимого второго операнда-источника и основную операцию (арифметическую, логическую, пересылки) над результатом сдвига и содержимым первого операнда-источника. Тем не менее, в состав команд процессора включены отдельные команды сдвига, которые могут выполнять *исключительно операцию сдвига*, рассматривая ее как основную, а не дополнительную:

```
ASR{S}{cond} Rd, Rm, Rs|#n
LSL{S}{cond} Rd, Rm, Rs|#n
LSR{S}{cond} Rd, Rm, Rs|#n
ROR{S}{cond} Rd, Rm, Rs|#n
RRX{S}{cond} Rd, Rm
```

Регистр Rs или константа #n – задает число разрядов сдвига
Операнд-источник – его содержимое сдвигается
Операнд-приемник. Не может быть опцией, указывается всегда
Код условного выполнения команды
S – (Set) – Опция установки флагов результатов операции: N, Z, C

ASR – Arithmetic Shift Right – Арифметический сдвиг вправо
LSL – Logical Shift Left – Логический сдвиг влево
LSR – Logical Shift Right- Логический сдвиг вправо
ROR – Rotate Right – Циклический сдвиг вправо
RRX – Rotate Right with Extend – Расширенный циклический сдвиг вправо на 1 разряд

Особенность команд в том, что тип операции сдвига определяется мнемокодом команды, а число разрядов сдвига – содержимым регистра сдвига Rs или непосредственной константой #n. Число разрядов сдвига можно задавать в диапазоне от 0 до 31. Исключение составляет операция арифметического сдвига вправо, для которой величина сдвига находится в диапазоне 0÷32. При n = 32 во все разряды регистра-приемника будет записан знаковый разряд операнда-источника. Для команд сдвига указание имени регистра-приемника обязательно.

Опция установки флагов «S» позволяет модифицировать флаги N, Z и C по результатам операции сдвига.

При нулевом числе разрядов сдвига предпочтительнее использовать вместо операции сдвига ее аналог – команду пересылки (см. табл. 12.2).

Таблица 12.2 Пример эквивалентных команд

Команда сдвига	**Эквивалентная команда пересылки**
LSL{S}{cond} Rd, Rm, #0	**MOV{S}{cond} Rd, Rm**

Напротив, вместо команды пересылки MOV со вторым гибким операндом, если число разрядов попутного сдвига отлично от нуля, целесообразно использовать ее аналог – команду сдвига (см. табл. 12.3).

Таблица 12.3 Пример эквивалентных команд

Команда пересылки	**Предпочтительный синтаксис с использованием команды сдвига**
MOV{S}{cond} Rd, Rm, ASR #n	**ASR{S}{cond} Rd, Rm, #n**
MOV{S}{cond} Rd, Rm, LSL #n (if n != 0)	**LSL{S}{cond} Rd, Rm, #n**
MOV{S}{cond} Rd, Rm, LSR #n	**LSR{S}{cond} Rd, Rm, #n**
MOV{S}{cond} Rd, Rm, ROR #n	**ROR{S}{cond} Rd, Rm, #n**
MOV{S}{cond} Rd, Rm, RRX	**RRX{S}{cond} Rd, Rm**

Транслятор *автоматически генерирует* команды предпочтительного синтаксиса, сохраняя исходный код Ваших программ неизменным. Вы можете убедиться в этом, просмотрев файл листинга программы-примера MyProg_13_1.s или изучив информацию в окне Дизассемблера в процессе отладки этой программы.

```
 7  MyProg
 8  ; Использование команд пересылки с попутным сдвигом
 9  ; и эквивалентных им обычных команд сдвига
10  ; Загрузить в регистр r1 начальные данные
11          LDR r1,=0xFFAABB55
12  ; Переслать в регистр r0 с предварительным
13  ; логическим сдвигом на 8 разрдяов влево
14          MOV r0,r1, LSL #8
15  ; Эквивалентная ей команда логического сдвига
16          LSL r0,r1,#8
17  ; Переслать в регистр r0 содержимое r1 с обменом
18  ; местами старшего и младшего полуслов
19          MOV r0,r1,ROR #16
20  ; Эквивалентная ей команда логического сдвига
21          ROR r0,r1,#16
22  ; Переслать в регистр r0 содержимое r1 с обменом
23  ; данными в старшем и младшем байтах слова
24  ; (средние байты могут быть изменены)
25          MOV r0,r1,ROR #8
26  ; Эквивалентная команда циклического сдвига
27          ROR r0,r1,#8
```

В примере демонстрируются команды пересылки слов с «попутным» сдвигом данных и команды логического сдвига, которые являются взаимозаменяемыми. Они позволяют, в частности, легко менять положение полуслов и байт в слове. Такие действия могут потребоваться при получении и обработке данных с внешних портов ввода или регистров периферийных устройств.

12.3 Программная реализация логических контроллеров

Логическими контроллерами называются устройства, которые, получая с портов ввода микропроцессорной системы вектор текущего состояния объекта управления (состояния дискретных датчиков и исполнительных устройств объекта), называемый *вектором входа* X (x_n, x_{n-1},…x_0), вырабатывают набор управляющих воздействий *вектор выхода* Y (y_m, y_{m-1},….y_0), в строгом соответствии с системой логических (Булевых) функций, описывающих поведение устройства:

$$\begin{cases} y_m = f(x_n, x_{n-1}, \dots x_0); \\ \dots \\ y_0 = f(x_n, x_{n-1}, \dots x_0); \end{cases} \quad (12.1)$$

Все входные переменные и выходные управляющие воздействия в таких устройствах являются *битовыми переменными*, которые могут принимать только два возможных значения «1» или «0». Входные переменные отражают состояние дискретных датчиков, установленных на объекте (давления, расхода, температуры, конечного положения исполнительных органов и т.д.), а выходные переменные являются командами на включение/выключение соответствующих исполнительных устройств (контакторов, реле, силовых ключей и т.д.).

В общем случае размерность вектора входа X может достигать нескольких сотен компонент, а вектора выхода Y – нескольких десятков или сотен. Логический контроллер может быть реализован аппаратно с помощью логических элементов малой степени интеграции (элементы И, ИЛИ, НЕ и др.), средней степени интеграции (мультиплексоры, шифраторы, триггеры, счетчики…) или большой степени интеграции (постоянная память ПЗУ, программируемые логические интегральные схемы (ПЛИС), микроконтроллеры…). Все решения предполагают реализацию системы логических функций (1). Рассмотрим способы программной реализации системы Булевых функций на базе микроконтроллеров с процессорами ARM Cortex-M3/M4.

12.3.1 Базовые принципы программной реализации логических контроллеров

1. Все порты ввода/вывода и регистры периферийных устройств в ARM-микроконтроллерах отображены на память, следовательно, получение входной информации (вектора входа X) – это обращение к соответствующим ячейкам памяти *по чтению*, а выдача управляющих воздействий (вектора выхода Y) – обращение к соответствующим ячейкам памяти *по записи*. Число таких ячеек памяти определяется разрядностью векторов входа и выхода и разрядностью портов ввода/вывода, обеспечивающих сопряжение микроконтроллера с внешними устройствами.
2. В общем случае (см. главу 10) обращение к портам ввода/вывода и регистрам периферийных устройств возможно по байтам (LDRB/STRB), полусловам (LDRH/STRH) или словам (LDR/STR). Однако, в любом случае данные при чтении попадают в 32-разрядные регистры ЦПУ, а при записи – извлекаются из них. Таким образом, компоненты вектора входа X упаковываются в несколько 32-разрядных слов в регистрах ЦПУ при опросе текущего состояния внешних входов, а компоненты вектора выхода Y рассчитываются и сохраняются в нескольких регистрах ЦПУ перед выдачей управляющих воздействий во вне.
3. Цикл работы логического контроллера (один проход, называемый *сканом)* состоит из трех блоков:

   1) Получение извне всех компонент вектора входа X и размещение их в регистрах ЦПУ;
   2) Вычисление всех компонент вектора выхода Y для текущего набора входных переменных X;
   3) Выдача вектора выхода Y вовне – вывод управляющих воздействий.
4. Этот цикл должен повторяться бесконечно, в результате чего каждый раз рассчитывается новый вектор выхода на основе текущего значения вектора входа. Считывание абсолютно всех компонент вектора входа перед началом расчета вектора выхода является обязательным. В противном случае, логический контроллер может выдать не предусмотренный алгоритмом вектор управляющих воздействий с непредсказуемыми результатами управления объектом.
5. Копия вектора входа X в регистрах ЦПУ является «образом входных битовых переменных», а рассчитанный вектор выхода Y до его выдачи вовне – «образом вектора выхода Y».
6. В большинстве практических случаев общего числа регистров ЦПУ для размещения в них входных и выходных битовых переменных вполне достаточно. Так, если число битовых входов не превышает 288, а число битовых выходов 64, возможно такое распределение регистров ЦПУ:
   1) (r1-r9) – компоненты вектора входа – до (9*32) = 288 бит;
   2) (r11-r12) – компоненты вектора выхода – до (2*32) = 64 бит.
7. Если регистров процессора все же недостаточно, для размещения компонент векторов входа и выхода используется память данных (ОЗУ). При этом ограничения по размерности векторов снимаются. Однако, лучше проанализировать задачу и разбить сложный логический контроллер на подконтроллеры, для каждого из которых число входов и выходов будет ограничено. Возможно, это будет группа контроллеров, работающих совместно, в том числе друг на друга.
8. Реализация логических контроллеров и дискретных автоматов с прямым доступом к так называемым «битовым лентам в памяти» ARM-процессоров будет рассмотрена в следующей главе.

12.3.2 Реализация логических контроллеров по таблице истинности

Если размерности векторов входа и выхода невелики, то программист может рассчитать и записать в кодовую память микропроцессорной системы управления *таблицу истинности для всей системы логических функций* (1). Проиллюстрируем сказанное на примере логического контроллера, обрабатывающего байтовый вектор входа (8 дискретных входов X0-X7) и генерирующего байтовый вектор выхода (8 битовых управляющих воздействий Y0-Y7). Таблица истинности содержит значения всех логических функций на всех возможных наборах входных переменных. Число таких наборов, т.е. строк в таблице истинности (пример см. в табл. 12.4), определяется размерностью вектора входа, в данном случае 256:

Таблица 12.4 Таблица истинности

Вектор входа X								**Вектор выхода Y**							
X7	**X6**	**X5**	**X4**	**X3**	**X2**	**X1**	**X0**	**Y7**	**Y6**	**Y5**	**Y4**	**Y3**	**Y2**	**Y1**	**Y0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**	**0**	**0**
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**	**…**
1	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**0**	**0**
1	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**1**	**0**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**

Таким образом, если вектор выхода имеет размерность 8, то таблица истинности занимает всего 256 байт в кодовой памяти. При увеличении размерности вектора выхода вдвое (16 бит) таблица истинности увеличивается до 256 полуслов, еще вдвое (до 32 бит) – до 256 полных 32-разрядных слов.

В чем состоит идея метода? Если таблица истинности рассчитана и размещена в памяти, то все что требуется – это считать из нее вектор выхода Y, зная значение вектора входа X. Если вектор выхода байтовый, то значение X точно соответствует индексу в таблице истинности (номеру строки) i=X. Если в таблице хранятся полуслова, то значение индекса должно быть удвоено i=(2·X), а если полные слова, то увеличено в 4 раза i=(4·X). Доступ к массивам данных по базовому адресу и индексу обеспечивается с использованием базово-индексной адресации [Rn, Rm] с пре-индексированием. В этом случае перед доступом к данным содержимое базового регистра Rn автоматически увеличивается на значение в индексном регистре Rm. Используя такую адресацию можно вообще без вычислений извлечь значение вектора выхода из таблицы истинности и выдать его для управления объектом. Быстро и эффективно.

Пример программной реализации логического контроллера по таблице истинности представлен в файле MyProg_13_2.s (проект CPU_13). Первое, что должен сделать программист, это заполнить таблицу истинности данными в соответствии с подлежащей реализации системой булевых функций. В нашем случае каждый вектор выхода представляет байт данных и для его размещения в кодовой памяти служит директива Ассемблера DCB. Таблица истинности размещается в самом конце кодовой секции и для ее заполнения можно использовать константы в любой удобной системе счисления (например, в двоичной).

```
35  ; Резервироание места в кодовой памяти и инициализация
36  ; таблицы истинности системы логических функций
37  Tab_Ist
38  ; Всего в таблице должно быть 256 байт данных
39  ; (показано только начало таблицы - 10 байт)
40          DCB   2_11000010
41          DCB   2_00001111
42          DCB   2_10111000
43          DCB   2_11000000
44          DCB   2_11011001
45          DCB   2_10101010
46          DCB   2_11011011
47          DCB   2_11011011
48          DCB   2_11011011
49          DCB   2_01111000
50  ;       ...
```

```
52  ; Объявление секции данных, в которой размещается
53  ; порт ввода вектора X и вывода вектора Y
54          AREA MyData, Data, ReadWrite
55  X       SPACE 1
56  Y       SPACE 1
```

Необходимо определиться с портами ввода входных битовых переменных и вывода управляющих воздействий. Если их адреса известны, то они должны быть предварительно описаны, например, с помощью директивы EQU (Эквивалентно). Так как мы будем отлаживать программу в симуляторе, объявим в ее конце секцию данных и зарезервируем в ней всего два байта: X – для симуляции байтового порта ввода и Y – для симуляции байтового порта вывода.

```
 7  MyProg
 8  ; Реализация логического контроллера по таблице
 9  ; истинности системы логических функций
10  LOG_CONTR
11  ; Проинициализировать базовый регистр r0
12  ; начальным адресом таблицы истинности
13          ADR r0,Tab_Ist
14  ; Проинициализировать базовый регистр r1
15  ; адресом порта ввода входных переменнх
16          LDR r1,=X
17  ; Проинициализировать базовый регистр r2
18  ; адресом порта вывода управляющих воздействий
19          LDR r2,=Y
```

Теперь можно приступить к написанию программы логического контроллера. Для доступа к таблице истинности будем использовать регистр-указатель r0, для доступа к порту ввода – регистр-указатель r1, к порту вывода – регистр-указатель r2. Так как содержимое базовых регистров не будет меняться, в теле основного вычислительного алгоритма их инициализация выполняется только один раз.

```
20  LOOP
21  ; Получить текущее состояние входных переменных
22  ; (вектор X)
23          LDRB r3,[r1]
24  ; Считать из таблицы истинности вектор выхода
25          LDRB r4,[r0,r3]
26  ; Выдать вектор выхода в порт вывода управляющих
27  ; воздействий
28          STRB r4,[r2]
29  ; Выполнить очередной скан расчета
30          B LOOP
```

Собственно, программа логического контроллера предельно проста: получение вектора входа из порта ввода с сохранением в регистре r3; извлечение вектора выхода в регистр r4 из таблицы истинности по базовому адресу в r0 и индексу в регистре r3; выдача вектора выхода из регистра r4 в порт вывода (по адресу в указателе r2). Дальше программа принудительно зацикливается для получения очередного набора входных данных и расчета выходных управляющих воздействий.

1) Выполните трансляцию и сборку проекта. Убедитесь в том, что при задании в окне памяти любого (в пределах 0 – 10) вектора входа из таблицы действительно извлекается нужное значение вектора выхода и сохраняется в соответствующей ячейке памяти.
2) В чем на Ваш взгляд преимущества этого метода решения?
3) В чем недостатки?
4) Почему используемый в программе метод адресации данных в таблице истинности называется базовым с пре-индексированием?
5) Что нужно изменить в программе, чтобы выполнялся доступ к 16-разрядным векторам выхода?

2) Компактность кода, предельно высокая скорость выполнения программы, максимально быстрая реакция на изменение входных переменных.
3) Составление таблицы истинности – неблагодарный труд. Более того, при любом изменении алгоритма ее коррекция требует времени. При числе входных переменных, превышающих уже несколько единиц, таблица получается объемной.
4) Индекс сначала добавляется к базовому адресу, и по полученному эффективному адресу уже извлекаются данные.
5) Первое: заменить суффикс «B» (байт) на суффикс «H» (полуслово) в командах основного тела алгоритма. Второе: после получения вектора входа из порта удвоить его значение.

12.3.3 Непосредственное вычисление логических функций с использованием логических команд

12.3.3.1 Имеется ли в процессорах Cortex-M3/M4 битовый сопроцессор?

В процессорах, ориентированных на эффективную реализацию логических контроллеров и дискретных автоматов (например, для использования в промышленных программируемых контроллерах, решающих задачи комплексной автоматизации производства), обычно применяются так называемые *битовые сопроцессоры*. Это устройства, которые могут оперировать с отдельными битами, не меняя состояния остальных бит. Они имеют *битовый аккумулятор* и допускают последовательное вычисление любых логических выражений с получением результата вычисления в битовом аккумуляторе (пример – микроконтроллеры Intel8051). Естественно, что настоящий битовый сопроцессор поддерживает:

1) *Загрузку* нужного бита в битовый аккумулятор;
2) *Логические операции* с текущим содержимым битового аккумулятора и любыми другими битами (И, ИЛИ, НЕ, ИЛИ НЕ, …);
3) *Сохранение* результата вычисления логического выражения в любом нужном бите (сохранение содержимого битового аккумулятора).

В таких процессорах определенная область встроенной памяти отводится для побитово-побайтово адресуемого ОЗУ, в котором поддерживается и байтовая адресация данных и *битовая адресация*. Это означает, что каждый отдельный бит имеет *свой персональный адрес*. Именно по этому адресу выполняется доступ к нужному биту в отмеченных выше операциях с битами. В процессорах Cortex-M3/M4 в явном виде битового сопроцессора нет, но он может быть «создан» самим программистом.

12.3.3.2 Как программно реализовать битовый сопроцессор в ARM Cortex-M3/M4?

Система команд, изучаемых нами процессоров, настолько мощная, что позволяет, при необходимости, создать битовый сопроцессор с использованием макросредств Ассемблера, то есть создать *библиотеку макрокоманд*, выполняющих функции битового сопроцессора, причем не менее эффективно чем настоящий, аппаратно реализованный битовый сопроцессор. Покажем, как это делается.

12.3.3.3 Функция битового аккумулятора

Эта функция может выполняться любым битом любого из регистров ЦПУ. Однако, наиболее удобно для этой цели использовать самый старший бит регистра. Договоримся называть этот регистр *аккумулятором* (символическое имя регистра-аккумулятора или формальный параметр, используемый для его обозначения в макро-определениях – $A). При вызовах макрокоманд оно будет заменяться реальным именем регистра-аккумулятора, например, r0. Все логические операции будут выполняться параллельно над всеми 32-мя битами регистров ЦПУ, но нас будет интересовать результат только в старшем 31-м бите – остальные биты, независимо от их содержимого, будут игнорироваться. Состояние 31-го бита легко проверяется по условиям: MI (N=1) – результат отрицательный (Bit_31=1) и PL (N=0) – результат положительный (Bit_31=0). Следовательно, после вычисления логического выражения можно будет сохранить полученный битовый результат в нужном месте *командами условного выполнения* по условию MI, или по условию PL.

12.3.3.4 Функция загрузки бита в битовый аккумулятор

Для ее реализации можно применить одну, из имеющихся в системе команд процессора, операций сдвига, например, операцию *логического сдвига влево* LSL

содержимого регистра-источника на нужное число бит с расширением нулями справа. Мы договорились, что образ вектора входа, то есть текущий набор входных битовых переменных, создается в регистрах общего назначения процессора. Пусть один из таких регистров имеет символическое имя $R, а конкретный бит, который должен быть загружен из него в битовый аккумулятор – номер $N (обычный литерал, например, 5).

```
; Макробиблиотека команд битового сопроцессора
;*******************************************************
; Загрузка бита с заданным номером в битовый аккумулятор
        MACRO
        L_Bit $A,$R,$N
        LSL $A,$R,#(31-$N)
        MEND
;*******************************************************
```

Число разрядов сдвига влево определяется выражением #(31-$N) и рассчитывается автоматически на стадии ассемблирования. В соответствии с синтаксисом команды LSL символ непосредственного операнда «#» перед выражением обязателен.

Создадим библиотеку команд битового сопроцессора в файле Macro_bit.lib (проект CPU_14).

При вызове макрокоманды формальный параметр $R будем заменять именем конкретного регистра ЦПУ (например, r1), а формальный параметр $N – номером бита, подлежащего загрузке (от 0 до 31). Вызов такой макрокоманды (по имени L_Bit) предельно прост L_Bit r0, r1,5: загрузить в 31-й бит регистра-аккумулятора r0 пятый бит регистра r1. В чем преимущество такой макрокоманды? Вы никогда не ошибетесь с расчетом величины сдвига: эту операцию будет выполнять сам транслятор на стадии ассемблирования программы.

12.3.3.5 Логические операции битового аккумулятора с битовыми операндами. Макробиблиотека битового сопроцессора

```
;*******************************************************
; "Логическое И" бита с битовым аккумулятором
        MACRO
        AND_Bit $A,$R,$N
        ANDS $A,$R,LSL #(31-$N)
        MEND
;*******************************************************
;*******************************************************
; "Логическое И НЕ" бита с битовым аккумулятором
        MACRO
        ANDN_Bit $A,$R,$N
        BICS $A,$R,LSL #(31-$N)
        MEND
;*******************************************************
;*******************************************************
; "Логическое ИЛИ" бита с битовым аккумулятором
        MACRO
        ORR_Bit $A,$R,$N
        ORRS $A,$R,LSL #(31-$N)
        MEND
;*******************************************************
;*******************************************************
; "Логическое ИЛИ НЕ" бита с битовым аккумулятором
        MACRO
        ORN_Bit $A,$R,$N
        ORNS $A,$R,LSL #(31-$N)
        MEND
;*******************************************************
;*******************************************************
; "Логическое Исключающее ИЛИ" бита с битовым аккумулятором
        MACRO
        EOR_Bit $A,$R,$N
        EORS $A,$R,LSL #(31-$N)
        MEND
;*******************************************************
```

В начале главы мы познакомились с логическими командами, которые выполняют сразу 32 параллельные операции над содержимым первого регистра-источника и второго-регистра источника, который может быть так называемым вторым универсальным операндом – регистром, содержимое которого предварительно сдвигается на нужное число разрядов с использованием одной из имеющихся операций сдвига. Это дает нам возможность создать макрокоманды, в которых первым регистром-источником и одновременно регистром-приемником будет регистр-аккумулятор $A, а вторым регистром – любой регистр ЦПУ $R, содержимое которого сдвигается логически влево LSL (в качестве примера) на число разрядов, определяемое номером нужного бита $N.

Это позволит выполнить прямую логическую операцию 31-го бита регистра-приемника (битового аккумулятора) и любого заданного бита регистра-источника.

Создадим такие макрокоманды для всех логических операций, поддерживаемых процессором: И, ИЛИ, И НЕ, ИЛИ НЕ, Исключающее ИЛИ. Напомним, что синтаксис второго универсального операнда Rm, Shift предполагает задание типа сдвига и через пробел – числа бит сдвига в виде непосредственного операнда, в нашем случае – LSL #n.

Обратите внимание на использование в макрокоманде «Логическое И НЕ» команды процессора BIC – «Очистить бит». Она сначала инвертирует нужный нам бит, а затем выполняет операцию «И» с содержимым аккумулятора. Такие же особенности

имеет и макрокоманда «Логическое ИЛИ НЕ». Команда процессора ORN вначале инвертирует нужный бит, а затем выполняет операцию логического «ИЛИ» с содержимым аккумулятора. Обратите внимание и на то, что все логические операции имеют суффикс «S» установки флагов. Флаги потребуются при сохранении результата расчета.

Добавим к этим макрокомандам еще одну – инвертирования содержимого битового аккумулятора. Получим *функционально полную (даже с избытком) группу логических операций*, с помощью которых можно легко реализовать любую, сколь угодно сложную, логическую функцию. Напомним, что в общем случае для реализации любой логической функции достаточно всего трех логических операций: И, ИЛИ, НЕ.

```
;*********************************************************
; "Логическое НЕ" битового аккумулятора
        MACRO
        NOT_A $A
        MVNS $A,$A
        MEND
;*********************************************************
```

Осталось разработать макроопределение записи содержимого битового аккумулятора в нужный бит одного из регистров, предназначенного для сохранения компонентов вектора выхода. Обратим внимание на то, что все наши макрокоманды выставляют флаги. Это не увеличивает время выполнения команд (все они выполняются за один такт), но позволяет сразу после завершения некоторой последовательности вычислений организовать эффективную пересылку содержимого битового аккумулятора $A в нужный бит регистра назначения $R. Номер этого бита будем задавать формальным параметром $N. Для установки бита по его номеру можно использовать логическую операцию ИЛИ с маской, в которой «1» записана в разряд, соответствующий номеру бита, а остальные разряды – нулевые. Получить такую маску можно автоматически, заставив транслятор вычислить выражение (1<< $N), в котором символ «<<» означает операцию логического сдвига влево операнда 1 на указанное после символа «<<» число разрядов.

Итак, любая макрокоманда обработки битов, выполненная последней, устанавливает флаги. Поэтому можно использовать последовательные две команды (ORR и BIC) с разными суффиксами условного выполнения:

1) «Логического ИЛИ» с суффиксом «MI», если битовый аккумулятор единичный;
2) «Логического И НЕ» с суффиксом «PL», если битовый аккумулятор нулевой.

```
;*********************************************************
; Сохранить содержимое битового аккумулятора
; в заданном бите регистра
        MACRO
        S_Bit $A, $R, $N
        ORRMI $R,#(1<<$N)
        BICPL $R,#(1<<$N)
        MEND
;*********************************************************
```

В результате будет выполнена только одна команда, для которой соответствующее условие выполнения окажется истинным: заданный бит в регистре назначения будет установлен или сброшен. Как обычно, в конце текста макробиблиотеки используется директива END.

В нашем распоряжении появились макрокоманды *битового сопроцессора*, каждая из которых соответствует одной реально существующей команде (за исключением команды S_Bit) и выполняется всего за один такт, что позволяет реализовать любую логическую функцию с предельно возможным быстродействием.

12.3.4 Пример программной реализации логического контроллера с использованием макробиблиотеки битового сопроцессора

Пусть число компонент вектора входа X и вектора выхода Y не превышает 32. Для образа вектора входа выделим регистр r1, для образа вектора выхода – регистр r2. Регистр r0 будем использовать в качестве битового аккумулятора. Требуется реализовать следующую систему логических функций:

$$\begin{cases} y_0 = x_0 \cdot x_5 \cdot \bar{x}_7 + (\bar{x}_2 + x_{31}) \cdot \overline{x_6 \oplus x_9}\,; \\ y_2 = \bar{x}_0 \cdot x_3 + (\bar{x}_4 \cdot x_{20} \cdot x_{18})\,; \\ y_7 = \bar{x}_0 \cdot \bar{x}_3\,; \\ y_8 = \overline{x_0 + x_3}\,; \\ y_{23} = \bar{x}_0 + \bar{x}_3\,; \\ y_{24} = \overline{x_0 \cdot x_3}\,; \\ y_{31} = \overline{x_1 \cdot x_2 \cdot \bar{x}_3} \end{cases} \qquad (12.2)$$

Первое, что мы сделаем – подключим в начале файла приложения MyProg_14.s (проект CPU_14) макробиблиотеку функций битового сопроцессора. Предполагается, что она находится в текущем каталоге. Определимся с тем, как при отладке программы будут вводиться входные битовые переменные и выводиться управляющие воздействия. Зарезервируем для этой цели по одному слову (X и Y) в секции данных. Модифицируя содержимое ячейки памяти X, можно задать любой вектор входа из 32 битовых компонент. За результатом работы логического контроллера будем наблюдать по содержимому ячейки памяти Y, симулирующей порт вывода.

```
1  ; Подключить файл с макробиблитекой функций
2  ; битового сопроцессора
3          INCLUDE Macro_bit.lib
```

```
58 ; Объявление секции данных в памят данных
59         AREA MyData, DATA, ReadWrite
60 ; Резервирование 2-х слов под
61 ; вектор входа X и вектор выхода Y
62 X       SPACE 4
63 Y       SPACE 4
```

Программа должна начинаться с ввода текущего состояния входных битовых переменных (вектора X). Используем для этого классическую косвенную адресацию, предварительно проинициализировав регистр указатель r5. Итак, содержимое вектора входа загружено в регистр r1, можно приступать к расчету битовых компонент вектора выхода и к их загрузке в регистр r2 (образ вектора выхода Y).

```
12 LOG_CONTR
13 ; Реализация логического контроллера с использованием
14 ; макробиблиотеки функций битового сопроцессора
15 ; Загрузить вектор входа X из памяти
16         LDR r5,=X
17         LDR r1,[r5]
```

```
19  ; Расчет функции y0
20          L_Bit r0,r1,6
21          EOR_Bit r0,r1,9
22          NOT_A r0
23          MOV r10,r0
24
25          L_Bit r0,r1,31
26          ORN_Bit r0,r1,2
27          AND r10,r0,r10
28
29          L_Bit r0,r1,0
30          AND_Bit r0,r1,5
31          ANDN_Bit r0,r1,7
32  ; Не забудьте установить флаги
33          ORRS r0,r10
34          S_Bit r0,r2,0
```

Проанализировав функцию y0, приходим к выводу, что сначала целесообразно найти выражение ($\overline{x_6 \oplus x_9}$) и временно сохранить результат в свободном регистре (например, r10). Дальше можно вычислить выражение ($\bar{x}_2 + x_{31}$) и логически умножить его на ранее сохраненное значение в r10, еще раз сохранив результат в регистре r10. Теперь можно приступить к вычислению выражения $(x_0 + x_5 + \bar{x}_7)$ и логически сложить его с содержимым регистра r10. Так как последняя логическая команда должна формировать флаги результата вычисления, используем в ее коде дополнительный суффикс «S» (установки флагов). Это позволит нам корректно воспользоваться макрокомандой установки результата вычисления в регистре r2 (образе вектора выхода Y).

```
36  ; Расчет функции y23
37          L_Bit r0,r1,0
38          NOT_A r0
39          ORN_Bit r0,r1,3
40          S_Bit r0,r2,23
41
42  ; Расчет функция y24
43          L_Bit r0,r1,0
44          AND_Bit r0,r1,3
45          NOT_A r0
46          S_Bit r0,r2,24
```

Для примера, вычислим еще две функции y23 и y24. Они достаточно просты, и все расчеты выполняются последовательно с использованием только макрокоманд библиотеки битовых операций. Никакого промежуточного сохранения результатов не требуется.

```
48  ; Вывести управляющие воздействия в порт вывода
49          LDR r5,=Y
50          STR r2,[r5]
51  ; Повторить цикл логического контроллера
52          B LOG_CONTR
```

В завершение выполним вывод управляющих воздействий в порт вывода и повторим цикл расчета заново для других значений входных переменных.

Как видите, пользоваться созданной макробиблиотекой функций битового сопроцессора – просто и эффективно!

1) Выполните трансляцию файла приложения. Исследуйте файл листинга и обратите внимание на то, как макровызовы превращаются в макрорасширения. Проследите за тем, как формальные параметры заменяются фактическими и какие константы сдвига (непосредственные операнды) генерирует транслятор по указанному номеру бита.
2) Выполните сборку проекта и запустите его на выполнение в отладчике. Откройте окно памяти, начиная с адреса расположения порта ввода X. Введите произвольное значение вектора X, получите и проанализируйте вектор выхода Y.
3) Почему значения функций y23 и y24 получаются одинаковыми на всех наборах переменных?
4) Доработайте программу MyProg_14.s, сохранив ее под именем MyProg_14_1.s и включив в нее реализацию остальных функций: y2, y7, y8, y31. Выполните отладку программы.
5) Есть ли необходимость в макроопределении функции S_Bit использовать формальный параметр $A?

1) В файле листинга указывается номер строки, ее адрес и символическое имя макрокоманды, но в колонке кода операции пока ничего не выводится. Дальше следует макрорасширение с реально сгенерированными командами процессора и их фактическими кодами. В позиции номера строки указывается номер соответствующей строки в файле макробиблиотеки, которая автоматически включается в начало файла листинга (для макрокомады L_Bit – строка 6). В непосредственных операндах указываются арифметические выражения, на базе которых транслятор делает авто-вычисления величин сдвига, что удобно при проверке правильности кода.

```
19 00000004                    ; Расчет функции y0
20 00000004                    L_Bit           r0,r1,6
 6 00000004 EA4F 6041          LSL             r0,r1,#(31-6)
21 00000008                    EOR_Bit         r0,r1,9
42 00000008 EA90 5081          EORS            r0,r1,LSL #(31-9)
22 0000000C                    NOT_A           r0
50 0000000C 43C0               MVNS            r0,r0
```

2) Самое простое – задать все нулевые значения компонент вектора X, а затем – все единичные значения, и на этих двух наборах зафиксировать значение вектора выхода Y. Для первого случая получим Y=0x01800001, для второго Y=0x00000001. Значения выходных переменных y0, y23 и y24 для этих двух наборов рассчитаны правильно. Убедитесь в этом.
3) Потому, что в соответствии с законами алгебры логики, называемыми правилами де Моргана, всегда справедливы соотношения: $\bar{x} + \bar{y} = \overline{x \cdot y}$; $\bar{x} \cdot \bar{y} = \overline{x + y}$.
5) Нет, этот формальный параметр можно опустить, так как имя регистра аккумулятора не используется в теле макроопределения. В нем используются только суффиксы условного выполнения, сформированные по результатам последней логической операции.

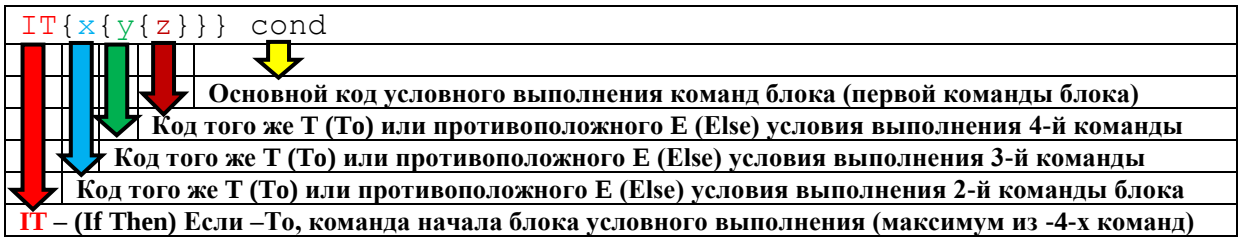
12.4 Блоки условного выполнения команд. Команда процессора IT (If-Then - Если - То)

Изучая файл листинга программы логического контроллера, Вы наверняка обратили внимание на то, что макрос сохранения битового аккумулятора расширяется несколько необычно. Перед кодом операции условного выполнения ORRMI – «Логического ИЛИ» содержимого регистра r2 с непосредственным операндом-маской стоит код еще какой-то «скрытой» команды процессора (BF4C). Разберемся, что это такое? В помощь нам окно Дизассемблера. Этот же фрагмент программы в нем выглядит так. Транслятор, встретив последовательность из двух команд условного выполнения, автоматически вставил перед ней команду IT – начала блока условного выполнения.

```
34 0000002C                    S_Bit           r0,r2,0
59 0000002C BF4C F042
           0201                ORRMI           r2,#(1<<0)
60 00000032 F022 0201          BICPL           r2,#(1<<0)
```

```
0x00000044 BF4C       ITE             MI
0x00000046 F0420201   ORRMI           r2,r2,#0x01
0x0000004A F0220201   BICPL           r2,r2,#0x01
```

Это особая команда, которая будет предварять любую отдельную команду условного выполнения или группу команд условного выполнения с числом команд в группе не более 4-х. Она называется *командой начала блока условного выполнения* и генерируется «интеллектуальным» транслятором автоматически, без какого-либо участия программиста. Команда IT имеет следующий синтаксис:



Команда введена в систему команд процессоров Cortex-M3/M4 для ускорения процесса выполнения групп команд с суффиксами условного выполнения.

К мнемокоду операции IT может быть добавлено до 3-х суффиксов, каждый из которых специфицирует условие выполнения всех команд блока, кроме первой. Для первой команды код условного выполнения «cond» указывается в качестве операнда команды IT. Особенности:

1) Любой блок условного выполнения может содержать команды условного выполнения *исключительно с прямым и оппозитным* (противоположным) условием, например, MI и PL (как в нашем случае), EQ и NE, GT и LE и т.д.
2) По коду условного выполнения в первой команде блока транслятор определяет основной код условного выполнения блока – это будет операнд команды IT.
3) Для всех последующих команд блока в виде *дополнительного суффикса* к мнемокоду IT транслятор генерирует признак соответствия основному коду (в первой команде), как бы *переключатель кода условия*: T – тот же код, что и для первой команды; E – оппозитный код. Число таких переключателей на 1 меньше числа команд в блоке.
4) Если конкретный алгоритм предполагает использование команд условного выполнения с *непарными условиями*, то для каждого из таких условий генерируется свой *собственный блок* условного выполнения.
5) Если среди команд блока условного выполнения имеется команда условной передачи управления, то она должна быть последней в блоке.
6) Блоки условного выполняения являются *прерываемыми*. Аппаратная часть процессоров Cortex-M3/M4 обеспечивет корректный возврат из процедур обработки прерываний/исключений в блок условного выполнения для продолжения работы.

Команда начала блока условного выполнения является как бы «переключателем» алгоритма выполнения команд блока – по прямому или оппозитному условию. Программист может не заботиться о ее создании – она генерируется автоматически. Но желательно разрабатывать алгоритм с последовательно расположенными командами условного выполнения (до 4-х команд), в которых будут использоваться *одни и те же прямые и оппозитные условия*. При этом минимизируется число необходимых команд IT, генерируемых транслятором.

1) Какой мнемокод команды блока условного выполнения сгенерирует транслятор, если в блоке только одна команда с условием NE?
2) Правильный ли код команды блока условного выполнения сгенерировал транслятор в нашем примере с библиотекой битовых операций?
3) В блоке 4 команды, из которых первая и вторая должны быть выполнены по условию EQ, а две последние – по условию NE. Какой мнемокод начала блока сгенерирует транслятор?
4) Может ли в блок условного выполнения входить команда, не имеющая в мнемокоде суффикса условного выполнения?
5) Прямое и оппозитное условие проверяются по одной и той же комбинации флагов?

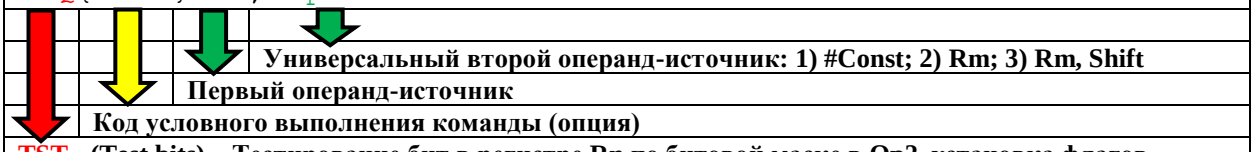
1) IT NE.
2) Генерируется правильный код ITE MI. Первая команда блока выполняется по условию MI, а вторая – по оппозитному условию PL (суффикс E- Else).
3) ITTEE EQ.

4) Нет. Все команды должны содержать суффикс условного выполнения.
5) Да. Например, в нашем примере проверяется только флаг N отрицательного результата.

12.5 Команды тестирования

В системе команд процессоров Cortex-M3/M4 имеются две специальные команды тестирования бит и битовых полей (TST), а также двух переменных на равенство со следующим синтаксисом:

```
TST{cond} Rn, Operand2
TEQ{cond} Rn, Operand2
```



TST - (Test bits) – Тестирование бит в регистре Rn по битовой маске в Op2, установка флагов
TEQ - (Test Equivalence) – Тестирование на эквивалентность переменных, установка флагов

Первая команда TST подобна операции «Логического И» двух операндов-источников:

```
AND{S}{cond} {Rd}, Rn, Op2
```

Однако, в отличие от нее, она не сохраняет результат операции (имя регистра-назначения отсутствует), а *только устанавливает флаги* результата операции N, Z, C (кроме V), которые дальше можно использовать для ветвления программы или условного выполнения последующих команд. Заметьте, что аналогичная команда AND требует дополнительной опции «S» для установки флагов.

Основное назначение команды TST – тестирование любого заданного бита в регистре-источнике Rn:

```
N     EQU 8                ; Определение номера бита
      TST r1, #(1<<N)      ; Тестирование бита с номером N
                           ; в регистре r1
      BNE Bit_Set          ; Переход на метку Bit_Set
                           ; если бит установлен
Bit_Clear                  ; Бит сброшен
;     …
```

Если номер бита заранее определен, как в примере выше, то в качестве непосредственного операнда целесообразно использовать выражение, которое будет автоматически рассчитано транслятором. Напомним, что символ «<<» является операцией логического сдвига операнда влево на заданное число разрядов, выполняемой на стадии ассемблерирования программы.

В нашем примере операнд равен 1. Следовательно, будет выполняться операция логического умножения на маску, в которой «1» находится в разряде с нужным номером бита, а все остальные биты маски – нулевые. Если тестируемый бит окажется нулевым, то флаг Z установится (код истинности условия EQ), в противном случае – сбросится (код истинности условия NE). Выполняйте ветвления в программе или условные команды по этим кодам:

Тестируемый бит	Код условного выполнения
0	EQ
1	NE

Вторая команда TEQ подобна операции «Исключающее ИЛИ» двух операндов источников:

```
EOR{S}{cond} {Rd}, Rn, Op2
```

Она отличатся тем, что не сохраняет результат операции, а только выставляет флаги. Как мы знаем, результат операции «Исключающее ИЛИ» будет нулевым только в том случае, *когда все биты обоих операндов будут одинаковыми*. Именно поэтому команда TEQ называется «тестированием на эквивалентность» любых двух переменных.

1) В чем отличие команд тестирования от команд сравнения CMP, которые тоже не сохраняют результат операции, а лишь выставляют флаги?
2) Как использовать команду TST для проверки «ненулевого» младшего байта регистра r1?
3) Как использовать команду TEQ для проверки одинаковости знаков двух исходных операндов-слов в дополнительном коде?

1) Главное отличие в том, что логические команды, к которым можно отнести команды тестирования (TST – «Логическое И», TEQ – Логическое «Исключающее ИЛИ»), вырабатывают только флаги Z, N, C. Флаг V (переполнения при работе с числами со знаком) не вырабатывается. Да и флаг переноса C может модифицироваться только в том случае, когда используется второй гибкий операнд с опцией «попутного» сдвига. Поэтому сравнивать числа с помощью команд тестирования на «больше», «меньше» и т.д. нельзя! Можно – только на полную эквивалентность!
2) Командой TST r1, #0xFF. Если младший байт будет содержать хотя бы один ненулевой бит – флаг Z сбросится (условие NE).
3) Для этого, после выполнения команды TEQ нужно проверять не условия (EQ или NE), а условия (MI или PL). Только в том случае, когда знаки тестируемых слов будут одинаковы, старший бит результата станет нулевым, а условие PL – истинным.

12.6 Реализация логических контроллеров методом тестирования битовых переменных

Рассмотрим пример логической функции, подлежащей реализации:

$$y_0 = x_0 + \overline{x_1} \cdot x_5 \qquad (12.3)$$

Конечно, можно использовать уже описанную выше технологию прямого вычисления функции с помощью логических команд процессора. Но, в нашем распоряжении имеются команды тестирования значений битовых переменных в любом

регистре ЦПУ, то есть тестирования компонент вектора входа X. Представим алгоритм решения задачи с помощью, так называемой, *граф-схемы алгоритма вычисления логической функции* – рис. 12.1. Нетрудно заметить, что если переменная x_0 равна «1», то значение второго выражения можно не проверять – достаточно просто установить значение выходной переменной y_0=1. Если же x_0=0, то результат будет зависеть от логического произведения $(\overline{x_1} \cdot x_5)$. Причем, если x_5=0, то дальнейшая проверка значения x_1 уже не требуется: нужно просто сбросить выходную переменную. В противном случае потребуется дополнительный анализ фактического значения переменной x_1.

Таким образом, граф-схема вычисления логической функции представляет собой, по существу, *блок-схему алгоритма решения задачи*. Для реализации такого алгоритма нужны лишь:

1) Команды тестирования входных битовых переменных (TST) в регистрах, содержащих компоненты вектора входа X;
2) Команды передачи управления по результатам тестирования (BEQ и BNE);
3) Команды установки (ORR) и сброса (AND) нужных бит в регистрах компонент вектора выхода Y.

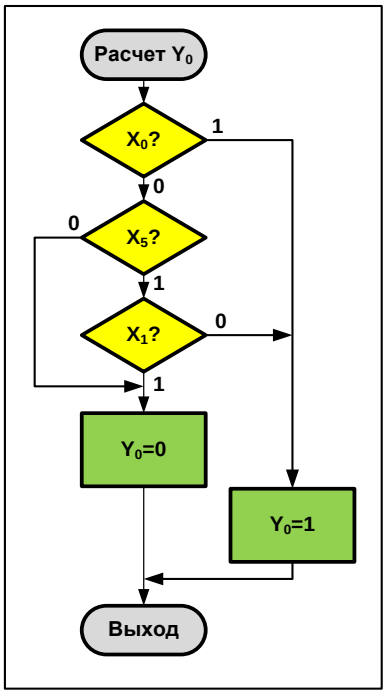


Рис. 12.1 Граф-схема вычисления логической функции Y_0

Пример решения задачи при условии, что вектор входа загружен в регистр r1, а вектор выхода формируется в регистре r2, представлен ниже:

```
; Подпрограмма расчета логической функции y0
; Тестирование x0
          TST r1, #(1<<0)
          BNE SET_Y0
; Тестирование x5
          TST r1, #(1<<5)
          BEQ CLR_Y0
; Тестирование x1
          TST r1, #(1<<1)
          BEQ SET_Y0
```

```
; Сбросить выходную переменную y0
CLR_Y0
          BIC r2, #(1<<0)
; Возврат в основную программу
          BX lr
; Установить выходную переменную y0
SET_Y0
          ORR r2, #(1<<0)
; Возврат в основную программу
          BX lr
```

С учетом того, что компоненты вектора входа и выхода могут размещаться в произвольных регистрах ЦПУ, а номера битов, подлежащих тестированию и сбросу или установке, могут быть любыми (0÷31), целесообразно разработать три макрокоманды, которые можно будет использовать при любом распределении регистров ЦПУ для любых номеров битовых переменных. Соответствующая макробиблиотека представлена в проекте CPU_ 15 (Macro_tbit.lib). Здесь, как и ранее, формальный параметр $R задает имя используемого регистра ЦПУ, а $N – номер бита, участвующего в операции. Еще раз отметим, что формальный параметр $N может использоваться в выражениях, вычисляемых на стадии ассемблирования. При макровызове он будет автоматически заменен конкретным номером бита. Как видите, единственное преимущество макрокоманды перед соответствующей командой процессора состоит в том, что можно использовать любые имена регистров и любые номера битов, что упрощает написание программ при большой размерности векторов входа и выхода.

```
; Макробиблиотека команд битового сопроцессора для
; вычисления логических функций методом тестирования бит
;*********************************************************
; Тестирование бита в регистре ЦПУ по его номеру
        MACRO
        TST_Bit $R,$N
        TST $R,#(1<<$N)
        MEND
;*********************************************************
;*********************************************************
; Установка бита в регистре ЦПУ по его номеру
        MACRO
        SET_Bit $R,$N
        ORR $R,#(1<<$N)
        MEND
;*********************************************************
;*********************************************************
; Сброс бита в регистре ЦПУ по его номеру
        MACRO
        CLR_Bit $R,$N
        BIC $R,#(1<<$N)
        MEND
;*********************************************************
```

```
34  ; Подпрограмма расчета функции y0
35  CALC_y0
36          TST_Bit r1,0
37          BNE S_y0
38          TST_Bit r1,5
39          BEQ C_y0
40          TST_Bit r1,1
41          BEQ S_y0
42  C_y0
43          CLR_Bit r2,0
44          BX lr
45  S_y0
46          SET_Bit r2,0
47          BX lr
```

```
50          ALIGN
51  ; Объявление секции данных в памят данных
52          AREA MyData, DATA, ReadWrite
53  ; Резервирование 2-х слов под
54  ; вектор входа X и вектор выхода Y
55  X       SPACE 4
56  Y       SPACE 4
```

Сохраним опробованный ранее метод разработки и отладки программ логических контроллеров с подключением макробиблиотеки и вводом вектора входа X из памяти и выводом вектора выхода Y в память, зарезервировав для них два слова в секции данных (MyProg_15.s).

Алгоритм расчета управляющего воздействия y_0 оформим в виде подпрограммы.

Будем вызывать ее из основной программы после получения текущего значения вектора входа X. После возврата из подпрограммы (расчета y_0) выведем вектор выхода в ячейку памяти Y, симулирующую порт вывода.

```
13  LOG_CONTR
14  ; Реализация логического контроллера методом
15  ; тестирования входных битовых переменных
16  ; Загрузить вектор входа X из памяти
17          LDR r5,=X
18          LDR r1,[r5]
19
20  ; Вызвать подпрограмму расчета функции y0
21          BL CALC_y0
22  ; Вызвать подпрограммы расчета других функций
23  ;       ...
24  ; Вывести управляющие воздействия в порт вывода
25          LDR r5,=Y
26          STR r2,[r5]
27  ; Повторить цикл логического контроллера
28          B LOG_CONTR
```

После завершения операции вывода цикл работы логического контроллера должен быть повторен, но уже на новом наборе входных переменных. Для каждой Булевой функции создается своя подпрограмма расчета с использованием макробиблиотеки. Все подпрограммы последовательно вызываются из основной программы логического контроллера.

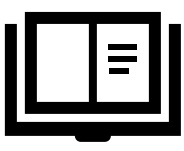
Рассмотрены два основных метода расчета любой, сколь угодно сложной системы Булевых функций: непосредственного вычисления значения функции с использование битового аккумулятора и логических операций; тестирования битовых переменных в соответствии с граф-схемой алгоритма расчета Булевой функции. Первый метод более универсален и обеспечивает фиксированное время расчета (одного скана логического контроллера), а второй может оказаться быстрее на некоторых наборах входных переменных, но из-за этого имеет «плавающее» время скана, что нужно учитывать в системах управления реального времени.

1) Изучите текст основной программы MyProg_15.s. Выполните ее трансляцию. Исследуйте файл листинга. Проследите, как макроопределения заменяются макрорасширениями.
2) Выполните отладку программы на нескольких наборах входных переменных.
3) Дополните список Булевых функций, подлежащих реализации, функциями из системы (2): y2, y7 и y8. Разработайте подпрограммы вычисления их значений методом, который посчитаете более эффективным. Выполните трансляцию, сборку и отладку проекта на нескольких наборах входных переменных.
4) Действительно ли функции y7 и y8 на всех наборах дают одинаковый результат?

2) Например, для входного вектора 0x20 Вы должны получить y0=1, а для вектора 0x22 y0=0.
4) Да, это так. Правила де Моргана работают при любых значениях битовых переменных.

12.7 Программная реализация дискретных управляющих автоматов

12.7.1 Дискретные управляющие автоматы



Принципиальное отличие *дискретного управляющего автомата* от логического контроллера состоит в том, что дискретный автомат имеет *память текущего состояния* и изменяет это состояние не только в зависимости от значений входных переменных, но и в зависимости от кода текущего состояния, сохраненного в памяти.

В общем случае, так же, как и логический контроллер, дискретный автомат принимает на обработку *вектор входных битовых переменных* **X** и вырабатывает вектор *выходных битовых управляющих воздействий* **Y.** *Вектор текущего состояния автомата* сохраняется в памяти **Q[k]** и вместе с вектором входных переменных **X** образует так называемый *вектор полного состояния автомата* **W (X, Q[k]).** Именно по вектору полного состояния автомата определяется номер очередного состояния, в которое автомат должен перейти **Q[k+1],** т.е. *состояния перехода*.

Этот переход может производиться *асинхронно*, как это происходит во всех *релейно-контакторных системах управления приводами* и технологическим оборудованием, или *синхронно*, как это делается во всех *цифровых, в том числе микропроцессорных* системах реализации дискретных управляющих автоматов. В первом случае дискретный автомат называется *асинхронным*, а во втором – *синхронным*. Сигнал синхронизации записи в аппаратных реализациях подается от специального тактового генератора, а в программных реализациях определяется интервалом между двумя последовательными вызовами автомата, называемым *интервалом квантования* (*дискретизации*) цифровой системы управления по времени.

На рис. 12.2 представлена обобщенная блок-схема аппаратной реализации дискретного управляющего автомата.

Вектор входных воздействий X поступает вместе с вектором текущего состояния автомата Q[k] на *входной логический контроллер*, в котором решается система логических уравнений **Q[k+1]=f(X, Q[k])** и определяется *код состояния перехода* – того состояния, в которое автомат перейдет с подачей очередного импульса синхронизации записи данных в память. Как только состояние автомата изменится, *выходной логический контроллер* рассчитает новое значение вектора выходных управляющих воздействий в соответствии с системой логических функций **Y=f(Q[k]).** Это воздействие будет выдано для управления элементами привода или исполнительными устройствами технологической автоматики. В соответствии с изменившимся состоянием автомата изменится и вектор полного состояния, который поступит на входной логический контроллер. Вновь произойдет оценка состояния перехода. С очередным тактовым импульсом текущее состояние автомата заменится новым состоянием перехода. Далее цикл работы автомата будет многократно повторяться.

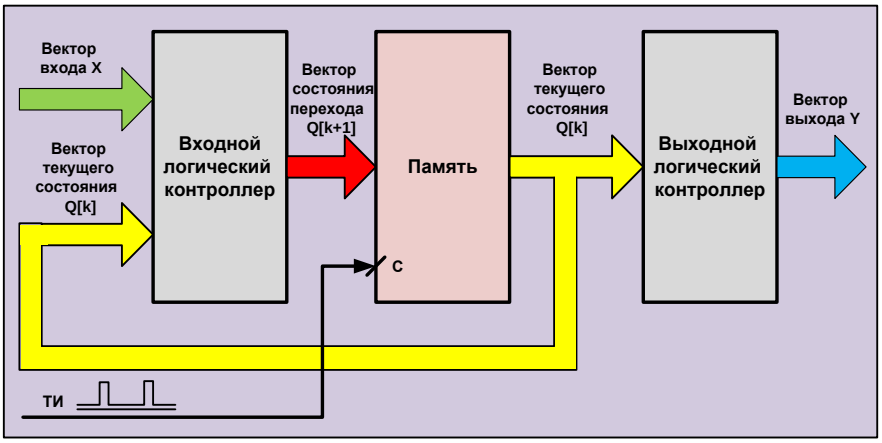


Рис. 12.2 Блок-схема дискретного управляющего автомата

Таким образом, дискретный управляющий автомат представляет собой *последовательный автомат*, который на каждом интервале дискретизации на основе оценки текущего значения вектора входа и вектора своего собственного состояния определяет состояние перехода и переходит в это новое состояние. В соответствии с текущим состоянием автомата рассчитываются и выдаются выходные управляющие

воздействия. Иногда структуру дискретного автомата удается упростить, - выходной логический контроллер может отсутствовать. При этом управление выходами осуществляется непосредственно по состоянию автомата.

Если вектор выхода формируется на основе только вектора текущего состояния автомата (как на рис. 12.2), то такой автомат называется *автоматом Мура*. В общем случае вектор входа X может поступать и на выходной логический контроллер – получим *автомат Мили*. В теории дискретных автоматов доказывается, что любой автомат Мили может быть преобразован к автомату Мура, поэтому далее рассмотрим реализацию только автоматов Мура.

Отметим, что по принципу *последовательного автомата* работают практически все *цифровые системы управления*, даже такие, которые в реальном времени решают целые системы дифференциальных уравнений. Разница состоит лишь в том, что компонентами всех векторов X, Y, Q могут быть не битовые переменные (как в дискретных управляющих автоматах), а переменные любой размерности. Такой подход соответствует *модели* объекта в так называемом *пространстве состояний*, которая все чаще применяется для моделирования динамических процессов и для управления сложным оборудованием.

12.7.2 Традиционные способы программной реализации дискретных управляющих автоматов. Недостатки и проблемы

Блок-схема автомата на рис. 12.2 предполагает, что разработчик дискретного управляющего автомата *предварительно синтезировал логические функции* как для входного, так и для выходного логического контроллера. Если это так, то задача программной реализации дискретного автомата сводится *к двум задачам реализации обычных логических контроллеров,* которые должны работать совместно.

Способы их программной реализации рассмотрены выше. Выбор конкретного способа определяется размерностью векторов входа, выхода и состояния автомата, сложностью систем логических функций.

Организация памяти текущего состояния автомата в процессорной системе никаких затруднений не вызывает – это переменная, хранящаяся в ОЗУ и обновляемая при каждом переходе автомата в новое состояние. Она должна содержать код состояния автомата, в простейшем случае просто номер состояния автомата.

Таким образом, основная сложность состоит в том, как для конкретной задачи автоматизации режимов работы устройства или технологической установки получить подлежащие реализации системы логических функций для входного и выходного логических контроллеров. Как закодировать состояния автомата, чтобы эти функции были наиболее простыми?

Если системы логических функций получены, то они могут быть минимизированы, например, с использованием законов алгебры логики или карт Карно, и реализованы.

На практике сложностей еще больше: если при синтезе этих функций Вы забыли какую-то входную переменную или если Вам потребуется изменить алгоритм работы автомата, или ввести несколько новых состояний, например, аварийных, то всю работу по синтезу систем логических функций придется выполнять заново! Разумеется, есть приемы и методы, описанные в теории дискретных управляющих автоматов, которые позволяют облегчить эту работу. Однако, это трудоемкий процесс, требующий не только времени, но и значительных материальных затрат. Такую работу обычно выполняют специализированные фирмы, занимающиеся промышленной автоматизацией.

К счастью, применительно к процессорным реализациям существует более простой и даже более эффективный способ: *прямая программная реализация дискретного автомата непосредственно по графу автомата*, который составляется в *строгом*

соответствии с технологической картой автоматизируемого процесса и является основой для написания программы автомата. Рассмотрим этот подход на простом примере системы управления режимами работы главного привода продольно-строгального станка.

12.7.3 Как составить граф дискретного автомата?

Рассмотрим методику составления графа автомата, управляющего главным приводом продольно-строгального станка как бы в диалоге технолога, знающего всю специфику объекта управления, и программиста, который получил задание создать программу управления объектом и является неплохим специалистом-электронщиком.

Программист: Что представляет собой продольно-строгальный станок?

Технолог: Станок, предназначенный для выполнения возвратно-поступательных перемещений резца, обрабатывающего плоские поверхности деталей. Основной привод должен перемещать резец в направлении «Вперед» (В) до конца детали, затем автоматически изменять направление движения на обратное «Назад» (Н) и после достижения начального положения детали, вновь начинать движение «Вперед».

Программист: Каким образом в начале каждого цикла движения резца «Вперед» осуществляется продольное смещение резца, чтобы обрабатывать каждый раз новую часть поверхности детали?

Технолог: Для этого станок оборудован вспомогательным приводом подачи, автоматизация которого пока не входит в нашу задачу.

Программист: Должен ли резец при движении назад автоматически подниматься?

Технолог: Да, конечно, но эту функцию также выполняет привод подачи.

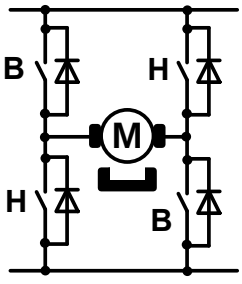
Программист: Какие датчики установлены на станке для контроля текущего положения режущей головки?

Технолог: Таких датчиков четыре. В допустимых конечных положениях режущей головки установлены датчики конечного положения: SQ1 – конечный выключатель при движении «Вперед» и SQ2 – конечный выключатель при движении «Назад».

Дополнительно предусмотрены аварийные датчики конечных положений SQ3 и SQ4, подающие команды в экстренных ситуациях – при выходе резца за пределы допустимой рабочей зоны. В разных модификациях станков установлены либо контактные датчики нажимного типа, либо бесконтактные (индуктивные, емкостные или на эффекте Холла). В любом случае схема предварительной обработки сигналов датчиков формирует при срабатывании соответствующего конечного выключателя сигнал логической «1», который должна принимать процессорная система управления.

Программист: Какой привод реализует главное движение? Не может ли получиться так, что при подаче на него команды «Останов» режущая головка пройдет еще некоторый путь и датчик конечного положения, например, SQ1, снимет сигнал срабатывания?

Технолог: Силовая часть привода главного движения показана ниже. В ней используется коллекторный двигатель постоянного тока с магнитоэлектрическим возбуждением, управляемый от реверсивного транзисторного силового моста. Силовые ключи показаны условно в виде контакторов В и Н и обратных диодов. Силовой преобразователь обеспечивает включение двигателя в любом из двух направлений (В или Н) и отключение двигателя от сети (Стоп). Скорость движения резца не регулируется. Так как привод редукторный, то выбег при останове двигателя



(когда оба ключа В и Н выключены) невелик, и датчики конечных положений, если они сработали, остаются во включенном состоянии.

Программист: Какие органы управления приводом главного движения имеются на станке?

Технолог: Кнопки «Пуск» и «Стоп».

Программист: Таким образом, вектор входных битовых переменных, поступающих на обработку в МПС, состоит из шести битовых компонент:

Компоненты вектора входа X					
X5	**X4**	**X3**	**X2**	**X1**	**X0**
«Стоп»	**«SQ4»**	**«SQ3»**	**«SQ2»**	**«SQ1»**	**«Пуск»**

Кроме команд управления силовыми ключами В и Н нужно ли еще чем-то управлять?

Технолог: Конечно. Если по каким-либо причинам режущая головка выйдет из допустимой зоны и сработает один из аварийных датчиков положения SQ3 или SQ4, то двигатель должен немедленно отключиться и должна сработать аварийная сигнализация (световая и звуковая).

Программист: Значит, выходных управляющих воздействий должно быть три: включения («1») /выключения («0») силовых ключей В или Н и включения/выключения аварийной сигнализации:

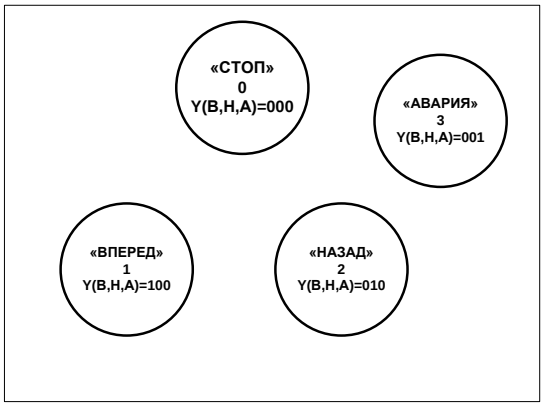
Компоненты вектора выхода Y		
Y2	**Y1**	**Y0**
«В»	**«Н»**	**«А»**

Дискретный управляющий автомат будет иметь 4 возможных состояния: «**СТОП**» – двигатель отключен; «**ВПЕРЕД**» – двигатель включен в направлении В; «**НАЗАД**» – двигатель включен в направлении Н; «**АВАРИЯ**» – двигатель отключен, включена аварийная сигнализация. Предлагаем последовательно пронумеровать все возможные состояния автомата и для каждого из них указать состояние вектора выхода, ему соответствующее:

Таблица 12.5 Состояния дискретного автомата

Состояние автомата	**Номер состояния (код)**	**Вектор выхода Y, соответствующий текущему состоянию, по-компонентно**		
		В	**Н**	**А**
«СТОП»	**0**	**0**	**0**	**0**
«ВПЕРЕД»	**1**	**1**	**0**	**0**
«НАЗАД»	**2**	**0**	**1**	**0**
«АВАРИЯ»	**3**	**0**	**0**	**1**

Авторы: Эта таблица (см. табл. 12.5) описывает функцию, подлежащую реализации в выходном логическом контроллере, то есть зависимость вектора выхода Y от текущего состояния автомата Q[k].



В *теории графов* каждое состояние автомата изображается кружком, внутри которого указывается имя текущего состояния, его код (возможно номер), а также состояние вектора выхода, соответствующее данному состоянию. Автомат в каждый момент времени может находиться *только в одном из возможных состояний*. Итак, если мы знаем код текущего состояния автомата, то знаем и вектор выхода Y, ему соответствующий. Следовательно, нет вообще никакой необходимости в расчете вектора выхода – он *может быть просто выдан по номеру состояния* в соответствующий порт вывода.

Кодирование состояний выполняется произвольным образом, однако рекомендуется *начинать с нулевого состояния*, которое должно соответствовать начальному состоянию автомата при включении питания и сбросе процессора. Это должно быть безопасное состояние, при котором все оборудование объекта выключено (в

нашем случае – «СТОП»). Все остальные состояния желательно *пронумеровать последовательно без пропусков номеров*. В дальнейшем, при программной реализации графа автомата, это позволит создать предельно короткий и компактный код.

Программист: В качестве начального состояния автомата мы выбрали «СТОП». В какое состояние должен перейти автомат, если оператор нажал кнопку «Пуск», – в состояние «ВПЕРЕД» или в состояние «НАЗАД»?

Технолог: Да, здесь и я вижу некоторую неопределенность. Давайте исключим ее и будем считать, что при нажатии кнопки «Пуск» движение всегда начинается «Вперед», в противном случае потребуется модификация аппаратной части станка: замена кнопочной панели с двумя кнопками («Пуск» и «Стоп») кнопочной панелью с тремя кнопками («Вперед», «Назад», «Стоп»), что нежелательно.

Программист: Как должна вести себя система управления, если по какой-то причине одновременно нажаты кнопки «Пуск» и «Стоп»?

Технолог: Команда «Пуск» должна быть заблокирована! Ведь нажатие кнопки «Стоп» может быть попыткой предотвращения аварии: пуска станка без установленной детали, попадания в рабочую зону посторонних предметов и т.д. Кнопка «Стоп» должна иметь приоритет перед кнопкой «Пуск».

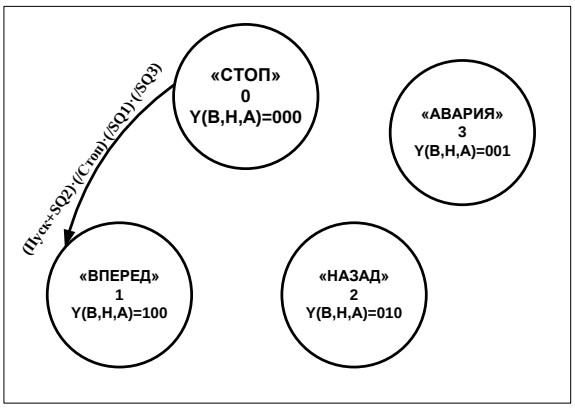
Программист: Как быть, если датчики, контролирующие достижение конечного положения при движении «Вперед» SQ1 и SQ3 неисправны, например, их контакты «приварились» и датчики не могут сработать?

Технолог: В этом случае движение «Вперед» также должно быть заблокировано. Ведь команда на смену направления движения или аварийный останов может не поступить!

Программист: Сформулируем условие включения привода в направлении «Вперед»: Нажата кнопка «Пуск», но не нажата кнопка «Стоп» и не сработали датчики конечного и аварийного положения при движении вперед **Пуск·(/Стоп)·(/SQ1)·(/SQ3).**

Технолог: Согласен. Но где же автоматическое включение в направлении «Вперед», если мы двигались в направлении «Назад» и сработал конечный включатель SQ2? Ведь наша цель автоматизировать возвратно-поступательные движения резца.

Программист: Уточним условие пуска привода в направлении «Вперед»: **(Пуск + SQ2)·(/Стоп)·(/SQ1)·(/SQ3).** Теперь на графе автомата можно нарисовать первую стрелку, которая будет отображать условие перехода дискретного автомата из состояния «Стоп» в состояние «Вперед».

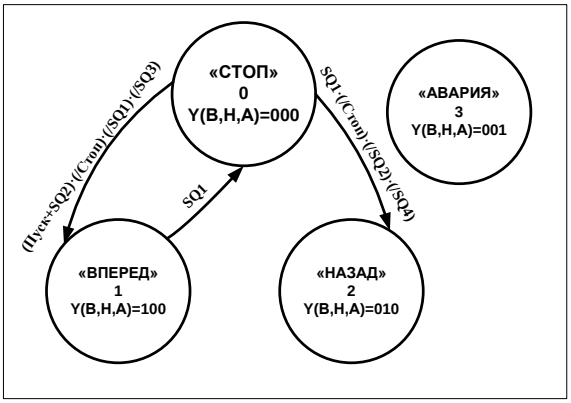


Технолог: Как только при движении «Вперед» мы достигнем конечного положения, и сработает датчик SQ1, мы должны немедленно реверсировать привод, включив его в направлении «Назад».

Программист: Это так, но лишь теоретически! Обратите внимание на схему силовой части привода. Если сразу после команды на отключение силового ключа «В» поступит команда на включение силового ключа «Н», а ключ «В» несколько «запоздает» с отключением, может возникнуть аварийная ситуация: оба ключа окажутся на некоторое время включенными одновременно – это сквозное короткое замыкание в силовой части привода, что недопустимо! Давайте поступим так. При достижении конечного положения (срабатывании датчика SQ1) переведем привод кратковременно в положение «Стоп», и только затем, уже из безопасного состояния, сделаем реверс.

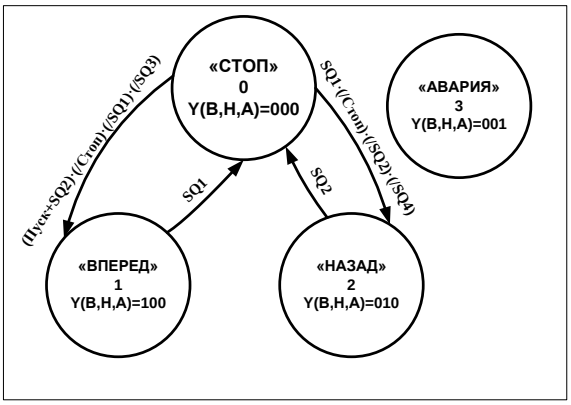
Технолог: Принимается. Пусть при достижении конечного положения SQ1 двигатель сначала отключается и только затем включается в противоположном направлении.

Программист: Итак, условие отключения при движении «Вперед»: **SQ1**. Условие запуска в направлении «Назад» из положения «Стоп»: **SQ1·(/Стоп)·(/SQ2)·(/SQ4)**.

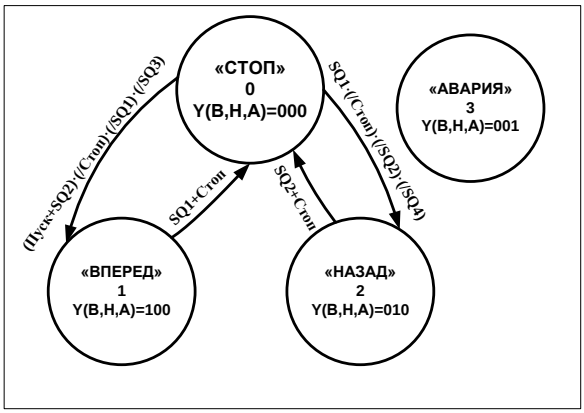


Технолог: Согласен. Мы таким образом реализуем и защиту от возможной неисправности датчиков конечного положения SQ2 и SQ4, контролирующих движение в направлении «Назад».

Программист: При движении «Назад» нужно поступить аналогично. При достижении резцом конечного рабочего положения (срабатывании датчика SQ2) сначала перевести привод в состояние «Стоп» и только из этого состояния реверсировать.



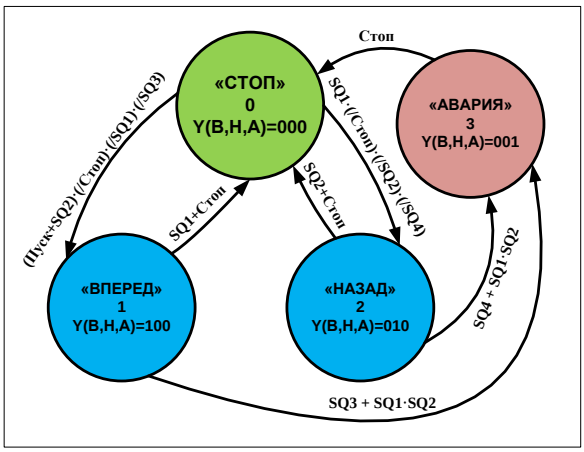
Технолог: Конечно, причем команду на реверс мы уже предусмотрели раньше, когда задумались об автоматизации процесса возвратно-поступательных движений резца (по срабатыванию датчика SQ2). Добавим на графе автомата стрелку с условием перехода из состояния «НАЗАД» в «СТОП».



Мы еще не решили, как будем реагировать на нажатие кнопки «Стоп». Предлагаю, при любом направлении движения – останавливать привод.

Программист: Согласен. В этом случае условия перехода из состояний «ВПЕРЕД» и «НАЗАД» в состояние «СТОП» нужно уточнить: **SQ1+Стоп** и **SQ2+Стоп**.

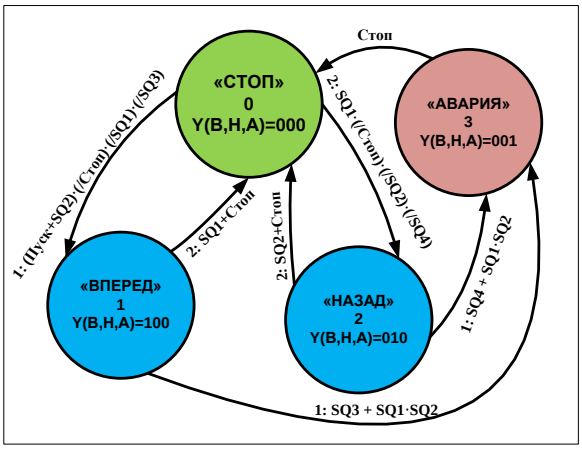
Технолог: Предлагаю контролировать аварийные ситуации по срабатыванию аварийных датчиков SQ3 (при движении Вперед) и SQ4 (при движении Назад). Отключать привод и включать аварийную сигнализацию.



Программист: Этого недостаточно. Что, если оба датчика конечного положения SQ1 и SQ2 оказались неисправны? При этом организовать цикл возвратно-поступательного движения в пределах рабочей зоны резца невозможно. Будем считать одновременное срабатывание этих датчиков **SQ1·SQ2** предпосылкой к аварии и останавливать привод.

Технолог: Предлагаю возвращать привод в состояние «СТОП» путем нажатия кнопки «Стоп». При этом световая и звуковая сигнализация будут отключены и, если неисправность устранена, можно вновь приступить к работе.

Программист: Из каждого состояния автомата, кроме аварийного, имеются как минимум два возможных перехода. Нужно оценить приоритеты этих переходов. Предлагаю самыми приоритетными сделать переходы в аварийное состояние для ускорения реакции системы управления на возможные неисправности, для предупреждения развития аварийных ситуаций. В окончательном, разработанном нами графе управляющего автомата, пометим стрелки переходов цифрами. Чем меньше номер на стрелке, тем приоритетнее переход и тем раньше его возможность должна проверяться в системе управления.



Граф дискретного автомата представляет собой совокупность вершин, отражающих возможные состояния автомата и стрелок переходов между вершинами, помеченных условиями переходов. Если переходов несколько, то для каждого из них назначается уровень приоритета. Чем выше уровень приоритета, тем раньше соответствующее условие перехода должно проверяться. Если условие перехода сработало, то текущий номер состояния заменяется состоянием перехода и все последующие, менее приоритетные условия переходов, не проверяются.

12.7.4 Программная реализация дискретного автомата

12.7.4.1 Понятие интерпретатора вершины графа

Введем понятие *интерпретатора вершины графа*, то есть программы, которая будет реализовывать все функции управления, предусмотренные в состоянии автомата с номером N[k]:

1) Вывод соответствующего текущему состоянию автомата управляющего воздействия. Эта операция может быть попутной при смене номера состояния автомата на состояние перехода.
2) Проверку истинности условий перехода из текущей вершины графа в другие вершины в соответствии с установленным для этой вершины приоритетом переходов.
3) Смену текущего состояния автомата N[k]=N[k+1], если одно из возможных условий перехода (первое в приоритетном списке) «сработает», то есть станет истинным. При этом проверка выполнения остальных условий прекращается, и обеспечивается досрочный выход из подпрограммы интерпретатора вершины. Смена номера состояния может сопровождаться выводом соответствующего ему управляющего воздействия Y.

Программа реализации дискретного управляющего автомата будет содержать несколько подпрограмм интерпретаторов вершин графа автомата, по числу его вершин. В любой момент времени автомат может находиться только в одном из возможных состояний, номер которого известен N[k]. Именно по этому номеру и должна быть вызвана соответствующая подпрограмма интерпретатора вершины графа. Если ни одно из условий перехода в другие состояния не сработает, то номер состояния автомата не

меняется. В противном случае текущее состояние заменяется состоянием перехода N[k]=N[k+1].

Программа дискретного автомата вызывается каждый раз *на периоде дискретизации системы управления по времени*. Естественно, что в ее начале должен быть считан номер текущего состояния автомата N[k] и текущий вектор входных переменных X. Дальше, по номеру текущего состояния автомата N[k] должен быть вызван интерпретатор соответствующей вершины графа. Он может как сохранить номер текущей вершины, так и заменить его на номер состояния перехода (вместе с выдачей нового вектора управляющих воздействий).

Перед первым запуском программы дискретного управляющего автомата должно быть обязательно установлено его начальное состояние вместе с соответствующим ему вектором выхода.

12.7.4.2 Технология выдачи управляющих воздействий

Отметим еще раз, что преимущество рассматриваемого метода состоит в том, что выходное управляющее воздействие не рассчитывается, а просто выдается вовне. Для нашего автомата справедлива таблица соответствия выдаваемого кода вектора Y номеру состояния (см. табл. 12.6).

Таблица 12.6 Таблица соответствия

Номер состояния N	**Вектор выхода Y (B, H, A)**	**Код**
0	**000**	**0**
1	**100**	**4**
2	**010**	**2**
3	**001**	**1**

Выдача управляющего воздействия может выполняться по-разному:

1) В начале подпрограммы интерпретатора вершины графа.
2) Одновременно со сменой номера состояния на номер состояния перехода.

Первый способ обеспечивает повторную выдачу того же вектора управляющих воздействий, если состояние автомата не меняется. Это может быть полезно с точки зрения повышения надежности системы управления. Однако работа автомата становится как бы двухступенчатой: во время первого скана номер состояния меняется на состояние перехода; во время второго скана для этого нового состояния выдаются управляющие воздействия. Появляется задержка выдачи управляющих воздействий на один такт автомата, что нежелательно.

Второй метод более предпочтителен по следующим причинам:

1) Время реакции автомата на изменение входных переменных уменьшается вдвое.
2) Только при смене текущего номера состояния на состояние перехода (один раз) выставляется новое значение вектора выхода, которое нет необходимости выдавать заново при каждом повторном входе в текущую вершину графа.
3) Обычно выводимая в порт вывода Y информация фиксируется в нем надежно и повторного вывода данных не требуется.

Целесообразно *совместить процедуру* смены номера состояния автомата на состояние перехода с выдачей для него нового вектора управляющих воздействий. При инициализации автомата вместе с начальным номером нужно также выдавать и соответствующие ему управляющие воздействия. Такой подход оказывается более эффективным и в том случае, если в качестве выходных воздействий требуется выполнение более сложных функций, чем выдача дискретного сигнала в порт вывода, например, при

необходимости передачи в локальные контроллеры распределенной системы управления специальных команд по интерфейсам, связывающим локальные контроллеры с контроллером-мастером, в котором реализуется дискретный автомат управления режимами работы мультипроцессорной системы.

Вернемся к нашему простому примеру. Автомат управления станком будет постоянно работать с тремя переменными в памяти системы управления: 1) вектором входных битовых воздействий (порт X); 2) номером текущего состояния автомата N; 3) вектором выходных управляющих воздействий (порт Y).

```
158  ; Выравнивние адресов переменных в памяти данных
159          ALIGN
160  ; Объявление секции данных в ОЗУ
161          AREA MyData, DATA, ReadWrite
162  ; Резервирование слов под
163  ; вектор входа X, вектор выхода Y
164  ; и номер текущего состояния автомата N
165          EXPORT X
166          EXPORT N
167          EXPORT Y
168  X       SPACE 4
169  N       SPACE 4
170  Y       SPACE 4
```

Зарезервируем для этих переменных по одному слову в оперативной памяти с использованием директивы Ассемблера SPACE. Объявим эти переменные глобальными (директива EXPORT), чтобы они попали в таблицу символов проекта и могли быть включены при отладке в список наблюдаемых переменных в окнах наблюдаемых переменных Watch.

В реальности адреса портов будут определяться архитектурой используемого микроконтроллера. Так как мы будем выполнять отладку в среде µVision в симуляторе, расположим эти переменные последовательно, чтобы можно было сразу считывать из памяти по два слова (X и N) и сохранять в памяти по два слова (N и Y). Это упростит написание программы автомата и ее отладку. Ниже в качестве примера показано начало основной программы проекта CPU_16 – MyProg_16.s. Так как состоянию автомата N=0 соответствует значение вектора выхода Y=0, то одной командой сохранения в памяти содержимого сразу двух регистров ЦПУ можно проинициализировать обе переменные в памяти N и Y. Заметьте, что в качестве регистров-источников данных Rt, Rt2 используется один и тот же регистр r1 с содержимым #0.

```
 7  MyProg
 8  ; Программная реализация дискретного управляющего
 9  ; автомата (главный привод продольно-строгального станка)
10  ; Инициализация
11  ; Установить начальное состояние автомата N=0
12  ; Вывести управляющее воздействие Y = 0
13          MOV r1,#0
14          LDR r0,=N
15          STRD r1, r1,[r0]  ; Сохранить сразу два слова N,Y
```

Напомним, что суффикс «D» в мнемонике команды STRD означает, что она выполняет сохранение в памяти сразу двух слов по косвенному адресу в регистре-указателе r0. Инкрементирование указателя на +4 при переходе к сохранению второго слова выполняется автоматически.

12.7.4.3 Технология проверки условий перехода и смены номера состояния

Таблица 12.7 Условия перехода между состояниями дискретного автомата

Текущее состояние N[k]	Условия перехода	Состояние перехода N[k+1]
0	1: (x0+x2)·(/x5) ·(/x1) ·(/x3)	1
	2: x1·(/x5) ·(/x2) ·(/x4)	2
1	1: x3+x1·x2	3
	2: x1+x5	0
2	1: x4+x1·x2	3
	2: x2+x5	0
3	x5	0

Для каждого состояния автомата на графе автомата в порядке приоритета указаны условия перехода, то есть логические функции, истинность которых вызывает переход (см. табл. 12.7).

В соответствии с принятым распределением входных битовых переменных в порту ввода X и в регистре ЦПУ – образе вектора входа, запишем условия переходов, которые должны проверяться для каждого текущего состояния автомата. Обратите внимание, что в нашем примере номера битовых входов x1, x2, x3, x4 строго соответствуют номерам датчиков конечного положения SQ1, SQ2, SQ3, SQ4. Очевидно, датчики должны быть подключены к одноименным битам порта ввода X. Такое соответствие облегчает программирование и последующее сопровождение программы.

Если вектор входа X считан и находится в одном из регистров ЦПУ (например, в r1), то расчет значения конкретной логической функции может быть выполнен одним из уже рассмотренных нами способов: *прямым вычислением с использованием логических команд процессора* или *вычислением с использованием метода тестирования битовых переменных*. В подпрограмме-интерпретаторе 0-й вершины применим первый метод.

```
40  ; Подпрограмма интерпретатора вершины 0 графа
41  INT_0
42
43  ; Проверить все условия перехода в порядке приоритета
44  ; Компоненты вектора X сохранены в регистре r1
45  ; Условие 1: (x0+x2)·(/x5) ·(/x1) ·(/x3)
46          LSL r0,r1,#(31-0)    ; Загрузка x0 в битовый аккумулятор
47          ORR r0,r1,LSL #(31-2)    ;(x0+x2)
48          BIC r0,r1,LSL #(31-5)    ; ·(/x5)
49          BIC r0,r1,LSL #(31-1)    ; ·(/x1)
50          BICS r0,r1,LSL #(31-3)   ; ·(/x3) и выставить флаги
```

Сначала одна из битовых переменных загружается в битовый аккумулятор (31-й бит регистра r0) с использованием операции логического сдвига влево, а затем последовательно выполняются нужные логические операции битового аккумулятора с битами в регистре r1 (образе вектора X), попутно сдвинутыми влево в положение 31-го разряда.

Последней логической операцией *обязательно устанавливаются флаги*.

```
51  ; Если условие истинно, сменить номер состояния N=1
52  ; и выдать управляющее воздействие Y=4
53          LDRMI r0,=N
54          MOVMI r3,#1
55          MOVMI r4,#4
56          STRDMI r3,r4,[r0] ; Сохранить сразу два слова N,Y
57  ; Досрочный выход из подпрограммы INT_0
58          BXMI lr         ; Остальные условия не проверять!
```

Напомним, что 31-й бит – это бит знака числа в дополнительном коде. Поэтому, истинности проверяемого логического выражения будет соответствовать условие MI – «Минус». По этому условию будем изменять текущий номер состояния автомата на состояние перехода и одновременно выдавать соответствующее ему управляющее воздействие. Регистр r0 выполняет функцию регистра-указателя и инициализируется начальным адресом блока переменных (N, Y) в памяти. Регистры r3, r4 будут содержать непосредственные операнды, которые должны быть записаны в эти переменные. Для записи используем команду двойного сохранения содержимого регистров r3, r4. Так как условие перехода в новое состояние выполнено, то делается досрочный выход из подпрограммы и другие условия переходов *не проверяются*.

Все эти команды являются *условно выполняемыми* по условию «MI» и автоматически включаются транслятором в блоки IT, содержащие не более 4-х команд. В нашем случае потребуется два таких блока.

```
60  ; Условие 2: x1·(/x5) ·(/x2) ·(/x4)
61          LSL r0,r1,#(31-1)    ; Загрузка x1 в битовый аккумулятор
62          BIC r0,r1,LSL #(31-5)    ; ·(/x5)
63          BIC r0,r1,LSL #(31-2)    ; ·(/x2)
64          BICS r0,r1,LSL #(31-4)   ; ·(/x4). Выставить флаги
```

Если первое условие перехода «ложное», то все команды блоков условного выполнения рассматриваются как «пустые» операции NOP, и мы автоматически переходим к проверке второго возможного условия для 0-й вершины графа. Технология вычисления значения логической функции сохраняется.

```
65  ; Если условие истинно, сменить номер состояния N=2
66  ; и выдать управляющее воздействие Y=2
67          LDRMI r0,=N
68          MOVMI r3,#2
69          STRDMI r3, r3,[r0]  ; Сохранить сразу два слова N,Y
70
71  ; Больше условий перехода нет, завершить подпрограмму
72          BX lr
```

Смена номера состояния на 2 и вывод такого же управляющего воздействия опять выполняется группой команд условного выполнения, но теперь их только три. Так как других условий перехода в текущем состоянии больше нет, делается безусловный возврат из подпрограммы.

Проверка условия перехода выполняется путем вычисления значения логической функции перехода. При ее истинности устанавливается новый номер состояния перехода, выводятся соответствующие ему управляющие воздействия, и выполняется досрочный выход из подпрограммы интерпретатора вершины. Для этой цели удобно использовать команды условного выполнения, которые будут автоматически оформлены транслятором в блоки условного выполнения.

12.7.4.4 Вызов интерпретатора вершины по таблице начальных адресов подпрограмм

Программа дискретного автомата должна быть устроена так, чтобы на каждом периоде дискретизации вызывался интерпретатор текущей вершины графа по номеру вершины N[k].

Этой операции соответствует широкоизвестная в языках высокого уровня *команда переключателя*, например, в языке СИ:

```
switch (номер состояния автомата)
{
      case 0: оператор_0
      case 1: оператор_1
      case 2: оператор_2
      …
}
```

В Ассемблере оператору-переключателю соответствует функция *косвенного запуска подпрограммы по ее начальному адресу* BLX Rm, считанному из таблицы начальных адресов подпрограмм и загруженному в регистр Rm.

```
148 ; Выравнивание кодовой памяти по границе полного слова
149 ; для размещения начальных адресов подпрограмм
150         ALIGN
151 ; Таблица начальных адресов интерпретаторов вершин графа
152 TAB_ADDR
153         DCD INT_0
154         DCD INT_1
155         DCD INT_2
156         DCD INT_3
```

Предположим, что все подпрограммы-интерпретаторы вершин графа написаны и их начальные адреса известны. Создадим в кодовой памяти таблицу точек входа в эти подпрограммы. Разместим ее, как обычно, в самом конце текущей кодовой секции, предварив директивой выравнивания по границе слова.

Используем для размещения начальных адресов директиву Ассемблера DCD с операндом – символическим именем начала соответствующего интерпретатора. Оно будет автоматически заменено фактическим адресом размещения подпрограммы в памяти в процессе трансляции и сборки проекта.

Вспомним (см. 10.7.4), что для доступа к данным в памяти можно пользоваться *базово-индексной адресацией*, когда начальный адрес таблицы размещается в базовом регистре (например, в r0), а индекс – в индексном регистре (например, r2). В этом случае

эффективный адрес доступа к памяти определяется как сумма содержимого базового и индексного регистров. Система команд ARM-процессоров позволяет предварительно промасштабировать содержимое индексного регистра в зависимости от формата расположенных в памяти данных. Так как в нашем случае в памяти расположены 32-разрядные слова (адреса подпрограмм), то для масштабирования можно применить *попутный логический сдвиг влево* индексного регистра на два разряда LSL #2. Последовательность действий:

1) Загрузить индексный регистр (r2) номером текущей вершины графа;
2) Загрузить базовый регистр (r0) начальным адресом таблицы точек входа в интерпретаторы вершин;
3) Считать (в r3) адрес интерпретатора вершины по эффективному адресу TAB_ADDR+4·N[k] с использованием команды LDR r3, [r0, r2, LSL #2];
4) Выполнить косвенную передачу управления по адресу, расположенному в регистре r3, с одновременным сохранением адреса возврата в регистре связи lr: BLX r3.

Программа дискретного управляющего автомата должна начинаться с опроса текущего вектора входных битовых переменных X и сохранения его образа в одном из регистров ЦПУ (в r1). Совместим эту операцию со считыванием номера текущего состояния автомата с сохранением его в регистре r2 с помощью команды двойной загрузки слов (X, N). Теперь можно обратиться к таблице начальных адресов подпрограмм и выполнить косвенную передачу управления на нужную подпрограмму интерпретатора вершины автомата.

```
17  ; Основная программа дискретного управляющего автомата
18  ; Вызывается с частотой дискретизации системы управления
19  DISK_AVT
20  ; Получить текущий вектор входных битовых переменных X
21  ; и сохранить в регистре r1
22  ; Получить текущий номер состояния автомата и
23  ; сохранить в регистре r2
24          LDR r0,=X
25          LDRD r1,r2,[r0]    ; Получить сразу два слова X,N
26
27  ; Загрузить в регистр r0 адрес таблицы точек входа
28  ; в подпрограммы – интерпретаторы вершин графа
29          ADR r0, TAB_ADDR
30  ; Считать из таблицы адрес интерпретатора, номер которого
31  ; уже загружен в регистр r2
32  ; Попутный авто-расчет смещения по таблице offset=(r2)*4
33          LDR r3,[r0, r2, LSL #2]
34  ; Косвенный вызов подпрограммы по ее начальному адресу
35          BLX r3
36  ;       ...
37  ; Повторить цикл дискретного управляющего автомата
38          B DISK_AVT
```

В основной программе могут решаться и другие задачи. В конце основной программы управление снова передается дискретному автомату. Как правило, дискретный автомат вызывается в процедуре обслуживания прерывания по таймеру, задающему частоту дискретизации системы управления по времени.

ARM-процессоры поддерживают механизм косвенной передачи управления в подпрограммы, начальные адреса которых предварительно записаны в таблицу точек входа в подпрограммы, расположенную в кодовой памяти. Это универсальный механизм, который можно использовать для запуска любых подпрограмм, номера которых задаются программистом, вводятся извне оператором или передаются в систему управления по любому из интерфейсов.

1) Разработайте подпрограммы интерпретаторов вершин 1, 2 и 3 графа автомата управления приводом продольно-строгального станка.
2) Интегрируйте их в основную программу. Выполните трансляцию и сборку проекта. Загрузите проект на отладку в симулятор. Установите точку останова на метке DISK_AVT. Определите последовательность значений вектора входа X, при которой автомат последовательно выполнит полный

цикл возвратно поступательного движения (B, H), затем перейдет в состояние (A-Авария), а после него – в исходное состояние «Стоп». Вводите в окне памяти очередной вектор входа и выполняйте программу до точки останова. Можете использовать для этой цели окно наблюдаемых переменных X, N, Y. Фиксируйте очередное значение вектора выхода и номер состояния. Убедитесь в правильности работы управляющего автомата.

3) Исследуйте механизм косвенного вызова подпрограммы в отладчике, анализируя содержимое регистра r3 и счетчика команд PC. Есть отличия?
4) Просмотрите файл листинга в местах условного выполнения групп команд и соответствующие фрагменты программы в окне Дизассемблера. Что означает, например, этот фрагмент?

```
0x00000046 BF41      ITTTT          MI
0x00000048 4829      LDRMI          r0,[pc,#164]  ; @0x000000F0
     54:                                 MOVMI r3,#1
0x0000004A 2301      MOVMI          r3,#0x01
     55:                                 MOVMI r4,#4
```

1) Не справились? Обратитесь к проекту CPU_16 – программа приложения MyProg_16.s. После анализа текста программы попробуйте все же самостоятельно создать подпрограмму любого из интерпретаторов вершин.
2) Можете воспользоваться при отладке следующей последовательностью векторов входных переменных: 0, 1, 2, 2, 4, 4, 8, 32. Сдвоенные значения соответствуют последовательным переходам в состояние «Стоп» и реверсу по датчикам конечного положения SQ1 и SQ2.

3) Да. При считывании адреса из таблицы его значение *автоматически* увеличивается на 1 транслятором с Ассемблера. Это сделано для того, чтобы сохранить набор команд Thumb-2 в качестве рабочего при косвенном вызове подпрограммы. В процессе выполнения команды BLX «лишняя» единица в младшем разряде автоматически сбрасывается. Напомним, что для сохранения совместимости с более ранними версиями процессоров ARM, в которых возможны переключения между режимами 32-разрядных команд (ARM) и 16-разрядных команд (Thumb) в процессорах Cortex-M сохранено назначение младшего бита счетчика команд в качестве переключателя текущего набора команд. Однако в Cortex-M используется один общий набор команд Thumb-2. Поэтому, младший бит адреса перехода должен быть установлен в 1 принудительно. Эту задачу и решает транслятор с Ассемблера. Можно не волноваться – ошибки не будет.
4) Транслятор объединил все команды условного выполнения в блоки, максимум по 4 команды, вставив дополнительные команды начала блоков условного выполнения. Команда блока ITTTT MI означает: основное условие для выполнения первой команды MI; такие же условия для трех последующих команд в блоке.

Список рекомендуемой литературы

1) Козаченко В.Ф. Микроконтроллеры: руководство по применению 16-разрядных микроконтроллеров Intel MCS-196/296 во встроенных системах управления. – М.: ЭКОМ, 1997. – 688 с.
2) Встраиваемые высокопроизводительные цифровые системы управления. Практический курс разработки и отладки программного обеспечения сигнальных

микроконтроллеров TMS320x28xxx в интегрированной среде Code Composer Studio: учеб. пособие / А.С. Анучин, Д.И. Алямкин, А.В. Дроздов и др.; под общ. ред. В.Ф. Козаченко, – М.: Издательский дом МЭИ, 2010. – 270 с.

3) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер.с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
4) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
5) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
6) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

13 ИСПОЛЬЗОВАНИЕ БИТОВЫХ ОБЛАСТЕЙ ПАМЯТИ ДАННЫХ И ПАМЯТИ ПЕРИФЕРИЙНЫХ УСТРОЙСТВ

Оглавление

13.1 Назначение и области преимущественного применения

В предыдущей главе мы отмечали, что большое число задач управления оборудованием в реальном времени связано с приемом и обработкой битовой (двоичной) информации. Это, прежде всего, логические контроллеры и дискретные управляющие автоматы. В любом современном силовом преобразователе или источнике питания с микропроцессорной системой управления обязательно имеется автомат управления режимами работы устройства. Любая задача автоматизации технологического процесса требует ввода информации с датчиков дискретных сигналов и выдачи логических команд управления. В современных системах автоматического управления на базе промышленных программируемых контроллеров число дискретных входов/выходов может достигать нескольких тысяч. Естественно, что для эффективной обработки такого большого объема битовой информации «хороший» микроконтроллер должен иметь как соответствующую аппаратную, так и программную поддержку.

В главе 12 мы рассмотрели общие принципы реализации логических контроллеров и дискретных управляющих автоматов с использованием классических областей памяти данных и периферии, адресуемых побайтно и пословно. При этом использовались логические команды процессора, которые работают одновременно с 32-мя битами данных в регистрах ЦПУ, но при правильном подходе могут эффективно выполнять функции и «битового сопроцессора», в частности, за счет «попутных» операций сдвига.

В этой главе мы рассмотрим дополнительные возможности битового сопроцессора, которые *«неявно» интегрированы в архитектуру процессорных ядер ARM*. Речь идет о *специальных областях* в памяти данных (в ОЗУ) и в памяти, отведенной под регистры

периферийных устройств, которые допускают не только пословное или побайтовое обращение к данным, но и *побитовое*. Эти области памяти называются «*битовыми лентами*». Каждый бит в такой ленте имеет свой персональный 32-разрядный адрес, который можно назвать *псевдо-адресом бита.*

Обращаясь по чтению к конкретному биту (указывая его 32-разрядный псевдо-адрес в памяти), Вы получаете информацию о состоянии только нужного бита без дополнительных битовых компонент, которые были бы считаны при побайтовом или пословном обращении. Обращаясь к биту по записи, Вы будете изменять состояние только этого конкретного бита, не меняя состояния остальных бит.

В процессорной технике такие области памяти называют «*семафорной памятью*»: они содержат битовые флаги, состояние которых может быть считано индивидуально или индивидуально изменено. Информация при опросе бита в семафорной памяти всегда попадает в младший разряд указанного Вами регистра ЦПУ, причем все старшие биты регистра назначения автоматически очищаются. Это позволяет:

1) Сразу после чтения проверить состояние полученного бита, например, командой тестирования бит TST, рассмотренной в предыдущей главе;
2) Сразу использовать значение бита при вычислениях любых логических выражений, так как все загруженные в регистры ЦПУ битовые переменные будут автоматически выравнены по младшему разряду и никаких дополнительных операций сдвига не потребуется.

Это существенное преимущество, так как в обычных микроконтроллерах, не имеющих «семафорной памяти», приходится перед вычислениями логических выражений выполнять так называемую предварительную распаковку битовых переменных, а перед выводом управляющих воздействий их упаковку (в байты, слова или полуслова). В процессорах ARM с поддержкой доступа к побитово адресуемой памяти (BitMapping) эти операции становятся лишними. Сразу после получения бита вектора входа с ним можно работать, а после вычисления бита выхода – выдавать значение, причем, непосредственно в битовый порт. Термин «BitMapping» можно перевести как биты, адреса которых отображены на память. Это означает, что некоторые области памяти и периферийных устройств используются специальным образом. Обращение к ним по записи или чтению эквивалентно обращению к одному конкретному биту. Руско-язычный термин «семафорная память» означает область памяти, в которой располагаются флаги (отдельные биты), каждый из которых может рассматриваться как битовый вход или битовый выход, а также как битовое состояние некоего внутреннего устройства или программы (например, дискретного автомата).

Особо отметим, что производители микроконтроллеров при подключении портов ввода/вывода и периферийных устройств собственной разработки к процессорному ядру должны стремиться к тому, чтобы адреса периферии (особенно портов ввода/вывода) по возможности были отображены на области «семафорной» памяти процессора! В этом случае пользователи микроконтроллеров автоматически получают уникальные возможности побитового доступа к содержимому регистров периферийных устройств. Выбирайте «правильные» микроконтроллеры, в которых это подключение сделано оптимально для конечных потребителей.

Итак, фирма ARM, хотя и не заявляет о том, что ее процессоры имеют встроенный битовый сопроцессор, но включила в их архитектуру развитые *средства поддержки операций работы с битами*, как в оперативной памяти, так и непосредственно в регистрах периферийных устройств. Рассмотрим их.

13.2 Области «семафорной памяти» на унифицированной карте памяти процессов ARM-Cortex-M3/M4

Микроконтроллеры фирмы ARM имеют стандартизованную карту памяти объемом 4 Гбайт, рассмотренную нами ранее (глава 6). Под кодовую память (память программ) выделена начальная область памяти объемом 0,5 Гбайт. Столько же (по 0,5 Гбайт) выделено для ОЗУ и регистров периферийных устройств микроконтроллера (ПУ) – устройств ввода-вывода (УВВ). Области ОЗУ и регистров периферийных устройств содержат по 8 разделов памяти объемом 64 Мбайт каждый, всего: 64 х 8 = 512 Мбайт = 0,5 Гбайт.

Первый раздел ОЗУ (64 Мбайт) – особый. Начиная с адреса 0x2000 0000 в нем располагается *побитово адресуемое ОЗУ* объемом 1Мбайт (8 Мбит). Оно предназначено для хранения битовых (Булевых) переменных проекта пользователя – флагов (семафоров), общее число которых может достигать 8 Мбит, что более чем достаточно даже для самых сложных проектов. Эта область памяти получила название *битового поля* или *битовой ленты* ОЗУ. Это обычная область памяти, доступ к которой возможен, как и ко всем остальным ячейкам ОЗУ – по байтам, полусловам и словам. Следовательно, возможны и обычные групповые операции считывания и записи бит, а также размещения в этой области памяти любых не битовых переменных проекта (8-, 16- или 32-разрядных).

Каждый бит битового ОЗУ имеет свой собственный 32-разрядный «псевдо-адрес» – так называемый *адрес доступа к биту* или «псевдоним бита» (alias). Любое обращение процессора по этому адресу заменяется обращением к соответствующему биту в битовом поле.

Псевдо-адреса доступа к битам располагаются в специальной области памяти 0x22000000-0x23FFFFFF. В этой «псевдо-памяти» теоретически можно разместить 32 Мбайт или 32/4 = 8 М слов, ровно столько, сколько отдельных бит содержит битовая лента. Таким образом, каждому реальному биту в битовой ленте ставится в соответствие его 32-разрядная *псевдо-ячейка памяти*, доступ к которой на самом деле аппаратным способом заменяется доступом к отдельному биту – рис. 13.1. Эта технология поддерживается архитектурой центрального процессора.

Если процессор обращается к 32-разрядным ячейкам памяти в области доступа к битам, то это обращение автоматически перенаправляется в область битовой памяти. При этом операция установки бита выполняется в режиме чтение-модификация-запись (read-modify-write) слова, содержащего нужный бит. Содержимое всех остальных бит слова в битовой ленте, кроме нужного бита, не меняется.

Считывание данных/команд из области псевдо-адресов доступа к битам невозможно, так как это не настоящая память, а *псевдо-память*. Конечно, такой подход к распределению памяти можно считать неэкономичным: под псевдо-адреса битов выделяется часть общего объема памяти процессора. Однако, при общем объеме прямо-адресуемой памяти процессора 4 Гбайт – это лишь незначительный объем: 32 Мб/4 Гб = 0,8%.

Аналогичная технология применяется и в сегменте памяти, отведенном под адреса периферийных устройств (ПУ). В начале сегмента располагается побитово адресуемая память объемом 1 Мбайт (8 Мбит), которая позволяет разместить в ней до 1Мбайт /4 = 256 К слов, т.е. 32-разрядных регистров ввода/вывода периферийных устройств. Это очень значительный объем периферии, который превышает потребности даже самых сложных специализированных микроконтроллеров, которыми являются, например, микроконтроллеры управления двигателями (MotorControl) и движениями (MotionControl). Размещение периферии по этим адресам предоставляет огромные преимущества пользователям микроконтроллеров – к каждому биту каждого регистра ПУ становится возможным индивидуальный доступ.

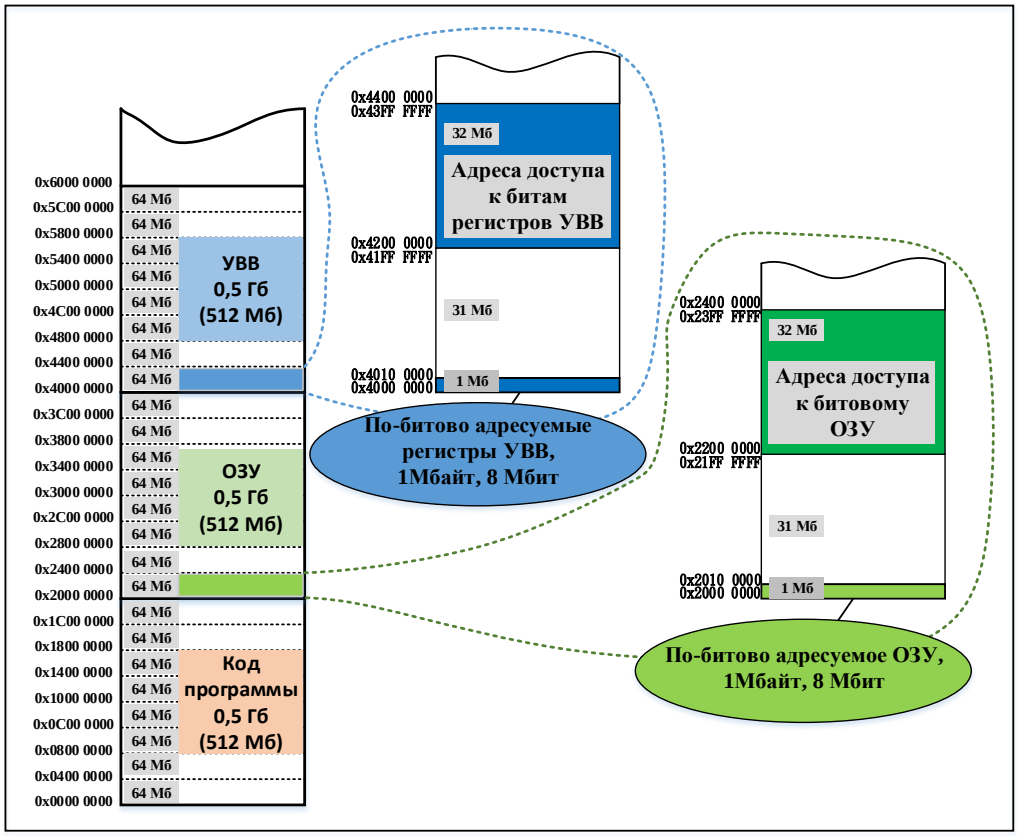


Рис. 13.1 Унифицированная карта памяти микроконтроллеров ARM с областями побитово адресуемого ОЗУ и побитово адресуемых регистров периферийных устройств

Каждый бит регистра ПУ будет иметь свой персональный 32-разрядный адрес (псевдо-адрес или «псевдоним» бита), по которому можно будет прочитать его текущее состояние или установить нужное.

13.3 Справочная информация для работы с картой памяти процессора

Для удобства работы с картой памяти процессора приведем общеизвестную информацию о десятичных весах некоторых наиболее важных разрядов 32-разрядного числа, а также десятичных весах шестнадцатеричных единиц числа, представленного в этой системе счисления – рис. 13.2.

Пример:

Диапазон адресов области доступа к битам в ОЗУ в соответствии с картой памяти: 0x22000000-0x23FFFFFF.

Размер этой области равен:(0x24000000-0x22000000)=0x0200 0000. Так как десятичный вес предпоследней шестнадцатеричной цифры числа 16M, то объем этой области памяти равен 32 Мбайт или 8 М словам – по одному 32-разрядному слову на каждый бит в битовой ленте.

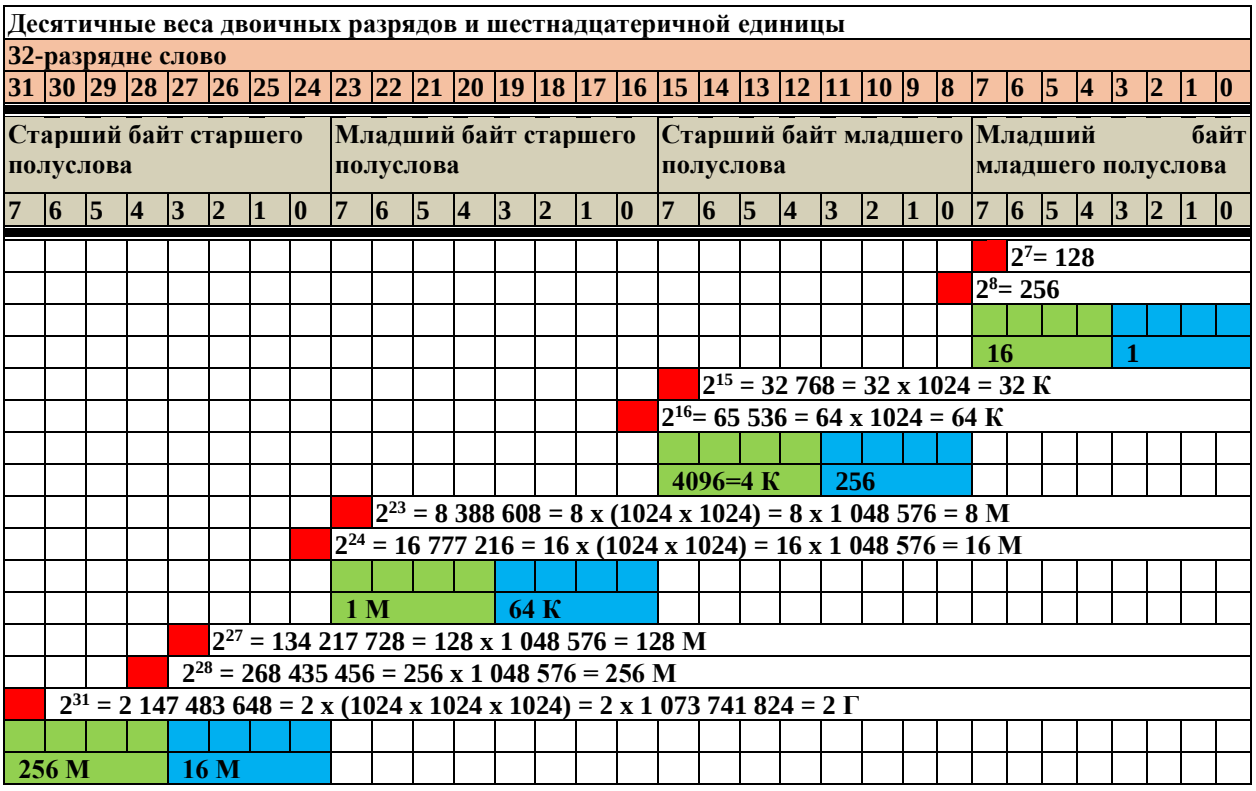


Рис. 13.2 Справочная информация для работы с картой памяти процессора

13.4 Технология установки/сброса бита в битовой памяти и чтения бита

1) Для того, чтобы установить какой-либо бит в битовом поле ОЗУ или в битовом поле регистров периферийных устройств, необходимо в 32-разрядную ячейку памяти с псевдо-адресом бита записать 1 в младший разряд. Это возможно с использованием исключительно косвенной адресации, предварительно установив в базовом регистре Rn псевдо-адрес нужного бита Adr_bit:

```
MOV Ri,#1;
MOV Rn,#Adr_bit  или  LDR Rn,=Adr_bit;
STR Ri,[Rn]
```

2) Для того, чтобы сбросить нужный бит в битовом поле ОЗУ или в битовом поле регистров периферийных устройств, нужно сохранить в 32-разрядной ячейке памяти с псевдо-адресом бита любое число, младший бит которого – нулевой.
3) Для того, чтобы прочитать состояние какого-либо бита в битовом ОЗУ или в битовом поле регистров периферийных устройств, необходимо выполнить операцию чтения из ячейки памяти с псевдо-адресом соответствующего бита. Текущее значение бита окажется в младшем разряде регистра-приемника данных Ri. Все старшие биты регистра приемника будет обнулены. Доступ к данным в памяти выполняется с помощью косвенной адресации:

```
MOV Rn,#Adr_bit  или  LDR Rn,=Adr_bit;
LDR Ri,[Rn]
```

4) Разумеется, возможен и обычный групповой доступ к битам битового поля с помощью команд чтения из битовой ленты слова, полуслова или байта.

Как показано на рис. 13.3, при записи любого слова в область памяти с псевдо-адресами бит только младший бит слова (выделен красным цветом) автоматически сохраняется в нужном бите «семафорной памяти». Остальные биты значения не имеют.

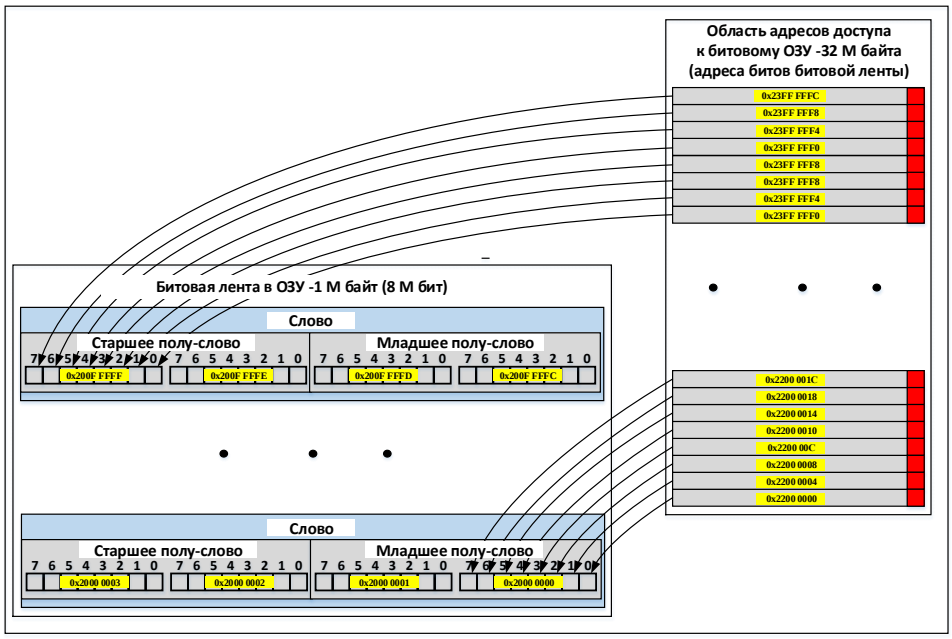


Рис. 13.3 Технология записи бита в «семафорную память»

Аналогичная картина имеет место при обращении к памяти с псевдо-адресами битов регистров периферийных устройств – рис. 13.4. Желтым цветом выделен «рабочий бит» слова, значение которого при записи поступает напрямую в соответствующий бит периферийного устройства (например, в порт вывода).

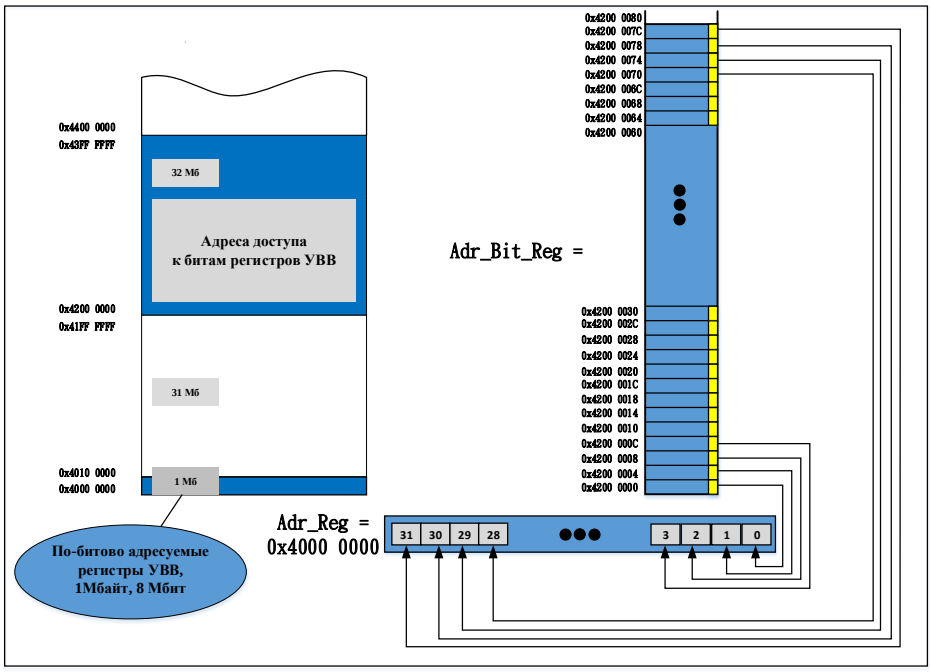


Рис. 13.4 Технология побитовой адресации регистров периферийных устройств

13.5 Как рассчитать псевдо-адрес бита?

Адреса битовых переменных в ОЗУ и в регистрах периферийных устройств, которые отображены на битовые ленты памяти, можно рассчитать. Более того, эту работу можно поручить транслятору с Ассемблера, который может выполнить эти расчеты на стадии ассемблирования программы.

Введем следующие обозначения (см. табл. 13.1):

Таблица 13.1 Условные обозначения адрессов

Adr_base_bit_band	**Базовый адрес битовой ленты в ОЗУ или в области регистров ПУ: 0x2000 0000 – начало битовой ленты в ОЗУ; 0x4000 0000 – начало битовой ленты в памяти регистров ПУ.**
Adr_base_bit_alias	**Базовый адрес области доступа к битовой памяти (псевдо-адресов битов): 0x2200 0000 – начало области доступа к битовой ленте в ОЗУ; 0x4200 0000 – начало области доступа к битовой ленте в памяти регистров ПУ.**
Adr_byte	**Адрес байта в битовой памяти, содержащего нужный бит**
Adr_word	**Адрес слова в битовой памяти, содержащего нужный бит**
N_bit	**Номер нужного бита в байте (от 0 до 7) или в слове (от 0 до 31)**
Adr_bit	**Псевдо-адрес бита в памяти доступа к битовым переменным**

Псевдо-адреса доступа к нужному биту рассчитываются так:

Adr_bit = Adr_base_bit_alias + [(Adr_byte)-(Adr_base_bit_band)]*32 + N_bit*4
или
Adr_bit = Adr_base_bit_alias + [(Adr_word)-(Adr_base_bit_band)]*32 + N_bit*4

Заметьте, что выражение в квадратных скобках представляет собой номер байта в битовой ленте данных. Так как каждый байт содержит 8 бит, а бит адресуется 32-разрядным адресом (4-мя байтами), то приращение адреса относительного базового составляет 8*4=32. Последнее выражение в формулах определяет смещение адреса в зависимости от номера бита – на 4.

Пример:

Требуется найти псевдо-адрес 30-го бита - рис.13.4.

Adr_bit_30 = 0x4200 0000 + 30*4 = 0x4200 0000 +120 =0x4200 0000 + 0x78 = 0x4200 0078, что и показано на рисунке.

При записи бита в область побитово адресуемой памяти значение имеет только младший бит Bit[0] 32-разрядного слова, сохраняемого в ячейке с псевдо-адресом бита. Остальные биты значения не имеют.

При чтении бита из области побитово адресуемой памяти его значение сохраняется в младшем бите регистра-приемника Bit[0]. Все старшие биты автоматически очищаются.

1) Эквивалентны ли команды установки бита в «семафорной памяти», если регистр-источник в ЦПУ содержит 0x01, 0xFF, 0xFFFF, или 0xFFFFFFFF?
2) Эквивалентны ли команды сброса бита в «семафорной памяти», если регистр- источник в ЦПУ содержит 0x00, 0xFE, 0xFFFF 0000?
3) Куда поступает бит, считанный из «семафорной памяти»?
4) Можно ли сразу после получения бита из «семафорной памяти» выполнить ветвление программы по условию EQ или NE?

1) Да, важно только, чтобы младший бит был равен 1.
2) Да, важно только, чтобы младший бит был равен 0.
3) В младший бит регистра-приемника. Все старшие биты обнуляются. Результат чтения бита может быть только 0x00000000 или 0x00000001.
4) Нет. Так как команда загрузки из памяти в регистр ЦПУ не устанавливает флагов. Предварительно нужно воспользоваться командой тестирования бита TST.

13.6 Рекомендации по эффективному использованию побитово адресуемой памяти данных и регистров периферийных устройств

1) Считывание битовых переменных из побитово адресуемой памяти (флагов/семафоров) или из побитово адресуемых регистров периферийных устройств (например, портов ввода) приводит к автоматическому выравниванию всех битовых переменных в регистрах процессора по младшему 0-му разряду с автоматическим обнулением всех старших незначимых разрядов.
2) Возможно считывание сразу группы битовых переменных с использованием команд множественной загрузки регистров LDMRn, {reglist} с автосмещением содержимого указателя в базовом регистре Rn на слово при каждом считывании очередного бита. Базовый регистр Rn должен предварительно инициализироваться псевдо-адресом первого нужного Вам бита.
3) После получения образа входных битовых переменных в регистрах ЦПУ можно использовать любые логические операции процессора (И, ИЛИ, Исключающее ИЛИ, НЕ,…) над содержимым регистров для вычисления логических функций любой сложности или систем логических функций. При этом один из свободных регистров ЦПУ будет использоваться в качестве битового аккумулятора. Опция попутного сдвига второго универсального операнда не должна использоваться, так как все битовые переменные уже отъюстированы по младшему разряду. Результат расчета также будет сохраняться в младшем бите регистра-аккумулятора Bit[0].
4) Значения нескольких логических функций рассчитываются последовательно и сразу выводятся в соответствующие биты «семафорной памяти» или напрямую в побитово-адресуемые порта вывода. Старшие биты Bits[31-1] аккумулятора никакого значения не имеют и их предварительный сброс командами маскирования не требуется.
5) Если Вы предпочитаете вычислять логические функции методом тестирования битовых переменных по граф-схеме вычисления логической функции, то это также возможно с использованием команды тестирования содержимого нужного регистра TSTRi, #1 и следующей за ней команды условной передачи управления BEQили BNE.
6) В последнем случае управление передается в ту точку программы, где выполняется установка или сброс бита в побитово адресуемом регистре вывода данных или побитово адресуемом ОЗУ.

Таким образом, аппаратно, на уровне ядра процессора и программно, на уровне его системы команд, поддерживаются самые эффективные на сегодня методы работы с битовыми переменными, что позволяет реализовывать логические контроллеры и дискретные управляющие автоматы практически любой сложности.

13.7 Пример доступа к побитово адресуемым регистрам периферийных устройств

Обратимся к проекту CPU_17. В программе приложения MyProg_17.s демонстрируются некоторые общие приемы работы с побитово адресуемыми регистрами периферийных устройств. Предположим, что в конкретном микроконтроллере порты ввода/вывода отображены на битовую область памяти с начальным адресом 0x40000000. Объявим адреса этих портов с помощью директивы Ассемблера EQU. С использованием такой же директивы объявим псевдо-адреса входных битовых переменных x0, x1,…,x7, вводимых в систему управления через порт Port_Xи управляющих воздействийy0, y1,…, выводимых через порт Port_Y после расчета их значений. Соответствие номеров битовых переменных их псевдо-адресам показано выше на рис. 13.4.

```
12  ; Объявление адресов регистров входных битовых переменных X
13  ; и выходных управляющих воздействий Y
14  ; в области по-битово, по-байтово адресуемой памяти ПУ
15  Port_X  EQU 0x40000000
16  Port_Y  EQU 0x40000004
17  ; Объявление псевдо-адресов доступа к битам вектора X
18  x0      EQU 0x42000000
19  x1      EQU 0x42000004
20  x2      EQU 0x42000008
21  x3      EQU 0x4200000c
22  x4      EQU 0x42000010
23  x5      EQU 0x42000014
24  x6      EQU 0x42000018
25  x7      EQU 0x4200001C
26  ; Объявление псевдо-адресов доступа к битам вектора Y
27  y0      EQU 0x42000080
28  y1      EQU 0x42000084
```

```
36  ; Считать битовые компоненты вектора X в соответствующие
37  ; регистры процесора x0-> r0;... x7->r7
38  Load_Bit
39  ; Инициализация базового регистра r10 адресом x0
40  ;       MOV r10, #x0
41  ; Получить очередной бит в одноименный регистр с пост-смещени
42  ; содержимого указателя на 4
43  ;       LDR r0,[r10],#4
44  ;       LDR r1,[r10],#4
45  ;       LDR r2,[r10],#4
46  ;       LDR r3,[r10],#4
47  ;       LDR r4,[r10],#4
48  ;       LDR r5,[r10],#4
49  ;       LDR r6,[r10],#4
50  ;       LDR r7,[r10]
```

Перед расчетом выходных управляющих воздействий нужно считать компоненты вектора входа X в регистры ЦПУ процессора. Одно из возможных, но не эффективных решений, предполагает сначала инициализацию базового регистра r10 начальным адресом области памяти, содержащей псевдо-адреса входных битовых переменных x0, а затем последовательную загрузку регистров ЦПУ r0-r7 по нарастающим псевдо-адресам битовых переменных (с авто-смещением содержимого указателя при каждом считывании на 4 байта).

```
52  ; Более эффективна групповая загрузка регистров ЦПУ
53  ; Инициализация базового регистра r10 адресом x0
54          MOV r10, #x0
55  ; Загрузка всех 8-и регистров ЦПУ битовыми переменными
56          LDM r10,{r0-r7}
```

Как видите, это решение «закомментировано», так как есть гораздо более экономичное с использованием команды множественной загрузки регистров ЦПУ командой LDM. Эта команда также требует загрузки базового регистра r10 начальным псевдо-адресом первого, нужного нам, бита и указания списка регистров ЦПУ, которые должны служить приемниками битовых переменных r0-r7. При каждой операции загрузки бита (как бы слова по псевдо-адресу бита) содержимое базового регистра будет автоматически инкрементироваться на 4, указывая на очередной псевдо-адрес битовой переменной. Таким образом, мы введем в ЦПУ значения всех компонент вектора входа x0-x7 (в регистры процессора с соответствующими номерами r0-r7).

```
59  ; Пример вычисления двух битовых управляющих воздействий
60  ; Расчет y0=(x0+x1)*/x7 (битовый аккумулятор r11)
61          ORR r11,r0,r1
62          BIC r11,r7
63  ; Вывод управляющего воздействия y0
64          LDR r10,=y0
65          STR r11, [r10]
66  ; Расчет y1=(x0*x1*/x2)+ x6 (битовый аккуумулятор r11)
67          AND r11,r0,r1
68          BIC r11,r2
69          ORR r11,r6
70  ; Вывод управляющего воздействия y1
71          LDR r10,=y1
72          STR r11, [r10]
```

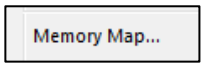
Итак, все входные битовые переменные расположены в регистрах процессора с автоматическим выравниванием по младшему биту. Можно приступать к вычислениям логических функций, определяющих компоненты вектора выхода Y. Используем в качестве битового аккумулятора регистр r11. Так как битовые переменные отъюстированы по младшему разряду, для вычислений используются обычные логические команды без каких-либо дополнительных «попутных» сдвигов.

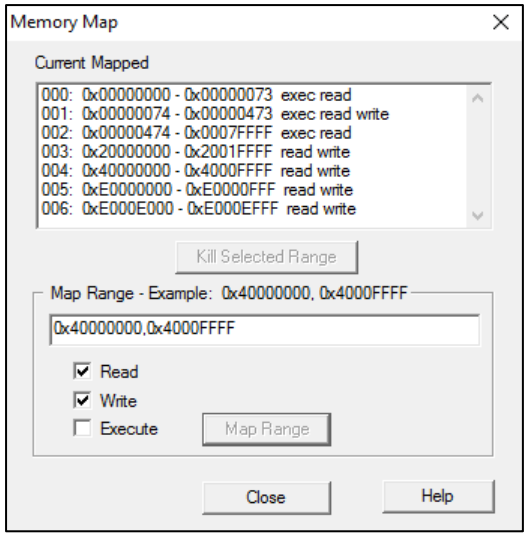
Полученные результаты (в младшем бите «аккумулятора») сразу готовы к выводу в битовые порты. Для этого достаточно проинициализировать базовый регистр r10 псевдо-адресом битового порта вывода и сохранить по этому адресу содержимое «аккумулятора».

Очень эффективная технология!

Выполните трансляцию файлов проекта и его сборку. Загрузите исполняемый файл в отладчик. Как обычно, откройте окно памяти, но с начального адреса 0x40000000, соответствующего началу битового поля в области памяти, отведенной для периферийных устройств. В первом слове (точнее в первом байте этого слова) должны располагаться компоненты вектора входа X. Попробуйте задать какое-либо значение этого вектора путем модификации содержимого ячейки памяти. Ничего не получается. Данные не записываются. В чем же дело? Попробуем разобраться.

Войдите в меню Debug (Отладка) и выберите команду «Карта памяти». В текущем списке доступных, симулируемых в отладчике, областей памяти Вы не найдете области памяти, выделенной для регистров периферийных устройств с начальным адресом 0x40000000. Ведь эта область не для начинающих пользователей, а для достаточно квалифицированных, уже знакомых с побитовой адресацией.





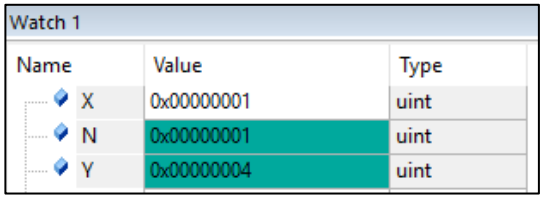
Добавим эту область, указав в специальном окне ее начальный и конечный адрес, а также способ доступа к данным по Чтению (Read) и по Записи (Write). После нажатия на клавишу MapRange (Карта диапазонов памяти) нужный нам диапазон адресов появится в окне под номером 004. Теперь симулятор будет воспринимать все вводимые в эту область памяти данные, более того, будет различать псевдо-адреса битов из этой области и полностью имитировать аппаратные средства процессора, предназначенные для работы с битовыми лентами в памяти.

1) Проверьте правильность расчета псевдо-адресов выходных битовых управляющих воздействий y0, y1 с помощью формул (п. 12.5).
2) Выполните отладку приложения на произвольных наборах входных битовых компонент вектора X. Убедитесь в правильности расчета логических функций y0, y1.

3) Модифицируйте программу для вычисления тех же функций y0, y1 методом тестирования битовых переменных. Для таких простых функций, какой метод Вы будете рекомендовать? Почему?
4) При расчете логических функций переходов в дискретных управляющих автоматах с использованием побитового доступа к компонентам вектора входа X на какое условие должно быть заменено условие «MI» (использованное в предыдущей главе для смены номера состояния автомата и выдачи новых управляющих воздействий)?
5) Дает ли технология «битовых лент» в памяти и периферии преимущества при выводе вектора управляющих воздействий Y в дискретных управляющих автоматах?
6) Модернизируйте программу управления главным приводом продольно-строгального станка (проект CPU_16) с использованием «семафорной памяти» процессора. Выполните отладку проекта. Отметьте основные преимущества решения.

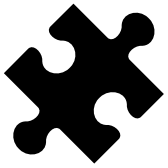
1) Расчет корректный.
2) Например, для входного вектора X=0xFFполучим (y1=1, y0=0), а для X=0x0 - (y1=0, y0=0).
3) Если не справились, обратитесь к файлу MyProg_17_1.s. Вы видите, что этот метод тоже работает, но требует для своей реализации существенно больших затрат – как памяти системы, так и времени программиста – он значительно сложнее в реализации. Метод вычисления логических функций с использованием логических команд существенно проще. Именно его и следует использовать при работе с битами в «битовых лентах».
4) Теперь Вы должны с помощью дополнительной команды TSTRi, #1 протестировать не старший, а младший бит битового «аккумулятора», содержащего результат вычисления функции. Условием истинности функции перехода будет «NE».
5) Практически нет. Причина – вектор выхода Y целесообразно выводить уже упакованным – в виде байта, полуслова или слова. Основная выгода – это быстрая оценка логических функций перехода автомата из текущего состояния в другие состояния.
6) Один из вариантов решения представлен в файле MyProg_17_2_s. Обратите внимание на то, что симулятор µVision по умолчанию поддерживает работу с 128 Кбайтами ОЗУ, начало которого соответствует началу «семафорной памяти» – 0x20000000. В этом случае нет необходимости добавлять новые области памяти в отладчике с использованием команды модификации «карты памяти». Однако, определение псевдо-адресов битов необходимо.

Советуем также зарезервировать основные переменные дискретного автомата X, N, Y в оперативной памяти и объявить их «глобальными». При этом Вы получите возможность ведения отладки с помощью окна наблюдаемых переменных Watch, содержащих эти переменные. Вводите очередное значение вектора X и проверяйте результат работы программы по состоянию переменных N, Y (в режиме «Прогона» до точки останова). Одна из возможных тестовых последовательностей показана в таблице справа.



X	0	1	2	2	4	4	8	32
N	0	1	0	2	0	1	3	0
Y	0	4	0	2	0	4	1	0

13.8 Итоговые рекомендации



Даже если порты ввода/вывода в конкретном микроконтроллере не отображены на побитово адресуемую память, целесообразно хранить копии векторов входа X и выхода Y *логических контроллеров* в «семафорной памяти». Определив псевдо-адреса битов, Вы получите простой доступ ко всем битовым компонентам вектора входа X и вектора выхода Y, сможете быстро рассчитать любую систему логических функций с сохранением результатов в каждом отдельном компоненте вектора выхода Y в «семафорной памяти». В конце всех вычислений автоматически упакованные значения векторов выхода Y (по байтам, полусловам или словам) просто скопируйте в реальные порты вывода микроконтроллера.

Строго соблюдайте правило: сначала создайте образ всех битовых входов X в «семафорной памяти», затем рассчитайте образ вектора выхода Y, и только затем – выдайте вовне управляющие воздействия. Эта последовательность полностью исключит несанкционированные алгоритмом значения выходных управляющих воздействий.

Для *дискретных управляющих автоматов* храните в «семафорной памяти» *только копию вектора входа* X. В интерпретаторе текущей вершины графа проверяйте все возможные условия переходов в другие состояния в порядке их приоритета, установленного графом автомата. Выполняйте переход путем смены номера текущего состояния на состояние перехода и выдачи нового вектора управляющих воздействий Y (сразу упакованного в байты, полуслова, слова). Эту операцию можно выполнить группой команд в блоке условного выполнения по условию «NE», соответствующему истинности проверяемого условия.

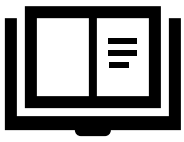
Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User’s Manual. Texas Instruments Inc. – 2011.
3) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
4) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.
5) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
6) ARM DDI 0337H, Cortex-M3. Technical Reference Manual. Arm Limited, 2010.
7) ARM DUI 0068B. ARM Developer Suite. Assembler Guide. Arm Limited, 2001.
8) ARM DUI 0553A. Cortex-M4 Devices. Generic User Guide. Arm Limited, 2010.
9) ARM DDI 0439D. ARM Cortex-M4 Processor. Technical Reference Manual. Arm Limited, 2013.
10) ARM DDI 0479B. Cortex-M System Design Kit. Technical Reference Manual. Arm Limited. – 2011.

14 РАБОТА СО СТЕКОМ. ВЛОЖЕННЫЕ ПОДПРОГРАММЫ

Оглавление

14.1 Типы стеков в микропроцессорных системах



В главе 7 мы познакомились с понятием стека (см. 7.7.3). В этой главе более подробно рассмотрим типы стеков, команды работы со стеками разных типов, способы сохранения/восстановления контекста при работе с подпрограммами и процедурами обслуживания прерываний и особенности организации вложенных подпрограмм.

В микропроцессорных системах различают несколько типов стека в зависимости от направления заполнения стека данными и места в памяти, на которое указывает указатель стека SP.

Классификация стеков по направлению заполнения данными

В зависимости от того, *в каком направлении заполняется стек* в процессе записи в него данных, различают:

- **Descending stack** – Декрементационный стек – **авто-декрементный**. Запись данных в стек выполняется по убывающим адресам памяти.
- **Ascending stack** - Инкрементационный – **авто-инкрементный**. Запись данных в стек выполняется по возрастающим адресам памяти.

Классификация стеков по содержимому указателя стека SP

В зависимости от того, адрес какой ячейки памяти содержит указатель стека, различают:

- **Full stack** – Указатель содержит адрес последней, *заполненной данными* ячейки памяти.
- **Empty stack** – Указатель содержит адрес *пустой, свободной для записи* ячейки памяти.

В соответствии с этими классификационными признаками в микропроцессорных системах встречаются 4 типа стека (их символические обозначения могут включаться в мнемокод ассемблерных команд) (см. табл. 14.1).

Таблица 14.1 Типы стека

Обозначение стека	Основные признаки
FD (Full Descending)	**С указателем на последнюю заполненную данными ячейку памяти (F), с авто-декрементированием при заполнении (D)**
FA (Full Ascending)	**С указателем на последнюю заполненную данными ячейку памяти (F), с авто-инкрементированием при заполнении (A)**
ED (Empty Descending)	**С указателем на пустую (свободную) ячейку памяти (E), с авто-декрементированием при заполнении (D)**
EA (Empty Ascending)	**С указателем на пустую (свободную) ячейку памяти (E), с авто-инкрементированием при заполнении (A)**

В процессорах ARM работа со стеком возможна двумя способами:

- С помощью специальных команд записи (PUSH) содержимого регистров ЦПУ в стек и извлечения данных из стека в регистры ЦПУ (POP);
- С использованием команд множественного сохранения (STM) в памяти содержимого регистров ЦПУ и множественного восстановления (LDM) их значений из памяти.

В процессорах Cortex-M3/M4 на уровне системы команд поддерживаются только два (из четырех возможных) типов стеков (выделены в таблице выше зеленым фоном).

Остальные типы стеков поддерживаются в более производительных процессорах фирмы ARM.

14.2 Основные команды работы со стеком

Имеются две, наиболее часто используемые, команды работы со стеком, синтаксис которых представлен ниже:

PUSH <cond> reglist POP <cond> reglist	
	reglist = {Ri-Rj} – список (листинг) регистров ЦПУ, участвующих в операции множественного сохранения данных в стеке/восстановления данных из стека
	Код условного выполнения команды (опция)
PUSH	**Сохранить содержимое регистров ЦПУ в стеке, начиная с регистра *с максимальным номером, в направлении убывающих номеров* регистров.**
POP	**Извлечь из стека данные и сохранить в регистрах ЦПУ, начиная с регистра *с минимальным номером в направлении возрастающих номеров* регистров.**

Эти команды работают со стеком типа FD (автодекрементным стеком, указатель которого SP всегда содержит адрес последней ячейки, заполненной данными).

По команде PUSH текущее содержимое регистров общего назначения, указанных в списке регистров {Ri-Rj}, сохраняется в стеке. При этом используется косвенная предекрементная адресация, когда перед каждой записью содержимое указателя стека SP четырежды декрементируется, чтобы указатель стека указывал на очередную свободную ячейку памяти. После этого производится сохранение в памяти содержимого очередного регистра, начиная с регистра, имеющего максимальный номер в списке в последовательности уменьшения номеров регистров.

По команде POP из стека извлекаются данные и сохраняются в регистрах ЦПУ, перечисленных в списке {Ri-Rj}. При этом используется постинкрементная адресация, когда после каждого извлечения данных из памяти указатель стека SP автоматически четырежды инкрементируется, показывая на очередные, еще не считанные из стека данные в памяти. Данные сохраняются в регистре ЦПУ с минимальным номером в направлении возрастающих номеров регистров. Самые последние данные восстанавливаются в регистр с максимальным номером, указанном в списке.

Имеются ограничения в использовании регистров – в список регистров нельзя включать счетчик команд PC. Однако, сохранять в стеке содержимое регистра связи LR можно и даже нужно при использовании так называемых вложенных подпрограмм. Эта технология будет рассмотрена в конце главы.

Имена регистров в списке могут быть указаны в любой последовательности, в том числе в виде интервалов или отдельных регистров, разделенных между собой запятой, например: {r0-r3, r5, r7-r10}. В любом случае сохранение данных начинается с регистра с максимальным номером r10 → r0, а восстановление данных – с регистра с минимальным номером r0 → r10. Последний сохраненный в стеке регистр восстанавливается первым, как это и должно быть в любых операциях со стеком.

14.3 Пример использования команд PUSH и POP

Рассмотрим простой пример (проект CPU_18, программа MyProg_18.s). Производится инициализация трех регистров ЦПУ начальными значениями и содержимое всех трех регистров сохраняется в стеке командой PUSH. Дальше регистры могут использоваться для других целей, как в подпрограммах или процедурах обслуживания прерываний, после чего их первоначальное содержимое восстанавливается из стека с помощью команды POP. Если Вы установите точку останова на метке LOOP, то сможете в процессе отладки программы в симуляторе модифицировать исходные значения регистров r1-r3 в регистровом окне и наблюдать, как эти значения восстанавливаются из стека после обнуления.

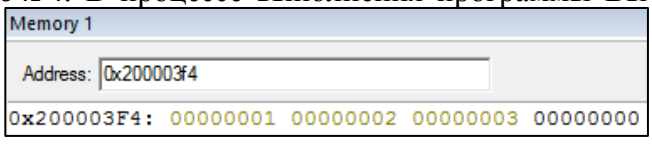
```
11  ; Инициализация трех регистров ЦПУ начальными данными
12          MOV r1,#1
13          MOV r2,#2
14          MOV r3,#3
15  ; Сохранить значения регистров в стеке
16  LOOP
17          PUSH {r1,r2,r3}
18  ;       ...
19  ; Использование регистов r1,r2,r3 (для примера, обнуление)
20          MOV r1,#0
21          MOV r2,#0
22          MOV R3,#0
23  ;       ...
24  ; Восстановить значения регистров из стека
25          POP {r1,r2,r3}
26  ; Повторить цикл сохранения/восстановления данных
27  ; в стеке для других значений в регистрах
28          B LOOP
```

Мы уже не раз отмечали, что перед передачей управления программе пользователя нужно выполнить инициализацию стека. Эта операция выполняется в стартовом файле StartUP_1.s, входящем в проект. Просмотрите этот файл, проследите за тем, как выделяется место под стек и выполняется установка первого вектора прерываний/исключений, содержащего значение вершины стека. Уточнить, где расположен стек и находится вершина стека, можно в файле с картой загрузки проекта CPU_18.map. Так как мы выделили для стека память объемом 1 Кбайт, и зарезервированных переменных в ОЗУ нет, то стек начинается с первого адреса оперативной памяти 0x20000000 и заканчивается на ячейке памяти с адресом 0x20003FFF. Следующий адрес является адресом уже «как бы заполненной данными» области памяти – адресом вершины стека 0x20000400.

Данные сохраняются в стеке и извлекаются из него исключительно 32-разрядными

```
STACK                    0x20000000   Section     1024  startup_1.o(STACK)
  __initial_sp           0x20000400   Data           0  startup_1.o(STACK)
```

полными словами. Так как стек автодекрементный, то очередные ячейки памяти для записи данных в стек будут иметь убывающие адреса 0x200004FC, 0x200004F8, 0x200004F4. Для наблюдения за состоянием стека при отладке программы откройте окно «дампа памяти», начиная с адреса 0x200004F4. В процессе выполнения программы Вы увидите в этом окне сохраненные значения регистров. Они же будут восстановлены из стека по команде POP.



1) Выполните отладку программы при произвольных значениях в регистрах ЦПУ, модифицируя их значения в регистровом окне.

2) Поменяйте порядок расположения регистров сначала в списке для команды PUSH, а затем POP. Программа работает?

3) Замените групповое сохранение/восстановление содержимого регистров сохранением/восстановлением содержимого каждого регистра в отдельности. Проверьте работу программы и последовательность сохранения данных в стеке, анализируя содержимое окна памяти.

2) Да. Порядок расположения регистров в листинге не имеет никакого значения. Их значения будут сохраняться, начиная с регистра с максимальным номером, а восстанавливаться – начиная с регистра с минимальным номером.

3) При индивидуальном сохранении значений регистров в стеке, когда в списке указан только один регистр, например, PUSH {r1}, порядок регистров имеет определяющее значение: последний сохраненный регистр должен восстанавливаться первым (см. программу MyProg_18_1.s). То есть, если регистры сохраняются в определенном порядке, заданном программистом, они должны восстанавливаться обязательно в обратном порядке.

14.4 Использование команд множественного сохранения/восстановления содержимого регистров в памяти для организации стека

Мы уже рассмотрели раньше две команды LDM и STM (см. 11.9) множественной загрузки регистров из памяти и множественного сохранения содержимого регистров в памяти. Обратитесь к этому параграфу еще раз. На самом деле возможности этих команд гораздо шире – они позволяют организовать в памяти процессора *стек нужного пользователю типа.* В таком применении к основному коду команд LDM и STM добавляется суффикс типа стека «Stack» или суффикс типа операции с указателем стека «OpSP»:

```
LDM{Stack/OpSP} Rn{!},reglist
STM{Stack/OpSP} Rn{!},reglist
```

reglist – Список (листинг) регистров ЦПУ, участвующих в операции множественной загрузки или сохранения, заключенный в фигурные скобки {Ri, Rj,…Rm}

! – Опция *авто-обновления* содержимого базового регистра (+4)/(-4) после выполнения очередной пересылки данных

Rn – Имя базового регистра ЦПУ, содержащего начальный адрес области памяти для восстановления/сохранения данных, может быть SP

Тип стека «Stack»	
FD	**Full Descending – SP на заполненную ячейку, заполнение по убыванию адресов**
EA	**Empty Ascending – SP на пустую ячейку, заполнение по возрастанию адресов**
Тип операции с указателем стека «OpSP»	
DB	**Decrement before – декрементирование Rn (SP) перед обращением к памяти**
IA	**Increment after – инкрементирование Rn (SP) после обращения к памяти**

LDM – (Load) – Множественная загрузка нескольких регистров ЦПУ из памяти
STM – (Store) – Множественное сохранение содержимого нескольких регистров в памяти

При организации стека теоретически в качестве базового регистра Rn можно использовать любой регистр ЦПУ. Однако, предпочтение следует отдавать указателю стека SP, который специально предназначен для этого. В этих командах символ косвенной адресации (квадратные скобки вокруг имени базового регистра) не используется. Это исключение из общих правил синтаксиса ассемблерных команд!

Для удобства читателей приведем команды-аналоги, которые могут заменять друг друга при работе с системным стеком, являющимся по умолчанию автодекрементным стеком типа FD (см. табл. 14.2).

Таблица 14.2 Команды-аналоги при работе со стеком

Команды-аналоги при работе со стеком «по умолчанию» типа FD (авто-декрементный по записи)	
Запись в стек	***Извлечение из стека***
PUSH reglist	**POP reglist**
STMFD SP!, reglist	**LDMFD SP!, reglist**
STMDB SP!, reglist	**LDMIA SP!, reglist**
	LDM SP!, reglist

При необходимости Вы можете создать свой собственный автоинкрементный стек типа EA и обеспечивать запись и извлечение данных из стека с помощью команд (см. табл. 14.3).

Таблица 14.3 Команды-аналоги при работе со стеком

Команды-аналоги при работе со стеком типа EA (авто-инкрементный по записи)	
Запись в стек	***Извлечение из стека***
STMEA SP!, reglist	**LDMEA SP!, reglist**
STMIA SP!, reglist	**LDMDB SP!, reglist**
STM SP!, reglist	

Если дополнительный суффикс в мнемокоде команды LDM или STM отсутствует, то он соответствует суффиксу **IA** (инкрементировать после обращения к памяти). Поэтому команда LDM SP!, reglist эквивалентна команде POP reglist при работе со стеком авто-декрементного типа. Напротив, при работе со стеком авто-инкрементного типа **EA** команда STM SP!, reglist соответствует команде записи в стек.

Принцип работы системного стека типа **FD** (указатель SP показывает на уже заполненную данными ячейку памяти, заполнение стека производится в сторону убывающих адресов) показан на рис. 14.1. Единицей данных, сохраняемых или извлекаемых из стека, является полное 32-разрядное слово (4 байта). Данные сохраняются только из регистров ЦПУ и восстанавливаются в них.

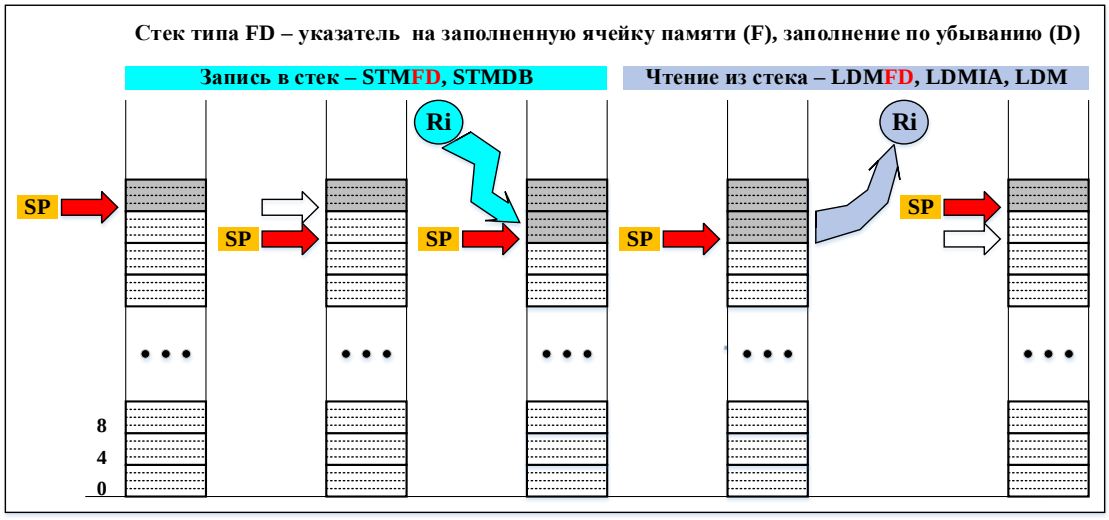


Рис. 14.1 Принцип работы системного стека типа FD

На рис. 14.1 иллюстрируется запись в системный стек и восстановление из стека содержимого одного регистра. При записи сначала декрементируется указатель стека SP, показывая на адрес очередной свободной ячейки памяти, а затем в ней сохраняются данные из регистра ЦПУ. При чтении данные сначала извлекаются по текущему содержимому указателю стека, а затем содержимое указателя авто-инкрементируется.

Графическая иллюстрация работы стека типа EA представлена на рис. 14.2.

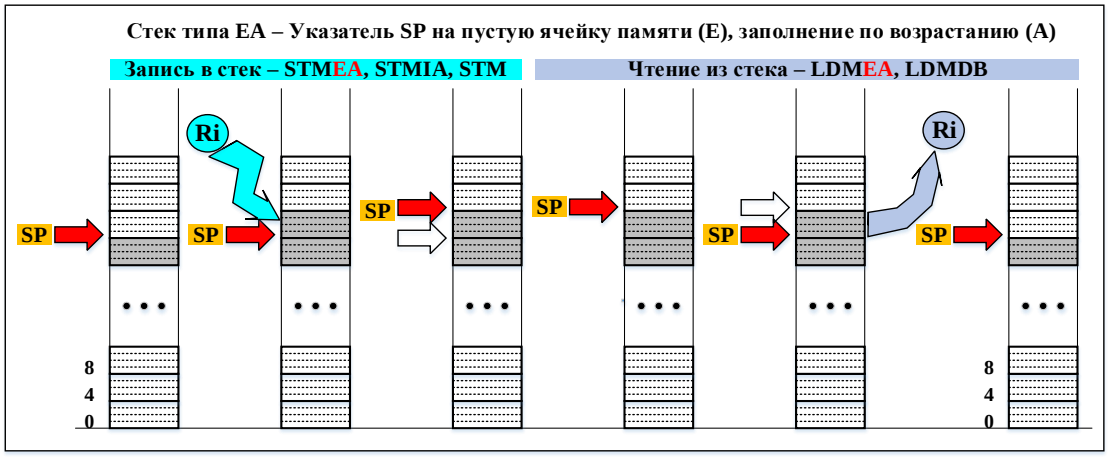


Рис. 14.2 Принцип работы стека типа EA

Здесь указатель стека SP указывает на адрес первой свободой ячейки памяти, в которую и выполняется сохранение содержимого регистра ЦПУ, после чего указатель стека авто-инкрементируется. При восстановлении данных из стека содержимое указателя стека сначала авто-декрементируется, после чего из соответствующей ячейки памяти данные извлекаются в регистр ЦПУ. Содержимое указателя стека SP не меняется, он указывает на адрес ставшей свободной ячейки памяти.

1) Эквивалентны ли команды LDM=LDMIA; STM= STMIA ?
2) Можно ли их использовать в стеке одного типа?
3) Что будет, если в команде STMDB SP{!}, reglist опция «!» будет опущена?
4) Заменяет ли следующая пара команд STMDB SP{!}, reglist; LDMIA SP {!}, reglist команды PUSH и POP при работе с системным стеком?
5) Можно ли заменить команду POP reglist командой LDM SP {!}, reglist при работе с системным стеком?

1) Да, в командах множественной загрузки/сохранения без суффикса по умолчанию всегда используется авто-пост-инкрементная адресация.
2) Нет, команды LDM и STM используются в разных типах стека. Их парное использование в стеке одного типа недопустимо. Рискуете извлечь из стека не те данные, которые сохраняли. Такие ошибки очень трудно идентифицировать!
3) Указатель стека не будет обновляться после каждой записи, хотя запись и будет выполнена правильно (только за счет авто-декрементирования смещения). Восстановление данных из такого стека станет невозможным. При работе со стеком опция «!» обязательна.
4) Да.
5) Да.

Графическая иллюстрация множественного сохранения содержимого регистров Ri, Rj, Rk, Rm с возрастающими номерами (i<j<k<m) в системном стеке и восстановления содержимого этих регистров из системного стека представлена на рис. 14.3.

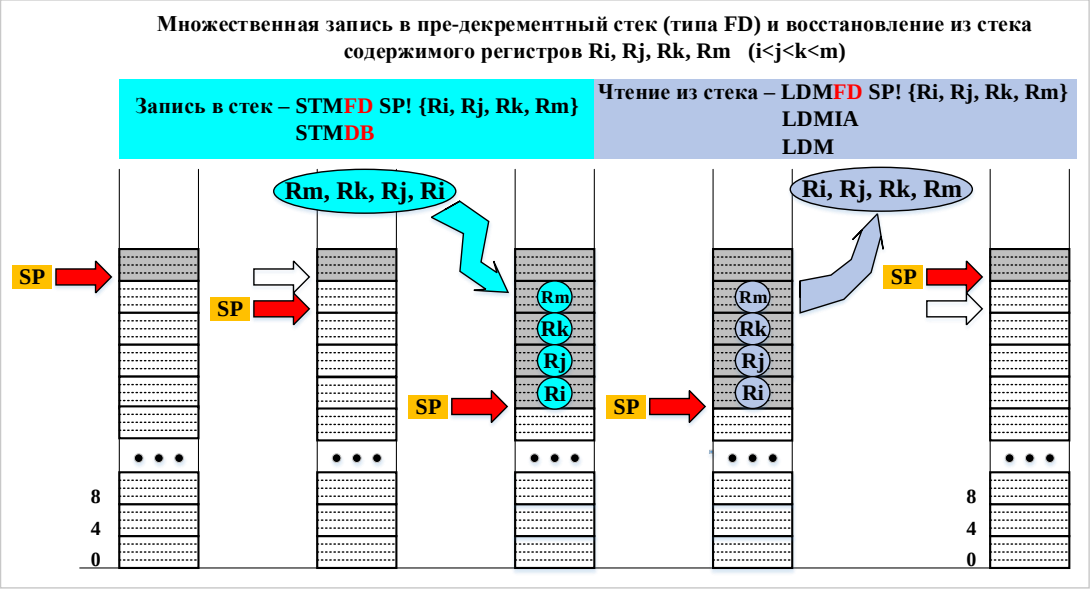


Рис. 14.3 Множественное сохранение/восстановление содержимого регистров ЦПУ в системном стеке

Еще раз обращаем внимание на очень важную деталь: регистры сохраняются в стеке *в порядке убывания их номеров* (от старшего – к младшему), а восстанавливаются - начиная *с самого младшего номера и кончая самым старшим*.

Разумеется, пользователь имеет право создать и использовать свой собственный стек, работающий по альтернативному принципу EA-стека. Приведем графическую иллюстрацию этой технологии, предполагая, что в качестве базового регистра используется указатель стека R13 (SP) – рис. 14.4.

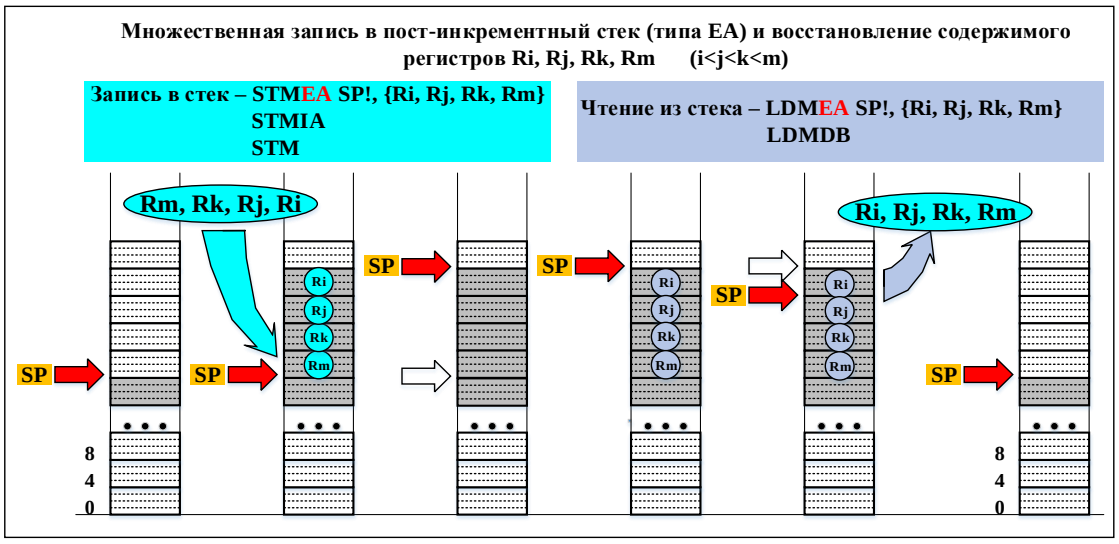


Рис. 14.4 Множественное сохранение/восстановление данных из альтернативного стека типа EA

И в стеке этого типа содержимое регистров ЦПУ сохраняется, начиная с регистра с самым старшим номером в порядке убывания номеров, а восстанавливается в обратном порядке – в направлении возрастания номеров регистров.

14.5 Пример использования команд множественной загрузки/сохранения регистров для доступа к системному стеку

Модифицируем программу приложения (CPU_18), заменив команды PUSH и POP их аналогами – командами множественного сохранения и восстановления содержимого регистров ЦПУ в памяти – MyPorg_18_2.s.

```
10  MyProg
11  ; Инициализация трех регистров ЦПУ начальными данными
12          MOV r1,#1
13          MOV r2,#2
14          MOV r3,#3
15  ; Сохранить значения регистров в стеке
16  LOOP
17          STMDB SP!,{r1-r3}
18  ;           ...
19  ; Использование регистов r1,r2,r3 (для примера, обнуление)
20          MOV r1,#0
21          MOV r2,#0
22          MOV R3,#0
23  ;           ...
24  ; Восстановить значения регистров из стека
25          LDMIA SP!,{r1-r3}
26  ; Повторить цикл сохранения/восстановления данных
27  ; в стеке для других значений в регистрах
28          B LOOP
```

Выполните отладку программы и убедитесь в том, что замена равноценная. Изучая содержимое стека в окне дампа памяти обратите особое внимание на последовательность сохранения данных в стеке – начиная с содержимого регистра ЦПУ с максимальным номером в направлении убывания номеров регистров. Проверьте, что данные из стека, наоборот, извлекаются сначала в регистр с наименьшим номером, а затем – в направлении возрастающих номеров регистров. Используйте другие команды-аналоги с суффиксом «FD». Убедитесь в их работоспособности (см. MyProg_18_3.s).

14.6 Технология организации вложенных подпрограмм

Обычно подпрограмма вызывается из основной программы. Будем называть такую подпрограмму *подпрограммой верхнего уровня*. При ее вызове используются команды передачи управления с обратной связью **`BL label`** или **`BLX Rm`**. В результате их выполнения адрес возврата сохраняется в регистре обратной связи LR (R14), а в счетчик команд PC (R15) загружается адрес точки входа в подпрограмму (см. 7.7.4). После завершения подпрограммы управление возвращается в основную программу командой косвенной передачи управления по содержимому регистра обратной связи **`BX LR`** – адрес возврата, сохраненный в регистре связи LR, загружается в счетчик команд.

Таким образом, при однократном вызове подпрограммы для сохранения адреса возврата используется специальный регистр ЦПУ – регистр связи LR, а не стек, как в большинстве процессоров других фирм.

Если подпрограмма использует регистры ЦПУ, значения которых должны быть сохранены для продолжения вычислений после возврата из подпрограммы, то их содержимое должно быть сохранено в системном стеке в самом начале подпрограммы и восстановлено из стека в ее конце, перед возвратом в основную программу.

Процедура сохранения содержимого нужных регистров в стеке и восстановления из стека называется *сохранением/восстановлением контекста*. Она применяется не только в подпрограммах, но и в процедурах обслуживания прерываний и исключений. Предположим, что в подпрограмме используются регистры ЦПУ r5-r9. Тогда для сохранения их значений в стеке можно использовать команду **`PUSH {r5-r9}`**, а для восстановления значений из стека – команду **`POP {r5-r9}`**. В этом случае структура подпрограммы должна быть следующей:

```
; Подпрограмма
; Точка входа в подпрограмму
SUBR
; Сохранить контекст
PUSH {r5-r9}
; Тело подпрограммы, в котором используются регистры r5-r9
;    …
; Восстановить контекст
POP {r5-r9}
; Возврат в основную программу (косвенный по содержимому LR)
BX LR
```

Перед вызовом подпрограммы (в основной программе) в нее *передаются параметры*, а перед возвратом – в основную программу (в подпрограмме) *возвращаются результаты* расчета. Среди множества способов передачи параметров и возврата результатов наиболее часто используется передача параметров по значениям в определенных регистрах ЦПУ. Пока, для определенности, будем считать, что это начальные регистры младшего регистрового банка ЦПУ, например, r0-r3. Важно, чтобы назначение этих регистров в основной программе и подпрограмме было одинаковым.

Теперь рассмотрим ситуацию, когда подпрограмма, вызванная из основной программы (подпрограмма верхнего уровня), в свою очередь вызывает другую подпрограмму – *подпрограмму нижнего уровня*. Это будет уже так называемая *вложенная подпрограмма*. Она вызывается такими же командами **`BL label`** или **`BLX Rm`**, которые сохраняют адрес возврата в регистре связи LR. Но в этом регистре уже был ранее сохранен адрес возврата в основную программу. Следовательно, он будет потерян.

Какой же выход? Очень простой! Программист, зная, что в теле подпрограммы верхнего уровня будет вызов подпрограммы нижнего уровня, в начале подпрограммы, кроме тех регистров, значения которых он планировал сохранить в стеке, должен дополнительно сохранить в нем содержимое регистра связи: **`PUSH {r5-r9, LR}`**.

После такой команды в теле подпрограммы можно обычным образом вызывать подпрограмму нижнего уровня. Возврат из нее также будет обычным. А вот возврат из подпрограммы верхнего уровня в основную программу должен быть выполнен по-другому: **`POP {r5-r9, PC}`**. Вместо регистра обратной связи LR в список регистров при восстановлении контекста нужно указать счетчик команд PC (R15).

Так как сохранение содержимого регистров в стеке выполняется, начиная с регистра с максимальным номером, то первым будет сохранен регистр связи LR (R14). Напротив, восстановление содержимого регистров будет выполняться в обратном порядке, начиная с регистра с минимальным номером в направлении возрастания номеров регистров. Поэтому последние данные, извлеченные из стека, а именно адрес возврата в основную программу, будет сохранен в регистре R15 (PC). По существу, будет выполнен косвенный возврат в основную программу, но данные для загрузки PC будут извлечены не из регистра связи LR, а из стека.

На рис. 14.5 иллюстрируется механизм взаимодействия основной программы и подпрограмм разного уровня, приводится содержимое регистра связи LR и стека. Адреса возврата для подпрограмм верхнего уровня, имеющих вложения других подпрограмм, принудительно сохраняются в стеке с использованием команды PUSH {LR} и восстанавливаются из него по команде POP {PC}. Подпрограмма самого низкого уровня, не имеющая вложений, вызывается как обычно.

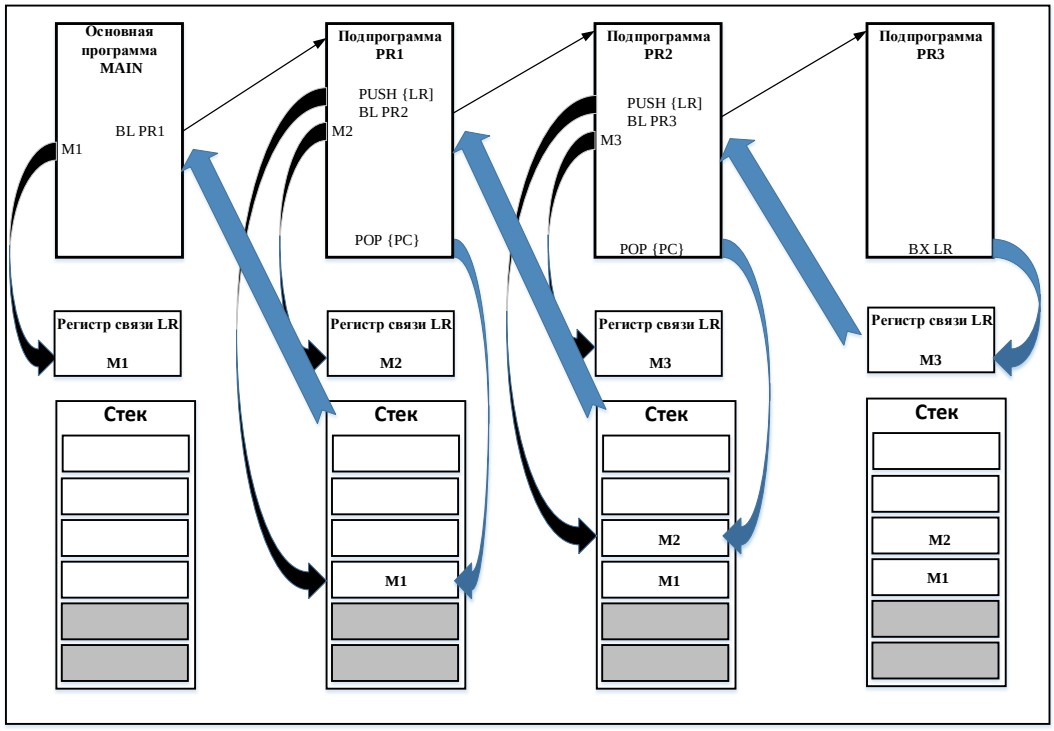


Рис. 14.5 Вложенные подпрограммы и стек

Как быть, если внутри подпрограммы последовательно вызываются несколько подпрограмм? Их может быть сколько угодно. Важно в начале подпрограммы сохранить адрес возврата в стеке с помощью команды PUSH {LR}, а в ее конце – выполнить косвенный возврат в вызывающую программу командой POP {PC}.

Если в подпрограмме не используются вызовы других подпрограмм, то сохранять в стеке можно любой контекст (любые регистры R0-R12 без каких-либо ограничений). Указатель стека SP (R13) при этом выполняет функцию базового регистра и его имя в листинг регистров не должно включаться.

Если в подпрограмме используются вложенные вызовы других подпрограмм (более низкого уровня), то, во избежание потери адреса возврата, в список регистров для сохранения должен обязательно включаться регистр связи LR. Его содержимое будет сохранено в стеке в последнюю очередь, как регистра с наибольшим номером. При восстановлении контекста вместо имени регистра связи LR достаточно указать имя регистра R15 (PC) – счетчика команд. В него и будут восстановлены самые последние данные из стека, а это и есть адрес возврата.

Все процедуры вызова подпрограмм в Ассемблере, а также в C/C++ *по умолчанию* используют предекрементный стек с указателем стека SP, который всегда показывает на адрес последних сохраненных в стеке данных. При необходимости программист может создать свой собственный преинкрементный стек.

14.7 Примеры вложенных подпрограмм копирования и сортировки данных в массиве

14.7.1 Основная программа и подпрограммы верхнего и нижнего уровня

Часто возникает задача копирования массива чисел из одной области памяти в другую с сортировкой массива, например, начиная с максимального значения в сторону минимального. Исходные числа могут быть знаковыми или числами без знака, байтами, полусловами, словами. Задача может быть решена в два этапа: 1 – простое копирование исходного массива по месту назначения; 2 – сортировка элементов массива-приемника. Предположим, что исходные данные – 32-разрядные числа без знака. Нам потребуются две подпрограммы нижнего уровня: 1 – Copy_Array_Word – копирования массива слов; 2 – сортировки слов в массиве-приемнике в направлении убывания значений Sort_Array_Max_Min. Обе подпрограммы (вложенные) будут последовательно вызываться из подпрограммы *верхнего уровня* Copy_Array_Sort.

```
10  MyProg
11  ; Основная программа
12  ; Число элементов исходного массива слов без знака
13  N       EQU   10
14  ;       ...
15  ; Передать параметры и вызвать подпрограмму копирования
16  ; и сортировки массива слов без знака
17          LDR r0,=Array_SRC    ; Адрес массива - источника
18          LDR r1,=Array_DST    ; Адрес массива - приемника
19          MOV r2,#N            ; Число слов в массиве
20          BL Copy_Array_Sort
21  ; Повторить для другого набора данных в исходном массиве
22          B MyProg
```

Пример решения задачи Вы найдете в проекте CPU_18 (программа приложения MyProg_18_4.s). Договоримся, что параметры будут передаваться в подпрограмму верхнего уровня по значениям в начальных регистрах младшего регистрового банка ЦПУ (r0-r3). Остальные регистры могут использоваться во вложенных подпрограммах без ограничений.

```
120 ; Объявить секцию данных в оперативной памяти и зарезервировать место
121 ; для массива- источника и массива-приемника данных
122          ALIGN     ; Выровнять по границе полного слова
123          AREA MyData, DATA, ReadWrite
124 Array_SRC SPACE N*4
125 Array_DST SPACE N*4
126
127 ; Конец ассемблерного текста
128         END
```

В подпрограмму верхнего уровня должны передаваться начальные адреса массивов – источника и приемника данных Array_ SRC, Array_DST, а также число элементов массива N. В конце программы-приложения откроем секцию данных и зарезервируем в ней две области памяти под массив-источник и массив-приемник из десяти 32-разрядных слов.

Обратите внимание на то, что основная программа зацикливается. Это позволяет выполнять ее отладку в режиме прогона с точкой останова в строке 22. Перед очередным прогоном Вы сможете поменять значения в массиве-источнике, а после останова – просмотреть содержимое регистра-приемника: правильно ли выполнена операция копирования и сортировки массива по убыванию значений.

```
24  ;*********************************************************************
25  ; Подпрограмма верхего уровня - копирование и сортировка массива
26  ; слов без знака
27  ; Входные переменные:
28  ; r0 - начальный адрес массива-источника
29  ; r1 - начальный адрес массива-приемника
30  ; r2 - число слов в массиве-источнике
31  ; Используемые регистры: r3-r7
32  ;*********************************************************************
33  Copy_Array_Sort
34  ; Сохранить в стеке значения используемых регистров и регистра связи,
35  ; так как в теле подпрограммы будут вложенные вызовы подпрограмм
36          PUSH {r3-r7, lr}
37  ; Передать параметры и вызвать подпрограмму копирования массива слов
38          MOV r4,r0    ; Начальный адрес массива-источника
39          MOV r5,r1    ; Начальный адерс массива-приемника
40          MOV r6,r2    ; Число элементов массива
41  ; Вызвать подпрограмму копирования массива слов
42          BL Copy_Array_Word
43  ; Вызвать подпрограмму сортировки слов по убыванию в массиве-приемнике
44          BL Sort_Array_Max_Min
45  ; Восстановить значения использованных регистров из стека
46  ; Возврат в основную программу
47          POP{r3-r7, pc}
48  ;*********************************************************************
```

Структура подпрограммы верхнего уровня, которая содержит вложенные подпрограммы, отличается от структуры обычной подпрограммы тем, что в ней выполняется не только сохранение контекста (содержимого тех регистров, которые будут использоваться в подпрограммах r3-r7), но и содержимого регистра обратной связи LR. Это необходимо для того, чтобы при вызове вложенной подпрограммы адрес

возврата в основную программу остался в стеке.

Вызов вложенных подпрограмм выполняется обычным образом. Если необходимо, как при вызове подпрограммы копирования массива Copy_Array_Word, перед вызовом выполняется передача параметров в подпрограмму. Если параметров, переданных в подпрограмму верхнего уровня, достаточно для вложенной подпрограммы (как в случае подпрограммы сортировки массива Sort_Array_Max_Min), и они не «перезаписываются» предыдущей подпрограммой, то выполняется только вызов подпрограммы. Возврат в основную программу обеспечивается командой восстановления значений, используемых в подпрограмме регистров, с одновременным восстановлением адреса возврата из стека в счетчик команд.

```
;*****************************************************************
; Подпрограмма копирования массива 32-разрядных слов
; Входы:
; r4 – начальный адрес массива-источника
; r5 – начальный адрес массива-приемника
; r6 – число слов в массиве
; Используемые регистры:
; r7 – текущее копируемое слово данных
;*****************************************************************
Copy_Array_Word
; Скопировать  очередное слово из массива-источника с
; пост-авто-смещением указателя r4 на +4 (число байт в слове)
        LDR r7,[r4],#4
; Сохранить это слово в массиве-приемнике с пост-авто-смещением
; указателя r5 на +4 (число байт в слове)
        STR r7,[r5],#4
; Декрементировать счетчик числа скопированных слов
        SUBS r6,#1  ; Выставить флаги результата «S»
; Если не все элементы массива скопированы, повторить
        BNE Copy_Array_Word
; Массив скопирован, возврат в вызывающую программму
        BX lr
;*****************************************************************
```

Технология копирования массивов данных из одного места памяти в другое подробно рассмотрена ранее (см. 10.10). Один из вариантов реализации подпрограммы предполагает пословное копирование из массива-источника в массив-приемник с использованием базовой адресации с пост-смещением указателей на длину слова (+4). В качестве базовых регистров используются регистры r4, r5, а в качестве счетчика числа циклов – регистр r6.

14.7.2 Алгоритм сортировки массива

Прежде чем перейти ко второй подпрограмме, рассмотрим суть алгоритма сортировки данных, который часто называют методом «всплывающего воздушного пузырька». Есть массив 32-разрядных целых чисел без знака a[i] из N элементов. Требуется выполнить сортировку элементов массива, расположив их по убыванию значений от максимального к минимальному Max → Min.

1) Пусть, для примера, в массиве только пять элементов (N=5). В ЦПУ считываются два первых элемента массива и сравниваются между собой. Для получения данных можно использовать команду двойной загрузки (двух слов сразу в два регистра процессора): LDRD Rt, Rt2, [Rn, #offset]. Базовый регистр Rn должен содержать начальный адрес массива, элементы которого упорядочиваются, смещение #offset – нулевое. Пусть данные считываются в регистры r4, r5, а в качестве базового регистра используется регистр r6: LDRD r4,r5,[r6].
2) Если первое слово (r4), как число без знака, выше второго (r5) или то же самое (условие HS), то порядок расположения пары элементов массива правильный и можно перейти к сравнению следующей пары элементов, просто увеличив содержимое базового регистра (r6) на длину слова (+4).
3) В противном случае нужно сохранить оба полученных элемента в массиве, но *в обратном порядке*. При этом можно использовать команду сохранения содержимого одновременно двух регистров ЦПУ, поменяв местами имена регистров-источников данных: STRD r5, r4, [r6]. Элементы в массиве меняются местами (большее число без знака как бы «всплывает» на одну позицию вверх).
4) Можно перейти к сравнению следующей пары чисел, увеличив содержимое базового регистра (r6) на +4.

5) Попарное сравнение элементов массива a[i] и a[i+1] выполняется в цикле. Если в массиве всего N элементов, то должно быть выполнено (N-1) сравнение. Назовем такое сравнение одним сканом (просмотром) массива.
6) Если максимальное значение оказалось в самом конце массива (как в примере, показанном ниже), то потребуется (N-1) таких сканов, чтобы максимальное значение оказалось вверху массива (чтобы «пузырек» всплыл на поверхность). Следовательно, в подпрограмме должно быть два цикла: внутренний – для последовательного сравнения всех пар чисел; и внешний – для выполнения нужного числа сканов массива. Состояние массива после одного скана показано в табл. 14.4.

Таблица 14.4 Иллюстрация работы алгоритма сортировки массива

Исходное состояние массива	После сравнения первой пары	После сравнения второй пары	После сравнения третьей пары	После сравнения четвертой пары
4	7	7	7	7
7	4	4	4	4
2	2	2	2	2
1	1	1	1	8
8	8	8	8	1

Обратите внимание, что по условию задачи мы сортируем числа без знака. Поэтому, в рассмотренном алгоритме используется условие «HS» – выше или то же самое.

14.7.3 Пример подпрограммы сортировки массива

```
74  ;*****************************************************************
75  ; Подпрограмма сортировки массива по убыванию значений
76  ; Входы:
77  ; r1 - начальный адрес массива, подлежащего сортировке
78  ; r2 - число элементов массива
79  ; Используемые регистры:
80  ; r3 - указатель начального адреса очередной пары слов массива
81  ; r4,r5 - регистры временного хранения очередной пары слов
82  ; r6 - счетчик числа пар слов в массиве
83  ; r7 - счетчик числа сканов массива
84  ;*****************************************************************
85  Sort_Array_Max_Min
86  ; Инициализация счетчика числа проходов массива
87          SUB r7,r2,#1
88  Out_Loop
89    ; Установить указатель r3 на начальный адрес массива
90          MOV r3,r1
91    ; Инициализация счетчика числа числа пар чисел в массиве
92          SUB r6,r2,#1
93  In_Loop
94    ; Обработать очередную пару элементов массива
95    ; Получить очередную пару слов в регистры r4,r5
96          LDRD r4,r5,[r3]
97    ; Сравнить первое и второе слово в паре
98          CMP r4,r5
99    ; Если первое слово "Выше или равно", сохранить порядок слов
100         BHS Next
101   ; Поменять местами элементы массива в текущей паре
102         STRD r5,r4,[r3]
103 Next
104   ; Переместить указатель на следующую пару элементов массива
105         ADD r3,#4
106   ; Декрементировать счетчик числа пар элементов массива
107         SUBS r6,#1
108   ; Если не все пары элементов массива обработаны - повторить
109   ; внутренний цикл
110         BNE In_Loop
111 ; Декрементировать счетчик числа сканов массива
112         SUBS r7,#1
113 ; Если не все сканы выполнены, повторить
114         BNE Out_Loop
115 ; Сортировка элементов массива выполнена, возврат в вызывающую
116 ; программу
117         BX lr
118 ;*****************************************************************
```

Разработанная в соответствии с описанным выше алгоритмом подпрограмма сортировки слов содержит два цикла, вложенных друг в друга: внешний, обеспечивающий заданное число проходов, и внутренний, обеспечивающий одно попарное сравнение всех слов в массиве.

Оба счетчика циклов инициализируются одним и тем же значением (N-1). При этом используется параметр (r2), переданный ранее из основной программы в подпрограмму верхнего уровня. Параметр (r1) используется для установки начального адреса массива (r3).

Каждая пара чисел считывается с помощью команды двойной загрузки слов из памяти и сравнивается командой сравнения CMP.

В конце подпрограммы проверяется сначала условие выхода из внутреннего цикла, а затем – из внешнего.

Возврат в вызывающую программу выполняется обычным образом.

Среди множества методов адресации памяти в командах двойной загрузки/сохранения слов можно использовать только базовую адресацию с непосредственным смещением [Rn {,#imm}]. При последовательной обработке массивов работать с непосредственным смещением неудобно. Поэтому рекомендуем использовать обычную косвенную адресацию [Rn] (без смещения). Для доступа к очередному элементу массива содержимое базового регистра нужно увеличить на +4 (размер слова в байтах).

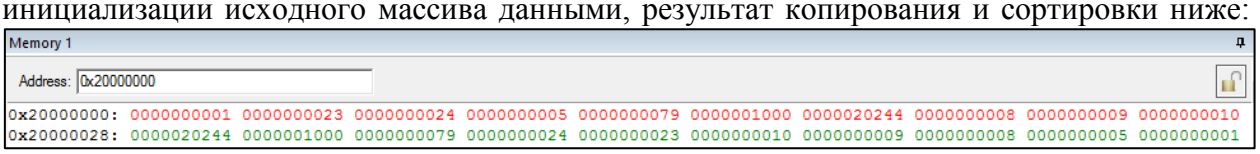
14.7.4 Проверка работоспособности программы

Выполните трансляцию и сборку файлов проекта. Загрузите программу в отладчик. Откройте окно памяти, начиная с адреса 0x20000000, и проинициализируйте массив-источник данных. Установите точку останова в конце основной программы. Выполните прогон программы, исследуйте результат в массиве-приемнике.

Для получения информации о расположении массивов данных и стека в оперативной памяти системы откройте файл с картой загрузки CPU_18.map, фрагмент которого свидетельствует о том, что массив источник имеет начальный адрес 0x20000000,

```
MyData                          0x20000000   Section       80  myprog_18_4.o(MyData)
Array_SRC                       0x20000000   Data          40  myprog_18_4.o(MyData)
Array_DST                       0x20000028   Data          40  myprog_18_4.o(MyData)
STACK                           0x20000050   Section     1024  startup_1.o(STACK)
__initial_sp                    0x20000450   Data           0  startup_1.o(STACK)
```

а массив-приемник 0x20000028. Измените размер окна дампа памяти так, чтобы содержимое обоих массивов было удобно для наблюдения. Один из вариантов инициализации исходного массива данными, результат копирования и сортировки ниже:



Содержимое регистров, используемых в подпрограммах, должно сохраняться в стеке в начале подпрограммы и восстанавливаться перед выходом из нее. Это позволит безопасно пользоваться подпрограммой, не боясь, что данные в вызывающей программе могут быть испорчены.

Если подпрограмма имеет встроенные вызовы подпрограмм, то вместе с сохранением контекста должно выполняться сохранение в стеке и содержимого регистра обратной связи LR. Его значение может восстанавливаться непосредственно в счетчик команд вместе с восстановлением содержимого других регистров.

1) Протестируйте программу на произвольных наборах исходных данных.

2) Регистры r0-r2 используются для передачи параметров в подпрограмму. Почему они не входят в список регистров для сохранения данных в стеке в начале подпрограммы верхнего уровня?

3) Будет ли правильной стратегия сохранения контекста, когда в подпрограмме верхнего уровня сохраняется/восстанавливается только содержимое регистра связи LR командами PUSH {lr}| POP {pc}, а содержимое всех регистров, используемых в подпрограммах нижнего уровня, сохраняется/восстанавливается в подпрограммах нижнего уровня?

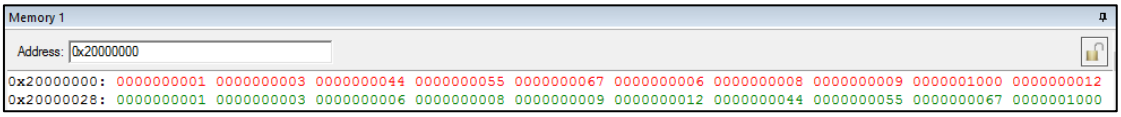
4) Измените подпрограмму сортировки слов без знака, чтобы данные располагались не в порядке убывания значений, а в порядке их возрастания.

5) Измените подпрограмму сортировки для обработки слов со знаком в дополнительном коде в порядке убывания значений.
6) То же для сортировки слов со знаком в порядке возрастания значений.

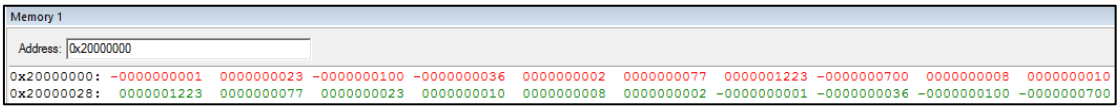
2) Это регистры, которые основная программа использует для передачи параметров в подпрограмму их значениями. Подпрограмма должна использовать эти данные в качестве входных переменных и, при необходимости, может вернуть в эти регистры результаты вычислений.

3) Да. Это даже более правильный путь, так как только вызываемая подпрограмма низкого уровня «знает», какие регистры она использует. Нам же пришлось заранее ознакомиться с этими подпрограммами, чтобы понять, какие из регистров ЦПУ следует сохранять/восстанавливать. Второй вариант подпрограмм верхнего и нижнего уровня см. в MyProg_18_5.s. Некоторый недостаток второго решения состоит в том, что каждая из подпрограмм нижнего уровня может сохранять и восстанавливать в стеке одни и те же регистры ЦПУ (в нашем случае r4-r7).

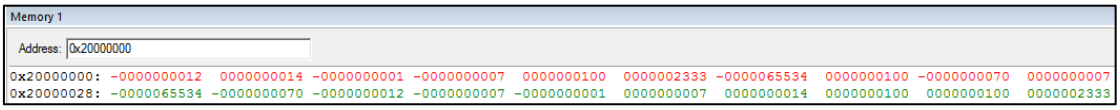
4) Для этого нужно всего лишь изменить условие при сравнении пары слов из массива с HS («Выше или то же самое») на LS («Ниже или то же самое»). Массив будет упорядочен в сторону возрастающих значений. Пример ниже (MyProg_18_6.s):



5) Условие сравнения элементов массива должно быть GT («Строго больше для чисел со знаком» (MyProg_18_7.s). Пример сортировки ниже:



6) Условие сравнения должно быть LE («Меньше или эквивалентно») – MyProg_18_8.s. Пример сортировки массива:



14.8 Блоковые пересылки данных при работе с массивами

Существенным преимуществом процессоров ARM Cortex-M3/M4 является возможность работы с памятью с использованием так называемых «блочных аккумуляторов». Под *блочным аккумулятором* понимается совокупность нескольких регистров ЦПУ, которые могут использоваться в операциях множественного копирования данных из одной области памяти и сохранения блока данных в другой области.

Такие операции могут потребоваться, например, при создании образа векторов входа в «семафорной памяти» для логических контроллеров и дискретных автоматов при загрузке данных из портов ввода, при пересылках массивов данных большой размерности.

В «блочный аккумулятор» можно загружать сразу несколько слов из памяти, например, восемь, а затем сохранять весь блок по месту назначения. Несмотря на то, что изучаемые процессорные ядра не имеют кэш-памяти, они содержат *встроенные средства оптимизации множественного доступа к последовательным областям памяти*. В результате – скорость пересылки данных заметно возрастает.

```
 8  MyProg
 9  ; Основная программа
10  ; Число элементов исходного массива слов
11  N       EQU   20
12  ;       ...
13  ; Передать параметры и вызвать подпрограмму
14  ; поблочного копирования данных
15          LDR r0,=Array_SRC    ; Адрес массива - источника
16          LDR r1,=Array_DST    ; Адрес массива - приемника
17          MOV r2,#N            ; Число слов в массиве
18          BL Copy_Array_Block
19  ; Повторить для другого набора данных в исходном массиве
20          B MyProg
```

Покажем, как использовать эти возможности на примере процедуры поблочного копирования массивов (проект CPU_18, MyProg_18_9.s). В основной программе процедуре поблочного копирования слов передаются начальные адреса массивов-источника и приемника, длина массива, выполняется вызов подпрограммы Copy_Array_Block.

Основная программа зацикливается для удобства отладки с точкой останова на строке 20.

```
22  ;*****************************************************************
23  ; Подпрограмма копирования массива слов с использованием
24  ; команд множественной загрузки/сохранения регистров процессора
25  ; Входы:
26  ; Параметры, передаваемые в подпрограмму содержимым регистров:
27  ; r0 - указатель на начальный адрес массива-источника
28  ; r1 - указатель на начальный адрес массива-приемника
29  ; r2 - число слов массива-источника, подлежащих копированию
30  ; Используемые регистры:
31  ; r3 - счетчик числа блоковых пересылок и регистр временного
32  ;      хранения слова при обычном копировании
33  ; r4-r11 - банк из 8-и регистров ЦПУ для временного приема данных
34  ;*****************************************************************
35  Copy_Array_Block
36  ; Сохранить в стеке содержимое используемого регистра r3
37        PUSH {r3}
38  ; Определить число блоков слов (по 8 слов) в массиве и сохранить в r3
39  ; То есть проинициализировать счетчик числа блочных операций
40  ; Логический попутный сдвиг на 3 разряда вправо (деление на 2*2*2=8)
41        MOVS r3,r2,LSR #3    ; «S» - сформировать флаги результата
42  ; Если число блочных операций равно 0, то скопировать оставшиеся слова
43        BEQ Copy_Rest
```

```
45  Block_Copy
46  ; Выполняем блочное копирование большей части элементов исходного массива
47  ; Сохранить текущее содержимое регистров (r4-r11), используемых в
48  ; в качестве 8-и словного «аккумулятора» в системном стеке (декрементном)
49        PUSH {r4-r11}        ; или STMDB sp!, {r4-r11}
50
51  ; По блочное копирование с использованием восьми регистров процессора
52  ; (r4-r11) в качестве временного «аккумулятора»
53  Copy_Oct
54  ; Загрузка 8-регистров данными из массива-источника с авто-смещением
55  ; содержимого регистра указателя r0 на +4 при каждой операции чтения
56  ; (с инкрементированием указателя r0 после доступа IA)
57        LDMIA r0!, {r4-r11}
58  ; Сохранение полученного блока из 8 слов в массиве-приемнике
59  ; с авто-смещением содержимого регистра указателя r1 при каждой
60  ; операции записи на +4  (тоже с инкрементированием после доступа)
61        STMIA r1!, {r4-r11}
62  ; Декрементировать счетчик числа блоковых пересылок
63        SUBS r3,#1  ; «S» - установить флаги результата
64  ; Если не все блоковые операции выполнены - продолжить
65        BNE Copy_Oct
66
67  ; Восстановить контекст из стека, т.к. регистровый блочный
68  ; «аккумулятор» (r4-r11) больше не потребуется
69        POP {r4-r11}         ; или LDMIA sp!, {r4-r11}
```

Подпрограмма построена так, что в начале определяется число блоков данных из 8-и слов, входящих в массив-источник, которые последовательно копируются в «блочный аккумулятор» на базе регистров ЦПУ r4-r11 и сохраняются из него в массиве-приемнике.

Если блочных пересылок нет, то пословно копируются только оставшиеся слова, число которых не превышает 8.

В самом начале подпрограммы в стеке сохраняется используемый в подпрограмме регистр r3, а в начале операции блокового копирования – регистры блокового аккумулятора r4-r11.

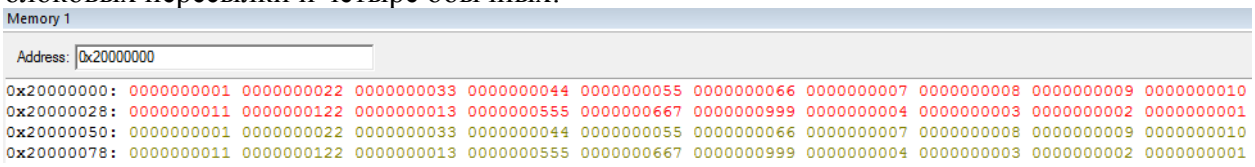
Операция поблочного копирования выполняется в цикле двумя командами: LDMIA – загрузки блока регистров ЦПУ по указателю r0 с инкрементированием после каждой загрузки (IA) и авто-обновлением содержимого указателя; STMIA – сохранения содержимого блокового аккумулятора по указателю r1 с инкрементированием после каждой загрузки (IA) и авто-обновлением содержимого указателя. После завершения операции поблочного копирования содержимое регистров «блокового аккумулятора» восстанавливается из стека.

```
71  ; По словное копирование оставшихся элементов массива
72  Copy_Rest
73  ; Выделим младшие три бита в числе слов, подлежащих копированию
74  ; По сути: получим остаток от деления этого числа на 8
75        ANDS r2,#7  ; «S» - выработать флаги результата
76  ; Если остаток нулевой - все слова скопированы
77        BEQ End_Sub
78  ; Обычное по-словное копирование
79  ; Оба указателя r0 и r1 проинициализированы за счет операции
80  ; авто-обновления «!» при множественной загрузке/сохранении регистров
81  Copy_Word
82        LDR r3,[r0],#4  ; Копировать из массива-источника
83        STR r3,[r1],#4  ; Сохранить в массиве-приемнике
84        SUBS r2,#1  ; Декрементировать счетчик числа циклов
85        BNE Copy_Word ; Повторить при необходимости
86  ; Все элементы скопированы.
87  End_Sub
88  ; Восстановить из стека содержимое регистра r3
89        POP {r3}
90  ; Возврат в основную программу
91        BX lr           ; или MOV pc,lr
92  ;*****************************************************************
```

Процедура завершается пересылкой оставшихся нескопированных слов в массив-приемник. Их число определяется как остаток от деления числа слов массива на размер блокового аккумулятора.

В конце процедуры из стека восстанавливается значение регистра r3 и выполняется возврат в вызывающую программу.

Ниже представлен дамп памяти, полученный при отладке программы. Исходный массив с начального адреса 0x20000000 длиной 20 слов полностью скопирован в выходной массив с начального адреса 0x20000050. При этом использовались две блоковых пересылки и четыре обычных:



14.9 Передача параметров в подпрограммы

14.9.1 Способы передачи параметров в подпрограммы

Существует несколько *способов передачи параметров* в подпрограммы и возврата из них результата. Главные из этих способов:

1) *Содержимым регистров* центрального процессора;
2) *Записью параметров в системный стек* и возврата результатов через системный стек.

До сих пор, во всех приведенных нами примерах, использовался первый метод, когда параметры предварительно записывались в определенные регистры ЦПУ и извлекались из них в процессе выполнения подпрограммы, после чего результаты работы подпрограммы возвращались в основную программу опять же содержимым регистров ЦПУ. В регистрах ЦПУ могут находиться: собственно, *числовые значения параметров*; *значения указателей* на места в памяти (адреса), где фактически располагаются параметры (в том числе массивы и структуры данных, подлежащие обработке). Во втором случае практически снимаются все ограничения по числу переданных в подпрограмму параметров.

Второй метод используется, главным образом, трансляторами с языков высокого уровня и предполагает *запись значений параметров в стек и извлечение их из стека* в подпрограмме. При достаточном объеме памяти, выделенной для системного стека в ОЗУ, ограничения по числу возможных параметров также снимаются. ARM-процессоры имеют развитые способы косвенной адресации, которые позволяют как записать данные в стек, так и извлечь их из стека для обработки внутри подпрограммы:

- Базовую адресацию с непосредственным смещением относительно текущего значения базового регистра (им может быть, в частности, указатель стека: [SP, #offset]). Зная последовательность записи данных в системный стек (фактически величины смещений по отношению к значению базового регистра SP) подпрограмма имеет возможность извлечения из стека любых параметров в любой требуемой последовательности.

- Методы множественной загрузки (STM) данных в память и извлечения данных из памяти (LDM), в которых в качестве базового регистра допускается использование указателя стека SP (подробно рассмотрены в этой главе).

14.9.2 Стандарт вызова процедур фирмы ARM

В процессорах ARM имеется всего 16 регистров общего назначения r0-r15 (см. табл. 14.5). Все они доступны через систему команд. Специалистами фирмы ARM разработана *спецификация (стандарт) вызова процедур* ARM Assembler Procedure Call Standard (AAPCS), которая рекомендуется для использования всем разработчикам как системного, так и прикладного программного обеспечения.

В этой спецификации все регистры центрального процессора делятся на три группы:

1) *Регистры специального назначения*, которые *не должны* использоваться для хранения параметров при вызове подпрограмм и возврата результатов. Это программный счетчик PC, регистр связи LR и указатель стека SP, а также специальный регистр r12, который предлагается использовать в качестве регистра связи при вложенных вызовах процедур, в том числе при «длинных» вызовах.
2) *Регистры локальных переменных* (r4-r11), которые могут использоваться внутри процедур произвольным образом по желанию программиста.
3) *Регистры аргументов/результатов* (r0-r3), предназначенные для передачи параметров в процедуры и возврата результатов.

Таблица 14.5 Регистры общего назначения

Регистр	Использование в подпрограммах		Роль в стандарте вызова процедур
r15	Регистры специального назначения	PC	Program Counter (Программный счетчик)
r14		LR	Link Register (Регистр связи)
r13		SP	Stack Pointer (Указатель стека)
r12		IP	Intra-Procedere-call scratch register (Регистр связи при вызовах внутри процедур)
r11	Локальные переменные	v8	Регистр локальной переменной8
r10		v7	Регистр локальной переменной7
r9		v6	Регистр локальной переменной6
r8		v5	Регистр локальной переменной5
r7		v4	Регистр локальной переменной4
r6		v3	Регистр локальной переменной3
r5		v2	Регистр локальной переменной2
r4		v1	Регистр локальной переменной1
r3	Аргументы	a4	Регистр аргумента4
r2		a3	Регистр аргумента3
r1		a2	Регистр аргумента2/результата2
r0		a1	Регистр аргумента1/результата1

Первые четыре регистра r0-r3 (a1-a4) в соответствии со стандартом используются для *передачи значений аргументов* в подпрограмму и для *получения обратно результатов* ее работы. Они могут также использоваться для хранения любых промежуточных данных между вызовами подпрограмм.

Следующие 8 регистров (r4-r11) являются *регистрами локальных переменных*, которые используются в подпрограммах произвольным образом и должны быть сохранены в стеке в начале подпрограммы и восстановлены в ее конце. Это так называемые регистры, значения которых сохраняются при вызове (call-saved variable register). Обратите внимание на то, что только четыре из этих регистров r4-r7 относятся к

младшему регистровому банку и могут использоваться во всех 16-разрядных командах набора Thumb-2. Остальные относятся к старшему регистровому банку, и их использование потребует уже 32-разрядных команд.

Регистр r9 в некоторых применениях имеет специальное назначение, которое зависит от используемой программной платформы.

Самые старшие четыре регистра r12-r15 выполняют специальные функции: IP, SP, LR и PC. Они не должны использоваться ни для передачи параметров, ни для хранения локальных переменных.

При работе с подпрограммами часто используется и регистр статуса программы пользователя (CPSR), биты которого отражают, в том числе, результаты выполнения операций в подпрограммах, если в них использованы команды с суффиксом разрешения формирования флагов «S» или команды сравнения/тестирования.

Напомним некоторые важные поля регистра статуса программы пользователя:

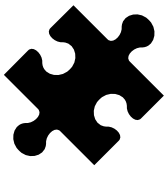
- N, Z, C, V и Q биты (bits 27-31) – основные флаги результатов операций;
- Бит E (bit 8) специфицирует два возможных порядка расположения байт в слове (прямой -little-endian mode) и обратный (big-endian-8 mode);
- Биты A, I, F и M[4:0] (bits 0-7) позволяют установить привилегированный или непривилегированный режим работы процессора.

Стандарт никак не лимитирует при вызове подпрограмм использование регистров сопроцессора. В Cortex-M4F мы имеем 32 регистра однократной точности s0-s31, которые могут быть доступны и как 16 регистров двойной точности d0-d15. Кроме того, возможно использование регистра статуса и управления FPSCR, который содержит результаты операций сравнения чисел в формате с плавающей точкой.

Используйте регистры r0-r3 процессора для передачи в подпрограммы аргументов по их значениям, а также для возврата результатов из подпрограмм.

В качестве регистров локальных переменных используйте регистры r4-r11. Старшие регистры ЦПУ r12-r15 применяйте исключительно в соответствии с их специальным назначением.

Если имеющихся регистров ЦПУ для передачи параметров и возврата результатов недостаточно – применяйте для этой цели системный стек. Компиляторы так и делают с автоматическим контролем нахождения указателя стека в допустимом диапазоне. Значение указателя стека всегда должно быть кратно 4-м, а при использовании специального программного обеспечения верхнего уровня (например, мониторов и ОСРВ) – кратно восьми.



Команда вызова подпрограммы BL (branch-with-link) всегда сохраняет адрес возврата (следующий адрес программы) в регистре связи, а затем загружает в счетчик команд адрес перехода – адрес точки входа в подпрограмму. Так как в Cortex-процессорах используется единая система команд Thumb-2, то бит 0 в регистре связи LR принудительно устанавливается в 1. Это признак набора команд Thumb-2. После возврата из подпрограммы процессор продолжит работу с тем же набором команд.

Любой процедуре в качестве дополнительного параметра (содержимым регистра связи LR) передается адрес возврата в основную программу. Загрузка адреса возврата из регистра связи LR в счетчик команд возвращает управление основной программе.

Последовательность вызова подпрограммы:

- LR[31:1] ← загрузка адреса возврата (return address) в регистр связи, кроме нулевого бита;

- LR[0] ← установка младшего бита в «1», так как процессоры Cortex-M поддерживают единую систему команд Thumb-2.
- PC ← загрузка адреса точки входа в подпрограмму (subroutine address) в счетчик команд.

Напомним, что любой косвенный адрес перехода (в том числе адрес возврата по содержимому регистра связи LR) задает рабочий набор команд процессоров ARM значением младшего разряда. Как только набор Thumb-2 подтвержден единицей в младшем бите, этот бит автоматически очищается (непосредственно перед загрузкой в счетчик команд PC). Эти операции не требуют участия программиста и выполняются автоматически.

14.9.3 Пример передачи параметров в подпрограмму через стек

Продемонстрируем технологию передачи параметров через стек и возврата результата их обработки также через стек на простом примере – проект CPU_18 (MyProg_18_10.s). Исходные значения переменных находятся в регистрах r1-r3 (это 32-разрядные слова 11, 12, 13). Для записи значений параметров в стек используем команду множественного сохранения в нем значений регистров PUSH {r1-r3}. Напомним, что они будут автоматически сохраняться в автодекрементном стеке в последовательности, начиная с регистра с максимальным номером: r3, r2, r1. Параметры переданы. Можно вызывать подпрограмму (обычным образом). Подпрограмма должна модифицировать параметры и вернуть их новые значения вызывающей программе опять же через стек. По адресу возврата разместим команду считывания результатов обработки данных из стека POP {r1-r3}. Вычисления в основной программе могут быть продолжены уже с модифицированными параметрами в регистрах r1-r3. Если в строке 23 поставить точку останова, то можно в отладчике менять исходные значения параметров (в регистровом окне) и запускать цикл обработки вновь, тестируя совместную работу программы и подпрограммы.

```
 8  MyProg
 9  ; Основная программа
10  ; Передать три числовых параметра в подпрограмму
11  ; через стек
12          MOV r1,#11      ;Var1
13          MOV r2,#12      ;Var2
14          MOV r3,#13      ;Var3
15  M1      PUSH {r1-r3}
16  ; Вызвать подпрограмму инкрементирования параметров INC_Var
17          BL INC_Var
18  ; Считать обработанные в подпрограмме результаты из стека
19          POP {r1-r3}
20  ; Продолжение вычислений
21  ;         ...
22  ; Повторить для другого набора данных в исходных регистрах
23          B M1
```

```
25  ;*****************************************************************
26  ; Подпрограмма инкрементирования значений параметров, переданных в
27  ; процедуру через системный стек
28  ; Входы: Три  параметра (слова) в вершине стека
29  ; Используемые регистры:
30  ; r4-r6 - локальные переменные подпрограммы
31  ;*****************************************************************
32  INC_Var
33  ; Сохранить в стеке содержимое локальных переменных
34          PUSH {r4-r6}
35  ; Добавить к указателю стека число 4*3=12, по
36  ; количеству сохраненных в стеке локальных переменных
37  ; Переместить указатель стека на начало блока переданных через стек
38  ; параметров
39          ADD SP, #(4*3)
40  ; Извлечь параметры из стека в регистры локальных переменных
41          POP {r4-r6}
42  ; Обработать переданные в процедуру параметры
43  ; Инеременитировать значения параметров
44          ADD r4,#1
45          ADD r5,#1
46          ADD r6,#1
47  ; Вернуть результаты обработки в стек (для вызывающей программы)
48          PUSH {r4-r6}
49  ; Переместить указатель стека на начало сохраненных в стеке
50  ; значений регистров r4-r6
51          SUB SP,#(4*3)
52  ; Восстановить контекст вызывающей программы
53          POP {r4-r6}
54  ; Возврат в вызывающую программу
55          BX lr
56  ;*****************************************************************
```

Решим в подпрограмме выше простую задачу – всего лишь инкрементируем значение каждого переданного в нее параметра.

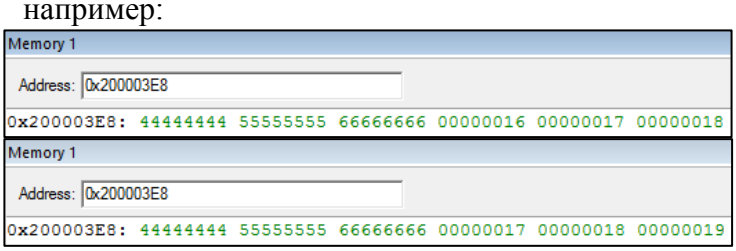
Первое, что нужно сделать – сохранить контекст вызывающей программы, то есть значения регистров ЦПУ, которые в подпрограмме будут использованы в качестве локальных переменных PUSH {r4-r6}. Они будут записаны «вслед» за уже размещенными в стеке значениями параметров.

Инкрементируем указатель SP на число байт, занимаемых сохраненным контекстом, чтобы он показывал на начало переданных в стек параметров, и извлечем их из стека, но уже в регистры локальных переменных POP {r4-r6}.

Теперь можно выполнить обработку параметров (в данном случае – простое инкрементирование их значений) и вернуть результат обработки вызывающей программе записью результатов в стек PUSH {r4-r6}.

Как обычно, в конце подпрограммы нужно восстановить контекст вызывающей программы, то есть исходные значение регистров, которые в подпрограмме использованы для хранения локальных переменных POP {r4-r6}. Перед этим придется уменьшить содержимое SP на длину в байтах этого блока данных. Возврат из подпрограммы традиционный – команда BX lr.

1) Выполните отладку проекта, предварительно открыв окно дампа памяти, начиная с адреса 0x200003E8 (начальный адрес первой из шести ячеек стека, в которые будут записываться и извлекаться данные в процессе работы программы и подпрограммы). Убедитесь в файле карты разгрузки проекта CPU_18.map, что значение адреса вершины стека 0x20000400. Не забудьте установить точку останова для упрощения отладки.
2) Исследуйте порядок сохранения/извлечения из стека содержимого регистров ЦПУ в пошаговом режиме работы.
3) Получите карту содержимого стека при двух последовательных командах выполнения программы до точки останова. Интерпретируйте данные в стеке, например:



3) Помните, что стек заполняется данными в сторону убывающих адресов (автодекрементный). Поэтому, справа налево в нем располагаются: сначала исходные параметры в регистрах (r3, r2, r1) и значения в регистрах (r6, r5, r4), установленные основной программой (сохраненный в стеке контекст); а при втором проходе – уже модифицированные подпрограммой (+1) значения в тех же регистрах (r3, r2, r1) и тот же самый контекст.

При работе со стеком обязательно следите за тем, чтобы каждой записи в стек соответствовало извлечение из стека такого же числа слов. Технология передачи параметров в подпрограммы через стек существенно сложнее передачи параметров содержимым регистров ЦПУ. Пользуйтесь ею при необходимости, обязательно проверяя «парность» записей и извлечений данных в/из стека, а также корректность принудительных смещений указателя стека. Ошибки при работе со стеком приводят к неработоспособности программы (например, к переполнению стека) и сложно идентифицируются.

15 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ТИПОВЫХ АЛГОРИТМИЧЕСКИХ СТРУКТУР

Оглавление

15.1 Возможно ли структурное программирование на Ассемблере?

Считается, что программирование на Ассемблере связано с широким применением команд условной и безусловной передачи управления и, строго говоря, не является *структурным программированием*, под которым понимается написание программы *исключительно на базе стандартизованных алгоритмических конструкций* таких, как условные операторы, операторы циклов, переключателей и т.д., широко используемых в языках высокого уровня. Операторы языков высокого уровня всегда создаются так, чтобы *иметь только один вход и один выход* (представляют собой программные блоки с одним входом и одним выходом). В этом случае их работа может быть легко протестирована.

Заметим, что одними из основных команд Ассемблера являются команды условной и безусловной передачи управления. В зависимости от значения проверяемого условия управление может быть передано в разные точки программы. Если ветвлений в программе много, то ее отладка затрудняется. При обнаружении ошибки приходится проверять все возможные ветви, по которым управление может быть передано в неправильно работающий фрагмент программы.

Однако, трансляторы с языков высокого уровня написаны именно на Ассемблере конкретных процессоров. Следовательно, реализация *типовых алгоритмических структур* на Ассемблере возможна и целесообразна. Это дает преимущества, свойственные современным системам модульного программирования:

1) Любая *типовая алгоритмическая структура* разрабатывается один раз и в дальнейшем (после отладки и документирования) может использоваться в качестве *шаблона для решения аналогичных задач* не только конкретным программистом, но и группой разработчиков, – быть своего рода *внутрифирменным стандартом*.
2) Типовая алгоритмическая структура всегда имеет *один вход и один выход*. Это существенно упрощает отладку, – нет необходимости тестировать множество возможных ветвей программы, в которых теоретически может содержаться ошибочный код.
3) Типовая алгоритмическая структура документируется и может быть представлена в удобной для изучения и использования графической форме – в виде *блок-схемы алгоритма* или *граф-схемы алгоритма*, иллюстрирующей последовательность всех выполняемых структурой действий.
4) Разработка структурированной программы – это разработка *модульной программы*, в которой каждый модуль выполняет свою, строго определенную, функцию. Функции оформляются в виде *подпрограмм*, отлаживаются один раз и вызываются по мере необходимости. Это соответствует концепции *внутренней модульности* программного обеспечения. Каждая подпрограмма сохраняется в отдельном файле на исходном языке и в виде выходного исполнительного файла в перемещаемом объектном коде.
5) Все подпрограммы тестируются на определенном наборе входных переменных, стандартном для конкретной прикладной области.
6) Любой программный модуль на исходном языке может быть подключен к любому новому проекту в виде готового файла.
7) Отлаженные подпрограммы могут объединяться в библиотеки и использоваться в любых других проектах без дополнительного тестирования. При необходимости разрабатываются или приобретаются *специализированные библиотеки*,

адаптированные к предметной области разработчика. Созданные пользователями объектно-ориентированные библиотеки могут продаваться или выставляться в Интернет для свободного использования всеми желающими.

8) Описание любого программного модуля должно быть доступным и содержать сведения о способе передачи в него параметров и возврата результатов.
9) Структурированная программа представляет собой набор определенного числа *программных модулей*, каждый из которых, в свою очередь, состоит из ряда *суб-модулей* (подпрограмм, функций). На самом нижнем уровне находятся *типовые алгоритмические структуры*, которые являются основой структурного программирования. По такому же принципу строятся все языки высокого уровня, в том числе язык C/C++, наиболее популярный у прикладных программистов.
10) Менеджер проекта обеспечивает доступ всех разработчиков к утвержденным на фирме стандартам ПО и шаблонам.

Язык ассемблера, сам по себе не являясь языком структурного программирования, позволяет *максимально эффективно* использовать всю систему команд процессора для создания типовых алгоритмических структур, *аналогичных операторам языков высокого уровня*.

В систему команд ARM-процессоров Cortex-M3/M4 включен *ряд очень мощных дополнительных команд*, которые по своим возможностям не уступают, а иногда и превосходят, соответствующие операторы языков высокого уровня (некоторые из них уже рассмотрены нами (такие, как команда IT блока условно-выполняемых команд), некоторые будут представлены в этой главе). Современный Ассемблер *позволяет создавать очень эффективные, полностью структурированные модульные программы* в соответствии с передовой концепцией структурного программирования.

15.2 Базовые принципы создания типовых алгоритмических структур

1) Все метки в модулях-шаблонах должны быть *локальными*, т.е. видимыми изнутри модуля и невидимыми извне. Для внешнего пользователя модуль должен представлять собой набор ассемблерных команд, помеченный меткой входа в модуль *Entry_Point* и меткой выхода из модуля *Exit_Point*.
2) Любой программный модуль-шаблон может содержать *блок сравнения/тестирования*, в котором выполняется сравнение каких-либо переменных между собой, тестируется некоторая битовая переменная или ранее рассчитанное значение логической (Булевой) функции, а также *исполнительный блок* – последовательность ассемблерных команд, предназначенную для выполнения при том или ином значении условия, выработанного в блоке сравнения/тестирования.
3) В модуле-шаблоне может быть несколько блоков сравнения/тестирования и несколько исполнительных блоков.
4) Для сравнения числовых переменных между собой используются команды сравнения CMP, CMN и команда тестирования на эквивалентность TEQ.
5) Битовые переменные тестируются командой TST или логическими командами с маской во втором универсальном операнде.
6) ARM-процессоры поддерживает опцию установки флагов результатов операции (дополнительный суффикс «S» в мнемокоде) при выполнении большинства команд. Поэтому, условие сравнения может быть выработано обычной командой с

дополнительным суффиксом «S», например, командой вычитания вместо команды сравнения или любой логической командой на завершающем этапе расчета Булевой функции. При этом *команда сравнения может в явном виде отсутствовать*. Ее функцию будет выполнять последняя команда в предшествующем исполнительном блоке, имеющая суффикс установки флагов «S».

7) Флаги в регистре статуса программы пользователя PSR автоматически обрабатываются процессором так (см. 7.9), что возможна последующая проверка сразу *комплексных условий.* Для этого в соответствующей команде условной передачи управления или условного выполнения достаточно использовать лишь нужный суффикс – код условного выполнения cond.
8) Назначение кодов условного выполнения и области их преимущественного применения с цветовым выделением парных условий (см. табл. 15.1).

Таблица 15.1 Коды условного выполнения

Код cond	Оператор сравнения	Описание	Проверяемое условие
EQ	**=**	**Равно**	**Эквивалентно/неэквивалентно при сравнении чисел со знаком и без знака;** **Бит сброшен/установлен при тестировании бит маскированием;** **Логическая функция «False»/«True»при тестировании входных битовых переменных в «семафорной» памяти.**
NE	**≠**	**Неравно**	
HI	**> Unsigned**	**Выше**	**Выше при сравнении чисел без знака;** **Ниже или то же самое при сравнении чисел без знака.**
LS	**≤ Unsigned**	**Ниже или то же самое**	
HS	**≥ Unsigned**	**Выше или то же самое**	**Выше или то же самое при сравнении чисел без знака;** **Ниже при сравнении чисел без знака.**
LO	**< Unsigned**	**Ниже**	
GT	**>Signed**	**Больше**	**Больше (строго) при сравнении чисел со знаком;** **Меньше или равно при сравнении чисел со знаком.**
LE	**≤ Signed**	**Меньше или равно**	
GE	**≥Signed**	**Больше или равно**	**Больше или равно при сравнении чисел со знаком;** **Меньше (строго) при сравнении чисел со знаком.**
LT	**<Signed**	**Меньше**	
MI	**Negative**	**Минус**	**Результат отрицательный/положительный;** **Старший бит установлен Bit[31]=1 или сброшен Bit[31]=0.** **Логическая функция «True»/«False»при вычислениях с использованием логических команд с попутным сдвигом битов в старший разряд «регистра-аккумулятора»**
PL	**Positive**	**Плюс**	
VS	**Overflow**	**Переполнение**	**Знаковое переполнение в предшествующем исполнительном блоке;** **Нет знакового переполнения.**
VC	**No Overflow**	**Нет переполнения**	

9) Все коды условного выполнения *парные*: прямой код условия всегда имеет свой собственный *альтернативный* (*противоположный*) код, соответствующий *оппозитному* значению условия. Если прямое условие «Истинно» (True), то оппозитное – «Ложно» (False), и наоборот. Такая возможность значительно упрощает программирование и позволяет выбирать самое оптимальное решение – быстро выполняемое и компактное.

При разработке типовых алгоритмических структур целесообразно использовать унифицированные обозначения, например, как показано в табл. 15.2:

Таблица 15.2 Унифицированные обозначения

Обозначение	Что означает
Cond	**Логическое условие (Boolean condition), соответствующее одному из кодов условного выполнения, условной передачи управления**
/Cond	**Обратное, противоположное (оппозитное) логическое условие (Oppositecond)**
Do	**Блок обработки или исполнительный блок – непрерывная последовательность ассемблерных команд, подлежащая выполнению**
Do_True	**Исполнительный блок, выполняемый при истинности условия**
Do_False	**Исполнительный блок, выполняемый при ложности условия**
Var	**Переменные (цифровые данные, расположенные в регистрах ЦПУ), в том числе битовые переменные.**
M_Entry	**Метка точки входа**
M_Exit	**Метка точки выхода**

Алгоритмические структуры-шаблоны должны быть «читабельными» – иметь по возможности не более 10 строк кода. При этом исполнительные блоки (*Do*) можно иллюстрировать одной – двумя командами, командой вызова процедуры, либо просто меткой начала блока M_Do.

В блок сравнения/тестирования в шаблоне целесообразно включить всего одну команду сравнения CMP, имея в виду, что на самом деле ему может предшествовать целая группа команд, последняя из которых будет иметь суффикс «S» выработки флагов, нужных для определения значения тестируемого в шаблоне логического условия. Именно эта команда фактически будет началом блока сравнения/тестирования.

В начале главы мы рассмотрим примеры алгоритмических структур, созданных без учета возможностей собственной мощной команды Ассемблера IT (Если … То). Затем покажем, как можно модифицировать эти структуры с использованием технологии условного выполнения команд в процессорах ARM, что позволит создавать более компактные и быстро выполняемые программные модули. Основания для такого подхода следующие:

1) В большинстве процессоров, выпускаемых другими фирмами, нет возможности условного выполнения команд, и приходится реализовывать типовые структуры классическими методами, знание которых необходимо.
2) Не всегда использование условно-выполняемых команд дает преимущества. Блоки IT имеют ограничения по числу включаемых в них команд (не более 4-х). При большом числе команд в таком блоке выгоднее по производительности может оказаться обычная классическая структура.

Хорошие трансляторы для процессоров ARM имеют встроенные возможности оптимизации кода с использованием команды процессора «Если…То (IT)». Если Вы создаете свои программы на языке высокого уровня, попробуйте оптимизировать их с использованием блоков условно-выполняемых команд.

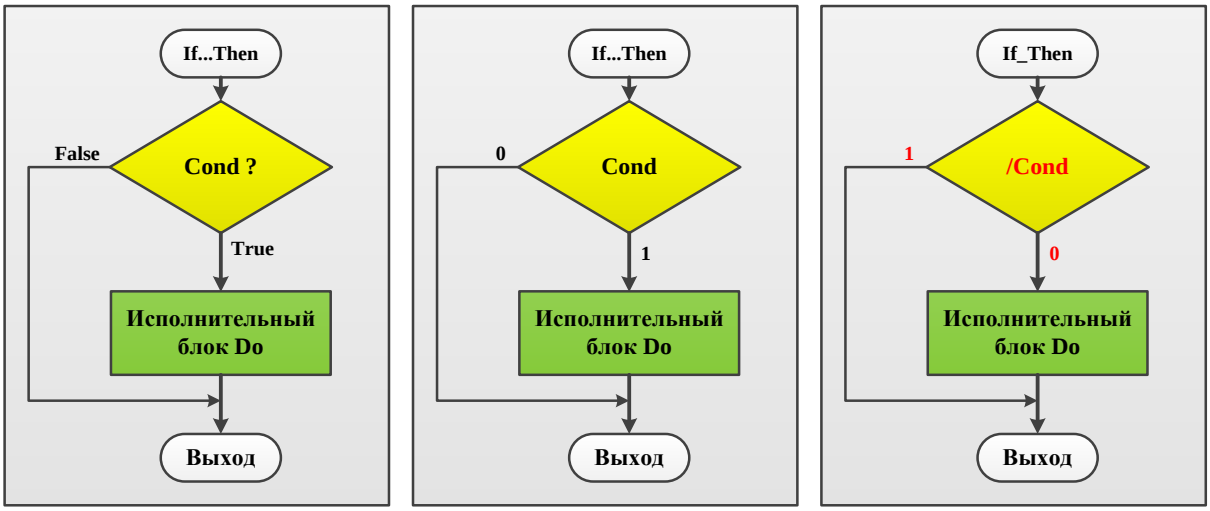
15.3 Типовые алгоритмические структуры

15.3.1 Оператор IF… Then (Если… То)

Это один из наиболее часто используемых операторов выбора альтернативы (Alternative Structure), который называется оператором простой или сокращенной альтернативы (Simple alternative). Если некоторое логическое условие истинно, то блок обработки Do выполняется, в противном случае – нет (пропускается). Для простой альтернативы отсутствует оператор Else (в противном случае) и соответствующий ему набор действий:

```
If (Cond)
Then Do
End If
```

На рис. 15.1 показана блок-схема алгоритма оператора If…Then и соответствующая ему граф-схема алгоритма, отличающаяся тем, что значения условий True/False на стрелках переходов заменены их числовыми значениями 1/0, что проще для восприятия.



а) б) в)

Рис. 15.1 а) Блок-схема оператора «Если… То»; б) граф-схема оператора «Если… То»; в) оптимальная структура для реализации на Ассемблере

Для реализации этой структуры нужен *блок сравнения/тестирования* (ромб на рис. 15.1), в котором выполняется операция сравнения/тестирования или любая другая операция с опцией выработки флагов «S». При этом формируется как прямое условие Cond, так и альтернативное - /Cond. Любая операция выработки флагов в регистре статуса программы пользователя PSR может быть началом блока If…Then.

Если выполнить переход на исполнительный блок Do по истинности прямого условия, то сразу после блока сравнения/тестирования придется применить дополнительную команду безусловной передачи управления на выход, чтобы обойти исполнительный блок при ложном значении проверяемого условия. Если этого не сделать, то исполнительный блок будет выполняться постоянно, при любом значении условия.

Последний вариант в) с точки зрения Ассемблера более правильный: проверяется не прямое, а оппозитное условие. Если оно истинно, то исполнительный блок обходится командой условной передачи управления на выход. В противном случае (истинно прямое условие) – выполняется блок Do. Такая структура требует только одной команды передачи управления на выход блока по истинности оппозитного условия.

Договоримся для определенности, что блок сравнения/тестирования в шаблонах алгоритмических структур будет представлен командой сравнения числовых переменных Var1 и Var2, расположенных в регистрах ЦПУ. Метка этой команды M_If и будет началом алгоритмической структуры. Метка начала исполнительного блока M_Do в данном случае выполняет лишь «декоративную роль» – комментария. Фактически используется только метка конца структуры M_Exit (следующей команды программы после данной структуры), на которую выполняется переход по оппозитному условию. Первые метки структуры M_Entry и M_If также выполняют роль дополнительного комментария:

Шаблон простой альтернативы If…Then:

```
M_Entry                          ; Точка входа
M_If      CMP Rvar1, Rvar2       ; Сравнение переменных
                                 ; Выработка условий Cond и /Cond
          B /Cond M_Exit         ; Переход по оппозитному условию
M_Do      ...                    ; Исполнительный блок Do
          ...
M_Exit                           ; Точка выхода из структуры
```

Пример:

Получить абсолютное значение числа со знаком в дополнительном коде, загруженного в регистр r1ЦПУ. Сравним это число с нулем, для формирования условий GE (Больше или эквивалентно для чисел со знаком) и LT (Строго меньше). Только во втором случае нужно инвертировать знак числа. Следовательно, прямым условием является LT, а альтернативным – GE. Именно по альтернативному условию GE будет выполняться обход исполнительного блока, в котором будет только одна команда – реверсивного вычитания из нуля значения в регистре r1с сохранением результата в том же регистре (фактически – команда инверсии знака числа):

```
M_If      CMP R1, #0             ; Сравнение с #0, установка флагов
          BGE M_Exit             ; Обход исполнительного блока
                                 ; по оппозитному условию >=
M_Do      RSB R1, #0             ; Исполнительный блок
                                 ; R1 ← -R1
M_Exit
```

Тот же пример, при использовании условно-выполняемых команд, можно модифицировать так:

```
M_If      CMP R1, #0             ; Сравнение с #0. Установка флагов
M_Do      RSB LT R1, #0          ; Исполнительный блок
                                 ; R1 ← -R1
M_Exit
```

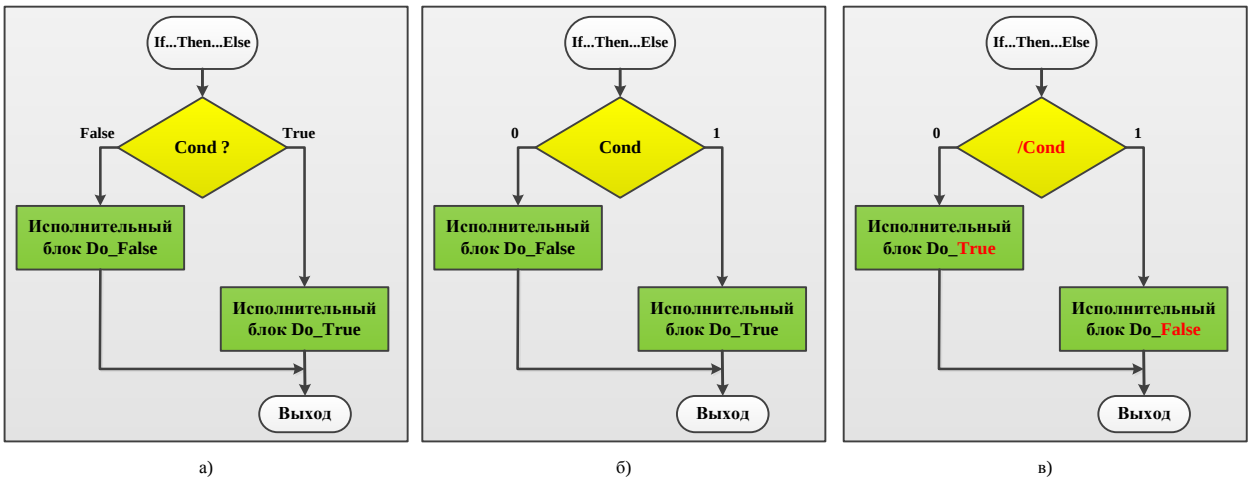
При этом команда исполнительного блока выполняется только при истинности прямого условия LT (Строго меньше), в противном случае – заменяется командой NOP (фактически пропускается). Получаем линейный программный модуль с одним входом и выходом, *полностью соответствующий концепции структурного программирования*.

15.3.2 Оператор If… Then… Else (Если… То… В противном случае)

Оператор полной альтернативы (Complete alternative) предполагает выполнение одного исполнительного блока Do_True при истинности проверяемого условия и второго исполнительного блока Do_False – в противном случае:

```
If (Cond)
     Then Do_True
Else
     Do_False
End If
```

Блок-схема и граф-схема алгоритма полного условного оператора представлены на рис. 15.2. При истинности проверяемого условия выполняется переход на исполнительный блок Do_True. В противном случае, выполняется второй исполнительный блок Do_False, после чего следует передача управления на выход условного оператора, в обход первого исполнительного блока Do_True. Возможна реализация полного условного оператора и с проверкой альтернативного условия, как показано на рис. 15.2, в). Однако, никаких преимуществ эта реализация не дает. Исполнительные блоки просто меняются местами, но дополнительная команда безусловной передачи управления в обход последнего исполнительного блока все равно потребуется.



а) б) в)

Рис. 15.2 Блок-схема *а*) и граф-схема *б*) полного условного оператора If…Then…Else; *в*) возможное альтернативное решение

Шаблон оператора полной альтернативы (If…Then…Else) для первого варианта представлен ниже:

```
M_Entry                          ; Точка входа
M_If      CMP Rvar1, Rvar2       ; Сравнение переменных
                                 ; Выработка условий Cond и /Cond
          B Cond M_Do_True       ; Переход по прямому условию
M_Do_False     ...               ; Исполнительный блок Do_False
               ...
          B M_Exit               ; Обойти блок Do_True
M_Do_True      ...               ; Исполнительный блок Do_True
               ...
M_Exit                           ; Точка выхода
```

Пример:

Протестировать значение младшего бита в регистре r0, полученного в результате вычисления логической функции с использованием битовых входов в «семафорной» памяти микроконтроллера. Если бит установлен, загрузить в регистр r1 код управляющего воздействия 0xAA, в противном случае – код 0x55.

```
M_Entry                          ; Точка входа
M_If        TST r0, #1           ; Тестирование нулевого бита Bit[0]
                                 ; Выработка условий NE(Bit=1)
                                 ; и EQ(Bit=0)
            BNE M_Do_True        ; Переход по прямому условию
M_Do_False  MOV r1, #0x55        ; Исполнительный блок Do_False
            B M_Exit             ; Обойти блок Do_True
M_Do_True   MOV r1 0xAA          ; Исполнительный блок Do_True
M_Exit                           ; Точка выхода
```

Тот же пример с использованием условно-выполняемых команд процессоров ARM существенно упрощается:

```
M_If        TST r0, #1           ; Тестирование нулевого бита Bit[0]
                                 ; Выработка условий NE(Bit=1)
                                 ; и EQ(Bit=0)
M_Do_False  MOVEQ r1, #0x55      ; Исполнительный блок Do_False
M_Do_True   MOVNE r1, #0xAA      ; Исполнительный блок Do_True
M_Exit                           ; Точка выхода
```

Мы получили простой программный модуль с одним входом и одним выходом, в котором вместо двух исполнительных модулей используется один, состоящий из двух условно-выполняемых команд по противоположным условиям (обязательно парным). Всегда будет выполняться только одна команда, для которой условие выполнения истинно, а вторая команда (для которой условие ложно) будет заменяться командой NOP – фактически пропускаться.

Напомним, что транслятор с Ассемблера *автоматически* создает блок условно-выполняемых команд с помощью специальной команды IT–команды начала блока условного выполнения (см. 12.4).

1) Мы привели список альтернативных кодов условий. Например, EQ и NE. Какой из этих кодов является прямым, какой оппозитным?

2) Какие условия нужно проверять, если вычислялось значение логической функции с использованием логических команд с попутным сдвигом битов в область битового аккумулятора – старшего разряда одного из регистров ЦПУ? При этом последняя логическая команда имела суффикс «S» установки флагов в мнемокоде операции. Условный оператор должен выполнить разные действия в зависимости от значения логической функции.

3) Разработайте шаблон полного условного оператора, если в качестве основного используется оппозитное условие (рис. 15.2, *в*). Будет ли эта реализация отличаться по объему кода и производительности от приведенной нами?

4) Можно ли в исполнительных блоках записывать условно-выполняемые команды с альтернативными условиями в любой последовательности?

1) Это зависит исключительно от решаемой задачи. Так, в приведенном выше примере прямым будет условие NE, так как проверяется ненулевое значение бита, а условие EQ – оппозитным. При сравнении двух чисел на равенство, наоборот, прямым будет условие EQ, а оппозитным NE.

2) Можно использовать условие MI в качестве идентификатора истинности логической функции и условие PL – ложности.

3) Нет. Они эквивалентны как по быстродействию, так и по объему кода.

4) Да. Последовательность будет иметь значение только в том случае, если для каждого из альтернативных условий в условном блоке будет больше одной команды. Все команды с одинаковым условием будут выполняться последовательно, так, как они запрограммированы.

15.3.3 Замена оператора полной альтернативы оператором простой альтернативы

Очень часто при программировании возникает ситуация, когда два исполнительных блока Do_True и Do_False являются взаимоисключающими. Набор команд в таких блоках практически одинаков. Отличаются только значения используемых в блоках переменных.

Можно выполнить один из исполнительных блоков «по умолчанию», без какой-либо проверки условия вообще. Далее, вслед за ним использовать простой оператор If…Then с проверкой только оппозитного условия. При его истинности сделанные по умолчанию установки нужно заменить новыми.

Пример:

Для предыдущей задачи тестирования бита и загрузки константы инициализации в регистр r0 возможна реализация:

```
M_Entry                    ; Точка входа
          MOV r1, #0xAA    ; Загрузить в r1 значение, равное
                           ; предположительно установленному биту
M_If      TST r0, #1       ; Протестировать нужный бит Bit[0]
                           ; Выработка условий NE(Bit=1) и
EQ(Bit=0)
          BNE M_Exit       ; Если бит действительно установлен,
                           ; сохранить инициализацию регистра r1
M_Do      MOV r1, #0x55    ; Пере-инициализировать регистр r1
M_Exit                     ; Точка выхода
```

При использовании условно-выполняемых команд, получим:

```
M_Entry                    ; Точка входа
          MOV r1, #0xAA    ; Загрузить в r1 значение, равное
                           ; предположительно установленному биту
M_If      TST r0, #1       ; Протестировать нулевой бит Bit[0]
                           ; Выработка условий NE(Bit=1) и
EQ(Bit=0)
          MOVEQ r1, #0x55; Если бит на самом деле сброшен,
                           ; выполнить
                           ; переинициализацию регистра r1
M_Exit                     ; Точка выхода
```

Как видите, технология условно выполняемых команд позволяет полностью исключить из алгоритма команды условной и безусловной передачи управления, создать *полностью линейный программный блок* с одним входом и одним выходом.

Один блок условного выполнения с автоматически созданной транслятором командой IT, предшествующей блоку, может содержать до четырех команд, в которых используются только парные (прямые или оппозитные) условия. Если условно-выполняемых команд больше четырех, то они автоматически будут объединены в группы условных блоков, каждый из которых будет работать только с одной парой взаимно-исключающих условий с числом команд в каждом блоке не более четырех.

Уникальные возможности процессоров ARM, допускающих условное выполнение практически всех, имеющихся в арсенале процессора команд, позволяют создавать на Ассемблере очень эффективные и интуитивно понятные программные модули типовых структур принятия альтернативных решений. Это будут программные модули на Ассемблере, *полностью соответствующие современной концепции структурного программирования*.

15.3.4 Итерационные алгоритмические структуры

15.3.4.1 Оператор цикла Repeat…Until (Повторять…Пока)

Итерационные структуры обеспечивают выполнение блока кода некоторое число раз и поэтому называются *операторами циклов*. Циклы могут организовываться для выполнения заданного числа операций или для получения результата вычислений с заданной точностью.

Структура цикла «Повторять пока» содержит исполнительный блок Do, после которого следует блок сравнения/тестирования, в котором вырабатывается и проверяется *условие повторения* цикла Cond.

```
Repeat
     Do
Until (Cond)
```

Пока условие Cond истинно, цикл повторяется, в противном случае выполняется выход из цикла и переход к следующей последней команде.

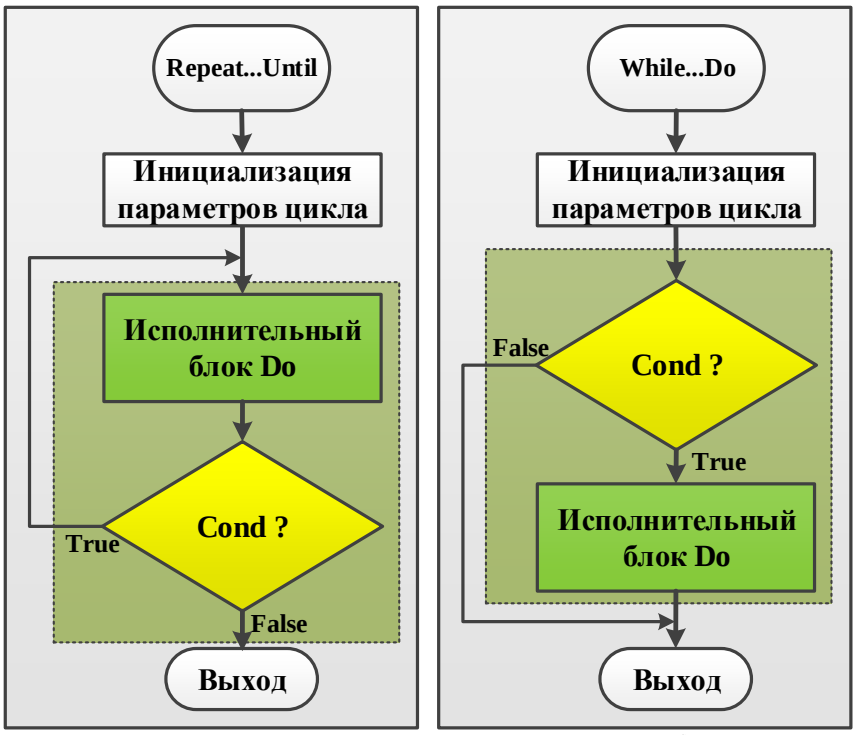
В такой структуре тело цикла выполняется, как минимум, один раз. Прежде чем выполнять цикл, нужно проинициализировать его *параметры*. В простейшем случае параметром цикла, задающим число его повторений, является *счетчик числа циклов* (может размещаться в одном из регистров ЦПУ). Он инициализируется перед началом цикла и модифицируется в его конце, при каждом проходе цикла. Операция модификации содержимого счетчика числа циклов обязательно должна сопровождаться выработкой флагов (например, опцией «S» в команде вычитания), после чего может следовать условный возврат в начало цикла.

Наиболее часто применяется *декрементирование* счетчика числа циклов. При этом условие повторения цикла – это ненулевое значение счетчика (NE), а условие выхода из цикла – альтернативное условие EQ. Если нужно повторить цикл N раз, то счетчик числа циклов инициализируется числом N.

На рис. 15.3, *а* показана блок-схема оператора цикла Repeat… Until.

Если в цикле обрабатываются массивы данных в памяти, то блок инициализации параметров цикла может содержать начальный и конечный адреса рабочей области памяти. Для доступа к данным используется косвенная адресация по содержимому

регистра–указателя (возможно с авто-смещением этого содержимого на размер обрабатываемых данных в байтах). В конце цикла в блоке сравнения/тестирования текущее содержимое указателя сравнивается с конечным адресом массива. По условию «Ниже или то же самое» (LS) управление передается в начало цикла. По альтернативному условию (HI) – выполняется выход из цикла.



а) б)

Рис. 15.3 Блок-схемы основных структур циклов: *а*) Repeat…Until; *б*) While…Do.

Шаблон структуры цикла Repeat…Until (Повторять… Пока):

```
M_Repeat                        ; Точка входа
M_Do        ...                 ; Исполнительный блок Do
                                ; В том числе модификация
                                ; переменной
                                ; управления циклом
M_Until
            CMP Rvar1, Rvar2    ; Сравнение переменных
                                ; Выработка условия
                                ; повторения цикла
            B Cond M_Do         ; Повторить цикл по условию Cond
M_Exit
```

Пример:

Выполнить блок кода M_Do заданное число раз N.

```
M_Repeat                        ; Точка входа
          MOV r10, #N           ; Инициализация счетчика
                                ; числа циклов
M_Do      ...                   ; Исполнительный блок Do
M_Until
          SUBS r10, #1          ; Декрементировать счетчик циклов
                                ; с установкой флагов («S»)
          BNE M_Do              ; Повторить цикл при условии NE
M_Exit
```

Циклы Repeat…Until строятся исключительно по классической технологии с использованием команды условной передачи управления в конце тела цикла.

15.3.4.2 Оператор цикла While…Do (Пока… Выполнять)

Этот оператор цикла похож на предыдущий – рис. 15.3, *б*). Отличие состоит лишь в том, что условие выхода из цикла проверяется не в конце, а в начале цикла, перед выполнением тела цикла – исполнительного блока Do. Поэтому, если заданное условие изначально ложно, то исполнительный блок не будет выполнен ни разу. Заметьте, что в первой структуре исполнительный блок выполняется, по крайней мере, один раз.

В начале этой алгоритмической структуры располагается блок сравнения/тестирования, после него – исполнительный блок:

```
While (Cond)
     Do
End While
```

С точки зрения языка Ассемблер удобно запрограммировать эту структуру с использованием оппозитного условия /Cond. Если оно выполняется, то делается переход в конец структуры, в противном случае – выполняется тело цикла.

Шаблон алгоритмической структуры While...Do (Пока…Выполнять):

```
M_While                         ; Точка входа
; Инициализация параметров управления циклом
          …
M_If      CMPR Var1, RVar2      ; Сравнить переменные
                                ; управления циклом
          B /Cond M_Exit        ; Переход в конец цикла
                                ; по оппозитному условию
M_Do      …                     ; Тело исполнительного блока
          …                     ; Модификация переменных управления
                                ; циклом
          BM_If                 ; Повторить сравнение
M_Exit                          ; Точка выхода из цикла
```

Пример:
Очистить область ОЗУ в диапазоне адресовRam_0-Ram_k.

```
M_Clear                         ; Точка входа
; Инициализация параметров управления циклом
          LDR r0, =Ram_0
          LDR r1, =Ram_k
; Инициализация константы заполнения памяти (шаблона)
          MOV r2, #0
M_If      CMP r0, r1            ; Сравнить начальный и
                                ; конечный адреса
          BHI M_Exit            ; Переход в конец цикла
                                ; по оппозитному условию HI
                                ; (начальный адрес строго выше
                                ; конечного адреса)
M_Do      STRB r2, [r0], #1     ; Тело исполнительного блока
                                ; Очистка очередного байта в памяти
                                ; с авто-инкрементированием
                                ; указателя в r0
          B M_If                ; Повторить цикл
M_Exit                          ; Точка выхода из цикла
```

1) Вспомните, как выполняется инициализация регистров ЦПУ произвольными словами с использованием псевдокоманды Ассемблера LDR Rn, =Const? Почему в этом случае нельзя воспользоваться обычной командой пересылки MOV?

2) Почему мы применили команду сохранения байта в памяти STRB, а не команду сохранения полуслова или даже слова, ведь это быстрее?

3) Можно ли наш цикл переоформить с использованием команд условного выполнения, включенных транслятором в условный блок IT?

1) Область ОЗУ начинается с адреса 0x20000000. Такая «длинная» 32-разрядная константа предварительно размещается транслятором в ПЗУ в виде «литерального пула» (см. 10.3) и к ней организуется доступ с использованием команды загрузки регистра с относительной адресацией по текущему состоянию счетчика команд PC и смещению:

```
LDR Rd, [PC, #offset]
```

2) Если задан произвольный конечный адрес, то это может быть адрес одного байта, а не полуслова или слова.

3) Да. Такая возможность есть:

```
M_If      CMP r0, r1           ; Сравнить начальный
                               ; и конечный адреса
M_Do      STRBLS r2, [r0], #1  ; Тело исполнительного блока
                               ; выполняется
                               ; если условие LS истинно
                               ; («Ниже или то же самое» при
                               ; сравнении чисел без знака)
          BLS M_If             ; Повторить цикл,
                               ; если конечный адрес не достигнут
M_Exit                         ; Точка выхода из цикла
```

Заметьте, что при большом числе команд в исполнительном блоке традиционное решение может оказаться предпочтительнее.

15.3.4.3 Оператор цикла For… (Для…)

Это одна из наиболее часто используемых в программировании алгоритмических структур:

```
For Var going from Var_0 to Var_k, increment h
     Do
End For
```

Для переменной Var в диапазоне от начального значения Var_0 до конечного значения Var_k с шагом h, выполнить блок Do.

Эта структура наиболее просто реализуется с помощью цикла While …Do (Пока…Выполнять):

```
Var=Var_0
While (Var≠Var_1)
     Do
     Var=Var + h
End While
```

Переменной Var присваивается начальное значение Var_0 и выполняется цикл (While …Do), в теле которого дополнительно происходит увеличение значения переменной на величину шага h. Как только переменная достигает конечного значения, цикл прерывается. Организация такого цикла уже рассмотрена нами выше.

15.3.5 Составные (комбинированные) условия

До сих пор в блоках сравнения/тестирования мы использовали простые условия, которые являлись комбинацией флагов состояния процессора и могли применяться во всех без исключения командах условной передачи управления, а также в специальной команде блока условного выполнения IT (Если… То).

На практике часто приходится проверять сложные (комбинированные) условия, которые вычисляются как логические функции, в состав которых входит несколько так называемых *операторов отношения*, например:

```
If ((N> 0) AND (Var1<Var2)) Then…
```

Если число N положительное и одновременно значение переменной Var1 меньше значения переменной Var2, выполнить… Операторы отношения объединяются в логическую функцию, истинность или ложность которой проверяется. В отличие от языков высокого уровня, операторы отношения в Ассемблере *не поддерживаются*. Как же быть?

Проблема может быть решена несколькими способами.

15.3.5.1 Универсальное решение

1) Каждый оператор отношения реализуется на Ассемблере в виде отдельного условного оператора If…Then, результатом работы которого является исполнительный блок, вырабатывающий значение логической функции, соответствующей оператору отношения. Для нашего примера рассчитываются функции: y0=(N>0) и y1=(Var1<Var2). Назовем их *функциями отношений*.
2) Значения функций отношения сохраняются в битах одного из свободных регистров ЦПУ с помощью операций установки/сброса битов.
3) Любое комбинированное условие представляется в виде логической функции Y битовых входов y0…yk, являющихся функциями отношений. Для нашего примера справедливо: Y=y0 AND y1.
4) Логическая функция Y рассчитывается методами, описанными ранее для программной реализации логических контроллеров (см. 12.3). Наиболее удобно использовать для этого метод прямых вычислений с использованием логических команд с попутным сдвигом битов в старший разряд, являющийся как бы «битовым аккумулятором».
5) Последняя логическая команда должна устанавливать флаги в регистре статуса программы PSR (иметь суффикс «S») и являться по сути командой выработки текущего значения сложного комбинированного условия (True или False).
6) Оператором If…Then проверяется значение этого сложного условия: MI – (Истина), PL – (Ложь).

Метод не имеет никаких ограничений на типы логических операций, которыми связываются между собой операторы отношений (И, ИЛИ, Исключающее ИЛИ, И НЕ и т.д.) и на число таких операций.

15.3.5.2 Последовательное выполнение блоков сравнения

Второй способ предполагает последовательное выполнение блоков сравнения по числу операторов отношения, входящих в комбинированное условие, связанных между собой логическими операциями «И» или «ИЛИ», наиболее часто применяемыми на практике.

15.3.5.3 Конъюнкция нескольких логических условий

Предположим, что проверяется «Логическое И» (*конъюнкция*) нескольких операторов отношения, каждый из которых соответствует одному из прямых или оппозитных условий, вырабатываемых процессором:

```
If (Cond_A AND Cond_B AND Cond_C)
Then
     Do
EndIf
```

В языках высокого уровня эта структура соответствует вложенным операторам If…Then, а в Ассемблере – нескольким последовательно выполняемым блокам сравнения/тестирования – рис. 15.4. На рис. 15.4, *а*) представлена граф-схема алгоритма проверки трех прямых условий по «И», а на рис. 15.4, *б*) и *в*) – варианты граф-схем алгоритмов проверки трех условий, одно из которых (Cond_B) – оппозитное.

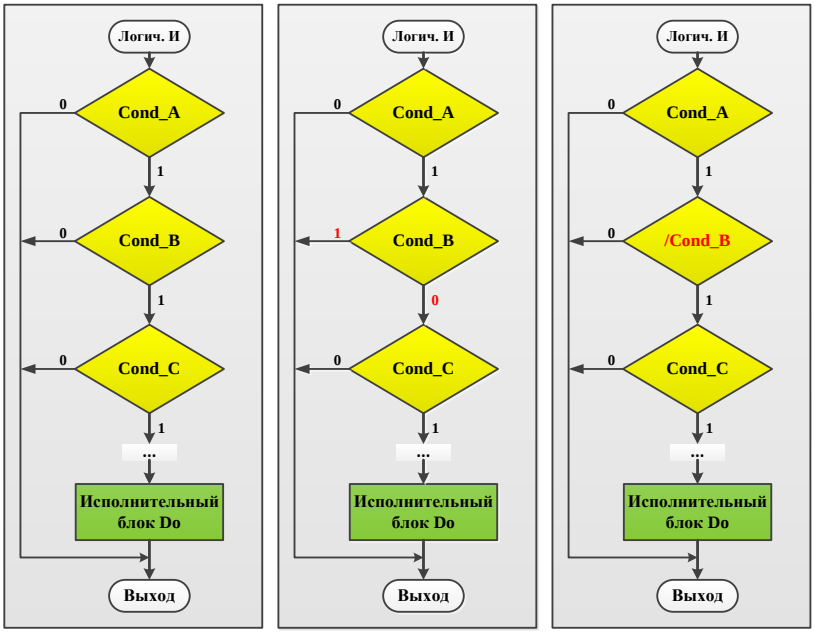


Рис. 15.4 Оператор If…Then с комбинированными условиями по «Логическому И»: *а*) все условия прямые; *б*) и *в*) одно из условий (Cond_B) – оппозитное

Если первое условие ложно, то второе условие вообще не проверяется, и выполняется досрочный выход из структуры в обход исполнительного модуля Do (по оппозитному условию /Cond_A. Аналогично проверяются и остальные условия, но только в том случае, когда предыдущие условия оказались истинными. Таким образом, исполнительный блок выполняется только тогда, когда все условия, объединенные по «Логическому И», окажутся истинными.

Шаблон реализации структуры If…Then (Если… То) с несколькими условиями, объединенными «Логическим И»:

```
M_If_AND                              ; Точка входа
M_IF_A          CMP RVar, RVar_A      ; Вычисление условия Cond_A
                B /Cond_A M_Exit      ; Выход по оппозитному условию
M_IF_B          CMPR Var, RVar_B      ; Вычисление условия Cond_B
                B /Cond_B M_Exit      ; Выход по оппозитному условию
M_IF_C          CMP RVar, RVar_C      ; Вычисление условия Cond_C
                B /Cond_C M_Exit      ; Выход по оппозитному условию
M_Do            ...                   ; Исполнительный блок
M_Exit          ...
```

Как показано на рис. 15.4, одно или несколько условий, объединенных «Логическим И», может быть оппозитным (для примера Cond_B). При этом граф-схема программной реализации структуры сохраняется, за исключением того, что досрочный

выход из блока сравнения с инверсным условием выполняется по прямому, а не по оппозитному условию.

Аналогичным образом создаются шаблоны циклов Repeat… Until и While… Do с комбинированными условиями выполнения цикла.

15.3.5.4 Дизъюнкция нескольких логических условий

Пусть требуется реализовать оператор If…Then с «Логическим ИЛИ» нескольких условий:

```
If (Cond_A OR Cond_B OR Cond_C)
Then
Do
EndIf
```

Если хотя бы одно из условий истинно, то блок Do выполняется. Этой структуре соответствует граф-схема алгоритма на рис. 15.5, *а*.

Ее недостаток: после блока сравнения по условию Cond_C придется использовать дополнительную команду безусловной передачи управления на выход структуры в обход исполнительного блока Do. Оптимизированная структура на рис. 15.5, *б* отличается тем, что в последнем блоке сравнения/тестирования выполняется переход на выход структуры по альтернативному условию. В результате – исключается команда безусловной передачи управления.

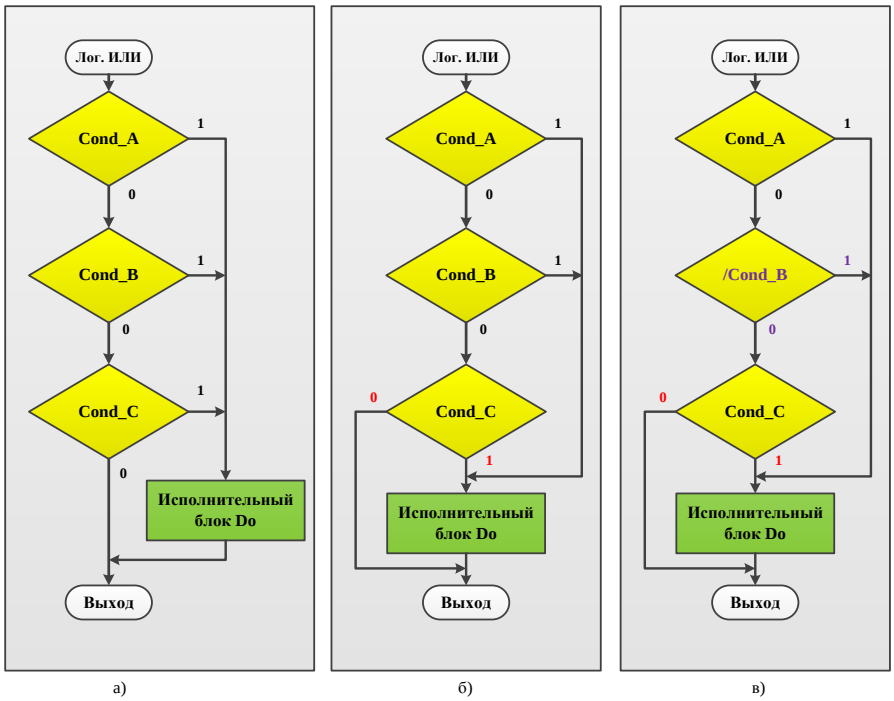


Рис. 15.5 Граф-схема реализации оператора If…Then с несколькими условиями по «ИЛИ»
а) общий случай; *б*) оптимизированная структура; *в*) с одним оппозитным условием

Если одно из проверяемых условий является инверсным, то алгоритмическая структура сохраняется, только это условие заменяется оппозитным – рис. 15.5, *в*.

Шаблон структуры If…Then (Если… То) с несколькими условиями, объединенными «Логическим ИЛИ»:

```
M_If_OR                              ; Точка входа
M_IF_A          CMP RVar, RVar_A     ; Вычисление условия Cond_A
                B Cond_A M_Do        ; Переход на вход блока Do
M_IF_B          CMP RVar, RVar_B     ; Вычисление условия Cond_B
                B /Cond_B M_Do       ; Переход на вход блока Do
M_IF_C          CMP RVar, RVar_C     ; Вычисление условия Cond_C
                B /Cond_C M_Exit     ; Выход по оппозитному условию
M_Do            ...                  ; Исполнительный блок Do
M_Exit          ...
```

Аналогичная проверка нескольких условий по ИЛИ может быть выполнена и в шаблонах структур операторов циклов.

15.3.5.5 Использование законов алгебры логики при программировании

При программировании логических функций рекомендуем пользоваться законами Булевой алгебры, как для возможного их упрощения, так и для получения более удобной для реализации на Ассемблере граф-схемы алгоритма.

Например, таблица истинности двух функций «И» (AND) и «ИЛИ» (OR) для всех возможных значений логических выражений A и B, а также их инверсных значений $\overline{A}$ и $\overline{B}$ показана ниже. Черта сверху над логической переменной означает ее инверсию:

Таблица 15.3 Таблица истинности двух функций

A	B	$\overline{A}$	$\overline{B}$	$A \cdot B$	$\overline{A} \cdot \overline{B}$	$A + B$	$\overline{A} + \overline{B}$
0	0	1	1	0	1	0	1
0	1	1	0	0	0	1	1
1	0	0	1	0	0	1	1
1	1	0	0	1	0	1	0

Эта таблица иллюстрирует хорошо известные правила Де-Моргана:

$\boldsymbol{A + B = \overline{\overline{A} \cdot \overline{B}}}$ – Логическая сумма двух логических выражений равна инверсии от логического произведения инверсных значений этих выражений;

$\boldsymbol{A \cdot B = \overline{\overline{A} + \overline{B}}}$ – Логическое произведение двух логических выражений равно инверсии от логической суммы инверсных значений этих выражений;

$\boldsymbol{\overline{A} \cdot \overline{B} = \overline{A + B}}$ – Логическое произведение инверсных значений двух логических выражений равно инверсии от логической суммы этих выражений;

$\boldsymbol{\overline{A} + \overline{B} = \overline{A \cdot B}}$ – Логическая сумма инверсных значений двух логических выражений равна инверсии логического произведения этих выражений;

Расширяя эти правила на любое число логических переменных, получим:

1) Инверсия нескольких логических условий, объединенных операцией «Логическое ИЛИ», эквивалентна «Логическому И» оппозитных условий.
2) Инверсия нескольких логических условий, объединенных операцией «Логическое И», эквивалентна «Логическому ИЛИ» оппозитных условий.

1) Как будет выглядеть структура цикла типа Repeat…Until при последнем комбинированном условии повторения цикла?

2) Требуется проверить текущее значение переменной со знаком в регистре r0 на попадание в допустимый коридор значений Max= (r1), Min= (r2), включая максимальное и минимальное допустимые значения. При истинности условия нужно установить в регистре r5 значение 0x55, в противном случае 0xAA. Предложите вариант решения.

3) Как изменится этот программный модуль, если границы отрезка должны быть исключены из допустимого коридора?

4) Имеет ли смысл модифицировать этот программный модуль с использованием технологии условного выполнения команд?

5) Требуется проверка условия $\overline{\overline{Cond\_A} + Cond\_B + \overline{Cond\_C}}$. Упростите его для более удобной программной реализации.

6) Требуется проверка условия $\overline{\overline{Cond\_A} \cdot Cond\_B \cdot \overline{Cond\_C}}$. Упростите его для более удобной программной реализации.

1) Один из возможных вариантов структуры:

```
M_Repeat              ; Точка входа (начало цикла)
; Инициализация параметров управления циклом
;           ...
M_Do        ...       ; Исполнительный блок Do
                      ; В том числе модификация
                      ; переменной
                      ; управления циклом (например, Var)
M_Until
          CMP RVar, RVar_A    ; Сравнение переменных
          B /Cond_A M_Exit    ; Выход по оппозитному условию
          CMP RVar, RVar_B    ; Сравнение переменных
          B Cond_B M_Exit     ; Выход по прямому условию
          CMP RVar, RVar_C    ; Сравнение переменных
          B Cond_C M_Do       ; Повторить цикл «Логическому И»
                              ; всех условий
M_Exit
```

2) Задача сводится к реализации оператора If ((Var≤Max) AND (Var≥Min)) Then… с двумя условиями по «Логическому И». Используем сокращенную форму этого оператора:

```
M_If_AND                      ; Точка входа
              MOV r5, #0xAA   ; Предположим, что условие не
                              ; выполняется
M_If_Max      CMP r0, r1      ; Сравнить с Max
                              ; (Прямое условие LE)
              BGTM_Exit       ; Выход по оппозитному условиюGT
M_If_Min      CMP r0, r2      ; Сравнить с Min
                              ; (Прямое условие GE)
              BLTM_Exit       ; Выход по оппозитному условиюLT
M_Do          MOV r5, #0x55   ; Исполнительный блок
M_Exit        ...
```

3) Следует изменить парные условия: вместо (LE/GT) использовать (LT/GE), а вместо (GE/LT) – (GT/LE).
4) Нет. Мы проверяем комбинированное условие по «Логическому И» двух «простых» условий, которые формируются процессором. В командах условного выполнения можно использовать только «простые» условия.
5) Заменим инверсию «Логического И» условий «Логическим ИЛИ» оппозитных условий $Cond\_A + \overline{Cond\_B} + Cond\_C$.
6) Заменим инверсию «Логического ИЛИ» условий «Логическим И» оппозитных условий $Cond\_A \cdot \overline{Cond\_B} \cdot Cond\_C$.

15.4 Преимущества реализации алгоритмов с использованием условно выполняемых команд

15.4.1 Важные положения

Одним из самых существенных преимуществ системы команд ARM-процессоров по сравнению с системами команд процессоров других фирм является возможность *условного выполнения практически всех команд процессора*. В этом случае любая команда просто снабжается соответствующим *суффиксом условного выполнения cond.* Если указанное в команде условие истинно, она выполняется, если нет, – пропускается (не выполняется, фактически заменяется командой NOP, нет операции). При этом на ее «выполнение или невыполнение» всегда расходуется один такт процессорного времени.

Для сравнения: если алгоритм решения задачи построен с использованием команд безусловной или условной передачи управления, то каждая такая команда будет выполняться не за такт, а уже за три. Дело в том, что выполнение команд передачи управления связано с очисткой конвейера («смывом» всех, ранее загруженных на него команд) и *полной перезагрузкой конвейера* новыми командами. На это нужны дополнительные два такта процессорного времени. Поэтому алгоритм, реализованный с использованием технологии условного выполнения команд ARM-процессоров, будет быстрее выполняться и, к тому же, будет более компактным, занимая меньший объем памяти, что также важно для встраиваемых микроконтроллерных систем управления оборудованием.

Напомним:

- Для того, чтобы какая-то команда могла быть выполнена условно, в любой из предшествующих ей команд нужно использовать дополнительный суффикс установки флагов «S». Это касается всех команд процессора, кроме команд сравнения CMP, CMN и команд тестирования TST, TEQ, назначение которых в том и состоит, чтобы формировать флаги в регистре статуса программы пользователя PSR.
- Целесообразно разрабатывать алгоритмы так, чтобы команды условного выполнения образовывали блоки до 4-х команд с альтернативными (противоположными) условиями выполнения (EQ, NE), (GT, LE), (MI, PL) и т.д. В этом случае транслятор будет автоматически добавлять к блоку условного выполнения специальную команду «Если То» IT, предваряющую блок и обеспечивающую быстрое выполнение каждой команды блока за один такт (см. 12.4).
- В том случае, если разработать алгоритм с оппозитными условиями не получается, – ничего страшного. В этом случае транслятору придется перед каждой командой

условного выполнения генерировать отдельную команду IT. Время выполнения алгоритма несколько замедлится, но, все равно будет меньше, чем при использовании команд условной передачи управления.

15.4.2 Пример программной реализации алгоритма поиска наибольшего общего делителя двух целых чисел

15.4.2.1 Алгоритм Эвклида

Этот алгоритм является классическим примером, широко используемым при изучении практически всех языков программирования. Сравниваются два целых числа «a» и «b». До тех пор, пока числа разные, из большего числа вычитается меньшее. Как только числа стали одинаковы, то полученное в них значение и будет содержать наибольший общий делитель чисел НОД – (Greatest Common Divisor).

Проверим работоспособность алгоритма Эвклида на примере двух произвольных чисел a=22 и b=4. Для нахождения результата придется выполнить 7 проходов (см. табл. 15.4).

Таблица 15.4 Иллюстрация работы алгоритма Эвклида

Номер прохода	a=	b=	a≠b?	a>b?	Действие:
1	22	4	Да	Да	a=a-b=22-4=18
2	18	4	Да	Да	a=a-b=18-4=14
3	14	4	Да	Да	a=a-b=14-4=10
4	10	4	Да	Да	a=a-b=10-4=6
5	6	4	Да	Да	a=a-b=6-4=2
6	2	4	Да	Нет	b=b-a=4-2=2
7	2	2	Нет		НОД=a=2

Как видите, алгоритм работает!

15.4.2.2 Программа на языке высокого уровня C/C++

```
int gcd (int a, int b)
{
     while (a!=b)
     {
          if (a>b)
               a=a-b;
          else
               b=b-a;
     }
     Return a;
}
```

В ней используются операторы языка Выполнять Пока (While…Do) и Если…То…В Противном случае (If…Then…Else). Цикл повторяется до тех пор, пока условие (a≠b) истинно. В противном случае цикл завершается и результат возвращается содержимым переменной a.

15.4.2.3 Классическое решение на Ассемблере с использованием команд условной передачи управления

В проекте CPU_19 программа приложения MyProg_19.s состоит из основной программы и подпрограммы поиска наибольшего общего делителя двух целых 32-разрядных чисел без знака GCD, значения которых передаются

в подпрограмму содержимым регистров r1 и r2. Основная программа инициализирует регистры r1, r2 начальными данными (передает параметры в подпрограмму) и вызывает подпрограмму GCD.

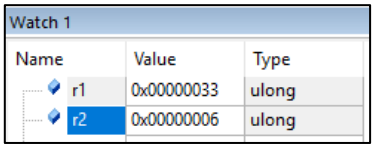
```
24  ;*******************************************************
25  ; Подпрограмма поиска наибольшего общего делителя
26  ; двух целых чисел НОД
27  ; Входы: r1, r2 – исходные целые числа без знака
28  ; Выход: НОД в регистре r1 и в регистре r2
29  ;*******************************************************
30  GCD
31  ; Сравнить исходные числа
32          CMP r1,r2
33  ; Если числа одинаковы, выйти из подпрограммы
34          BXEQ LR
35  ; Если первое число больше второго, вычесть из него второе
36  ; и повторить цикл расчета
37          BHI   HIGH_1
38  ; В проитивном случае вычесть из второго числа первое
39  ; и повторить цикл расчета
40          SUB r2,r1
41          B GCD
42  HIGH_1
43          SUB r1,r2
44          B GCD
45  ; Конец подпрограммы
46  ;*******************************************************
```

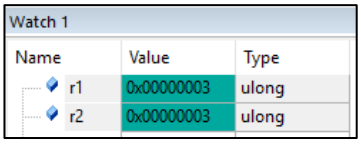
В подпрограмме выполняется сравнение чисел. Если они одинаковы, выполняется досрочный выход из подпрограммы (результат автоматически остается в r1 и в r2). В противном случае анализируется, какое из двух чисел больше. При этом используется условие HI («Выше»), применяемое исключительно для сравнения чисел без знака. По этому условию выполняется передача управления на фрагмент подпрограммы, в котором из первого числа вычитается второе и сравнение выполняется вновь. В противном случае из второго числа вычитается первое и цикл сравнения продолжается.

```
 9  MyProg
10  ; Основная программа поиска наибольшего общего делителя
11  ; двух целых чисел без знака в регистрах r1, r2
12
13  ; Передать параметры в подпрограмму – два целых числа
14  ; без знака
15          MOV r1,#34
16          MOV r2,#16
17  ; Вызвать подпрограмму
18  M1
19          BL GCD
20  ; Повторить вызов подпрограммы для других значений
21  ; в регистрах, модифицированных при отладке
22          B M1
```

Реализация алгоритма классическая, с использованием команд условной и безусловной передачи управления. Основная программа предельно проста. В ней выполняется вызов подпрограммы, и цикл вызова повторяется. Это позволяет поставить точку останова на метке M1 и после каждого вызова подпрограммы изменять исходные числа в регистрах r1, r2 для следующего вызова.

Рекомендуем открыть окно наблюдаемых переменных Watch и включить в него в качестве переменных регистры r1, r2. Целесообразно также выполнить в командном окне среды µVision операцию установки десятичного формата представления информации в окне Watch: Radix=10.





Каждый раз, запуская программу в режиме прогона с точки останова, Вы сможете наблюдать в окне Watch результат поиска наибольшего общего делителя для текущей пары чисел. После останова поменяйте содержимое регистров r1, r2 (прямо в окне Watch) и запустите программу вновь. Таким образом Вы сможете отладить программу на любом наборе переменных. Как видите, для исходных чисел 33 и 6 наибольший общий делитель равен 3.

15.4.2.4 Реализация алгоритма с использованием условно выполняемых команд

Модернизируем подпрограмму GCD (см. MyProg_19_1.s), используя две подряд команды вычитания с альтернативными условиями «HI» (Выше) и «LO» (Ниже). На равенство числа уже были проверены в команде условного выхода из процедуры. Как видите, подпрограмма стала и более компактной, и более понятной. Полностью исключены команды условной передачи управления.

Отладка на любых наборах переменных подтверждает правильность работы программы. Убедитесь в этом сами.

```
24  ;*******************************************************
25  ; Подпрограмма поиска наибольшего общего делителя
26  ; двух целых чисел НОД
27  ; Входы: r1, r2 - исходные целые числа без знака
28  ; Выход: НОД в регистре r1 и в регистре r2
29  ; Использование команд условного выполнения
30  ;*******************************************************
31  GCD
32  ; Сравнить исходные числа
33          CMP r1,r2
34  ; Если числа одинаковы, выйти из подпрограммы
35          BXEQ LR
36  ; Если первое число больше второго, вычесть из него второе
37          SUBHI r1,r2
38  ; В противном случае вычесть из второго числа первое
39          SUBLO r2,r1
40  ; Повторить цикл расчета
41          B GCD
42  ; Конец подпрограммы
43  ;*******************************************************
```

1) Исследуйте файл листинга программы MyProg_19_1.s. Обратите внимание на то, как кодируются команды условного выполнения. Для сравнения откройте те же фрагменты программы в окне Дизассемблера. Теперь кодировка стала более понятной?

2) Почему каждая команда условного выполнения попала в свой собственный блок условного выполнения?
3) Почему транслятор заменил команду SUBLO на команду SUBCC?
4) Измените программу так, чтобы две последние команды попали в один блок условного выполнения. Выполните ее отладку.

1) Все команды условного выполнения предваряются командой блока условного выполнения IT.
2) Дело в том, что у нас условия выполнения команд не являются противоположными (альтернативными). Поэтому, каждая команда попадает в свой собственный блок условного выполнения. Условия HI (>) и LO(<), строго говоря, не альтернативные. Альтернативными могли бы быть условия HI (>) и LS (<=).
3) Это эквивалентные суффиксы условного выполнения – CC означает: флаг C очищен. Очистка флага C выполняется только при возникновении заема, когда из меньшего числа вычитается большее (для чисел без знака).
4) Замените в последней команде условие LO на условие LS (Меньше или то же самое) – см. MyProg_19_2.s. Это два альтернативных условия, и транслятор может объединить обе команды в один блок, что эффективнее с точки зрения скорости выполнения программы. На равенство проверка выполняется выше. Поэтому логика алгоритма не нарушается, а реализация становится более эффективной.

Специальная команда Ассемблера IT (Если То) позволяет создавать существенно более производительный и компактный код, чем при использовании обычных арифметических и логических команд вместе с командами условной и безусловной передачи управления. Хорошие трансляторы имеют встроенные возможности оптимизации кода с

использованием команды генерации IT-блоков. Пользуйтесь этими возможностями при разработке программного обеспечения на языках высокого уровня.

Блоки условного выполнения IT работают максимум с 4-мя командами процессора, поэтому, при большом числе команд в исполнительных модулях, требующих условного выполнения, эффективнее может оказаться классическая технология использования команд условной передачи управления.

15.5 Множественный выбор

Операторы множественного выбора обеспечивают выбор одной альтернативы из множества возможных на основе текущего значения «переменной-переключателя». Переменная-переключатель, как правило, является целочисленной переменной. Для каждого из ее возможных значений выполняется один исполнительный блок.

Алгоритмическая структура оператора Switch с переменной-переключателем Var представлена ниже:

```
Switch (Var)
{     Case Var_0      : Do_0
      Case Var_1      : Do_1
      Case Var_2      : Do_2
      (...)
      Default         : Do_default
}
End Switch
```

В зависимости от значения переменной-переключателя Var выполняется соответствующий этому значению исполнительный блок команд. Если значение переключателя не соответствует ни одной из возможных версий, то выполняется блок «по умолчанию», который, в частном случае, может быть «пустым» блоком.

Такие структуры поддерживаются во всех языках программирования высокого уровня, например, в языке C/C++ с помощью оператора Switch. Причем, в качестве переменной-переключателя могут использоваться даже символьные и строковые переменные.

15.5.1 Структура множественного выбора с использованием оператора If…Then (Если…То)

Интуитивно понятный (но далеко не оптимальный) метод реализации такой алгоритмической структуры состоит в использовании структуры If…Then (Если…То), уже рассмотренной нами:

```
If Var = Value_0
      Then Do_0
Else If Var = Value_1
           Then Do_1
      Else If Var = Value_2
           Then Do_2
           (… ит.д.)
                Else Do_default
```

Продемонстрируем ее реализацию на Ассемблере, ограничиваясь всего тремя возможностями выбора. Предположим, что значение переключателя получено не из памяти или порта ввода (как обычно), а является предварительно определенной константой. Для каждой из возможных альтернатив будем вызывать соответствующую ей подпрограмму и после ее выполнения передавать управление на выход.

Шаблон переключателя выбора Switch:

```
M_Switch            ; Точка входа в структуру переключателя
     MOV r0, #Var   ; Загрузить в регистр-переключательr0
; значение «ключа» выбора альтернативы
; 0-я альтернатива?
M_0  CMP r0, #Value_0
     BNE M_1        ; Нет, проверить следующую альтернативу
     BL Do_0        ; Да, выполнить подпрограмму Do_0
     BM_Exit        ; Досрочный выход из структуры Switch
; 1-я альтернатива?
M_1  CMP r0, #Value_1
     BNE M_2        ; Нет, проверить следующую альтернативу
     BL Do_1        ; Да, выполнить подпрограмму Do_1
     B M_Exit       ; Досрочный выход из структуры Switch
; 2-я альтернатива?
M_2  CMP r0, #Value_2
     BNE M_Default  ; Нет, выполнить блок по умолчанию
     BL Do_2        ; Да, выполнить подпрограмму Do_2
     B M_Exit       ; В конец структуры Switch
M_Default
     BL Do_default
M_Exit
```

Эта структура не лишена недостатков:

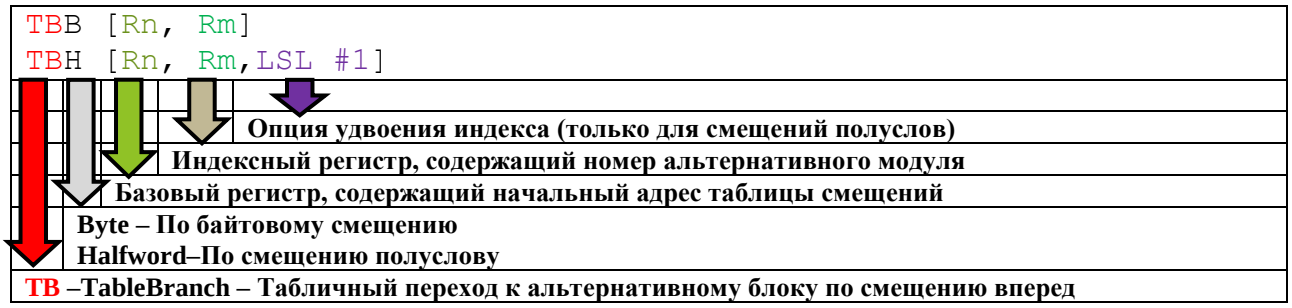
- Текст программного модуля довольно длинный – его нужно писать внимательно, чтобы избежать ошибок.
- Неудобно модифицировать такую структуру, если придется добавлять еще одну или несколько альтернатив.
- Время выполнения каждой из альтернатив разное. Чем дальше расположена альтернатива от начала блока, тем больше процессорного времени потребуется, чтобы до нее «добраться». Это неудобно и нежелательно в системах управления.

15.5.2 Команды перехода к альтернативным блокам по таблице смещений

Разработчики системы команд процессоров ARM Cortex-M3/M4 предусмотрели специальные команды передачи управления альтернативным программным модулям по так называемой *таблице смещений до точек входа в модули* и *индексу-селектору*, задающему номер модуля, подлежащего выполнению.

Имеются два варианта команд табличного перехода TBB и TBH. Первый предполагает использование коротких (байтовых) смещений вперед относительно текущего значения в счетчике команд, а второй – использование16-разрядных смещений-полуслов. В обеих командах применяется относительная адресация по текущему содержимому счетчика команд PC (адресу следующей после TBB или TBH команды) и считанному из таблицы смещений значению: (PC)=(PC)+Offset.

Номер подлежащего выполнению программного блока должен быть предварительно записан в индексный регистр Rm. Команды имеют следующий синтаксис:



Адрес размещения таблицы смещений задается в базовом регистре Rn. Таблица смещений должна быть выровнена в кодовой памяти по адресу, кратному четырем (адресу слова). В нее должно записываться уменьшенное вдвое значение смещения. При выполнении команды TBB или TBH по индексу (номеру) программного модуля в регистре индекса Rm, значение смещения считывается из таблицы смещений и автоматически удваиваться. Это сделано для того, чтобы расширить область возможного размещения альтернативных программных модулей или подпрограмм: для команды TBB–до 0,5 К байт, а для команды TBH – до 128 К байт.

В первом случае (TBB) число возможных модулей по выбору невелико. Это должны быть простые модули, состоящие из небольшого числа команд. Во втором случае (TBH) блок альтернативных модулей может иметь существенно большую длину, которой обычно достаточно для размещения даже сложных операторов Switch.

Для задания базового адреса таблицы смещений используется или один из регистров ЦПУRn, или счетчик команд PC. В первом случае таблица смещений может размещаться вслед за группой альтернативных модулей, во втором – сразу после команды табличного перехода – перед группой альтернативных модулей.

Естественно, что таблица смещений до альтернативных модулей должна быть предварительно запрограммирована и размещена в кодовой памяти системы, а ее начальный адрес должен быть загружен в регистр базового адреса Rn перед выполнением команд TBB/TBH.

Предполагается следующий порядок работы с командами табличного альтернативного выбора:

1) Программируются все возможные альтернативные блоки. Каждый из них помечается меткой, соответствующей его начальному адресу: CASE0, CASE1…
2) Программируется таблица смещений от места расположения команды TBB/TBH до точек входа в альтернативные блоки. Величины смещений в таблице определяются как половины фактических смещений.
3) Перед выполнением команды TBB/TBH инициализируются значения регистров Rn (начальным адресом таблицы смещений) и Rm (номером нужного программного блока). Если в качестве базового регистра указывается PC, то предполагается, что таблица смещений располагается сразу после команды табличного перехода.
4) Каждый альтернативный программный блок после выполнения передает управление в конец структуры Switch.
5) Контроль допустимых индексов выполняется перед структурой переключателя. Если индекс выходит за допустимый диапазон, управление передается блоку по умолчанию.

Продемонстрируем, как можно пользоваться этими достаточно сложными комплексными командами процессора (проект CPU_19). В программе пользователя MyProg_19_TBB.s структура блока альтернативного выбора начинается с метки Switch и заканчивается меткой Switch_End. Перед обращением к структуре нужно указать номер альтернативного блока, подлежащего выполнению, – инициализировать селектор выбора – индексный регистр r1.

```
12  ; Задать номер блока, подлежащего выполнению (в r1)
13          MOV r1,#0
14  Switch
15  ; Проверить, не превышает ли номер максимально допустимый?
16          CMP r1,#3
17  ; Если да, то выполнить блок команд по умолчанию
18          BHI Default
19  ; Задать начальный адрес таблицы байтовых смещений (в r0)
20          ADR r0, Tab_Byte_Offset
21  ; Выполнить команду по номеру блока в индексном регистре r1
22          TBB [r0,r1]
23  ; Сектор размещения альтернативных блоков команд
24  CASE0
25          MOV r2,#(0+10)
26          B Switch_End
27  CASE1
28          MOV r2,#(1+10)
29          B Switch_End
30  CASE2
31          MOV r2,#(2+10)
32          B Switch_End
33  CASE3
34          MOV r2,#13
35          B Switch_End
36  Default
37          MOV r2,#0xFF
38  ; Конец структуры множественного выбора Switch
39  Switch_End
40  ;       ...
41  ; Повторить оператор-переключатель Switch для отладки
42          B Switch
43
44  ; Выравнивание таблицы смещений по границе полного слова
45          ALIGN
46  ; Таблица байтовых смещений от команды TBB до начала блоков CASE
47  ; При выполнении команды TBB смещения, полученные из таблицы
48  ; будут автоматически удваиваться
49  Tab_Byte_Offset
50          DCB 0
51          DCB (CASE1-CASE0)/2
52          DCB (CASE2-CASE0)/2
53          DCB (CASE3-CASE0)/2
```

Структура начинается с проверки допустимости номера выполняемой команды. Если он превышает максимальный (в данном случае 3), то управление передается на блок по умолчанию Defaultи далее – на выход.

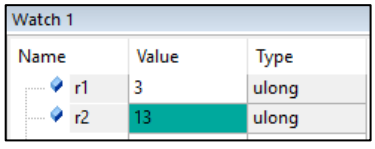
После инициализации регистра-переключателя выбора r1 выполняется загрузка базового регистра r0 адресом начала таблицы байтовых смещений от начала команды TBB до соответствующего альтернативного блока команд CASEn.

Далее следует собственно команда табличного вызова альтернативы TBB, а сразу после нее размещается блок альтернативных команд.

Для иллюстрации выполнения альтернативного блока в отладчике в каждой команде номер команды увеличивается на 10 и сохраняется в регистре r2. Все альтернативные блоки заканчиваются командой передачи управления в конец структуры Switch.

Перед началом размещения таблицы смещений требуется директива выравнивания адреса по границе полного слова ALIGN. Вычисление смещений «поручается» транслятору с Ассемблера, так как именно он «знает» фактические значения адресов входа во все альтернативные блоки и может сам вычислить такое выражение, как ((Case2-Case0)/2).

Для упрощения отладки поставьте точку останова в начале структуры Switch (строка 16) и откройте окно наблюдаемых переменных с текущими значениями переключателя (регистр r1) и выходной переменной оператора Switch (регистр r2). Для удобства анализа представьте переменные в десятичной системе счисления – выполните команду Radix=10 в командном окне отладчика. Приведем пример выполнения команды n с номером 3: Убедитесь, что структура выбора альтернативы работает при любом допустимом номере команды (0-3), а при недопустимом – возвращает результат по умолчанию 0xFF.



1) Как будет работать структура альтернативного выбора, если блок по умолчанию отсутствует и задан номер команды, превышающий максимальный, например, 4? Проверьте в отладчике.

2) Можно ли в блоках альтернативного выполнения вместо команды загрузки регистра r2 использовать вызов соответствующей подпрограммы. Попробуйте.

3) Модифицируйте текущую программу с использованием команды TBH, имея в виду выделение большего места под расположение альтернативно выполняемых блоков.

1) Программа будет работать непредсказуемо. Последует обращение к таблице смещений с не определенным значением (отсутствующим). Если это место в памяти содержит нули, то будет выполняться нулевая альтернатива, а если какие-то данные? Они будут интерпретированы как смещение и будет сделан переход по случайному адресу в памяти. Не допускайте при программировании неопределенностей!

2) Да. Именно так обычно и делается, чтобы уменьшить объем памяти, занимаемый альтернативными блоками. При этом можно использовать более простую команду TBB с байтовыми смещениями.

3) При написании программы обратите внимание на то, что таблица смещений будет содержать теперь не байты, а полуслова. Для их резервирования потребуется директива DCW вместо директивы DCB. Кроме того, для получения смещения адреса при доступе к таблице смещений, придется предварительно удвоить номер выполняемой команды (в таблице полуслова, а не байты). Если не справились – изучите программу MyProg_19_TBH.s в том же каталоге CPU_19. Подключите файл к проекту и выполните его отладку.

Отличительной особенностью команд табличного выбора альтернативы с регистром PC в качестве базового регистра является то, что в этом случае инициализация базового регистра не требуется. В процессе выполнения команды TBB/TBH счетчик команд будет содержать адрес очередной команды, который и будет начальным адресом размещения таблицы смещений.

При этом порядок размещения исполнительных блоков и таблицы смещений меняется: вначале размещается таблица смещений, а затем – альтернативные блоки. Проиллюстрируем это на примере программы MyProg_19_TBH_PC.s.

Выполните отладку этой программы и убедитесь в работоспособности структуры Switch. По мнению авторов, такое размещение программных модулей внутри структуры более логично. Структура заканчивается блоком по умолчанию, а не таблицей смещений, как в предыдущем случае.

Обратите внимание, что при использовании смещений-полуслов, каждое из них занимает в памяти два байта. Поэтому, для доступа к нужному смещению значение индекса (номера альтернативы) должно быть удвоено – операция LSL #1 в команде TBH.

Величины смещений, записываемые в таблицу смещений, по-прежнему должны быть равны половине адресного расстояния от команды TBH (следующей после нее команды) до адреса расположения альтернативного модуля. Они будут автоматически удвоены в процессе выполнения команды табличной передачи управления.

Как видите, после выполнения любого из альтернативных блоков управление передается в конец оператора множественной альтернативы. Программа может быть продолжена дальше. Технология модульного программирования не нарушается: оператор множественной альтернативы имеет один вход и один выход.

```
 7  MyProg
 8  ; Команда TBH – табличной передачи управления
 9  ; по смещению – полуслову на альтернативный блок команд
10  ; Особенность – в качестве базового регистра используется PC
11  ; Поэтому, таблица смещений располагается сразу за командой TBH
12  ; Аналог оператора Switch в языке высокого уровня C/C++
13
14  ; Задать номер выполняемого блока в индексном регистре r1
15          MOV r1,#0
16  Switch
17  ; Проверить, не выходит ли номер за допустимый диапазон?
18          CMP r1, #3
19  ; Если да, то выполнить блок команд по умолчанию
20          BHI Default
21  ; Выполнить команду по номеру блока в индексном регистре r1
22          TBH [PC,r1, LSL #1]
23  ; Таблица смещений-полуслов от команды TBH до начала блоков CASE
24  Tab_Hword_Offset
25          DCW (CASE0-Tab_Hword_Offset)/2
26          DCW (CASE1-Tab_Hword_Offset)/2
27          DCW (CASE2-Tab_Hword_Offset)/2
28          DCW (CASE3-Tab_Hword_Offset)/2
29  ; Альтернативные блоки, которые должны быть
30  ; выполнены по номеру операции n
31  CASE0
32          MOV r2,#(0+10)
33          B Switch_End
34  CASE1
35          MOV r2,#(1+10)
36          B Switch_End
37  CASE2
38          MOV r2,#(2+10)
39          B Switch_End
40  CASE3
41          MOV r2,#(3+10)
42          B Switch_End
43  Default MOV r2,#0xFF
44  Switch_End
45          B Switch
```

Ассемблер ARM-процессоров по своим возможностям приближается к языкам высокого уровня. Он позволяет относительно просто реализовать большинство операторов языков высокого уровня, в том числе оператор альтернативного выбора. При этом обеспечивается максимальное быстродействие всех алгоритмических структур и поддерживается концепция модульного структурного программирования.

15.5.3 Универсальный интерпретатор команд

Анализируя технологию программной реализации дискретных управляющих автоматов (см. 12.7), мы уже обратили внимание читателя на то, что программа управляющего автомата по существу представляет собой совокупность отдельных подпрограмм – интерпретаторов вершин графа автомата. В зависимости от номера состояния автомата вызывается и запускается соответствующая подпрограмма. На самом деле таких задач очень много, например: запуск нужной функции (или режима работы устройства) в зависимости от номера, указанного пользователем (оператором) в меню пульта управления; запуск нужной функции в зависимости от команды, полученной от системы управления верхнего уровня по одному из интерфейсов и т.д. В любом случае задача сводится к запуску подпрограммы по ее номеру.

Универсальное решение состоит в том, что все начальные адреса подпрограмм (точки входа в подпрограммы) сохраняются в специальной таблице – таблице точек входа в подпрограммы. Это будет таблица 32-разрядных кодов, сохраняемых в памяти с помощью директивы резервирования слов в кодовой памяти DCD. Назовем эту таблицу таблицей начальных адресов подпрограмм Table_Addr (векторов перехода на подпрограммы).

Предположим, что регистр r0 является *базовым регистром* и содержит начальный адрес таблицы векторов, а регистр r1 – *индексным регистром* и содержит номер подлежащей выполнению подпрограммы.

ARM-процессоры поддерживают механизм доступа к данным с использованием базового-индексной адресации (см. 10.7.4), допускающей *попутное авто-масштабирование индекса* в зависимости от размера данных, расположенных в памяти.

Так как в таблице векторов находятся 32-разрядные адреса подпрограмм, то для получения смещения индекса «i» придется выполнить попутную операцию его умножения на 4, что эквивалентно операции попутного логического сдвига содержимого индексного регистра влево на 2 бита LSL #2.

Итак, для программного запуска подпрограммы по ее номеру нужно:

1) Записать все начальные адреса подпрограмм в таблицу векторов перехода на подпрограммы:

```
Table_Addr
          DCD Adr_0
          DCD Adr_1
          …
```

2) Проинициализировать базовый регистр (r0) начальным адресом таблицы векторов, а индексный регистр – номером подлежащей выполнению подпрограммы:

```
LDR r0, =Table_Addr
MOV r1, #N
```

3) Загрузить в один из свободных регистров ЦПУ начальный адрес соответствующей подпрограммы с авто-масштабированием индекса:

```
LDR r3, [r0, r2 LSL #2]
```

4) Выполнить *косвенный переход на подпрограмму* по адресу в регистре r3 с сохранением адреса возврата в стеке:

```
BLX r3
```

Это очень эффективный и предельно компактный метод создания интерпретаторов команд. По существу, таблица векторов переходов на подпрограммы представляет собой *таблицу указателей* на расположение функций (подпрограмм), широко используемую в языках высокого уровня. Для вызова нужной функции достаточно знать лишь ее номер. Начальный адрес функции будет автоматически извлечен из таблицы и по нему будет выполнен переход в начало процедуры.

Если процедуры для каких-то номеров функций еще не созданы, запишите в таблицу векторов адрес функции по умолчанию. Это может быть просто пустая процедура – возврат в вызывающую программу.

Перед запуском интерпретатора команд проверяйте допустимость номера выполняемой процедуры. При выходе из допустимого диапазона – запускайте «пустую» процедуру.

При выполнении косвенных переходов транслятор автоматически увеличивает значение адреса перехода на 1. Это делается для того, чтобы подтвердить центральному процессору режим работы с единым унифицированным набором команд Thumb-2. В процессе выполнения команды перехода младший бит адреса перехода автоматически обнуляется и переход выполняется по правильному четному адресу, заданному программистом. Можете не волноваться! Эта функция не требует участия программиста – автоматически выполняется интеллектуальным транслятором!

1) Где должна располагаться таблица векторов перехода на подпрограммы – в ОЗУ или в ПЗУ?
2) Будет ли в интерпретаторе команд, созданном по описанной выше технологии, одинаковое время вызова любой из процедур? Подходит ли метод для систем реального времени?
3) Обязательно ли проверять возможность выхода номера процедуры за допустимый диапазон?

1) Более правильно в ПЗУ. В этом случае ее трудно повредить извне. Теоретически возможно размещение таблицы векторов перехода на подпрограммы в ОЗУ. Она может быть в этом случае динамической (перепрограммируемой), – набор подпрограмм будет как бы «сменным». Не рекомендуется начинающим программистам.
2) Да.
3) Да. Иначе Вы рискуете случайно вызвать несуществующую подпрограмму с непредсказуемыми последствиями.

Система команд процессоров ARM оптимизирована с точки зрения создания типовых алгоритмических структур языков высокого уровня. Именно поэтому процессоры с ядрами ARM позволяют создавать предельно оптимизированные компиляторы и эффективно использовать их в разработках конечных изделий на языках высокого уровня C/C++. Система команд ядер Cortex-M4 расширена командами табличного выбора альтернативы для облегчения разработки широко используемых на практике программных модулей.

Список рекомендуемой литературы

1) Козаченко В.Ф. Микроконтроллеры: руководство по применению 16-разрядных микроконтроллеров Intel MCS-196/296 во встроенных системах управления. – М.: ЭКОМ, 1997. – 688 с.
2) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
3) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

16 ВВЕДЕНИЕ В ЦИФРОВУЮ ОБРАБОТКУ СИГНАЛОВ

Оглавление

16.1 Цифровое управление и регулирование

Любая микропроцессорная система управления (МПСУ), встроенная в оборудование (источник питания, преобразователь частоты, насос, станок, робот, конвейерную линию), предназначена для того, чтобы на основе информации, полученной с задающих устройств (потенциометры задания, пульты оперативного управления и пр.) и датчиков обратной связи (положения рабочего органа, скорости, ускорения, температуры, давления и т.д.), вырабатывать управляющие сигналы на исполнительные устройства в соответствии с разработанной структурой и параметрами системы управления – рис. 16.1.

С задающих устройств в МПСУ вводятся сигналы *требуемых значений переменных*, а с датчиков – сигналы обратной связи – *фактических значений переменных*. Внутри МПСУ выполняется сравнение сигналов задания с сигналами обратной связи, и на основе полученных величин рассогласования вырабатываются управляющие воздействия, поступающие на исполнительные устройства. Такое управление называется *управлением с обратными связями*.

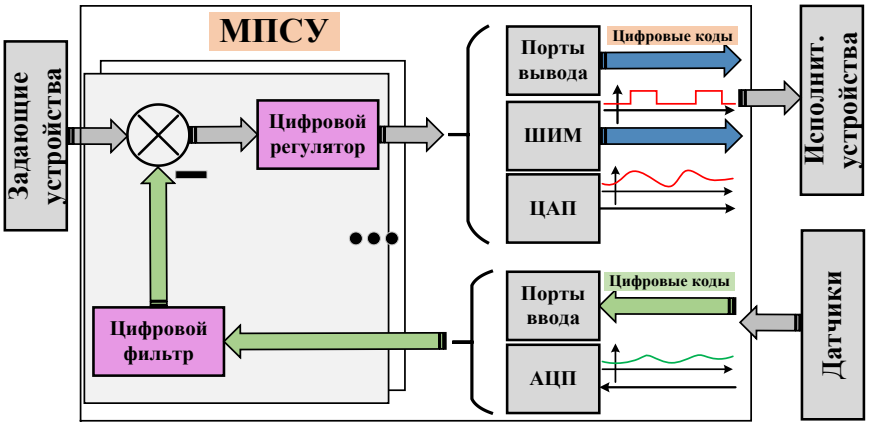


Рис. 16.1 Обобщенная структура микропроцессорной системы управления

Сигналы с датчиков обратной связи могут вводиться в МПСУ в виде цифровых кодов или в аналоговом виде. В первом случае для их ввода используются порты ввода данных микроконтроллера или стандартные интерфейсы, а во втором – аналого-цифровые преобразователи, которые преобразуют непрерывные входные сигналы в цифровые коды. Сигналы с задающих устройств обычно поступают в цифровой форме, в том числе от систем верхнего уровня управления по типовым интерфейсам, хотя возможен и аналоговый ввод. В последнем случае аналоговый сигнал, например, с задающего потенциометра, также предварительно обрабатывается в АЦП.

Сигналы задания и сигналы обратной связи с датчиков могут предварительно поступать в *цифровые фильтры*, например, для выделения полезного сигнала на фоне имеющихся в системе помех (фильтры низкой, высокой частоты, полосовые фильтры, режекторные и т.д.).

Полезные сигналы на выходе фильтров сравниваются с задающими сигналами, и величины отклонений поступают на входы *цифровых регуляторов*, вырабатывающих управляющие воздействия на исполнительные устройства. Регуляторов может быть столько, сколько регулируемых переменных в системе.

Управляющие воздействия могут выдаваться в разных форматах: 1) непосредственно в виде цифровых кодов управления оборудованием – в *порты вывода* или по типовым интерфейсам; 2) в виде широтно-импульсного (ШИМ) или частотно-импульсного сигнала (ЧИМ), в котором скважность или частота сигнала является

информационной составляющей: в состав МПСУ при этом включаются специализированные периферийные устройства – *ШИМ/ЧИМ-генераторы*; 3) в аналоговой форме: в состав МПСУ включаются *цифро-аналоговые преобразователи*, выполняющие преобразование цифрового кода в непрерывный сигнал.

Современная концепция *прямого цифрового управления оборудованием* предполагает интеграцию на кристалл микроконтроллера *специализированных периферийных устройств*, имеющих *прямой цифровой интерфейс* как с датчиками обратной связи, так и с исполнительными устройствами: модулей захвата/сравнения, квадратурных декодеров, менеджеров событий с ШИМ-генераторами, многоканальных АЦП, ЦАП и др. Особенность современного подхода – прямое управление каждым силовым ключом силового преобразователя любой сложности и прямое сопряжение с типовыми датчиками обратной связи (например, импульсными датчиками положения и скорости).

Цифровые системы управления последовательно выполняют ряд функций:

1) Ввод цифровых сигналов задания и обратных связей из портов ввода или по типовым интерфейсам;
2) Запуск преобразования в АЦП и оцифровка аналоговых (непрерывных) сигналов;
3) Цифровая фильтрация входных, в основном аналоговых, сигналов;
4) Расчет выходных управляющих воздействий в цифровых регуляторах;
5) Вывод управляющих воздействий в *цифровой* (порты вывода), *импульсной* (частотно или широтно-импульсный сигнал) или *аналоговой* форме.

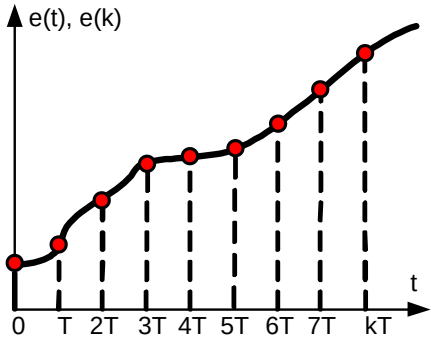


Рис. 16.2. Непрерывная и решетчатая функции

Действия, описанные выше, выполняются за конечное время, называемое интервалом квантования цифровой системы управления по времени T. Внутри системы управления любой непрерывный сигнал представлен только его отдельными выборками (дискретами) – значениями, фиксированными на каждом интервале квантования – рис. 16.2 (красные кружки). Такая последовательность значений непрерывной функции называется решетчатой функцией.

После преобразования в АЦП любой аналоговый сигнал является еще и квантованным по уровню. Это означает, что каждая выборка представляется числом ограниченной разрядности.

Так, выходной код АЦП может быть 10-, 12-, 16-разрядным, в редких случаях – большей разрядности. Следовательно, принципиальными особенностями цифровых систем управления по сравнению с аналоговыми является квантование всех сигналов по времени и по уровню.

Для синтеза структур и параметров систем управления разработана специальная теория – *теория систем автоматического управления (ТАУ)*. Синтез возможен как в непрерывной области (что существенно проще), так и в дискретной. Для синтеза структур и параметров цифровых фильтров применяется *теория цифровой обработки сигналов (ЦОС)*.

Оба направления – это отдельные области прикладной науки, без знания которых невозможна разработка сколько-нибудь эффективной системы цифрового управления. Наша книга посвящена вопросам реализации цифровых систем управления, главным образом, на базе процессоров фирмы ARM. Нас интересуют, прежде всего, следующие вопросы: что должен уметь делать микропроцессор, чтобы эффективно решать задачи

цифрового управления и цифровой фильтрации? Какая поддержка в системе команд ARM-процессоров имеется для решения этих специфических задач?

Поступим так: на примере простейших структур цифровых регуляторов покажем, какие операции являются *типовыми в цифровом регулировании и цифровой фильтрации*. Рассмотрим команды поддержки этих операций при работе с переменными в формате с фиксированной точкой, а затем (в последующих главах) – и в формате с плавающей точкой. Покажем, как в архитектуре и системе команд ARM Cortex-M3/M4 обеспечивается поддержка решения типовых задач, наиболее часто встречающихся при цифровой обработке сигналов: создания и использования линейных и кольцевых буферов в памяти данных; обработки данных в этих буферах с использованием операций умножения с накоплением; ограничения возможных переполнений; предварительной обработки входных данных с портов ввода и периферийных устройств.

Ядро процессоров Cortex-M4 расширено эффективными *командами цифровой обработки сигналов (ЦОС)* – командами умножения с накоплением и насыщения (ограничения) для работы с числами в формате с фиксированной точкой. Ядро процессоров Cortex-M4F дополнительно поддерживает операции ЦОС в формате с плавающей точкой. Процессорные ядра Cortex-M4/M4F по функциональным возможностям и производительности не уступают, а порой и превосходят специализированные процессоры для цифровой обработки сигналов (DSP) и пригодны для решения самых сложных задач управления реального времени. Большинство команд, в том числе умножения с накоплением, выполняются, как и положено в RISC-процессорах, всего за один цикл.

Приведем в этой главе материал, дающий начальное представление о методах решения задач ЦОС. Он позволит понять, зачем в систему команд процессора включены соответствующие специальные команды. Рассмотрим более подробно особенности и ограничения, возникающие при работе с числами в формате с фиксированной точкой.

16.2 Разностные уравнения как способ описания цифровых регуляторов и фильтров

Существуют различные методы синтеза цифровых регуляторов и цифровых фильтров. Все один достаточно сложны и требуют специальной подготовки, особенно в том случае, когда нужно учесть эффекты квантования сигналов по уровню и времени.

Процессоры ARM являются 32-разрядными. По сравнению с 8- и 16-разрядными процессорами эффекты квантования по уровню в них могут быть сведены к минимуму, особенно при переходе к обработке данных в формате с плавающей точкой. Более того, скорость вычислительных операций в RISC-процессорах столь велика, что в подавляющем большинстве реальных задач период квантования T может быть на порядок меньше периода самого высокочастотного сигнала в системе. В этом случае появляется возможность синтезировать систему управления в непрерывной области, получив ее описание в виде обычной системы дифференциальных уравнений, а затем – записать для нее эквивалентную систему разностных уравнений, пригодную для непосредственной программной реализации в микроконтроллере.

16.2.1 Теорема Котельникова (Найквиста, Шеннона)

В теории цифровой обработки сигналов вводится понятие частоты самого высокочастотного входного сигнала f_{max}, подлежащего цифровой обработке. В системах регулирования и управления существует понятие самой маленькой постоянной времени объекта управления T_{μ}. Ей соответствует максимальная частота возможного изменения переменных объекта управления $f_{max}= 1/T_{\mu}$.

Теорема Котельникова гласит: для того, чтобы входной аналоговый сигнал можно было полностью восстановить по его дискретным выборкам, частота дискретизации $f_{дискр}$ цифровой системы управления должна быть, как минимум, вдвое больше максимальной частоты сигнала: $f_{дискр} \geq 2f_{max}$.

На практике лучше иметь превышение частоты дискретизации на порядок. При этом Вы получаете колоссальное преимущество – можете проектировать систему управления, как полностью аналоговую, а в конце – просто перейти от дифференциальных уравнений к разностным (покажем, как это делается).

Таким образом, период дискретизации должен быть, как минимум, вдвое меньше периода самого высокочастотного сигнала или самой маленькой постоянной времени объекта управления $T \leq 0.5T_{\mu}$, – лучше $T \leq (0.1 \div 0.2)T_{\mu}$.

16.2.2 Технология перехода от дифференциальных уравнений к разностным на примере типовых регуляторов. Методы дискретизации

Проиллюстрируем методы перехода от аналоговых регуляторов к цифровым на примере ПИД-регулятора, имеющего следующую передаточную функцию:

$$W(p) = k_{п} + \frac{1}{T_{и} p} + T_{д} p \ , \qquad (16.1)$$

где: $k_{п}$ – коэффициент пропорциональной части регулятора; $T_{и}$ – постоянная интегрирования; $T_{д}$ – постоянная дифференцирования.

Обозначим сигнал рассогласования на входе регулятора через $e(t)= x_{з}(t) – x_{факт}(t)$, а выходной сигнал регулятора (управляющее воздействие) – через u(t). Передаточной функции (16.1) соответствует дифференциальное уравнение:

$$u(t) = k_{п} e(t) + \frac{1}{T_{и}} \int_{0}^{t} e(t)dt + T_{д} \frac{de(t)}{dt} \qquad (16.2)$$

16.2.2.1 Первый метод дискретизации

Состоит в непосредственной замене всех переменных дифференциального уравнения (16.2) их дискретными значениями. При этом производная заменяется *левой разностью первого порядка* (рис. 16.3, а), а *интеграл – суммой*, вычисленной по методу прямоугольников (рис. 16.3, б) или трапеций (рис. 16.3, в).

В первом случае (16.2) приводится к виду:

$$u(k) = k_{п} e(k) + \frac{T}{T_{и}} \sum_{i=0}^{(k-1)} e(i) + \frac{T_{д}}{T} [e(k) - e(k-1)] \qquad (16.3)$$

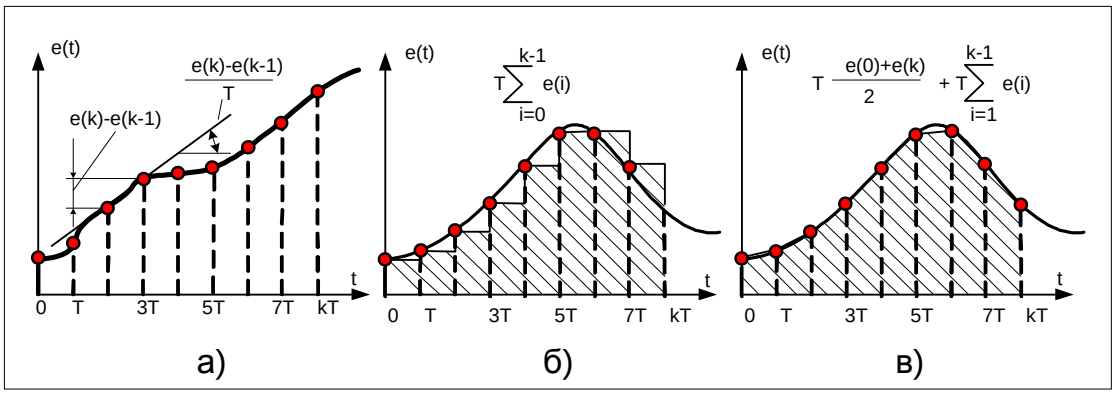


Рис. 16.3 Дискретизация уравнений непрерывных систем

Во втором случае – к виду:

$$u(k) = k_{\text{п}} e(k) + \frac{T}{T_{\text{и}}}\left[\frac{e(0)+e(k)}{2} + \sum_{i=1}^{(k-1)} e(i)\right] + \frac{T_{\text{д}}}{T}[e(k) - e(k-1)]. \qquad (16.4)$$

Полученные выражения (16.3) и (16.4) соответствуют так называемым «*позиционным*» алгоритмам вычисления управляющих воздействий, когда для формирования суммы требуется запоминание всех предыдущих значений сигналов рассогласования, что крайне неэффективно. Для более удобной реализации перейдем от «позиционного» к так называемому «*рекуррентному*» алгоритму, когда текущее значение управляющего воздействия u(k) вычисляется через значение управляющего воздействия на предыдущем такте управления u(k-1) и некоторую добавку Δu(k):

$$u(k) = u(k-1) + \Delta u(k). \qquad (16.5)$$

Подставим в уравнения (16.3) и (16.4) вместо k → (k-1) и вычислим $u(k) - u(k-1)$. Результат для методов прямоугольников и трапеций, соответственно:

$$\Delta u(k) = (k_{\text{п}} + \frac{T_{\text{д}}}{T})e(k) - (k_{\text{п}} + 2\frac{T_{\text{д}}}{T} - \frac{T}{T_{\text{и}}})e(k-1) + \frac{T_{\text{д}}}{T}e(k-2). \qquad (16.6)$$

$$\Delta u(k) = (k_{\text{п}} + \frac{T_{\text{д}}}{T} + \frac{1}{2}\frac{T}{T_{\text{и}}})e(k) - (k_{\text{п}} + 2\frac{T_{\text{д}}}{T} - \frac{1}{2}\frac{T}{T_{\text{и}}})e(k-1) + \frac{T_{\text{д}}}{T}e(k-2). \qquad (16.7)$$

Результаты похожи, но отличаются точностью. Метод трапеций позволяет вычислить интеграл по дискретным выборкам точнее, чем метод прямоугольников. Мы получили «скоростной» алгоритм вычисления управляющего воздействия, основанный на расчете текущего значения приращения управляющего воздействия:

$$\begin{cases} \Delta u(k) = k_0 e(k) - k_1 e(k-1) + k_2 e(k-2) \\ u(k) = u(k-1) + \Delta u(k) \end{cases}, \qquad (16.8)$$

в котором коэффициенты k_0, k_1, k_2 зависят от параметров регулятора и интервала дискретизации системы управления по времени T. Его особенность в том, что запоминаются только три последних выборки сигнала рассогласования e(k), e(k-1), e(k-2). Этих выборок и значения управляющего воздействия на предыдущем шаге достаточно, чтобы рассчитать новое значение управляющего воздействия.

16.2.2.2 Второй метод дискретизации

Состоит в дифференцировании уравнения (16.2) для исключения интеграла и в последующей замене второй производной *левой разностью второго порядка*. Получим:

$$\frac{du(t)}{dt} = k_{\text{п}}\frac{de(t)}{dt} - \frac{1}{T_{\text{и}}}e(t) + T_{\text{д}}\frac{d^2e(t)}{dt^2}, \qquad (16.9)$$

откуда:

$$\frac{u(k)-u(k-1)}{T} = k_{\text{п}}\frac{e(k)-e(k-1)}{T} - \frac{1}{T_{\text{и}}}e(k) + T_{\text{д}}\frac{e(k)-2e(k-1)+e(k-2)}{T^2},$$

или после преобразований:

$$\Delta u(k) = (k_{\text{п}} + \frac{T}{T_{\text{и}}} + \frac{T_{\text{д}}}{T})e(k) - (k_{\text{п}} + 2\frac{T_{\text{д}}}{T})e(k-1) + \frac{T_{\text{д}}}{T}e(k-2). \qquad (16.10)$$

Это разностное уравнение может быть найдено также после представления (16.9) системой дифференциальных уравнений первого порядка в форме Коши:

$$\begin{cases} \dfrac{du(t)}{dt} = k_{\text{п}}\dot{e}(t) + \dfrac{1}{T_{\text{и}}}e(t) + T_{\text{д}}\ddot{e}(t); \\ \dfrac{de(t)}{dt} = \dot{e}(t); \\ \dfrac{d\dot{e}(t)}{dt} = \ddot{e}(t). \end{cases} \qquad (16.11)$$

Если заменить все производные левыми разностями и выполнить несложные преобразования, получим (16.10). Замена производных левыми разностями в системе дифференциальных уравнений, записанной в форме Коши, соответствует хорошо известному из математики численному методу Эйлера решения дифференциальных уравнений на компьютерах.

Одной и той же системе дифференциальных уравнений может соответствовать несколько дискретных моделей, представленных в виде разностных уравнений. Их реализация «в цифре» может отличаться от реализации «в аналоге» точностью. По мере уменьшения интервала квантования по времени эта разница постепенно нивелируется и любое цифровое решение приближается к своему аналоговому прототипу. То же происходит при численном решении систем дифференциальных уравнений на компьютерах при уменьшении шага интегрирования. Дискретные модели отличаются между собой тем, сколько входных выборок и предыдущих значений управляющих воздействий использовано в алгоритме (порядком разностного уравнения). Чем выше порядок разностного уравнения, тем точнее оно приближается к своему аналоговому прототипу.

16.3 Обобщенная дискретная модель цифрового регулятора и цифрового фильтра

Дадим сводку разностных уравнений, описывающих в дискретной форме пропорционально двукратно-дифференциальный (ПДД), пропорционально-интегрально-дифференциальный (ПИД) и пропорционально двукратно-интегральный (ПИИ) регуляторы:

$$\begin{cases} u(k) = k_0 e(k) - k_1 e(k-1) + k_2 e(k-2); \\ u(k) = u(k-1) + k_0 e(k) - k_1 e(k-1) + k_2 e(k-2); \\ u(k) = 2u(k-1) - u(k-2) + k_0 e(k) - k_1 e(k-1) + k_2 e(k-2). \end{cases} \qquad (16.12)$$

Можно показать, что любой, сколь угодно сложный цифровой регулятор или цифровой фильтр, описывается разностным уравнением вида:

$$y[k] = b_0 \cdot x[k] + b_1 \cdot x[k-1] + b_2 \cdot x[k-2] + \cdots + b_n \cdot x[k-n] + \\ + a_1 \cdot y[k-1] + a_2 \cdot y[k-2] + \cdots + a_n \cdot y[k-n] = S_1 + S_2 \qquad (16.13)$$

Здесь:

$x[k], x[k-1], \ldots$ – дискретные выборки входной переменной цифрового регулятора или цифрового фильтра: текущая, предыдущая, …;

$y[k], y[k-1], \ldots$ – выходные значения управляющих воздействий цифрового регулятора или выходные значения цифрового фильтра: текущее, предыдущее, …;

$b_0, b_1, \ldots$ – коэффициенты при соответствующих выборках входных переменных;

$a_0, a_1, \ldots$ – коэффициенты при полученных ранее (на предыдущих тактах) значениях выходных переменных (или управляющих воздействий);

n – порядок цифрового регулятора/фильтра, соответствующий порядку дифференциального уравнения-прототипа.

В уравнении (16.13) отдельные коэффициенты или их часть могут отсутствовать (быть нулевыми). Цифровой фильтр, в котором отсутствуют коэффициенты a_i и имеются только коэффициенты b_i, называется *фильтром с конечной импульсной характеристикой* – КИХ-фильтром, а фильтр, в котором присутствуют и коэффициенты a_i и b_i – *фильтром с бесконечной импульсной характеристикой* – БИХ-фильтром. В зависимости от требований конкретного приложения выбирается нужный тип фильтра и синтезируются его коэффициенты.

Таким образом, имеются две типовые операции, которые должны выполняться при решении любой задачи цифрового регулирования/фильтрации - *операция умножения с накоплением результата* для входных выборок:

$$S_1 = \sum_{i=0}^{n} b_i \cdot x[k-i] \qquad (16.14)$$

и операция умножения с накоплением результата для предыдущих значений выходной переменной:

$$S_2 = \sum_{i=1}^{n} a_i \cdot y[k-i] \qquad (16.15)$$

1) Выполните дискретизацию уравнения ПИД-регулятора с использованием метода описанных прямоугольников, когда используются не предыдущие значения выборок, а текущие.

2) Сравните с результатом по методу вписанных прямоугольников. Даст ли этот метод более высокую точность?

3) Какому из методов представления интеграла соответствует классический метод Эйлера?

2) Точность разностного уравнения будет такой же низкой по сравнению с точностью уравнения, полученного методом трапеций, что особенно заметно при монотонности исходной непрерывной функции.

3) Методу вписанных прямоугольников.

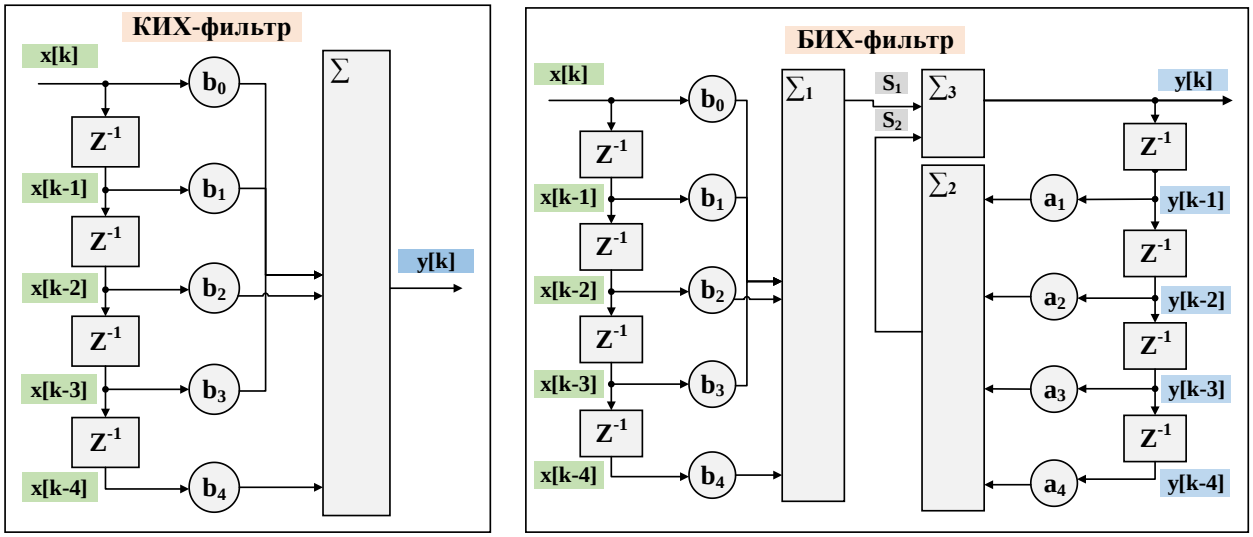
16.4 Типовые структуры цифровых фильтров

Типовая структура КИХ-фильтра, пригодная для его программной реализации, представлена на рис. 16.4а на примере фильтра 4-го порядка. Очередное значение входной переменной x[k] считывается из порта ввода или АЦП и умножается на коэффициент b_0. Блок с обозначением z^{-1} представляет собой элемент задержки входного сигнала на один интервал квантования системы управления T. На его выходе получаем предыдущую выборку входного сигнала x[k-1], которая должна быть умножена на коэффициент b_1 и просуммирована с ранее полученным произведением.

Аналогично обрабатываются пост-предыдущие выборки x[k-2], x[k-3] и x[k-4]: умножаются на коэффициенты b_2, b_3, b_4 и все произведения суммируются. Значение на выходе сумматора y[k] является выходом фильтра.

БИХ-фильтры отличаются тем, что в них присутствует *обратная связь* по выходной переменной фильтра, рассчитанной на предыдущих тактах его работы – рис. 16.4б. По некоторым коэффициентам эта обратная связь может быть положительной, а по некоторым – отрицательной. Это определяется требованиями к фильтру. В общем случае

эта структура потребует двух операций умножения с накоплением для вычисления S1 и S2 и одной дополнительной операции суммирования S3.



а) б)

Рис. 16.4 Структура цифровых фильтров, а – КИХ-фильтр, б – БИХ-фильтр

Тип фильтра, число и значения коэффициентов зависят от требований к выходному сигналу. Чем они жестче, тем больше коэффициентов может входить в фильтр. Однако, приведенные на рис. 16.4а и рис. 16.4б обобщенные структуры фильтров/регуляторов сохраняются.

16.5 Операционная поддержка цифровой обработки сигналов

Итак, основной операцией, которой должны обладать процессоры для эффективной цифровой обработки сигналов (ЦОС), должна быть *операция умножения с накоплением.* Она может выполняться над числами в формате с фиксированной точкой или над числами в формате с плавающей точкой. В общем случае все коэффициенты и выборки входной x или выходной y переменной являются числами со знаком, поэтому преимущественно должны использоваться знаковые операции умножения с накоплением. Процессоры, которые имеют в своем арсенале команды умножения с накоплением, относятся к классу *сигнальных процессоров*.

Процессоры ARM Cortex-M4/M4F можно назвать *сигнальными RISC-процессорами*: они поддерживают не только операции умножения с накоплением чисел в формате с фиксированной точкой (Cortex-M4), но и с плавающей точкой (в Cortex-M4F), причем делают эти операции максимально быстро – всего лишь за один такт.

Для чисел в формате с фиксированной точкой исходные операнды могут быть двух типов: с 16-разрядными полусловами (16)*(16) или с 32-разрядными словами (32)*(32). Результат умножения с накоплением может быть либо 32-разрядным, либо 64-разрядным. Поддерживаются также операции умножения с накоплением типа (16)*(32), когда или выборки данных, или коэффициенты имеют вдвое большую разрядность.

При выполнении операций умножения с накоплением возможны переполнения – выходы за пределы допустимого выходного формата. Поэтому должны быть предусмотрены и средства борьбы с ними:

1) Ограничение результата умножения с накоплением на уровне максимально допустимых значений с использованием специальных *команд насыщения*.
2) Использование *«длинных» операций умножения с накоплением*, когда аккумулятор состоит из двух 32-разрядных регистров ЦПУ (64 разряда) для гарантированного исключения переполнений.
3) Использование операций умножения с накоплением чисел с плавающей точкой (для Cortex-M4F), когда переполнения исключаются самим форматом чисел.

Кроме того, система команд процессора должна иметь команды, позволяющие считывать из портов ввода и периферийных устройств байты и полуслова входных данных и *преобразовывать их в нужный формат* – полуслов или полных слов. При этом могут потребоваться операции знакового или *беззнакового расширения байта или полуслова к формату полного слова*, а также специальные *операции работы с битовыми полями*, когда нужная информация выделяется из битового поля заданной длины и преобразуется к 32-разрядному слову, обрабатываемому процессором.

Обратите особое внимание на то, где должны храниться выборки входных переменных и значения выходных переменных на предыдущих тактах работы системы управления. Для этого предусматриваются *специальные буфера в оперативной памяти* системы управления, емкость которых зависит от порядка цифрового фильтра/регулятора. Так, если порядок фильтра не превышает 4, то для каждого буфера входных и выходных переменных достаточно всего 4-х ячеек памяти. С ростом порядка емкость буферов растет пропорционально порядку.

Желательно так организовать эти буфера, чтобы они хранили только нужную информацию – только последние, самые «свежие» данные, чтобы новые, поступающие в них данные автоматически замещали самые «старые», которые больше не потребуются. Такие буфера в памяти могут быть двух типов: *линейными* или *кольцевыми*. В любом случае могут потребоваться специальные команды или процедуры для перезаписи выборок в буферах и специальные средства адресации данных в кольцевых буферах и соответствующие подпрограммы.

Процессоры ARM Cortex-M4/M4F, имея развитую универсальную систему адресации памяти, поддерживаемую соответствующими командами, а также ряд специальных дополнительных команд, позволяют эффективно решить все сформулированные выше задачи по цифровой обработке сигналов, в том числе – создания и работы с линейными и кольцевыми буферами данных в памяти.

16.6 Технология организации буферов данных в ОЗУ

16.6.1 Линейный буфер

Для реализации любого цифрового регулятора/фильтра достаточно высокого порядка необходим буфер входных выборок и (возможно) значений предыдущих выходных управляющих воздействий. *Линейный буфер выборок* показан на рис. 16.5.

Перед тем, как новое значение входной переменной x[k] записывается в буфер на очередном такте расчета регулятора/фильтра, все «старые» выборки *перезаписываются* (сдвигаются) на одну позицию вниз, а «самая старая» x[k-n] выдвигается за пределы буфера и теряется (как бы попадает в «мусорное ведро»). После этого в освободившуюся ячейку памяти считывается новая входная выборка. Буфер выборок готов к очередному такту обработки данных.

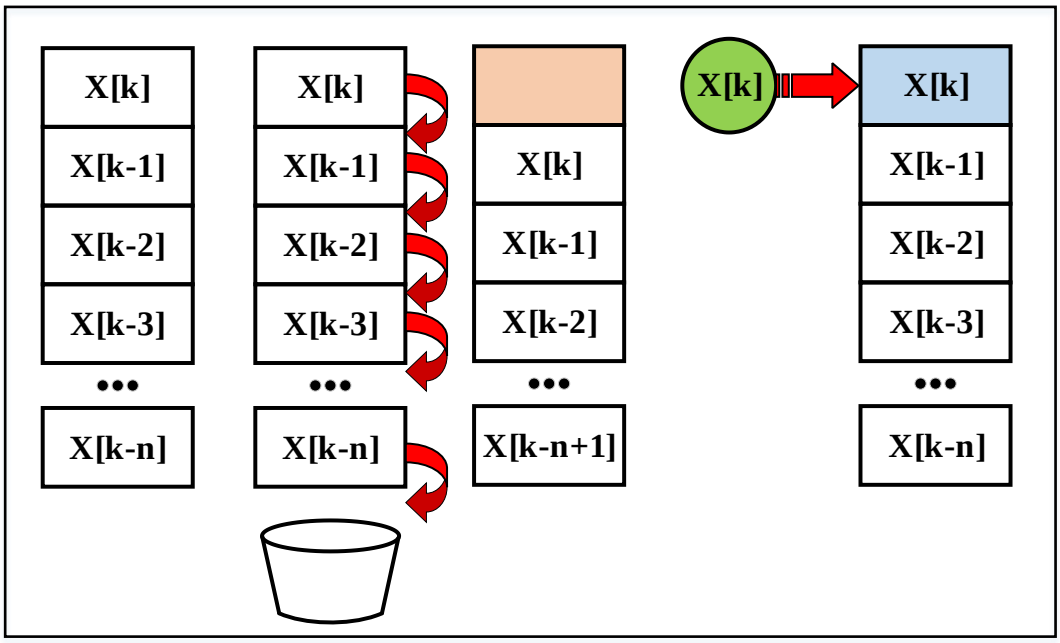


Рис. 16.5 Линейный буфер выборок

Технология работы с линейным буфером данных демонстрируется в проекте CPU_20 (файл MyProg_20.s). В начале программы объявляется переменная – длина буфера N, а в конце – резервируется сам буфер в ОЗУ и дополнительная ячейка памяти, имитирующая порт ввода входных данных Port_X.

```
10  ; Основная программа
11  ; Определение переменной "емкость" линейного буфера
12  ; (Число 32-разрядных слов)
13  N       EQU   10
14  ; Передать параметры в подпрограмму очистки буфера
15          LDR r0,=Lin_Buffer  ; Начальный адрес буфера
16          MOV r1,#N           ; Число слов в буфере
17  ; Вызов подпрограммы начальной очистки буфера
18          BL Clear_Lin_Buffer
```

```
90  ; Объявить секцию данных в ОЗУ для размещения линейного
91  ; буфера выборок
92          AREA MyData, DATA, ReadWrite
93  Lin_Buffer
94          SPACE N*4
95  ; Зарезервировать одну ячейку памяти для иммитации порта ввода
96  Port_X  SPACE 4
```

```
31  ;*************************************************************
32  ; Подпрограмма очистки линейного буфера данных в ОЗУ
33  ; Входы: r0 - указатель начального адреса буфера
34  ;         r1 - число 32-разрядных слов данных в буфере
35  ; Используемые регистры:
36  ;         r2 - "шаблон" заполнения буфера данными
37  ; Выход: очищенный буфер данных в ОЗУ
38  ;*************************************************************
39  Clear_Lin_Buffer
40  ; Загрузить "шаблон" заполнения буфера данными
41          MOV r2,#0
42  Loop
43  ; Очистить текущеую ячейку с пост-авто-смещением указателя
44          STR r2, [r0],#4
45  ; Декрементировать счетчик числа циклов с установкой флагов
46          SUBS r1, #1
47  ; Если не все ячейки буфера обнулены, повторить цикл
48          BNE Loop
49  ; Буфер очищен, выйти из подпрограммы
50          BX LR
51  ; Конец подпрограммы
52  ;*************************************************************
```

Прежде чем в буфер будут поступать входные данные, он должен быть очищен. С этой целью в начале основной программы вызывается подпрограмма предварительной очистки буфера Clear_Lin_Buffer, которая выполняет функцию его инициализации. Содержимым регистров ЦПУ ей передается начальный адрес буфера и его длина.

В подпрограмме очистки буфера используется стандартная команда сохранения данных STR из регистра ЦПУ с косвенной адресацией памяти и пост-авто-смещением указателя на 4 (длину слова в байтах), включенная в тело цикла. Цикл выполняется столько раз, сколько ячеек памяти находится в буфере.

Запись новой выборки в буфер должна сопровождаться перезаписью всех предыдущих выборок с освобождением места под новую выборку. Эту функцию выполняет подпрограмма Write_Lin_Buffer.

```
20  ; Передать параметры в подпрограмму перезаписи "старых"
21  ; выборок в буфере и записи в него новой выборки
22  M1      LDR r0,=Port_X      ; Считать новую выборку
23          LDR r2, [r0]        ; из порта ввода
24          LDR r0,=Lin_Buffer  ; Начальный адрес буфера
25          MOV r1,#N           ; Число слов в буфере
26  ; Вызов подпрограммы записи в буфер новой выборки
27          BL Write_Lin_Buffer
28  ; Повторить для нового значение входной переменной
29          B M1
```

```
54  ;*********************************************************
55  ; Подпрограмма записи  новой выборки в линейный буфер
56  ; и перезаписи предыдущих выборок
57  ; Входы: r0 - указатель начального адреса буфера
58  ;         r1 - число 32-разрядных слов данных в буфере
59  ;         r2 - новое значение выбоки для записи в буфер
60  ; Используемые регистры:
61  ;         r3 - указатель адреса предыдущей ячейки
62  ;         r4 - указатель адреса последующей ячейки
63  ;         r5 - регистр временного хранения выборки
64  ; Выход: модифицированный линейный буфер
65  ;*********************************************************
66  Write_Lin_Buffer
67  ; Установить указатели адресов r3,r4 на предпоследнюю и
68  ; последнюю ячейки буфера
69          SUB r1,#1       ; r1 <- (N-1)
70          MOV r5,#4       ; длина слова в байтах
71          MUL r5,r5,r1    ; смещение адреса: r5 <- (N-1)*4
72          ADD r4,r0,r5    ; адрес последней ячейки буфера
73          SUB r3,r4,#4    ; адрес предпоследней ячейки буфера
74  M_Loop
75  ; Перезаписать данные из предыдущей ячейки в следующую
76  ; с пост-авто-декрементированием указателей на 4
77          LDR r5, [r3],#-4
78          STR r5, [r4],#-4
79  ; Декрементировать счетчик числа циклов с установкой флагов
80          SUBS r1, #1
81  ; Если не все ячейки буфера обработаны, повторить цикл
82          BNE M_Loop
83  ; Записать в буфер новую выборку
84          STR r2, [r0]
85  ; Буфер перезаписан, выйти из подпрограммы
86          BX LR
87  ; Конец подпрограммы
88  ;*********************************************************
```

Она вызывается из основной программы и, кроме параметров буфера (начального адреса и длины), ей передается также значение новой выборки. Для отладки подпрограммы она вызывается многократно.

Вы можете поставить точку останова в строке 29, изменить содержимое ячейки памяти, имитирующей порт ввода, и выполнить подпрограмму еще раз. Новая выборка будет записана в вершину буфера, а все «старые» выборки сместятся на одну позицию вниз.

Обратите внимание на то, что перезапись данных в буфере выполняется, начиная с двух последних ячеек памяти, и продолжается вплоть до первой ячейки, которая в результате освобождается и принимает новую выборку. Адреса двух последних ячеек буфера рассчитываются в начале подпрограммы и сохраняются в указателях (r3, r4). В цикле перезаписи ячеек буфера оба указателя пост-авто-декрементируются на 4.

Изучите файл карты памяти CPU_20.map и определите место расположения буфера и порта ввода в памяти. При отладке откройте окно дампа памяти и проследите, как очередные значения из ячейки 0x20000028 (имитирует Port_X), вводимые Вами в точке останова программы, при повторном ее запуске записываются в вершину буфера.

Пример:

Memory 1

Address: 0x20000000

```
0x20000000:  00000044  00000022  00000077  00000077  0000002D  0000000A  0000000A  00000001  00000002  00000003
0x20000028:  00000044  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
```

Memory 1

Address: 0x20000000

```
0x20000000:  00000079  00000044  00000022  00000077  00000077  0000002D  0000000A  0000000A  00000001  00000002
0x20000028:  00000079  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000  00000000
```

Развитые способы адресации памяти в процессорах ARM Cortex-M4/M4F с возможностью использования нескольких указателей (в данном случае – двух) с пост- авто-декрементированием позволяют легко обновлять данные в линейных буферах и записывать в них новые входные данные для выполнения очередного такта расчета цифрового регулятора/фильтра. При этом никакие специальные команды не требуются.

16.6.2 Кольцевой буфер

В линейном кольцевом буфере с каждым тактом работы цифрового регулятора/фильтра все данные в буфере перезаписываются, освобождая место для новой выборки. Эта операция требует времени, особенно при большой емкости буфера. Можно организовать в памяти *кольцевой буфер*, в котором новые данные будут автоматически *замещать* самые старые и уже ненужные данные, а все предыдущие выборки вообще не трогать, как показано на рис. 16.6.

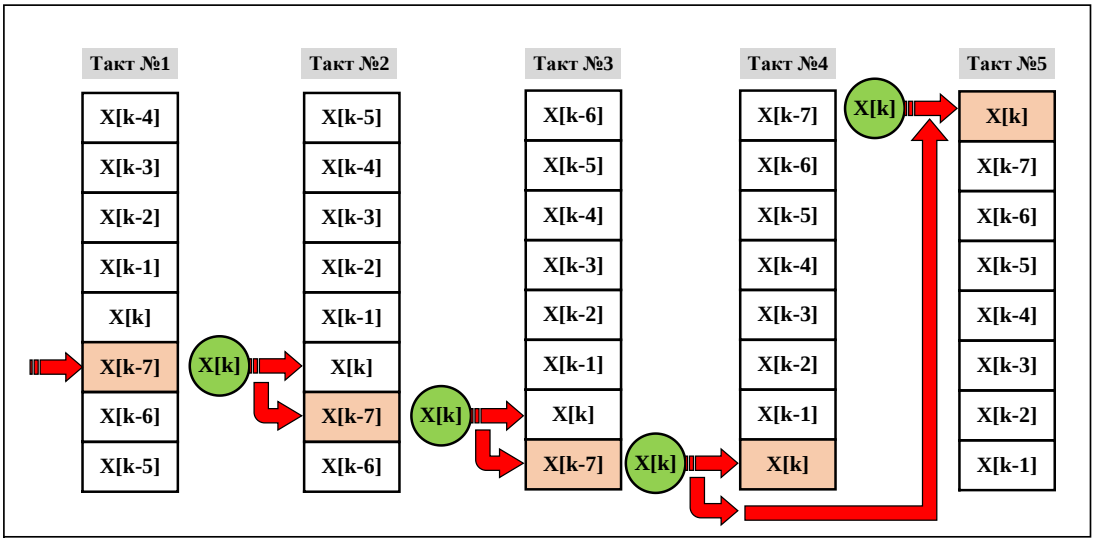


Рис. 16.6 Кольцевой буфер выборок

Пусть (для примера) буфер рассчитан на 8 выборок и его начальное состояние (на такте № 1) таково, что *указатель кольцевого буфера* (красная стрелка) показывает на адрес расположения самой «старой» выборки x[k-7]. Данные в буфере уже обработаны и выборка x[k-7] больше не нужна. Она может быть перезаписана значением входной переменной x[k] на следующем такте работы регулятора/фильтра. Если это сделать, то в соответствующей ячейке памяти появится самая «свежая» выборка x[k], а во всех предыдущих ячейках данные останутся прежними, но для нового такта работы они будут представлять собой выборки, задержанные относительно новой выборки на один такт.

Пусть в процессе записи новой выборки в буфер *указатель кольцевого буфера автоматически пост-инкрементируется на длину данных в одной ячейке*. Если его новое значение будет превышать максимальный адрес буфера (как на 4-м такте), указатель будет пере-инициализироваться минимальным (начальным) адресом буфера.

Таким образом, в буфере всегда будет находиться ровно столько выборок, сколько нужно для расчета выходной переменной с использованием операции умножения на коэффициенты и накопления. Никакой перезаписи выборок не потребуется. Это и есть *кольцевой буфер*.

Для работы с данными в кольцевом буфере достаточно знать местоположение *указателя кольцевого буфера*. Нужно пре-декрементировать указатель, и он автоматически покажет на самую последнюю выборку x[k]. Разумеется, после операции декрементирования необходимо проверить выход за нижнюю границу кольцевого буфера. Если это произошло, то указатель нужно пере-инициализировать значением максимального адреса буфера.

```
MyProg
; Основная программа
; Определение переменной "емкость" кольцевого буфера
; (Число 32-разрядных слов в буфере)
N       EQU   10
```

```
; Объявить секцию данных в OSY для размещения кольцевого
; буфера выборок
        ALIGN
        AREA MyData, DATA, ReadWrite
ADR_0
        SPACE N*4
ADR_1
; Зарезервировать одну ячейку памяти для иммитации порта ввода
Port_X  SPACE 4
; Конец ассемблерного текста
        END
```

```
;*******************************************************
; Подпрограмма инициализации параметров кольцевого буфера
; с автоматической очисткой буфера как линейного буфера
; Используемые регистры: r4,r5
; Выход: очищенный кольцевой буфер данных в OSY
;   r0 – минимально допустимый адрес кольцевого буфера
;   r1 – указатель текущей ячейки кольцевого буфера
;   r2 – максимально допустимый адрес кольцевого буфера
;   r3 – адрес порта ввода входной переменной x[k]
;*******************************************************
Inicy_Circ_Buffer
; Загрузить в r0 начальный адрес кольцевого буфера
        LDR r0,=ADR_0
; Скопировать в регистры r1 и r4
        MOV r1, r0
        MOV r4, r0
; Загрузить в r2 конечный адрес кольцевого буфера
        LDR r2,=ADR_1
        SUB r2,#4
; Очистить все ячейки кольцевого буфера, как линейного
; Загрузить "шаблон" заполнения буфера данными
        MOV r3,#0
Loop
; Очистить текущую ячейку с пост-авто-смещением указателя
        STR r3, [r4],#4
; Сравнить текущий адрес с последним адресом буфера
        CMP r4,r2
; Пока буфер не очищен полностью, продолжать
        BLS Loop
; Установить указатель r3 на адрес порта Port_X
        LDR r3,=Port_X
; Выход из подпрограммы
```

```
;*******************************************************
; Подпрограмма записи  новой выборки в кольцевой буфер
; Входы: новое значение входной переменной x[k] (Port_X)
; Используемые регистры (дополнительно): r4
; Выход: модифицированный кольцевой буфер
;*******************************************************
Write_Circ_Buffer
; Считать выборку x[k] из порта ввода в регистр r4
        LDR r4,[r3]
; Записать эти данные по текущему указателю адреса
; "свободной" ячейки в кольцевом буфере с авто-смещением
; указателя кольцевого буфера
        STR r4, [r1],#4
; Проверить превышение максимально допустимого адреса
        CMP r1, r2
; Если да, пере-инициализировать указатель кольцевого
; буфера начальным адресом буфера
        MOVHI r1,r0
; Выход из подпрограммы
        BX lr
;*******************************************************
```

Продемонстрируем особенности программирования кольцевых буферов – проект CPU_20, файл MyProg_20_1.s. Закрепим за кольцевым буфером регистры r0, r1, r2 в качестве указателей минимального, текущего и максимального адреса кольцевого буфера. Выделим дополнительно регистр r3 в качестве указателя адреса порта ввода данных (входной переменной x[k]). Регистры r4, r5 зарезервируем для временного использования в подпрограммах обслуживания регулятора/фильтра.

Как обычно, в начале основной программы определим переменную длины кольцевого буфера, а в ее конце – зарезервируем память под буфер. Ячейка памяти Port_X будет имитировать порт ввода данных.

В состав подпрограмм обслуживания кольцевого буфера включим процедуру его инициализации. Она выполняет начальную установку регистров ЦПУ, содержащих параметры кольцевого буфера, а также предварительно очищает сам буфер.

Обратите внимание на то, что установленные в этой подпрограмме параметры кольцевого буфера будут использоваться во всех подпрограммах обслуживания кольцевого буфера – регистры r0-r3 будут выполнять функцию «зарезервированных» для текущей задачи регистров. Так, регистр r0 будет содержать начальный адрес, а регистр r2 - конечный адрес буфера.

Регистр r1 будет указателем очередной самой «старой» ячейки буфера, в которую можно записать очередную выборку x[k].

Сама подпрограмма записи новых данных в кольцевой буфер при таком подходе существенно упрощается. Данные считываются из порта ввода и сразу записываются на место самой «старой» выборки с авто-пост смещением указателя на длину слова. После чего следует проверка выхода указателя кольцевого буфера за максимально допустимое значение. Если это так, то указатель пере-инициализируется начальным адресом буфера. Обратите внимание на то, что эта операция выполняется с помощью условно выполняемой команды MOVHI. Если содержимое указателя, как числа без знака, строго выше максимально-допустимого адреса – следует перезапись. В противном случае команда просто пропускается.

Мы предложили вариант, когда для доступа к данным в кольцевом буфере при его обработке используется следующий порядок извлечения выборок: x[k], x[k-1],…x[k-n] – начиная с самой «новой» в направлении самой «старой». Напишем подпрограмму, которая будет извлекать одну выборку, автоматически переставляя указатель и проверяя возможный его выход за пределы буфера.

```
88  ;*********************************************************
89  ; Подпрограмма чтения очередной выборки из
90  ; кольцевого буфера с предварительным декрементированием
91  ; указателя и проверкой выхода за минимально допустимый
92  ; адрес
93  ; Выход: r4 - считанные из буфера данные
94  ;*********************************************************
95  Read_Circ_Buffer
96  ; Декрементировать на 4 текущее значение в указателе
97  ; кольцевого буфера
98          SUB r1,#4
99  ; Сравнить с минимально допустимым адресом
100         CMP r1, r0
101 ; Если "строго ниже", заменить максимальным адресом
102         MOVLO r1,r2
103 ; Считать выборку в регистр r4
104         LDR r4, [r1]
105 ; Выход из подпрограммы
106         BX lr
107 ;*********************************************************
```

В ее начале следует декрементирование указателя кольцевого буфера на длину слова и проверка возможного выхода за пределы буфера «вниз» – в направлении уменьшающихся адресов. Если указатель выходит за нижнюю границу, он автоматически перезаписывается максимальным адресом буфера. В противном случае команда перезаписи MOVLO – пропускается.

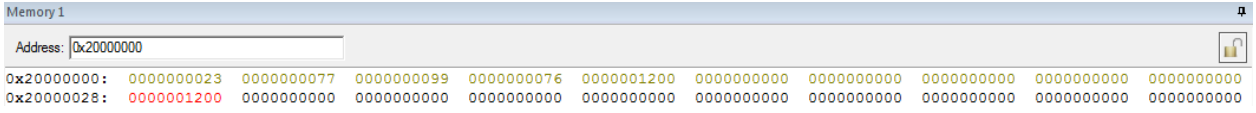
Далее (глава 17) мы покажем, как пользоваться этой подпрограммой при обработке данных (при выполнении умножений выборок на коэффициенты фильтра и накоплении).

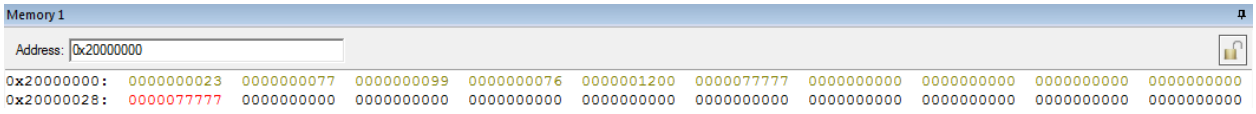
Здесь же проверим работу подпрограммы записи новых данных в кольцевой буфер.

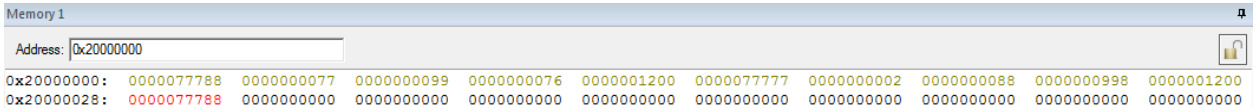
```
22  ; Инициализация параметров кольцевого буфера,
23  ; очистка кольцевого буфера
24          BL Inicy_Circ_Buffer
25  M1
26  ; Вызов подпрограммы записи в буфер новой выборки
27          BL Write_Circ_Buffer
28  ; Повторить для нового значение входной переменной
29          B M1
```

В основной программе вначале вызывается процедура инициализации кольцевого буфера, а затем (многократно) процедура записи в него новых данных.

Поставьте точку останова при отладке программы в строке 29. Произойдет очистка кольцевого буфера, и первая выборка из порта ввода (0x20000028) запишется в буфер. Модифицируйте содержимое ячейки, содержащей входную переменную, и следите за тем, как очередная выборка заполняет буфер:







После того, как все 10 ячеек буфера будут заполнены, процесс записи автоматически продолжится с первой ячейки буфера:

1) Чем кольцевой буфер выборок существенно отличается от линейного буфера?

2) Подпрограмма считывания данных из кольцевого буфера предполагает следующий порядок доступа к выборкам: x[k], x[k-1], … При этом указатель вначале пре-декрементируется, а затем выполняется

считывание выборки из буфера. На какой ячейке кольцевого буфера окажется указатель, если будут считаны все выборки?

3) В каком порядке при обработке данных должны считываться коэффициенты фильтра?
4) Можно ли вести обработку данных в кольцевом буфере, начиная не с самой «новой», а с самой «старой» выборки? Как в этом случае изменится подпрограмма чтения данных из кольцевого буфера? В каком порядке при этом должны располагаться коэффициенты фильтра?
5) Каковы отличия буфера предыдущих значений выходных управляющих воздействий y[k-1], y[k-2],… от буфера входных выборок x[k], x[k-1], …?

1) В таком буфере данные не перезаписываются, лишь смещается указатель самой «старой» выборки (ячейки, «свободной» для новой записи). Это существенно повышает быстродействие при обработке данных в регуляторах/фильтрах высокого порядка.

2) На той же самой, на которой он был до обработки данных. После обработки можно вновь вызывать подпрограмму считывания в буфер очередной выборки.

3) В порядке b_0, b_1, b_2… Именно в таком порядке они и должны располагаться в таблице коэффициентов фильтра, записанной в ПЗУ.
4) Да. Нужно вначале считать данные из буфера по текущему адресу, а затем – инкрементировать указатель (+4) с проверкой достижения не нижней, а уже верхней допустимой границы буфера. Коэффициенты должны располагаться в таблице в обратном порядке b_n…b_2, b_1, b_0. Обратите внимание на то, что при таком подходе указатель кольцевого буфера перемещается только «вперед» и требуется контроль только верхней границы буфера.
5) Принцип работы этого буфера такой же, он так же очищается перед началом работы регулятора/фильтра, и указатель текущей ячейки устанавливается на первую ячейку. Отличие: в буфер поступает значение выходной переменной y[k], рассчитанное на предыдущем такте, то есть тогда, когда выполнена обработка обоих буферов БИХ-фильтра, рассчитаны значения сумм S1, S2 и S3 (см. рис. 16.4б).

Мощные средства работы с указателями данных позволяют в процессорах ARM легко реализовывать кольцевые буфера с минимальными затратами как памяти, так и процессорного времени. При этом отсутствует необходимость в использовании специальных методов адресации кольцевых буферов, как в классических сигнальных процессорах и микроконтроллерах.

16.7 Методы отладки программ с использованием логического анализатора

16.7.1 Назначение и возможности логического анализатора

Средства графического представления информации в процессе отладки программного обеспечения являются незаменимыми помощниками разработчиков сложных систем управления, содержащих цифровые регуляторы и фильтры. Это средства *цифрового*

осциллографирования динамических процессов. Они позволяют вывести на экран компьютера графики изменения в функции времени сразу нескольких переменных (как входных, так и выходных) и визуально оценить правильность реализации алгоритма, быстро найти ошибки и неточности.

В комплект интегрированных средств разработчика ПО для микропроцессорных систем на базе ARM-ядер µVision входит *логический анализатор сигналов (Logic Analyzer)* – мощный инструмент для отладки программ с использованием симулятора или аппаратных трассировщиков данных. Его основные возможности:

- Одновременный вывод на экран графиков максимум 4-х переменных в функции времени. Переменные должны быть *глобальными переменными* проекта.
- Автоматическое включение в список *переменных для трассировки* (авто-съема данных по факту доступа к ним по записи/чтению) выбранных программистом наблюдаемых переменных. После запуска программы пользователя в режиме RUN (в том числе до точки останова) начинается трассировка данных, и графики изменения переменных немедленно отображаются в окне логического анализатора.
- Поддерживается несколько способов отображения информации в окне логического анализатора: в виде графиков аналоговых сигналов, сигналов состояния (переключаемых сигналов), битовых сигналов (флагов).
- Значения переменных, выводимых на экран, могут предварительно маскироваться и сдвигаться – можно графически исследовать любой байт, полуслово, битовое поле или бит наблюдаемой глобальной переменной.
- Имеются средства масштабирования графиков для более подробного изучения деталей динамических процессов и просмотра нужного участка графика (скроллинг по оси времени).
- Предусмотрена оцифровка любой точки графика как по оси ординат (значение переменной), так и по оси абсцисс (времени) с использованием специальных контрольных линий.
- Данные регистрации значений наблюдаемой переменной можно сохранить в файле на диске для последующего автономного наблюдения в логическом анализаторе (в режиме «off-line») или изучения с использованием других компьютерных программ графического отображения данных.
- Есть возможность виртуального симулирования в выбранном Вами микроконтроллере периферийных устройств (порты ввода/вывода, АЦП, ЦАП, ШИМ-генераторы и др.), но только для микроконтроллеров известных производителей и исключительно при отладке в симуляторе.

16.7.2 Имеющиеся ограничения

- Регистрация одновременно не более 4-х динамических процессов. Если при отладке программы Вы используете точки останова по факту доступа к памяти (по записи или чтению), то число наблюдаемых переменных сокращается на число точек останова.
- Все наблюдаемые переменные должны быть глобальными – при программировании на Ассемблере – объявленными с помощью директивы EXPORT.
- Возможно представление графиков только в функции времени – фазовые портреты не поддерживаются.
- Графическая регистрация содержимого регистров ЦПУ – не поддерживается. Однако, Вы можете скопировать содержимое нужного регистра в переменную в памяти (глобальную) и снять это ограничение.

- Не поддерживается регистрация содержимого реальных, встроенных в микроконтроллер, регистров периферийных устройств.

Рассмотрим особенности работы с логическим анализатором сначала на примере простого многоканального генератора периодических сигналов, а затем исследуем с его помощью работу кольцевого буферного регистра.

16.7.3 Отладка программы генератора периодических сигналов с использованием логического анализатора

Рассмотрим технологию использования логического анализатора для отладки программ на конкретном примере. Требуется создать четыре типа генераторов периодических сигналов (простых, без использования системы прерываний):

1) *Линейно нарастающего* сигнала в диапазоне 0 – 255 с последующим авто-сбросом в 0 и повторной генерацией очередного периода.
2) *Треугольного сигнала*, нарастающего от 0 до 127 в первой половине полупериода сигнала и спадающего с 127 до 0 во второй половине полупериода сигнала.
3) *Переключаемого сигнала*, который имеет значение 3 в первом полупериоде линейного сигнала и значение 12 во втором полупериоде линейного сигнала.
4) *Битового сигнала*, состояние которого равно 0 в первом полупериоде и 1 – во втором полупериоде линейного сигнала.

```
88  ;*************************************************************
89  ; Объявить секцию данных в ОЗУ для размещения наблюдаемых
90  ; переменных - выходов генераторов периодических сигналов
91  ; Все переменные должны быть глобальными
92              ALIGN
93              AREA MyData, DATA, ReadWrite
94  ; Выход генератора линейного сигнала
95  Lin_Signal  SPACE 4
96              EXPORT Lin_Signal
97  ; Вход генератора треугольного сигнала
98  Tri_Signal  SPACE 4
99              EXPORT Tri_Signal
100 ; Выход генератора переключаемого сигнала (меандра)
101 Rec_Signal  SPACE 4
102             EXPORT Rec_Signal
103 ; Выход генератора битового флага
104 Bit_Signal  SPACE 4
105             EXPORT Bit_Signal
106 ;*************************************************************
```

Требуется сгенерировать заданное число периодов всех 4-х сигналов и графически отобразить работу генераторов в окне логического анализатора. Создадим для решения задачи отдельный проект CPU_21 (программа приложения MyProg_21.s). Заметим, что при инкрементировании любого из регистров ЦПУ его состояние линейно нарастает от 0 до (232-1). Такой регистр можно использовать в качестве генератора опорного сигнала, так как состояние каждого байта, полуслова и даже битового поля этого регистра периодически повторяется. Пусть опорный сигнал генерируется в регистре r0, а для представления нужных нам 4-х сигналов используются регистры r1, r2, r3, r4. Так как содержимое регистров ЦПУ не может быть отображено в логическом анализаторе.

Для каждого из генераторов предусмотрим отдельное слово в памяти данных, которое будем рассматривать как *выход* соответствующего *генератора*. Объявим эти ячейки памяти глобальными переменными для возможности трассировки данных при доступе к ним. Как обычно, резервирование данных в ОЗУ выполним в конце программы. Для доступа к выходным переменным генераторов будем использовать четыре регистра-указателя: r5, r6, r7, r8

```
Gen_Signal
; Основная программа
; Генератор четырех периодических сигналов разной формы
;*********************************************************
; r0 - линейно нарастающий сигнал без ограничений (в слове)
;      опорный сигнал
; r1 - линейно нанастающий сигнал в диапазоне от 0 до 255
;      и далее авто-повторяемый
; r2 - треугольный периодический сигнал (растет от 0 до 127,
;      далее уменьшается до 0 и вновь нарастает)
; r3 - меандр (в первой половине периода треугольного сигнала
;      3, во второй 12)
; r4 - битовый (в первой половине периода треугольного сигнала
;      0, во второй 1
;*********************************************************
; Число отображаемых на графиках периодов сигналов
N       EQU  3
; Инициализировать начальные значения переменных в регистрах
        MOV r0,#0
        MOV r1,r0
        MOV r2,r0
        MOV r3,#3
        MOV r4,r0
; Инициализировать указатели выходных переменных в памяти
        LDR r5,=Lin_Signal
        LDR r6,=Tri_Signal
        LDR r7,=Rec_Signal
        LDR r8,=Bit_Signal
; Выдать начальные значения сигналов генераторов
        STRH r1,[r5]
        STRH r2,[r6]
        STRH r3,[r7]
        STRH r4,[r8]

; Основной цикл генерации периодических сигналов
; Инициализировать счетчик числа периодов r9
        MOV r9,#N
```

```
Loop
;*************************************************
; Генератор линейно-нарастающего опорного сигнала
        ADD r0,#1
; Выделить из опорного сигнала только младший байт
        ANDS r1,r0,#0xFF
; Если очередной период отработан (его состояние 0)
; декрементировать счетчик числа отработанных периодов
        SUBEQ r9,#1
;*************************************************

; Половина периода линейного сигнала отработана?
        CMP r1,#127
;*************************************************
; Генератор треугольного периодического сигнала
; Нет, выдать сигнал линейного генератора
        MOVLS r2,r1
; Да, выдать сигнал: (255-сигнал линейного генератора)
        RSBHI r2,r1,#255
;*************************************************
;*************************************************
; Генератор переключаемого сигнала (меандра)
        MOVLS r3,#3
        MOVHI r3,#12
;*************************************************
;*************************************************
; Генератор битового сигнала - (флага)
        MOVLS r4,#0
        MOVHI r4,#1
;*************************************************

; Выдать текущие значения сигналов генераторов
        STRH r1,[r5]
        STRH r2,[r6]
        STRH r3,[r7]
        STRH r4,[r8]
; Все периоды отработаны?
        CMP r9, #0
        BNE Loop    ; Повторить генерацию очередного периода
; Да. Завершить работу генератора периодических сигналов
```

Выполним инициализацию начальных значений всех сигналов, а также указателей выходных переменных генераторов. Зададим нужное число периодов сигналов.

Сигнал первого генератора r1 можно легко получить из опорного сигнала, просто выделив младший байт регистра r0. Второй сигнал требует сравнения первого сигнала со значением 127. Если он «ниже или тот же самый», то второй сигнал должен просто повторить первый r2=(r1). В противном случае – генерируется спадающая часть второго сигнала – по альтернативному условию «строго выше» (от 128 до 255). Один из возможных вариантов реализации:

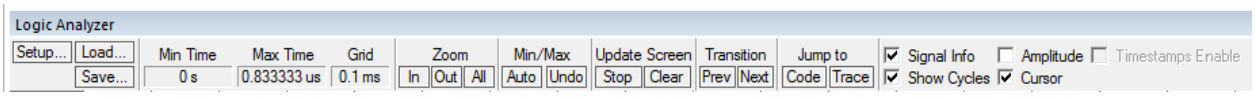
r2 = [255-(r1)].

Как видите, мы точно определили середину периода первого сигнала и для каждого полупериода отдельно сформировали второй выходной сигнал.

Третий и четвертый сигналы генерируются по тому же условию достижения половины периода первого.

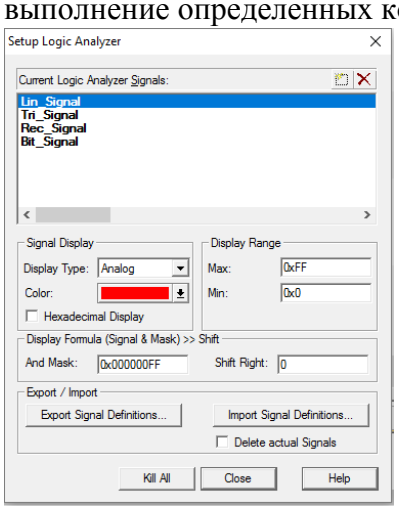
Команда сравнения CMP r1,#127 используется только один раз. Дальше применены несколько команд условного выполнения по уже сформированному условию (LS/HI). Этот пример еще раз доказывает преимущества технологии ARM, обеспечивающей условное выполнение последовательной группы команд без дополнительных команд условной передачи управления. В конце программы выполняется обновление выходных переменных генераторов и, если не все периоды отработаны, то генерация продолжается.

Итак, после трансляции и сборки проекта перейдем в режим отладки с использованием симулятора и вызовем логический анализатор:



Это можно сделать двумя способами – пиктограммой в строке команд отладчика или из меню **View - Analysis Windows (Просмотр – Окно Анализатора)**. Перед Вами появится окно логического анализатора с расположенными в его верхней части

названиями меню (показана только верхняя часть окна). Каждое из меню обеспечивает выполнение определенных команд управления логическим анализатором.



С помощью меню Setup Вы можете выбрать *наблюдаемые переменные* из списка глобальных переменных проекта. Вызовите это меню и последовательно вводите имена переменных для их графического отображения в окне логического анализатора. В нашем случае это переменные выходов генераторов: Lin_signal, Tri_signal, Rec_signal и Bit_signal. В верхней части меню Setup имеются две пиктограммы-команды: первая для включения новой переменной в список наблюдаемых, а вторая – для удаления из списка выделенной переменной. Выделение выполняется щелчком клавиши мыши на имени переменной.

Для каждой из наблюдаемых переменных можно выбрать тип отображения сигнала на графике (**Display Type**), цвет сигнала (**Color**) и опцию оцифровки значений в шестнадцатеричном формате (**Hexadecimal Display**). По умолчанию оцифровка выполняется в десятичной системе счисления. Поддерживаются три типа отображения сигналов:

- *Analog* – обычные аналоговые сигналы (приемлемо для большинства переменных);
- *State* – переменные, меняющие состояние с одного на другое (как на временных диаграммах сигналов в процессорной технике);
- *Bit* – битовые переменные (флаги), состояние которых может быть только 1 или 0.

Для каждого из наблюдаемых сигналов независимо можно выбрать так называемую Дисплейную формулу **Display Formula (Signal & Mask) >> Shift**, которой задается 32-разрядная маска, позволяющая выделить из наблюдаемой переменной любое полуслово, байт или битовое поле и дополнительно сдвинуть его на заданное число разрядов вправо. Применительно к нашей задаче для первых двух сигналов выделим младшие байты, для третьего сигнала – младший нибл, а для четвертого – младший бит. Для всех этих сигналов – сдвиг вправо не требуется (0).

В нижней части меню имеются подменю, которые позволяют экспортировать наблюдаемый сигнал в файл на диске с целью его дальнейшего исследования либо в логическом анализаторе (в автономном режиме), либо в стандартных компьютерных программах построения графиков.

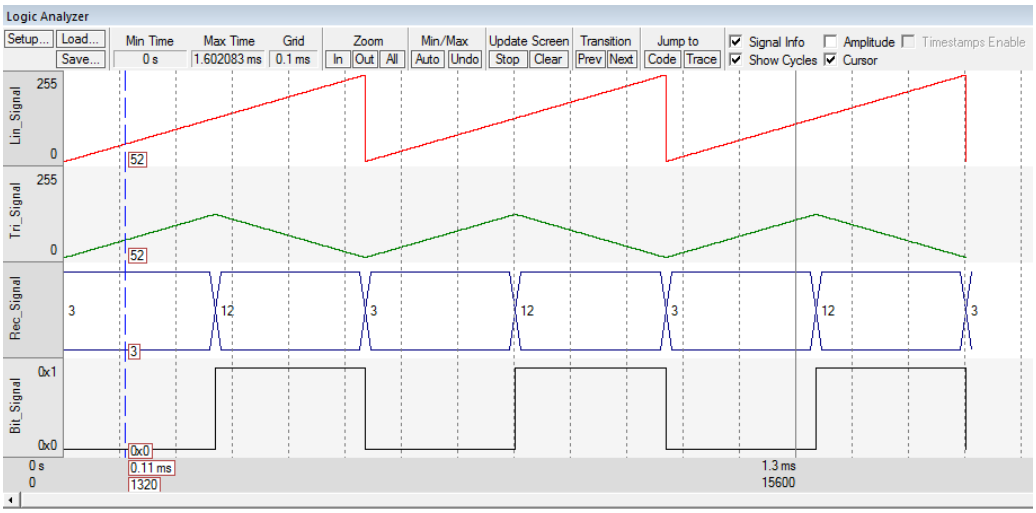


Рис. 16.7 Окно графического анализатора

```
85          BNE Loop    ; Повторить генерацию очередного периода
86  ; Да. Завершить работу генератора периодических сигналов
87  ; Предотвратить выполнение несуществующего кода
88          B .
```

Установим точку останова на первой команде после программы генератора периодических сигналов, запустим программу на выполнение. Начнется автоматический съем и регистрация данных с одновременным отображением состояния переменных в окне анализатора (рис. 16.7).

На рис. 16.7 вы видите – сгенерировано три периода сигналов, как и требовалось по заданию. Первые два сигнала – аналоговые, третий – сигнал состояния, четвертый – битовый. Обратите внимание на то, что сигнал состояния отображается *переходом* переменной от одного значения к другому. При этом численное значение переменной отображается внутри диаграммы смены состояния.

В левой части окна указаны символические имена переменных и диапазоны их изменения. В окнах Min Time и Max Time указано время начала и конца регистрации данных, а в окне Grid (Сетка) – временной интервал между вертикальными серыми пунктирными линиями.

Изменить диапазон отображаемых на графике значений можно с помощью меню «Min/Max». Очень удобна команда «Auto», которая масштабирует выделенный сигнал автоматически в зависимости от фактического диапазона считанных значений.

Вертикальная синяя контрольная линия может быть передвинута мышью в любое место окна. Она показывает оцифрованные текущие значения всех наблюдаемых переменных, а также времени (в красных прямоугольных окошках). Обратите внимание на то, что текущие значения 1-го и 2-го сигналов на первом полупериоде совпадают, значение переменной состояния равно 3, а битовой переменной 0, как и должно быть.

Если опция «Show Cycles» (Показать циклы) в меню логического анализатора включена, то дополнительно к времени выводится также число процессорных циклов, выполненных на данный момент. Вы можете, щелкнув клавишей мыши, переставить контрольную линию в другое место. В этом случае окошки оцифровки будут показывать не только новые значения переменных, но и их приращения по отношению к предыдущим значениям (например, для первого сигнала d:49). Если включить опцию «Signal Info», то будет также выведена дополнительная информация о точке графика, на которую указывает курсор мыши.

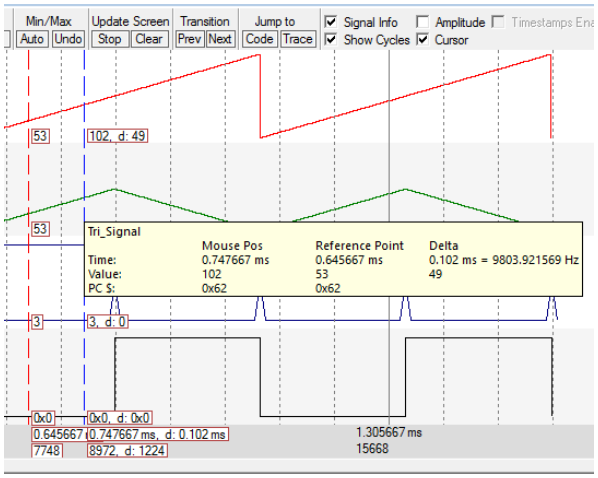


Рис. 16.8 Окно графического анализатора

Переключатель «Amplitude» разрешает включение горизонтальных контрольных линий (см. рис. 16.8), которые могут служить измерителями приращений сигналов по вертикали (по амплитуде). Возможны переключения от одной контрольной линии к другой – меню «Transition» (Переход), а также переход к просмотру соответствующего

фрагмента кода, вызвавшего изменение сигнала в данной точке графика – меню «Jump to» – «Code».

Меню «Zoom» (Зуммирование) позволяет укрупнить график (In) для детального просмотра деталей (растянуть ось времени) или, наоборот, уменьшить (Out) – сжать ее для наблюдения процесса на большем временном интервале. Кроме того, Вы можете воспользоваться клавишей прокрутки изображения в нижней части окна, если вся область регистрации данных не поместилась в текущее окно логического анализатора.

1) Поэкспериментируйте с командами управления логическим анализатором. Укрупните график в области начала периода. Убедитесь, что все генераторы работают синхронно друг с другом.

2) Какие изменения нужно внести в программу, чтобы второй генератор работал как вычитающий счетчик с двойной частотой первого генератора и начальным значением, равным 127, как показано на графике на рис. 16.9?

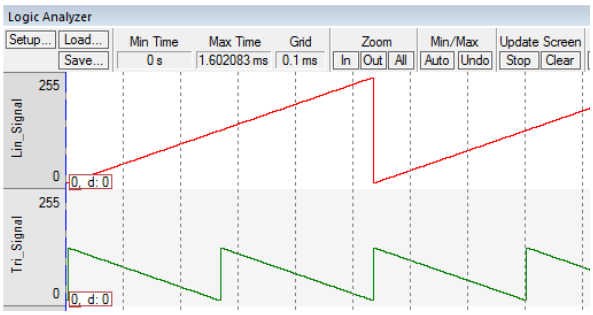


Рис. 16.9 Окно графического анализатора

2) Одно из возможных решений (MyProg_21_1.s) использует такой же способ определения полупериода первого сигнала и две команды реверсивного вычитания содержимого r1 сначала из числа 127, а затем – из числа 255 для генерации второго сигнала.

```
59          CMP r1,#127
60  ;*************************************************
61  ; Генератор треугольного периодического сигнала
62  ; Нет, выдать сигна равный 255-(r1)
63          RSBLS r2,r1,#127
64  ; Да, выдать сигнал линейного генератора
65          RSBHI r2,r1,#255
66  ;*************************************************
```

Применяйте логический анализатор среды µVision для визуализации динамических процессов в системах цифрового управления и быстрой проверки правильности реализации своих алгоритмов. Помните, что для наблюдения доступны только глобальные переменные – слова. Их трассировка выполняется по записи и по считыванию. Значения переменных только в эти моменты времени будут выведены на экран логического анализатора. Значения переменных между точками трассировки считаются неизменными. Используйте механизм маскирования и сдвига данных в слове для выделения нужных фрагментов данных и изображения их на графиках в максимально удобном для анализа диапазоне.

16.7.4 Визуализация состояния кольцевого буфера данных при отладке

В системах цифрового регулирования и фильтрации важнейшее значение имеет правильность работы подпрограмм обслуживания буферов выборок (входных данных) и предыдущих значений выходных управляющих воздействий. Малейшая неточность при работе с указателями данных приводит к полной неработоспособности программного обеспечения. Именно поэтому подпрограммы работы с буферами данных должны быть тщательно отлажены, желательно с использованием средств графической визуализации их текущего состояния. Покажем, как это можно сделать, на примере кольцевого буфера данных, уже отлаженного нами ранее с использованием обычных средств исследования содержимого ячеек памяти (см. 16.6.2 и программу MyProg_20_1.s проекта CPU_20).

```
138 ; Определение начальных 4-х ячеек кольцевого буфера
139 ; глобальными переменными для графического анализа в
140 ; логическом анализаторе
141 Circ_0  EQU ADR_0
142 Circ_1  EQU ADR_0+4
143 Circ_2  EQU ADR_0+8
144 Circ_3  EQU ADR_0+12
145         EXPORT Circ_0
146         EXPORT Circ_1
147         EXPORT Circ_2
148         EXPORT Circ_3
```

Первое, что нужно сделать – это модифицировать исходную программу так, чтобы переменные кольцевого буфера были доступны для наблюдения – сделать их глобальными. Уменьшим число переменных в кольцевом буфере до 4-х, чтобы все они могли «уместиться» на экране логического анализатора, и присвоим каждой переменной в буфере персональное имя: Circ_0, Circ_1 и т.д. (см. MyPorg_20_2.s).

Не меняя подпрограмм инициализации кольцевого буфера, записи и считывания в него очередных данных, несколько модифицируем основную программу. Сразу после инициализации буфера выполним запись в него последовательно 20 нарастающих кодов от 1 до 20, чтобы проследить, правильно ли работает система авто-смещения указателей в начало кольцевого буфера при его полном заполнении.

Затем – последовательно считаем из кольцевого буфера его текущее содержимое (5 раз), чтобы проверить, работает ли система авто-смещения указателя при выходе за пределы буфера в сторону меньших адресов.

```
19 MyProg
20 ; Основная программа
21 ; Определение переменной "емкость" кольцевого буфера
22 ; (Число 32-разрядных слов в буфере)
23 N       EQU   4
24 ; Инициализация параметров кольцевого буфера,
25 ; очистка кольцевого буфера
26         BL Inicy_Circ_Buffer
27 ; Заполнить кольцевой буфер выборок нарастающим кодом от 1 до 20
28 ; для графической отладки подпрограмм в логическом анализаторе
29 ; Число записей в кольцевой буфер
30         MOV r5, #20
31 ; Начальное значение выборки
32         MOV r6, #0
33 ; Инкрементировать значение выборки (имитация нового значения)
34 M1      ADD r6,#1
35 ; Записать значение "новой" выборки в Port_X
36         STR r6,[r3]
37 ; Сохранить значение выборки в кольцевом буфере
38         BL Write_Circ_Buffer
39 ; Декрементировать счетчик числа записей в кольцевой буфер
40         SUBS r5,#1
41 ; Если не все записи сделаны, повторить для новой выборки
42         BNE M1
43 ; Процесс последовательного заполнения кольцевого буфера завершен
44 ; Извечь 5 выборок из кольцевого буфера
45         MOV r5,#5
46 M2
47         BL  Read_Circ_Buffer
48         SUBS r5,#1
49         BNE M2
50 ; Ограничить ход выполнения программы
51         B .
```

Поставим первую точку останова в конце записи данных в буфер, а вторую – в конце считывания данных из буфера.

Убедитесь, что содержимое подпрограмм работы с кольцевым буфером осталось прежним (сравните MyProg_20_2.s и MyProg_20_1.s). Теперь можно включить в проект новую программу, выполнить ее трансляцию и сборку проекта. Начните отладку в симуляторе, откройте окно логического анализатора и воспользовавшись командой Setup задайте имена наблюдаемых переменных (ячеек кольцевого буфера) и параметры представления сигналов на экране логического анализатора. Запустите процесс выполнения до первой точки останова. Содержимое кольцевого буфера будет отображено на экране (рис. 16.10).

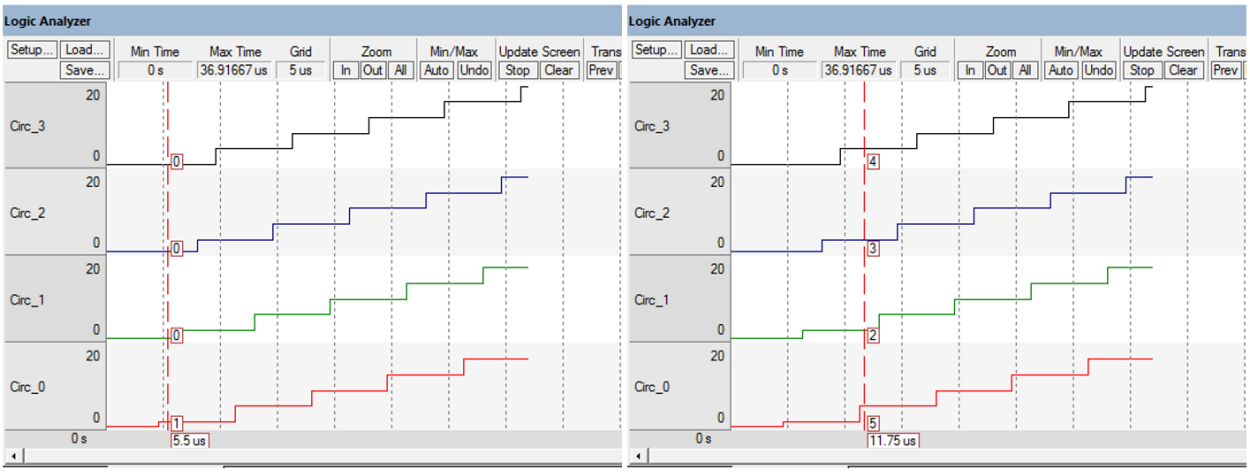


Рис. 16.10 Окно графического анализатора

Как видите (рис. 16.10), в начале программы выполняется обнуление данных в кольцевом буфере, а затем в него последовательно записываются числа 1, 2, 3, 4. Запись числа 5 выполняется уже в начальную ячейку буфера Circ_0. Следовательно, механизм авто-смещения указателя кольцевого буфера при записи в него новых данных – работает. Следующие записи делаются в очередные «свободные» ячейки буфера и самые «старые» данные автоматически замещаются новыми.

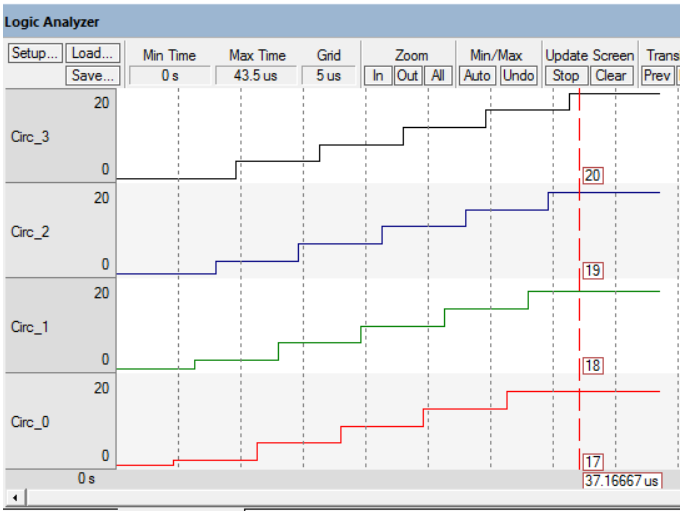


Рис. 16.11 Окно графического анализатора

Продолжим выполнение программы до следующей точки останова (рис. 16.11). В это время содержимое буфера не обновляется, а данные из него просто считываются. Как мы уже отмечали, съем данных для графического представления выполняется как по записи, так и по чтению. Продолжение графиков свидетельствует о том, что данные из буфера действительно считываются, но их содержимое в буфере не меняется (рис. 16.11).

Таким образом, правильность алгоритма обслуживания кольцевого буфера данных в памяти подтверждается.

1) Измените набор данных для заполнения кольцевого буфера и их число. Выполните отладку программы.

2) Исследуйте возможности маскирования значений наблюдаемых переменных и их сдвига, авто-настройки диапазона вывода данных на экран по амплитуде, масштабирования графиков по оси времени.

16.8 Преимущества переменных в относительных единицах

Цифровая система управления может обрабатывать данные в *физических величинах* (Метрах, Амперах, Вольтах, …) или в *относительных величинах*. С математической точки зрения *относительное значение физической переменной* – это отношение ее фактического значения к номинальному значению, которое обычно соответствует так называемому номинальному режиму работы устройства:

$$x^* = \frac{x}{x_{\text{ном}}}. \qquad (16.16)$$

При этом номинальное значение переменной обычно отличается от максимально возможного значения. Так, при пуске двигателя пусковые токи могут в 5 – 7 раз превышать номинальные значения. Существует такое понятие, как возможная *перегрузка* по току или моменту двигателя, в общем случае – *коэффициент превышения максимально возможного значения переменной над ее номинальным значением*:

$$k_{max} = \frac{x_{max}}{x_{\text{ном}}}. \qquad (16.17)$$

Коэффициент k_{max} в реальных системах управления обычно не больше 10. Если физические значения переменных очень сильно отличаются друг от друга, то относительные значения – нет. Так, для электрического двигателя значения токов могут измеряться амперами и килоамперами, – то есть отличаться друг от друга на несколько порядков. Те же значения в относительных единицах будут находиться для всех типов двигателей примерно в одном и том же диапазоне значений 0÷±7. Более того, диапазоны всех переменных в системе управления становятся *сопоставимыми* – рис. 16.12.

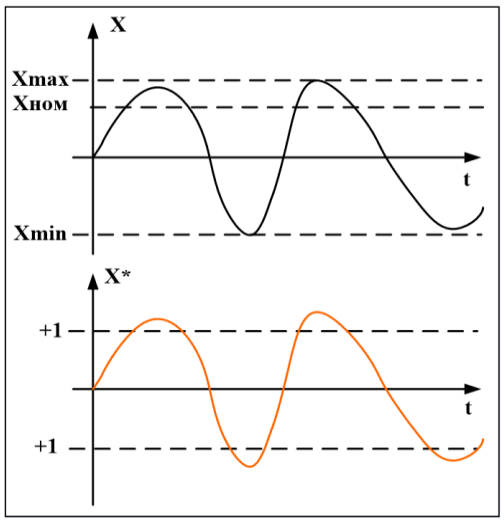


Рис. 16.12. Представление переменных систем управления в относительных единицах.

Так, скорость в относительных единицах может изменяться примерно в том же диапазоне, что и токи: 0÷±7. Это существенное преимущество, которое особенно важно при использовании представления переменных *в формате с фиксированной точкой*. Если максимальное значение целой части всех относительных переменных не превышает ±7, то для их представления в МПСУ достаточно выбрать формат вещественных чисел с фиксированной точкой S4.28 или даже S4.12 (см. 3.8.3). Это числа со знаком, в которых для целой части числа отведено 4 разряда, включая знаковый, а для дробной – 28 или 12 разрядов, соответственно. С учетом ограниченной разрядности вводимых в систему управления переменных, например, с 10- – 12-разрядного АЦП, точности формата S4.28 или даже S4.12, используемого внутри системы управления, обычно вполне достаточно.

Если за номинальные значения принимаются максимально возможные значение переменных, то их относительные значения никогда не превышают ±1, то есть будут дробными числами, которые можно представить в форматах S1.31 или S1.15. Основное преимущество такого представления – при умножении двух дробей произведение всегда меньше любого из операндов. Следовательно, в операциях умножения можно не контролировать возможные переполнения. Впрочем, *при выполнении операций накопления (сложения) произведений такой контроль все равно понадобится*.

Отметим, что значительное число фирм разрабатывают и предлагают разработчикам библиотеки поддержки вычислений в так называемых i.q-форматах (форматах с фиксированной точкой) для разных типов процессоров, в том числе для ARM. Вы можете выбрать приемлемый для Вашей задачи формат представления переменных и, используя такую библиотеку, выполнять все вычисления в этом формате.

Таким образом, имеются две возможности выбора форматов переменных в системах цифрового управления:

1) Формат вещественных чисел с фиксированной точкой (операционная поддержка в процессорных ядрах Cortex-M4 и M4F). При этом целесообразен переход от физических переменных к относительным для повышения точности расчетов.
2) Формат вещественных чисел с плавающей точкой (операционная поддержка только в ядрах Cortex-M4F). Можно работать с переменными как в относительных, так и физических единицах, что удобнее при отладке реальных объектов.

16.9 Особенности чисел с фиксированной точкой и операций с ними

16.9.1 Неявное расположение фиксированной точки

Процессор при выполнении любых команд воспринимает операнды как простые битовые наборы. Напомним, как в процессоре представляются, например, *дробные числа со зн*аком – рис. 16.13. Старший разряд числа является знаковым, после него «как бы» стоит десятичная точка, а далее располагаются двоичные разряды дробной части, весовые коэффициенты которых, в соответствии с двоичной позиционной системой счисления, равны: 2^{-1}, 2^{-2}, …

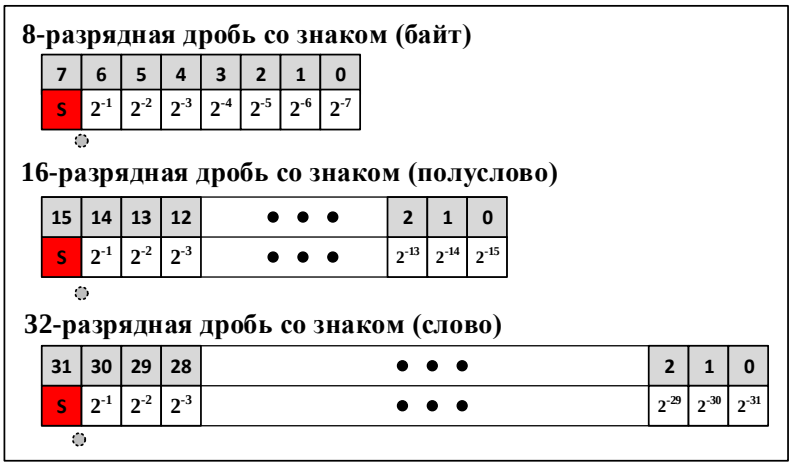


Рис. 16.13 Представление дробей

Явное место (выделенный специально разряд) для расположения десятичной точки *не предусматривается* – ее положение знает только программист и только он, зная это положение, может правильно выбрать необходимую операцию и оценить результат ее выполнения.

При выбранной разрядности дробного числа n общее число возможных дробных чисел со знаком равно 2^n, из них половина чисел, включая ноль – положительные, а половина - отрицательные. Представление таких чисел в формате S1.3 показано на рис. 16.14 в качестве примера. Обратите внимание на то, что *ноль в таком представлении относится к диапазону положительных чисел (+0.0).*

Дробь	Двоичный код
+0.0	0000
+0.125	0001
+0.25	0010
+0.375	0011
+0.5	0100
+0.625	0101
+0.75	0110
+0.875	0111
-1.0	1000
-0.875	1001
-0.75	1010
-0.625	1011
-0.5	1100
-0.375	1101
-0.25	1110
-0.125	1111

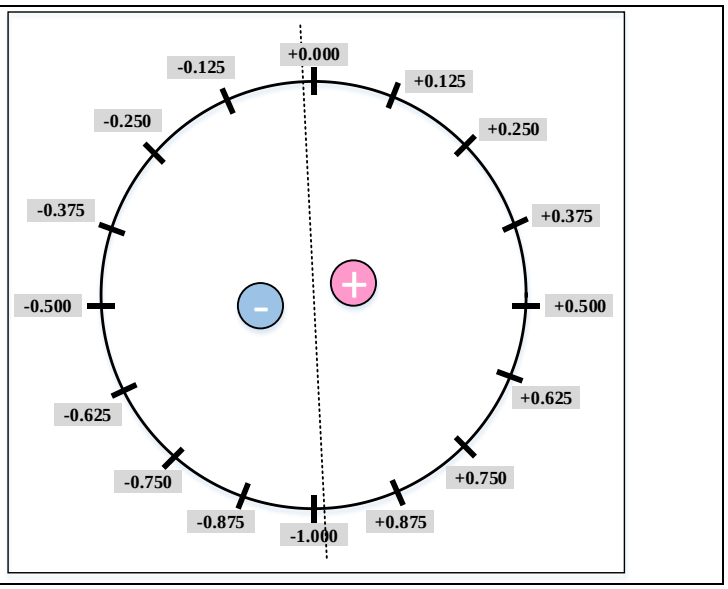


Рис. 16.14 Пример кодировки 4-битовых дробей со знаком в формате S1.3

Если известен двоичный код дроби, то ее десятичное значение определяется так:

Дробь$_{10}$ = S·(-1) + (десятичные веса всех, установленных в 1 разрядов дробной части).

Значение знакового разряда S для положительных чисел S=0, для отрицательных S=1.

В формате вещественных чисел с фиксированной точкой (i.q) положение десятичной точки также неявное: она располагается как бы между младшим разрядом целой части числа и старшим разрядом дробной части – рис. 16.15.

Рис. 16.15 Вещественное число с фиксированной точкой

Чтобы различать вещественные числа с фиксированной точкой без знака и со знаком, к обозначению формата числа добавляется префикс «U» (Unsigned) или «S» (Signed). Так, число в формате U4.28 является вещественным числом без знака, а число в формате S4.28 – числом со знаком в дополнительном коде.

В формате 32-разрядного слова ARM-процессоров существует большое многообразие форматов чисел с фиксированной точкой, например, (4.28), (8.24), (16.16) и т.д. Выбор конкретного формата зависит от приложения, квалификации программиста, и конечно, от требуемой точности представления переменных. В первом формате под целую часть программист отвел только 4 разряда, во втором – 8, а в третьем – 16. Остальные разряды представляют собой дробную часть числа. Очевидно, что точность представления чисел в первом случае – максимальная, *если используются относительные единицы*.

Точность представления переменных во всех форматах с фиксированной точкой определяется десятичным весом самого младшего разряда дробной части числа. Так для двоичных дробей в наиболее употребительных форматах получим результат, представленный в табл. 16.1.

Таблица 16.1 Точность представления относительных переменных в формате дробей со знаком

Формат дроби	Десятичный вес младшего разряда	Точность в %
S1.7 (байт)	**2^{-7} = 0.0078125**	**≈0.8**
S1.15 (полуслово)	**2^{-15} = 0.000030517578125**	**≈0.003**
S1.31 (слово)	**2^{-31} = 0.00000000046566128…**	**≈0.00000005**

Уже при 16-разрядном формате относительной величины (полуслово) точность можно считать достаточной для большинства систем управления, не говоря уже о 32-разрядном формате (слова), где точность более чем достаточная. При переходе к относительным единицам с учетом превышения переменной ее номинального значения, когда под целую часть числа (включая знак) отводится некоторое число разрядов, превышающее один (знаковый), точность будет падать.

Положение десятичной точки во всех форматах чисел с фиксированной точкой знает только программист. Арифметические операции сложения и вычитания таких чисел могут выполняться только в том случае, если положение точек в операндах одинаково, в противном случае необходимы дополнительные операции преобразования формата одного из операндов к формату другого.

1) Можно ли в формате дробного числа точно представить (+1.0)?
2) Переменные в системе представлены вещественными числами в формате S4.12. Какова точность их представления?
3) Если все переменные проекта представлены в относительных единицах и диапазон их изменения не более ±5, какой формат представления чисел Вы выберете?
4) Какому десятичному числу соответствует вещественное число с фиксированной точкой в указанном ниже формате: а) **U4.4** = 2_11110000; б) **S4.4** = 2_11110000; в) **S8.8** = 0xFF00; г) **S16.16** = 0xFFFFFFFF.

1) Нет. В формате дроби точное представление имеет только отрицательное число (-1.0). Положительное (+1.0) может быть представлено только приближенно. Так, в байтовом формате S1.7 получим 0111 1111, что соответствует числу +0.9921875 (почти +1.0);
2) Вес младшего разряда дробной части равен 2^{-12}, что соответствует точности представления относительных переменных 0,024%.
3) Формат S4.28. Возможен также 16-разрядный формат S4.12, если точность устроит разработчика.
4) а) число без знака 15.0; б) число со знаком -1.0; в) то же самое число -1.0, но в формате S8.8; г) самое маленькое отрицательное число в формате S16.16 = $-2^{(-16)}$=0.000015258

16.9.2 Особенности умножения чисел в формате с фиксированной точкой

Самая главная особенность умножения числе в формате с фиксированной точкой состоит в том, что разрядность результата умножения равна сумме разрядов множителей – то есть удваивается при одинаковой разрядности операндов. При этом *разрядности целой части множителей складываются, как и разрядности дробной части*. Это означает, что при выполнении операций умножения чисел с фиксированной точкой справедливы

следующие правила формирования формата произведения и расположения в нем двоичной точки:

$$Um.n * Up.q = U(m+p).(n+q);$$
$$Sm.n * Sp.q = S(m+p).(n+q). \quad (16.18)$$

Правила (16.18) справедливы для любых чисел с фиксированной точкой: целых, дробных, вещественных, без знака и со знаком в дополнительном коде. Причем, число разрядов целой части в исходных операндах может отличаться. Заметьте, что это число определяется с учетом знакового разряда S.

Если целая часть содержит только знаковый разряд (для двоичных дробей), то знаковый разряд в произведении удваивается. В этом случае говорят о «размножении» знакового разряда:

$$S1.15*S1.15=S2.30 \quad (16.19)$$

Соотношение (16.19) свидетельствует о том, что при умножении дробных чисел со знаком получим произведение с двумя знаковыми разрядами и 30-ю разрядами дробной части. «Лишний» знаковый разряд можно исключить за счет последующего *логического сдвига произведения на один разряд влево*:

$$S2.30 \text{ LSL } \#1 = S1.31 \quad (16.20)$$

Отбросив «малозначимое» младшее слово результата, получим произведение в формате исходных множителей S1.15 (с точностью исходных множителей).

Итак, в 32-разрядном процессоре при использовании в качестве операндов 16-разрядных полуслов, произведения никогда не выйдут за формат 32-разрядного слова:

$$U4.12*U4.12=U8.24;$$
$$S4.12*S4.12=S8.24. \quad (16.21)$$

Однако – не обольщайтесь! Как мы показали, для реализации цифровых регуляторов/фильтров нужна не только операция умножения, но и операция умножения с накоплением. Это означает, что в процессе накопления сумма может выйти за пределы 32-х разрядов. Следовательно, необходимы или специальные меры контроля переполнений в операциях умножения с накоплением, или – использование так называемого «длинного» умножения с накоплением, когда результат является 64-битовым и переполнения фактически исключаются.

Покажем справедливость соотношений (16.18) на нескольких простых примерах. Предположим, что умножаются две пары целых чисел без знака в форматах U4.0 – 5*7 и 15*15. Поскольку числа целые, можно воспользоваться классической процедурой умножения двоичных чисел «в столбик» с последующим суммированием частных произведений. Апострофами над некоторыми разрядами частных произведений отметим переносы из младшего разряда в старший, возникающие при сложении частных произведений (см. рис. 16.16).

(5)					0	1	0	1	(15)						1	1	1	1
(7)					0	1	1	1	(15)						1	1	1	1
*									*									
					0`	1	0	1							1``	1`	1	1
+				0`	1	0	1		+					1```	1	1	1	
+			0`	1	0	1			+				1```	1	1	1		
+		0	0	0	0				+			1``	1	1	1			
(35)	=	0	1	0	0	0	1	1	(225)	=	1	1	1	0	0	0	0	1
(35)	=		32				2	1	(225)	=	128	64	32					1

Рис. 16.16 Пример двоичного умножения чисел без знака в форматах U4.0

В нижней части рисунка показаны десятичные веса единичных разрядов произведений, сумма которых и дает искомый результат в десятичной форме. Как видите, в обоих случаях получено правильное произведение в формате U8.0. Даже при максимальных значениях множителей результат не превышает максимально возможное целое число в формате U8.0 – 255.

Теперь поступим так: предположим, что те же самые битовые наборы входных операндов представляют не целые числа, а дроби без знака в формате U0.4, вещественные числа без знака в формате U2.2 и даже вещественные числа без знака в разных форматах U1.3 и U2.2. Выполним двоичное умножение, как показано на рис. 16.16, не обращая внимание на формат исходных операндов, и проверим, соответствует ли полученное произведение выходному формату, определенному соотношением (16.18).

Результаты умножений для первого битового набора сведены в таблицу 16.2.

Таблица 16.2 Точность представления относительных переменных в формате дробей со знаком

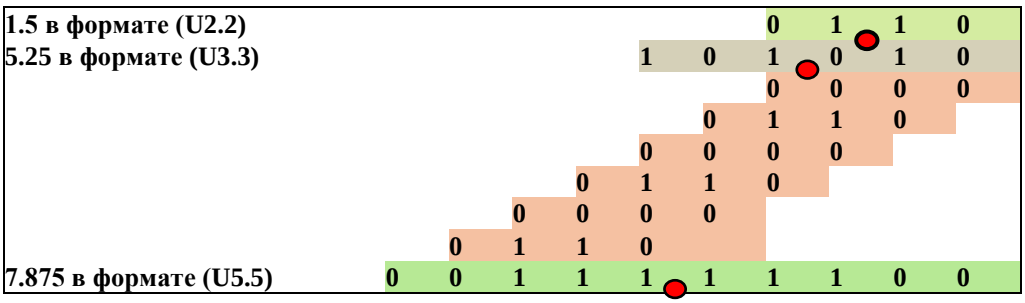
Умножение чисел без знака в различных форматах с фиксированной точкой		
U0.4*U0.4		**= U0.8**
Op1	**Op2**	**Op1*Op2**
0.0101 = $2^{(-2)}$ + $2^{(-4)}$ = 0.25 + 0.0625 = 0.3125	**0.0111 = $2^{(-2)}$ + $2^{(-3)}$ + $2^{(-4)}$ = 0.25 + 0.125 + 0.0625 = 0.4375.**	**0.00100011 = $2^{(-3)}$ + $2^{(-7)}$ + $2^{(-8)}$ = 0.125 + 0.0078125 + 0.00390625 = 0.13671875 =0.3125*0.4375**
U2.2*U2.2		**= U4.4**
Op1	**Op2**	**Op1*Op2**
01.01= 1 + 0.25 = 1.25	**01.11 = 1 + 0.5 + 0.25 = 1.75**	**0010.0011 = 2 + 0.125 + 0.0625 = 2.1875 = 1.25 * 1.75**
U1.3*U2.2		**= U3.5**
Op1	**Op2**	**Op1*Op2**
0.101 = 0.5 + 0.125 = 0.625	**01.11 = 1 + 0.5 + 0.25 = 1.75**	**001.00011 = 1 + $2^{(-4)}$ + $2^{(-5)}$ = 1 + 0.0625 + 0.03125 = 1.09375 = 0.625 * 1.75**

Таким образом, сформулированное выше правило умножения чисел без знака в разных форматах с фиксированной точкой выполняется.

Мы уже отмечали, что использование обычной технологии двоичного умножения чисел со знаком в дополнительном коде (см. 11.8) дает неверный результат. Для умножения и умножения-накопления чисел со знаком должны использоваться специальные команды, имеющие перед мнемокодом операции суффикс S (Signed) и правильно обрабатывающие знаковый разряд числа. Можно показать, что для всех знаковых операций умножения правило определения формата произведения и положения в нем двоичной точки (16.18) также выполняется.

1) Проверьте, что правило умножения чисел с фиксированной точкой выполняется и для второго набора битовых операндов (рис. 16.12) при любом положении двоичной точки в множителях.
2) Убедитесь, что при умножении чисел U2.2=1.5 и U3.3=5.25 результат будет правильным: U5.5 = 7.875.
3) При умножении дробных числе в формате полуслова S1.15 число дробных разрядов произведения удваивается S2.30. Повышается ли фактическая точность?

2) Результат умножения чисел в форматах U2.2 и U3.3:



3) Нет. Точность будет соответствовать точности множителей. Поэтому младшее полуслово произведения можно отбросить. Его целесообразно сохранять только в операциях умножения с накоплением (для повышения точности суммы).

В системе команд ARM-процессоров должны быть и имеются две группы команд умножения и умножения с накоплением: с префиксом U (для множителей без знака) и префиксом S (для множителей со знаком). Все команды умножения/накопления выполняются на аппаратном уровне всего за один такт работы процессора.

Правильный выбор типа операции, форматов исходных операндов, а также трактовка полученного результата – исключительно прерогатива программиста. Только программист, за счет корректного выбора форматов исходных данных и соответствующих им команд, может гарантированно исключить возможные переполнения в операциях умножения-накопления чисел в формате с фиксированной точкой.

16.9.3 Преобразование физических переменных к нужному формату чисел с фиксированной точкой

Оцифровка входных переменных для реализации цифровых регуляторов и фильтров обычно выполняется в АЦП. Большинство современных АЦП имеют однополярные (нереверсивные) входы, которые в состоянии обрабатывать аналоговые сигналы в диапазоне (0÷3.3 В) или (0÷5 В). Будем называть этот диапазон *номинальным диапазоном входных сигналов АЦП*.

Разработчики систем цифрового управления снабжают свои контроллеры схемами предварительной обработки сигналов (и, возможно, дополнительной аналоговой фильтрации для защиты от помех), которые позволяют принимать сигналы с датчиков в типовых *общепромышленных диапазонах*, в частности:

- Для однополярных (нереверсивных) входных сигналов: (0÷5 В), (0÷10 В) или (0÷20 мА);
- Для двухполярных (реверсивных) входных сигналов: (-5 В ÷ +5 В), (-10 В ÷ +10 В).

При этом гарантируется, что если входной аналоговый сигнал с датчика поступает на вход контроллера в стандартном диапазоне, то он автоматически преобразуется к номинальному входному диапазону АЦП, например, так:

$(0 \div X_{max}) \rightarrow (0 \div 5В) \rightarrow (0 \div 3.3В)$;

$(-X_{max} \div +X_{max}) \rightarrow (-5В \div +5В) \rightarrow (0 \div 3.3В)$.

16.9.3.1 Однополярные аналоговые сигналы

Преобразование к нужному формату зависит от разрядности АЦП. Так, при 10 разрядах максимальному входному сигналу X_{max} будет соответствовать максимальный выходной код 2^{10}-1 = 1023: целое число без знака в формате U10.0 Соответствующая передаточная функция АЦП представлена на рис. 16.17.

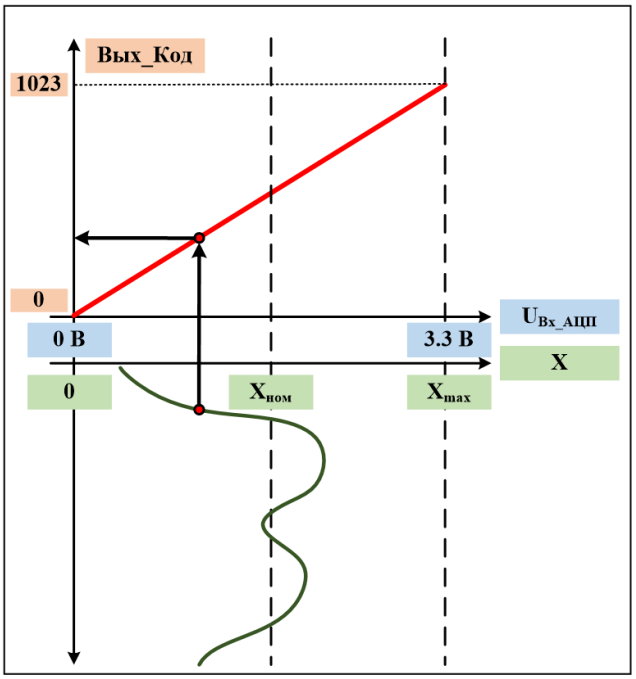


Рис. 16.17 Передаточная функция АЦП для однополярного входного сигнала

Если за базовое значение физической величины принять ее максимальное значение, то код с АЦП можно считать 10-разрядной дробью без знака в формате U0.10. Предположим, что этот код загружается в регистр ЦПУ с использованием команды загрузки из памяти полуслова без знака LDRH. Он автоматически попадает в младшее слово регистра и расширяется нулями (как число без знака) в область старших разрядов. Полученное значение в регистре будет соответствовать формату U6.10, если использовать только младшее полуслово регистра или формату U(16+6).10 = U22.10, если использовать все 32-разрядное слово (см. рис. 16.18).

15	10	9	0
		Вых. код АЦП (10 разрядов)	

31	16	15	10	9	0
				Вых. код АЦП (10 разрядов)	

Рис. 16.18 Представление данных в разных форматах

Мы получили относительную переменную (дробь) без знака в формате вещественного числа U6.10 или U22.10.

На самом деле за базу для относительных единиц принимается не максимально возможное значение, а номинальное $x_{ном}$. При этом справедливо:

$$x^* = \frac{x}{x_{ном}} = \frac{x}{x_{max}} \cdot \frac{x_{max}}{x_{ном}} = \frac{x}{x_{max}} \cdot k_{max} = x^*_{max} \cdot k_{max} \qquad (16.22)$$

То есть, реальная относительная переменная больше на коэффициент перегрузки k_{max} (коэффициент возможного превышения переменной ее номинального значения). Пусть он равен 2 (как на рис. 16.17). Тогда двоичная точка в преобразованном числе на самом деле сдвинута на один разряд вправо и фактический формат представления

относительной переменной в процессоре – U7.9 или U23.9. Для преобразования этой переменной в нужный формат, например, U4.12 нужно всего лишь логически сдвинуть полученный код на 3 разряда влево с помощью команды LSL Rm, #3:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	Относительная переменная x* (10 разрядов)									
Логический сдвиг на три разряда влево LSL Rm, #3															
0	0	0	Относительная переменная x* (13 разрядов)										0	0	0
Относительная переменная x* в формате U4.12															

1) Какие преобразования нужно выполнить, если АЦП не 10-, а 12-разрядный и переменная без знака в относительных единицах должна быть представлена в формате U4.12, причем k_{max}=2?
2) В некоторых АЦП выходной код располагается не в младшей части регистра, а в старшей, например, для 10-разрядного АЦП занимает старшие 10 разрядов. Как нужно поступить в этом случае?
3) При переходе к 16-разрядному формату от 10- или 12-разрядного формата АЦП повышается ли реальная точность представления аналоговых переменных?

1) Теперь загруженный в регистр ЦПУ код будет занимать 12 младших разрядов (11-0). Если коэффициент превышения переменной ее номинального значения по-прежнему равен 2, то относительная переменная (полуслово) будет представлена в вещественном формате U5.11. Для его преобразования в формат U4.12 потребуется логический сдвиг влево на один разряд LSL Rm, #1.
2) При том же коэффициенте превышения переменной номинального значения, считанные из АЦП данные будут в вещественном формате U1.15. При этом младшие 6 разрядов числа будут нулевыми. Для приведения к формату U4.12 нужно выполнить логический сдвиг *вправо* на 3 разряда LSR Rm, #3.
3) Нет. В новом формате значимыми остаются лишь 10 или 12 разрядов, полученные с АЦП. Точность выборок данных не превышает точности АЦП.

16.9.3.2 Разнополярные аналоговые сигналы

Как мы уже отмечали, разнополярные аналоговые сигналы перед преобразованием в АЦП приводятся к нереверсивному входу. Поэтому, полученный с АЦП код будет по-прежнему целым числом без знака.

Так, для 10-разрядного АЦП он будет иметь формат U10.0 (0÷1023). При считывании в ЦПУ код автоматически расширяется нулями в область старших разрядов. Чтобы преобразовать это значение в число со знаком в дополнительном коде нужно вычесть из него число, равное половине диапазона АЦП: 2^9=512, как показано ниже на рис. 16.19.

Получим формат целого числа со знаком S10.0. Этот формат можно рассматривать и как формат дроби со знаком S1.9, представляющей относительную входную переменную, в которой за базовое значение принято x_{max}.

С учетом коэффициента превышения номинального значения переменной k_{max}=2 истинное относительное значение будет представлено в формате S2.8 (младшее полуслово – в формате S8.8, все 32-разрядное слово – в формате S24.8).
Для преобразования в стандартный формат S4.12 достаточно выполнить логический сдвиг числа влево на 4 разряда LSL Rm, #4.

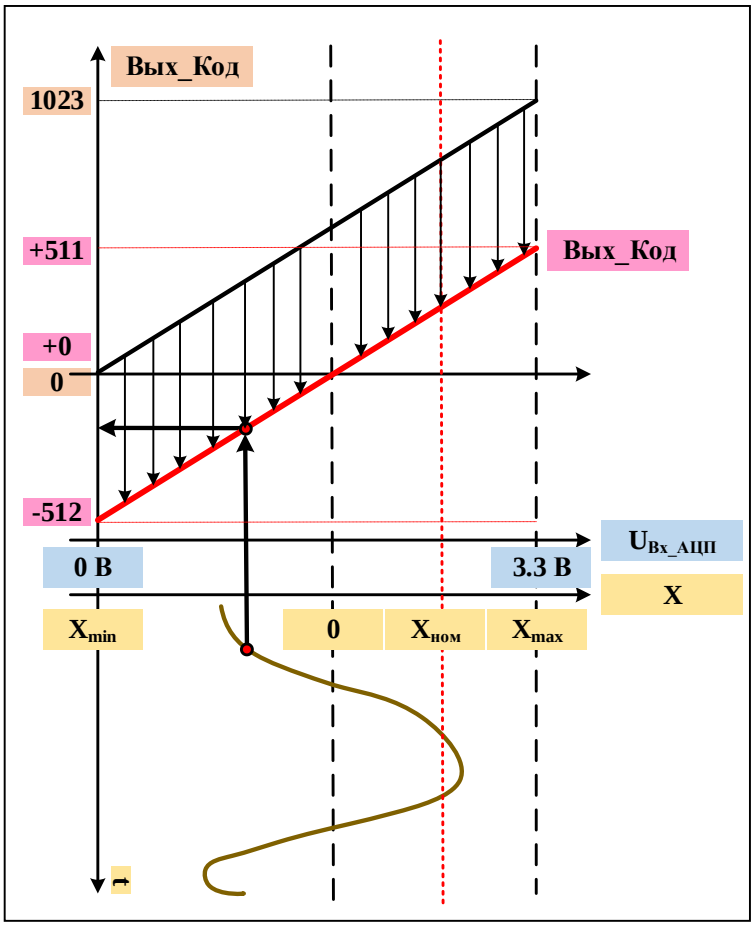


Рис. 16.19 Преобразование разнополярного сигнала

Для преобразования кода аналоговой переменной, считанного с АЦП в нужный формат числа с фиксированной точкой, достаточно обычных команд загрузки данных из регистров периферийных устройств (отображенных на память), а также обычных арифметических и логических команд процессора.

Список рекомендуемой литературы

1) Встраиваемые высокопроизводительные цифровые системы управления. Практический курс разработки и отладки программного обеспечения сигнальных микроконтроллеров TMS320x28xxx в интегрированной среде Code Composer Studio: учеб. пособие / А.С. Анучин, Д.И. Алямкин, А.В. Дроздов и др.; под общ. ред. В.Ф. Козаченко, – М.: Издательский дом МЭИ, 2010. – 270 с.
2) Айфичер Э., Джервис Б. Цифровая обработка сигналов: практический подход, 2-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. – 992с. ил.
3) Анучин А.С. Системы управления электроприводов: учебник для вузов. – М.: Издательский дом МЭИ, 2015. – 373.с: ил.

17 КОМАНДЫ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

Оглавление

В предыдущей главе показано, что основными специализированными командами, предназначенными для эффективной реализации цифровых регуляторов/фильтров являются *команды умножения с накоплением*. Конечно, эти операции можно выполнить и с использованием обычных арифметических команд (умножения, сложения), рассмотренных нами в главе 11. Однако, наличие специальных команд ЦОС в системе команд процессоров Cortex-M4 позволяет решать задачи цифрового управления проще, эффективнее и переводит эти процессоры по их возможностям в разряд сигнальных процессоров.

17.1 Признаки классификации команд умножения/умножения с накоплением

Система команд Cortex-M4 содержит большое число команд умножения/умножения с накоплением. Выбор нужной команды по алфавитному списку весьма затруднен. Мы будем классифицировать эти команды, прежде всего, *по формату операндов*, участвующих в операциях умножения (множителей), а также – по *формату генерируемого командой произведения*. Это позволит учитывать специфику конкретного приложения, для которого разрабатывается программа, в том числе формат исходных данных, подлежащих обработке, а также требуемую точность и формат результата.

Возможны 8-, 16-, 32-разрядные форматы входных данных и 32-разрядные или 64-разрядные форматы выходных данных (произведения или накопленного значения). Для обозначения форматов ранее были введены соответствующие обозначения Ui.q и Si.q (см. 16.9), в которых префикс «U» специфицирует числа без знака, а префикс «S» – числа со знаком. В главе 16 показано, что одни и те же команды умножения/умножения с накоплением можно использовать для работы с *целыми*, *дробными* и *вещественными* числами, главное – знать общую разрядность числа. Поэтому будем различать операнды, прежде всего, по общей разрядности числа:

Таблица 17.1 Типы данных в операциях умножения/умножения с накоплением

Обозначение	Тип данных
U8, S8	**8-разрядные байты без знака и со знаком**
U16, S16	**16-разрядное полуслово без знака и со знаком**
U32, S32	**32-разрядное слово без знака и со знаком**
U64, S64	**64-разрядное двойное слово без знака и со знаком**

Все операции умножения/умножения с накоплением выполняются *исключительно* над 32-разрядными или 16-разрядными операндами: (32)*(32); (16)*(16). Имеется также группа команд умножения слова на полуслово (32)*(16).

Если требуется операция умножения байт, то предварительно необходимо расширить байт до полуслова (или слова) и только затем выполнить умножение. Расширение для знаковых чисел выполняется старшим знаковым разрядом, а для беззнаковых – нулем. Расширение до слова автоматически выполняется при загрузке байта/полуслова из памяти или порта ввода с помощью команд загрузки регистров LDRB, LDRSB с суффиксами B (байта без знака) и SB (байта со знаком) и команд LDRH, LDRSH с суффиксами H (полуслова без знака) и SH (полуслова со знаком) – *см. 10.7.1*. После загрузки выполняется преобразование данных к нужному формату (глава 16).

Введем дополнительные обозначения для расширенных байт или полуслов, которые содержат только ноль или знак во всех разрядах:

Таблица 17.2 Условные обозначения расширенных байт и полуслов

Обозначение	Тип данных
EZ8	**Расширенный нулями байт**
ES8	**Байт, расширенный знаковым разрядом**
EZ16	**Расширенное нулями полуслово**
ES16	**Полуслово, расширенное знаковым разрядом**

17.2 Особенности синтаксиса команд умножения/умножения с накоплением

Приведем краткие общие сведения о принципах формирования мнемокодов команд умножения/умножения с накоплением в процессорах ARM.

В поле операндов команд умножения/умножения с накоплением, как обычно, вначале указывается имя регистра-приемника результата операции **Rd** (произведения), затем имена двух регистров, содержащих операнды-множители: регистра первого-множителя **Rn**, и, через запятую, – регистра второго множителя **Rm**:

Op {Rd}, Rn, Rm

При этом все регистры являются 32-разрядными регистрами ЦПУ. Множители могут быть как полноценными 32-разрядными словами в регистрах, так и полусловами, расположенными в соответствующих словах Rn и Rm. В последнем случае в мнемокоде команды будут присутствовать дополнительные суффиксы, указывающие на то, какие именно полуслова из регистров-источников (младшее или старшее) следует извлекать при выполнении операции умножения.

Для обозначения *операции умножения* в мнемокоде команды используются три корня: **MUL**, **ML** или **MU** (от Multiplication). Второе обозначение применяется в большинстве операций с дополнительными суффиксами-спецификаторами множителей для сокращения общего числа символов в мнемокоде команды, а также в операциях умножения с накоплением. Третье обозначение применяется только в командах двойного умножения полуслов со сложением или вычитанием полученных произведений.

Как и во всех других командах, префикс «**S**» специфицирует знаковую операцию, а префикс «**U**» – беззнаковую. Исключение – самые первые, появившиеся в системе команд ARM, операции умножения 32-разрядных слов **MUL** и умножения 32-разрядных слов с накоплением **MLA** и **MLS**, в символике которых первый символ «S» опущен, хотя команды и выполняют умножение слов со знаком в дополнительном коде.

Синтаксис *команд умножения с накоплением* отличается тем, что в строке операндов появляется имя четвертого регистра – 32-разрядного аккумулятора-накопителя **Ra** (при накоплении слов), содержащего ранее накопленное 32-разрядное произведение, или сразу двух регистров 64-разрядного аккумулятора-накопителя **RdLo**, **RdHi** (при накоплении двойных слов – так называемом «длинном накоплении»):

Op Rd, Rn, Rm, Ra

Op RdLo, RdHi, Rn, Rm

Заметьте, что *регистр-аккумулятор* – это уже четвертый регистр, который может быть указан в строке операндов команды, выполняющей 32-разрядное умножение. При этом результат текущей операции умножения с накоплением всегда поступает в регистр приемник данных Rd. Если операция умножения с накоплением выполняется многократно (а это в большинстве случаев именно так), то регистр-приемник и регистр-аккумулятор – должны быть одним и тем же регистром:

Op Ra, Rn, Rm, Ra

В этом случае его имя обязательно должно быть указано и в качестве имени регистра-приемника Rd=Ra и в качестве имени регистра аккумулятора-накопителя Ra.

При накоплении 64-битных произведений первым в строке операндов указывается имя младшего регистра аккумулятора RdLo, а затем – старшего RdHi. При многократных вызовах 64-битный аккумулятор накапливает все произведения чисел, расположенных в регистрах множителей Rn и Rm, то есть по умолчанию является приемником операции умножения с накоплением, в отличие от 32-разрядных операций, где имя приемника должны быть указано отдельно (но тоже в первой позиции).

К корневому обозначению операции умножения с накоплением добавляется:

- суффикс «**A**», означающий (Accumulate – положительное аккумулирование);
- суффикс «**S**», означающий (Subtrucrt – отрицательное аккумулирование);
- суффикс «**L**» (Long – длинное) – «длинное» аккумулирование в 64-битовом слове.

Заметьте, что если в обычных операциях умножения имя регистра-приемника – опция (может быть опущено), то в операциях умножения с накоплением – оно является обязательным атрибутом команды и присутствует всегда.

17.3 Путеводитель по командам умножения/умножения с накоплением

В этом параграфе дадим краткий обзор имеющихся в системе команд операций умножения /умножения с накоплением, который позволит читателю быстро выбрать нужную команду в соответствии с реальной задачей и форматами входных и выходных переменных. Обратим особое внимание на *преимущественные области применения* команд каждой группы. Более подробные сведения, включая конкретные примеры использования команд и имеющиеся ограничения, Вы найдете в электронном справочнике по системе команд (приложение 1). Далее в этой главе наиболее употребительные команды рассматриваются подробно с примерами использования в конкретных проектах.

При умножении слов в общем случае получается 64-разрядное произведение. В 32-разрядном процессоре имеются три возможности: вычислить только младшее слово произведения, только старшее слово произведения или все произведение целиком:

1. (32*32) → Low (64)
2. (32*32) → High (64)
3. (32*32) → (64)

Именно по этим признакам мы классифицировали все команды умножения/умножения с накоплением и дали начальные сведения об области их преимущественного применения, имеющимся ограничениям. Особое внимание следует обратить на то, что только некоторые команды умножения с накоплением могут формировать флаг «насыщения» S, который в подобных операциях можно рассматривать как флаг переполнения – выхода результата накопления за пределы допустимого формата выходных данных.

Во всех таблицах красными прямоугольниками в левой части таблицы выделены команды умножения, а оранжевым – умножения с накоплением. В мнемонике команд использовано *цветовое деление*, которое поможет Вам быстро освоить принцип кодировки операций, разработанный инженерами фирмы ARM. Он такой же, как и во всей книге. Краткое описание действия, выполняемого командой, и информация о выработке флагов результатов операции позволит Вам быстро сделать предварительный выбор типа нужной команды из приведенного ниже справочника.

17.3.1 Умножение/деление/накопление 32-разрядных слов

Таблица 17.3 Команды обработки слов с получением младшего слова результата

Команды обработки 32-разрядных слов с получением 32-разрядного результата (при умножении – только младшего слова произведения)				
	Мнемоника	**Операнды**	**Краткое описание**	**Флаги**
	MUL	**{Rd,}, Rn, Rm**	**(S32) ← Low [(S32)*(S32)] Умножение слов со знаком в дополнительном коде с получением младшего слова произведения**	-
	UDIV	**{Rd,}, Rn, Rm**	**(U32) ← (U32)/(U32) Деление слов без знака с получением частного – слова**	-
	SDIV	**{Rd,}, Rn, Rm**	**(S32) ← (S32)/(S32) Деление слов со знаком с получением частного – слова**	-
	MLA	**Rd, Rn, Rm, Ra**	**(S32) ← (S32) + Low [(S32)*(S32)] Умножение слов, прибавление младшего слова произведения к содержимому аккумулятора и сохранение**	-
	MLS	**Rd, Rn, Rm, Ra**	**(S32) ← (S32) - Low [(S32)*(S32)] Умножение слов, вычитание младшего слова произведения из содержимого аккумулятора и сохранение**	-

Табл. 17.3 для общности включает и базовые арифметические операции MUL, DIV, уже рассмотренные нами ранее (глава 11) и поддерживаемые во всех ядрах Cortex-M3/M4/M4F. Две последние операции относятся к командам ЦОС – умножения/накопления 32-разрядных операндов с получением 32-разрядного результата.

Эта группа команд вычисляет *только младшее слово произведения*. Оно же и накапливается в операциях положительного (MLA) и отрицательного накопления (MLS). Это означает, что при выходе произведения за пределы разрядной сетки процессора (32 разряда) старшие разряды *автоматически «усекаются» (отбрасываются)*. Обратите особое внимание на то, что никаких флагов «переполнения» при выполнении как операций умножения, так и накопления – *не вырабатывается*.

Команды этой группы можно использовать тогда, когда множители представляют собой числа, результат умножения которых никогда не выходит за пределы слова, например, целые со знаком (S16.0), расширенные до формата слова (S32.0) = (ES16, S16.0). Программист должен быть уверен, что переполнения исключены не только при операциях умножения, но и при операциях сложения (при накоплении). Это можно обеспечить только правильным выбором формата исходных данных и порядка цифрового регулятора/фильтра. Так как эта группа команд умножения с накоплением не формирует никаких флагов (в том числе флага «насыщения» при операции накопления), пользоваться этими командами нужно осторожно.

17.3.2 Умножение/умножение с накоплением 32-разрядных слов с получением только старшего слова произведения

К корневому имени MUL или ML для этой группы команд добавляется префикс «**M**», означающий получение *только старшего значащего слова произведения* (Most significant word). Команды этой группы обрабатывают только слова со знаком префикс «S» и так же, как и в предыдущем случае, не вырабатывают флагов – табл. 17.4.

Таблица 17.4 Команды обработки слов с получением старшего слова результата

Команды умножения слов с получением только старшего слова произведения				
	Мнемоника	**Операнды**	**Краткое описание**	**Флаги**
	SMMUL **SMMULR**	**{Rd,}, Rn, Rm**	**(S32) ← High[(S32)*(S32)]** **Умножение слов со знаком с получением старшего слова произведения без округления или с округлением (суффикс R)**	-
	SMMLA **SMMLAR**	**Rd, Rn, Rm, Ra**	**(S32) ← (S32) + High[(S32)*(S32)]** **Умножение слов со знаком с получением старшего слова произведения без округления или с округлением, сложение с содержимым аккумулятора – слова, сохранение**	-
	SMMLS **SMMLSR**	**Rd, Rn, Rm, Ra**	**(S32) ← (S32) - High[(S32)*(S32)]** **Умножение слов со знаком с получением старшего слова произведения без округления или с округлением, вычитание из содержимого аккумулятора – слова, сохранение**	-

Команды этой группы с суффиксом «**A**» выполняют положительное аккумулирование произведения (сложение с содержимым регистра-аккумулятора), а с суффиксом «**S**» – отрицательное аккумулирование (вычитание). Это единственная группа команд, в которой младшее слово результата может быть либо просто отброшено, либо отброшено после округления до младшего бита старшего слова произведения. В последнем случае к мнемокоду команды добавляется суффикс «**R**» (Round – округлить).

Эта группа команд обычно используется для умножения переменных и коэффициентов, представленных дробями или вещественными переменными с малым числом разрядов целой части, когда потеря малозначимого дробного остатка произведения несущественна, например:

(S1.31)*(S1.31) = (S2.62) = (S2.30) + малозначимый 32-разрядный остаток
(S4.28)*(S4.28) = (S8.56) = (S8.24) + малозначимый 32-разрядный остаток
(S8.24)*(S8.24) = (S16.48) = (S16.16) + малозначимый 32-разрядный остаток

Эта группа команд предназначена для разработчиков систем управления, работающих с переменными в относительных единицах, причем с незначительной кратностью возможного превышения номинальных значений. Она обеспечивает максимальную точность цифрового регулирования/фильтрации в рамках 32-разрядной сетки процессора.

17.3.3 Умножение/умножение с накоплением с получением полного произведения

Команды этой группы генерируют полное 64-разрядное произведение, которое может накапливаться в аккумуляторе – двойном слове. При правильном выборе форматов исходных данных переполнения при накоплении могут быть полностью исключены. Поэтому эти команды не формируют флаг переполнения Q (он попросту не нужен) – табл. 17.5.

Таблица 17.5 Команды обработки слов с получением результата – двойного слова

Команды умножения слов с получением полного 64-разрядного произведения				
	Мнемоника	**Операнды**	**Краткое описание**	**Флаги**
	UMULL	**RdLo, RdHi, Rn, Rm**	**(U64) ← (U32)*(U32)** **Умножение слов без знака с получением произведения – двойного слова**	-
	SMULL	**RdLo, RdHi, Rn, Rm**	**(S64) ← (S32)*(S32)** **Умножение слов со знаком с получением произведения – двойного слова**	-
	UMLAL	**RdLo, RdHi, Rn, Rm**	**(U64) ← (U64) + [(U32)*(U32)]** **Умножение слов без знака с получением произведения – двойного слова и накоплением в аккумуляторе**	-
	SMLAL	**RdLo, RdHi, Rn, Rm**	**(S64) ← (S64) + [(S32)*(S32)]** **Умножение слов со знаком с получением произведения – двойного слова и накоплением в аккумуляторе**	-
	UMAAL	**RdLo, RdHi, Rn, Rm**	**(U64) ← [(U32)*(U32)] + (U32_RdLo) + (U32_RdHi)** **Умножение слов без знака с получением произведения – двойного слова, добавлением к нему двух слов аккумулятора (младшего и старшего) и сохранением результата в аккумуляторе**	-

Команды «длинного» умножения подробно рассмотрены нами ранее в главе 11. Команды умножения с накоплением при ограниченном порядке цифрового регулятора/фильтра полностью исключают возникновение переполнения при накоплении.

Операции «длинного» умножения и умножения с накоплением являются настоящим достоянием ARM-процессоров. Они выполняются всего за такт и исключают возможные переполнения при работе с цифровыми регуляторами/фильтрами даже очень высокого порядка. При необходимости программист может выполнить ограничение старшего слова результата с помощью специальных команд насыщения SSAT, рассматриваемых далее в этой главе.

17.3.4 Умножение слова на полуслово с получением старшего слова произведения

Результатом такого умножения является 48-битное произведение, старшее 32-разрядное слово которого сохраняется, а младшее 16-разрядное полуслово – просто отбрасывается – табл. 17.6.

Таблица 17.6 Команды работы со словами и полусловами и получением старшего слова произведения

Команды умножения слова на полуслово с получением старшего слова произведения				
	Мнемоника	**Операнды**	**Краткое описание**	**Флаги**
	SMULWB **SMULWT**	**{Rd,}, Rn, Rm**	**(S32) ← High [(S32)*(S16)]** **Умножение слова на полуслово с сохранением старшего слова произведения**	-
	SMLAWB **SMLAWT**	**Rd, Rn, Rm, Ra**	**(S32) ← (S32) + High [(S32)*(S16)]** **Умножение слова на полуслово, сложение старшего слова произведения с содержимым аккумулятора, сохранение**	**Q**

Первый регистр-источник **Rn** должен содержать слово (**W**), участвующее в операции, а второй регистр-источник **Rm** – полуслово. Для его спецификации используется суффикс «**B**» или «**T**». К корневому обозначению команды при этом добавляются оба суффикса. Префикс «**S**» свидетельствует о том, что рассматриваемые операции – знаковые.

Ниже показано, как суффиксы в коде операции специфицируют операнды-множители.

Таблица 17.7 Суффиксы спецификации операндов в мнемокоде команд группы

1-й регистр-источник Rn	**2-й регистр источник Rm**		**Суффиксы операции умножения слова на полуслово**
Слово	**Ст. п.-слово**	**Мл. п.-слово**	
W		**Bottom (B)**	**WB**
	Top (T)		**WT**

Использование команд этой группы возможно для повышения точности 16-разрядных операций, когда-либо коэффициенты, либо выборки входных данных регулятора/фильтра представляются в 32-разрядном формате. При этом формат второго операнда остается 16-разрядным. Это компромисс, на который идут, если точности 16-разрядных вычислений недостаточно. Примеры форматов входных операндов:

(S1.31)*(S1.15) = (S2.46) = S2.30 + малозначимый 16-разрядный остаток;
(S4.28)*(S4.12) = (S8.40) = S8.24 + малозначимый 16-разрядный остаток;
(S8.24)*(S8.8) = (S16.32) = S16.16 + малозначимый 16-разрядный остаток;

Команды умножения с накоплением работают с 32-разрядным аккумулятором и в отличие от многих других команд при переполнении *формируют флаг насыщения* Q. Это позволяет в реальном времени при реализации цифровых регуляторов и фильтров проверить, было ли в процессе аккумулирования данных переполнение или нет. Если было – результат некорректный. Нужно либо установить предельно возможное значение выхода, либо, если это произошло в процессе отладки, – изменить форматы выходных переменных и вновь протестировать программу на наличие переполнения.

Умножение целых чисел с использованием команд этой группы недопустимо, так как может привести к полной потере значимости, например:

(S32.0)*(S16.0) = (S48.0). При отбрасывании младших 16-и разрядов значимость будет полностью потеряна.

Используйте команды этой группы, если разрядности полуслов недостаточно для получения требуемой точности. Обычно повышается разрядность коэффициентов цифрового регулятора/фильтра, а разрядность выборок остается 16-разрядной.

17.3.5 Умножение/умножение с накоплением полуслов

Это самая значительная группа команд, так как в микроконтроллерных применениях формат 16-разрядных данных используется очень часто – табл. 17.8.

Каждый из двух 32-разрядных регистров-источников Rn (первый) и Rm (второй), указанных в поле операндов команды умножения/умножения с накоплением, может содержать по два полуслова. Существуют команды *одиночного* умножения полуслов, когда в операции участвуют только два полуслова из возможных 4-х, расположенных в

регистрах-источниках Rn, Rm (остальные полуслова не используются), и операции *двойного* умножения, когда одной командой выполняются сразу две операции умножения полуслов, расположенных в регистрах-источниках Rn, Rm.

Для операций умножения с накоплением в операции накопления может использоваться либо одно 32-разрядное произведение (при одиночном умножении), либо сразу два 32- разрядных произведения (при двойном умножении). Причем, все эти нужные действия будут выполняться за один такт процессора, что повышает производительность операций при цифровой обработке сигналов.

Таблица 17.8 Команды обработки полуслов

Команды умножения полуслов с получением 32-разрядного произведения – слова				
	Мнемоника	**Операнды**	**Краткое описание**	**Фла-ги**
	SMULBB BT TB TT	{Rd,}, Rn, Rm	(S32) ← (S16)*(S16) Умножение двух полуслов со знаком с получением произведения – слова	-
	SMLABB BT TB TT	Rd, Rn, Rm, Ra	(S32) ← (S32) + [(S16)*(S16)] Умножение двух полуслов со знаком, сложение произведения с содержимым аккумулятора-слова, сохранение	Q
	SMLALBB BT TB TT	RdLo, RdHi, Rn, Rm	(S64) ← (S64) + [(S16)*(S16)] Умножение двух полуслов со знаком, сложение с содержимым аккумулятора-двойного слова и сохранение в нем же	-
	SMUAD SMUADX	{Rd,}, Rn, Rm	(S32) ← (S16)*(S16) + (S16)*(S16) Двойное умножение полуслов со знаком, прямое или перекрестное, суммирование произведений и сохранение суммы	Q
	SMUSD SMUSDX	{Rd,}, Rn, Rm	(S32) ← (S16)*(S16) - (S16)*(S16) Двойное умножение полуслов со знаком, прямое или перекрестное, вычитание произведений и сохранение разности	-
	SMLAD SMLADX	Rd, Rn, Rm, Ra	(S32) ← (S32) + [(S16)*(S16)+(S16)*(S16)] Двойное умножение полуслов со знаком, прямое или перекрестное, сложение произведений, добавление к содержимому аккумулятора и сохранение	Q
	SMLSD SMLSDX	Rd, Rn, Rm, Ra	(S32) ← (S32) + [(S16)*(S16) - (S16)*(S16)] Двойное умножение полуслов со знаком, прямое или перекрестное, вычитание произведений, добавление разности к содержимому аккумулятора, сохранение	Q
	SMLALD SMLALDX	RdLo, RdHi, Rn, Rm	(S64) ← (S64) + [(S16)*(S16) + (S16)*(S16)] Двойное умножение полуслов со знаком, прямое или перекрестное, сложение произведений и накопление суммы в аккумуляторе – двойном слове	-
	SMLSLD SMLSLDX	RdLo, RdHi, Rn, Rm	(S64) ← (S64) + [(S16)*(S16) - (S16)*(S16)] Двойное умножение полуслов со знаком, прямое или перекрестное, вычитание произведений и накопление разности в аккумуляторе – двойном слове	-

В операциях умножения полуслов с помощью дополнительных суффиксов специфицируется, какие именно полуслова являются множителями. Если в команде используются только два 16-разрядных полуслова-множителя (16*16), а два оставшихся полуслова – нет, то возможны следующие суффиксы-спецификаторы – табл. 17.9:

Таблица 17.9 Суффиксы-спецификаторы места нахождения операндов-полуслов в регистрах источниках Rn, Rm

1-й регистр-источник Rn		2-й регистр источник Rm		Операция умножения	Суффикс
Ст. п.-слово	Мл. п.-слово.	Ст. п.-слово	Мл. п.-слово		
	Bottom (B)		Bottom (B)	Мл. п.-слово*Мл. п.-слово	BB
	Bottom (B)	Top (T)		Мл. п.-слово*Ст. п.-слово	BT
Top (T)			Bottom (B)	Ст. п.-слово*Мл. п.-слово	TB
Top (T)		Top (T)		Ст. п.-слово*Ст. п.-слово	TT

Если из каждого регистра источника Rn и Rm в операции умножения/умножения с накоплением используются оба полуслова-множителя (в командах двойного умножения полуслов – Dual), то в мнемонике команды используется дополнительный суффикс **D.** При этом применяется прямой порядок выборки полуслов из регистров-источников Rn и Rm. Если программист желает изменить порядок выборки полуслов *из второго регистра-источника* Rm, то он должен к суффиксу **D** в мнемокоде команды добавить суффикс **X** (от E**x**change) – табл. Таблица 17.10.

Таблица 17.10 Суффиксы-спецификаторы команд двойного умножения полуслов

1-й регистр-источник Rn		2-й регистр источник Rm		Операции умножения	Суффикс
Ст. п.-слово	Мл. п.-слово.	Ст. п.-слово	Мл. п.-слово		
	Bottom (B)		Bottom (B)	1-е произведение	D
Top (T)		Top (T)		2-е произведение	
	Bottom (B)	Top (T)		1-е произведение	DX
Top (T)			Bottom (B)	2-е произведение	

В любом случае сначала извлекается младшее слово из первого регистра–источника и умножается на одно из слов второго регистра-источника (при прямом порядке – на младшее слово, при обратном – на старшее).

Результат умножения любых двух полуслов – 32-разрядное слово, для сохранения которого достаточно одного регистра общего назначения процессора – регистра-приемника Rd, имя которого указывается первым в поле операндов команды умножения/умножения с накоплением. Для операций накопления в качестве 4-го регистра в поле операндов команды указывается имя регистра-аккумулятора Ra.

При умножении полуслов возможно как обычное 32-разрядное накопление, так и «длинное» 64-разрядное, которое учитывает возможные переполнения при сложении 32-разрядных произведений. В этом случае в мнемокоде команды присутствует суффикс «**L**». Группа команд с 32-разрядным накоплением формирует флаг переполнения **Q**, позволяющий проконтролировать выход результата накопления за пределы разрядной сетки процессора. В группе команд 64-разрядного накопления необходимость в выработке флага Q отпадает.

При вводе данных из портов ввода или регистров АЦП разрядность данных, как правило, не превышает 16 бит. При этом входные переменные представляются в относительных единицах. Наиболее употребительные форматы: S1.15; S4.12.

Если для коэффициентов регулятора/фильтра выбрать такие же форматы, то произведение никогда не превысит формата слова:

(S1.15)*(S1.15) = (S2.30);
(S4.12)*(S4.12) = (S8.24).

Все команды обычного накопления при умножении полуслов, в которых используется 32-разрядный регистр–аккумулятор, вырабатывают флаг насыщения Q, который позволяет идентифицировать переполнение. Команды умножения с «длинным» накоплением этот флаг не вырабатывают, так как на практике число возможных накоплений ограничивается порядком цифрового регулятора/фильтра, который вряд ли превысит 100. При этом отсутствие переполнений гарантировано самим форматом исходных 16-разрядных множителей.

17.4 Контроль переполнений в операциях умножения с накоплением

Большинство команд умножения с накоплением не устанавливают флаг переполнения разрядной сетки Q – это команды, использующие 32-разрядные операнды с получением 32-разрядного или 64-разрядного результата. При этом вся ответственность за отсутствие переполнения лежит на программисте – он должен так выбрать форматы входных данных и коэффициентов, а также порядок цифрового регулятора/фильтра, чтобы полностью исключить переполнения.

При «длинных» операциях умножения с накоплением переполнения исключаются даже при очень большом порядке фильтра. При этом имеется возможность ограничения полученного результата накопления либо с помощью обычных команд процессора (сравнения, условного выполнения), либо с помощью специальных *команд насыщения* (см. далее в этой главе).

Для команд умножения с накоплением, которые работают с 16-разрядными операндами (S16)*(S16), а также с операндами (S32)*(S16), сумма накапливается в 32-разрядном слове, и флаг насыщения Q в регистре статуса приложения пользователя APSR *может установиться* при выходе результата из допустимого диапазона 32-разрядных чисел со знаком. Этот флаг является «*флагом-липучкой*»: один раз возникнув, он не исчезает, даже если при выполнении следующих операций накопления переполнение не фиксируется. Таким образом, флаг Q свидетельствует о том, что полученному в результате обработки данных результату верить нельзя – он неправильный.

Следовательно, прежде чем выдать рассчитанное выходное воздействие, нужно проверить его на «достоверность» – отсутствие флага насыщения Q. Если флага Q нет, расчет достоверный, можно выдавать управляющее воздействие, в противном случае – нужно останавливать программу и скорректировать алгоритм и/или форматы входных переменных.

Сложность заключается в том, что команд условной передачи управления или условного выполнения по содержимому флага Q – нет. Напомним, что этот флаг располагается в 27-ом бите регистра статуса приложения пользователя APSR:

Регистр статуса программы приложения APSR																													
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
N	**Z**	**C**	**V**	**Q**																											

17.4.1 Команды доступа к регистрам специального назначения

Единственная возможность проконтролировать флаг S состоит в считывании содержимого регистра APSR в один из регистров ЦПУ с последующим тестированием 27-го бита этого регистра. Для работы с регистрами специального назначения, в том числе с регистром APSR, предусмотрены две команды:

```
MRS{cond} Rd, spec_reg
MSR{cond} spec_reg, Rn
```

Первая выполняет считывание содержимого регистра специального назначения spec_reg в регистр ЦПУ Rd, а вторая – обратное действие: сохраняет текущее содержимое регистра ЦПУ Rn в регистре специального назначения. Корень оптокода – M (от Move – переслать данные). Суффикс «R» символизирует регистр общего назначения, а суффикс «S» – регистр специального назначения.

Регистрами специального назначения являются регистры статуса APSR, IPSR, EPSR, управления процессором CONTROL, встроенного контроллера прерываний и др. Особенности доступа:

- Доступ по чтению к регистрам специального назначения возможен всегда;
- Доступ по записи зависит от текущего режима работы процессора:
    - В непривилегированном режиме Unprivileged (пользовательском режиме) запись возможна только в регистр статуса приложения пользователя APSR. Запись в остальные регистры блокируется.
    - В привилегированном режиме Privileged доступ возможен ко всем без исключения регистрам специального назначения.

17.4.2 Как проконтролировать флаг насыщения Q

Нужно считать содержимое регистра статуса программы приложения пользователя APSR в один из регистров ЦПУ и выполнить тестирование 27-го бита этого регистра. В зависимости от результата тестирования: либо выдать рассчитанное управляющее воздействие, либо остановить выполнение программы:

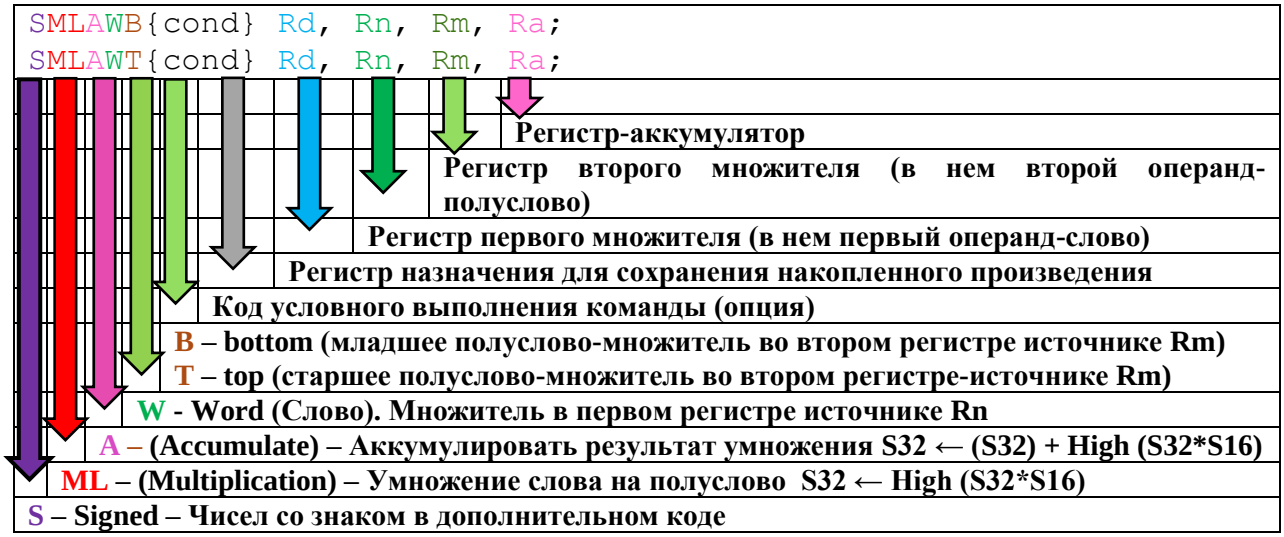
```
; Считать состояние регистра APSR
     MRS r5, APSR
; Протестировать состояние 27-го бита
     TST r5, #(01 LSL 27)
; Q=0. Продолжить, выдать рассчитанное управляющее воздействие
     BEQ Output
; Q=1. Возникло переполнение. Остановить выполнение программы
STOP …
; Выдать управляющее воздействие
Output
```

В том случае, когда программист уверен, что после возникновения переполнения можно выдать корректное управляющее воздействие, он, прежде всего, должен сбросить «флаг-липучку» Q. Для этого нужно модифицировать считанное из регистра APSR состояние процессора, установив в 27-м бите «0», и сохранить новое значение в регистре APSR. Такая операция является операцией «Чтение-Модификация-Запись». Она предусматривает сохранение всех исходных битов регистра APSR неизменными, кроме 27-го бита, и фактически выполняет сброс флага Q:

```
; Сбросить 27-й бит в регистре APSR (флаг Q)
     BIC r5, #(01 LSL 27)
     MSR APSR, r5
; Продолжить программу с выдачей управляющего воздействия
Output
;     …
```

17.4.3 Контроль флага насыщения Q на примере операций умножения с накоплением слова на полуслово и полуслова на полуслово

17.4.3.1 Синтаксис операций умножения с накоплением слова на полуслово



Графическая иллюстрация алгоритма выполнения команды SMLAWB дана на рис. Рис. 17.1

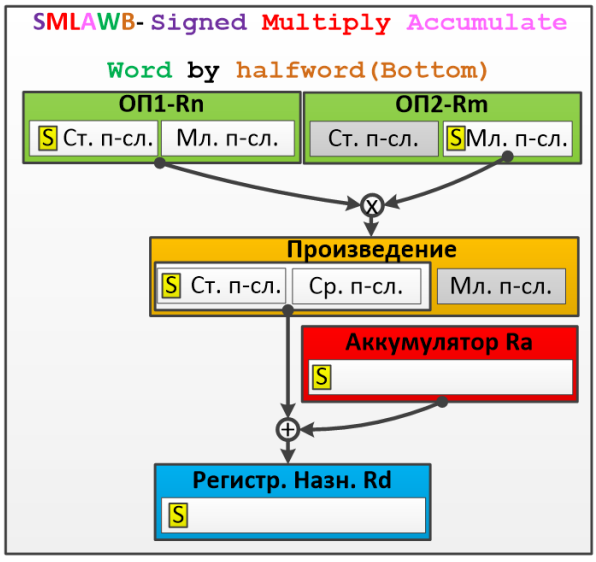


Рис. 17.1 Алгоритм выполнения команды SMLAWB

Первый *операнд-слово* извлекается из регистра первого множителя Rn, а второй – *младшее полуслово* – из регистра второго множителя Rm. Старшее слово 48-битного произведения складывается с содержимым регистра аккумулятора Ra и сохраняется в регистре назначения Rd.

При операциях множественного умножения с накоплением Вы должны использовать в качестве регистра назначения Rd и в качестве регистра-аккумулятора Ra один и тоже же регистр ЦПУ:

Ra = Ra + High [(Rn)* (полуслово из Rm)].

Как уже упоминалось, вместо суффикса B можно использовать суффикс T. В этом случае вместо младшего будет извлекаться старшее полуслово из регистра-источника данных Rm.

17.4.3.2 Программная реализация ядра цифрового регулятора/фильтра

Предположим, что выборки данных являются полусловами со знаком в формате S4.12, а коэффициенты – словами со знаком в формате S4.28. В них дробная часть на 16 разрядов больше, чем в выборках, что обеспечивает более высокую точность алгоритма цифрового регулятора/фильтра. Обратимся к проекту CPU_22 (файл MyProg_22.s). В нем демонстрируется возможность возникновения флага переполнения Q при выполнении операций умножения с накоплением и технология проверки состояния этого флага.

```
57              ALIGN
58  ; Резервирование кодовой памяти под 32-х разрядные
59  ; коэффициенты цифрового регулятора/фильтра
60  ; в относительных единицах в формате (S4.28)
61  Tab_koef
62  B_0         DCD +7*268435456   ; в относ. ед.: +7
63  B_1         DCD +7*268435456   ; в относ. ед.: +7
64  B_2         DCD +7*268435456   ; в относ. ед.: +7
65
66  ; Резервирование кодовой помяти под 16-и разрядные
67  ; значения выборок в относительных единицах (S4.12)
68  ; Имитация буфера выборок в ОЗУ
69  Buf_X
70  X_k         DCW +7*4096        ; в относ. ед.: +7
71  X_k_1       DCW +7*4096        ; в относ. ед.: +7
72  X_k_2       DCW +7*4096        ; в относ. ед.: +7
73
74  ; Объявить секцию данных в ОЗУ для порта выдачи
75  ; управляющего воздействия в формате (S8.24)
76              ALIGN
77              AREA MyData, DATA, ReadWrite
78  Port_Out    SPACE 4
```

В конце программы резервируется место в кодовой памяти для трех одинаковых коэффициентов фильтра +7.0 и трех одинаковых выборок +7.0. Реально расположенный в ОЗУ буфер выборок имитируется уже заполненным данными буфером в ПЗУ для демонстрации только особенностей операции умножения с накоплением слова на полуслово на одном проходе регулятора/фильтра.

Коэффициенты фильтра должны иметь формат (S4.28). Поэтому, эквивалентный 32-разрядный код для резервирования в памяти определяется как $(+7)*2^{28}$, где 28 – число разрядов дробной части числа. Выборки должны быть в формате (S4.12), и эквивалентный код для резервирования полуслов в памяти определяется как $(+7)*2^{12}$.

В самом конце программы резервируется место в ОЗУ под порт вывода управляющего воздействия. Если возникнет переполнение, то вывод некорректных данных будет заблокирован.

Формат накопленного произведения (S4.28)*(S4.12)= (S8.24) с учетом того, что малозначимое младшее полуслово произведения отбрасывается. Следовательно, целая часть результата является 8-разрядной. Диапазон ее возможных значений: (-128 ÷ +127). Мы выбрали для примера значения коэффициентов и выборок с максимальным значением целой части (+7). Посмотрите, сколько произведений можно накопить при изначально обнуленном аккумуляторе до момента возникновения флага насыщения Q. Уже третье умножение с накоплением должно сопровождаться выработкой флага Q (результат расчета – некорректный). Проверим это!

№	Операция	Флаг насыщения Q
1	(+7)*(+7) + 0 = +49 = 0x31	0
2	(+7)*(+7) + 49 =+98 = 0x62	0
3	(+7)*(+7) + 98 =+147 = 0x93	1
4	(+7)*(+7) + 147 = +196 = 0xC4	

В начале программы выполняется инициализация двух указателей в регистрах ЦПУ: r0 – начальным адресом таблицы коэффициентов, r1 – начальным адресом буфера выборок. Задается число операций умножения с накоплением (в r2) и обнуляется регистр-аккумулятор r5.

```
12  MyProg
13  ; Основная программа
14  ; Загрузить указатели на коэффициенты и выборки
15  ; начальными адресами их расположения в памяти
16  ; r0 - указатель на коэффициенты (S4.28)
17  ; r1 - указатель на выборки (S4.12)
18          LDR r0,=Tab_koef
19          LDR r1,=Buf_X
20  ; Задать число операций умножения с накоплением
21          MOV r2,#3
22  ; Очистить регистр-аккумулятор r5
23          MOV r5,#0
```

Собственно, процедура цифровой фильтрации заключается в считывании в регистр r3 очередного коэффициента в формате (S32) с использованием косвенной адресации с пост-авто-смещением содержимого указателя на +4 (длину коэффициента в байтах), в считывании в регистр r4 очередной выборки также с пост-авто-

смещением содержимого указателя на +2 (длину выборки в байтах) и в выполнении операции умножения коэффициента на выборку с накоплением произведения в регистре-аккумуляторе r5. Эти три команды включаются в цикл с заданным числом повторений в регистре r2. После завершения процедуры цифровой фильтрации проверяется достоверность полученного результата.

```
28  Filter
29  ; Получить очередные множители из памяти
30  ; с авто-пост инкрементированим указателей
31          LDR r3,[r0],#4     ; (S32)
32          LDRH r4,[r1],#2    ; (S16)
33  ; Умножение с накоплением в регистре r5
34          SMLAWB r5,r3,r4,r5
35  ; Декрементировать счетчик числа умножений и, если
36  ; не все операции выполнены, повторить
37          SUBS r2,#1
38          BNE Filter
```

Содержимое статусного регистра APSR считывается в регистр r6 ЦПУ и выделяется 27-й бит (флаг Q) с использованием операции маскирования (Логическое И).

```
39  ; Проверить возможное переполнние (флаг Q)
40          MRS r6, APSR     ; Считать статусный регистр
41          AND r6, #(01 :SHL: 27)  ; Выделить флаг Q
42          BEQ Output       ; Нет переполнения
43  ; Есть переполнение. Очистить флаг Q.
44  ; Остановить выполнение программы
45  CLR_Q
46          BIC r6, #(01 :SHL: 27)  ; Очистить флаг Q
47          MSR APSR, r6     ; Сохранить в APSR
48  Stop
49          B Stop
50  Output
51  ; Выдать управляющее воздействие в порт вывода
52          LDR r0,=Port_Out
53          STR r5,[r0]
54  ; Завершение цифрового регулирования/фильтрации
55          B .
```

Если насыщение отсутствует, то результат достоверный – можно выдавать управляющее воздействие.

В противном случае флаг Q сбрасывается с использованием логической операции «Очистить бит» и содержимое статусного регистра восстанавливается, но уже с очищенным флагом Q. Далее программист может выдать «аварийное» управляющее воздействие (на случай переполнения) и продолжить выполнение программы, либо остановить программу с выдачей уведомления о некорректности расчета.

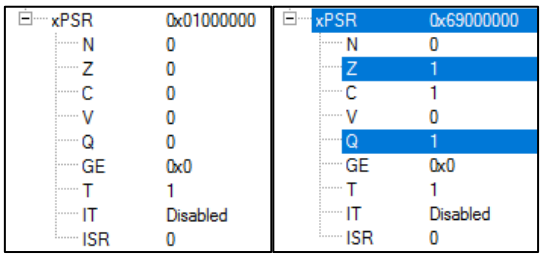
1) Выполните трансляцию файла приложения. Изучите файл листинга. Убедитесь в том, что коэффициенты фильтра и выборки сохранены в памяти в нужном формате – (S4.28) и (S4.12), соответственно.

2) Выполните сборку проекта и загрузите его на отладку в симулятор. Поставьте точку останова в строке 38 программы (после завершения одного прохода фильтра). Раскройте в окне регистров содержимое регистра статуса программы пользователя xPSR. Выполняйте программу до точки останова и следите за состоянием флага насыщения Q. Когда он возникнет?

1) Обратите внимание, что в самом старшем байте числа находится 7. Это как раз и соответствует числу +7.0 в формате (S4.28):

```
61 00000038          Tab_koef
62 00000038 70000000
                     B_0      DCD              +7*268435456 ; в относ. ед.: +7
```

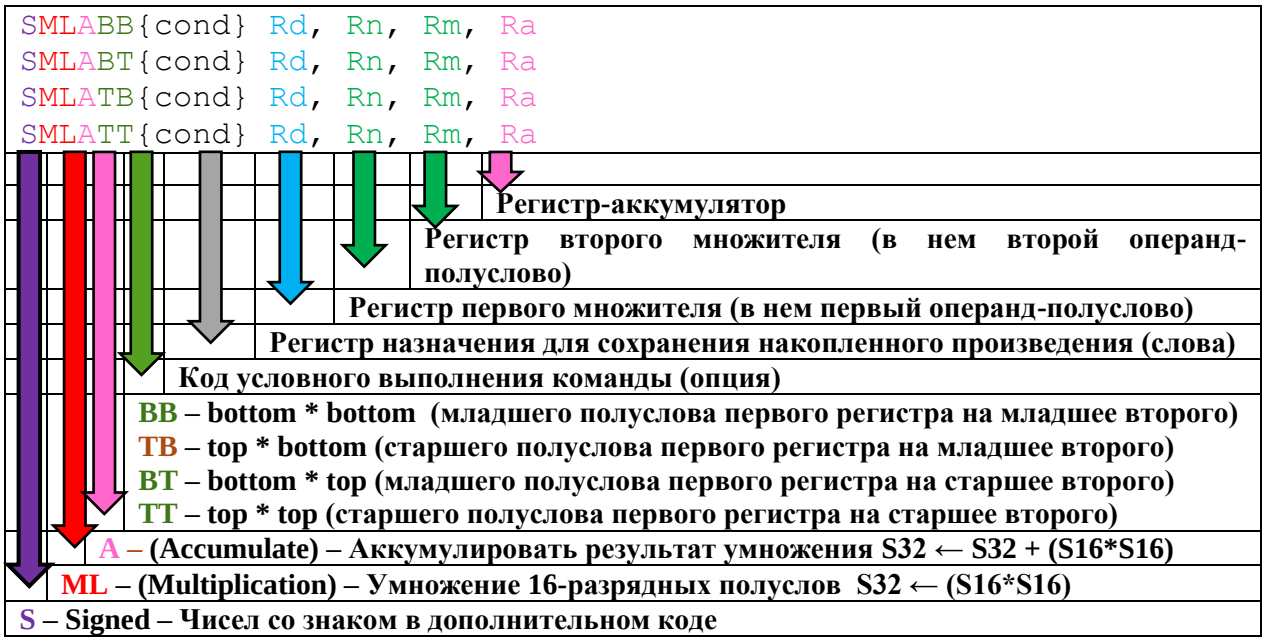
2) Флаг действительно возникает, свидетельствуя о переполнении при выполнении всех заданных проходов фильтра. Приведем для справки начальное состояние регистра статуса программы пользователя и состояние после третьего прохода фильтра. Именно после этого прохода и должно возникнуть переполнение:



Процессоры ARM имеют в регистре статуса программы пользователя специальный флаг Q, который может быть признаком переполнения разрядной сетки при выполнении операций умножения-накопления 32-разрядных произведений. Если Вы пользуетесь такими операциями, – применяйте средства контроля состояния флага насыщения S.

17.4.3.3 Синтаксис операций умножения с накоплением полуслов

Обычно коэффициенты и выборки регулятора/фильтра представляются в одном и том же формате, причем точности 16-разрядных слов часто оказывается вполне достаточно. Эти операции так же, как и рассмотренные выше, формируют 32-разрядный результат накопления с возможной установкой флага переполнения (флаг Q).



Если возник флаг Q – результат расчета некорректен. Управляющее воздействие может измениться скачкообразно, что вызовет сбои и даже автоколебания в системе цифрового управления. Выдача такого некорректного воздействия должна предотвращаться программистом.

Иллюстрация алгоритма выполнения одной из 4-х возможных команд этого типа представлена на рис. 17.2. В остальных вариантах команды меняются только активные полуслова в регистрах-источниках.

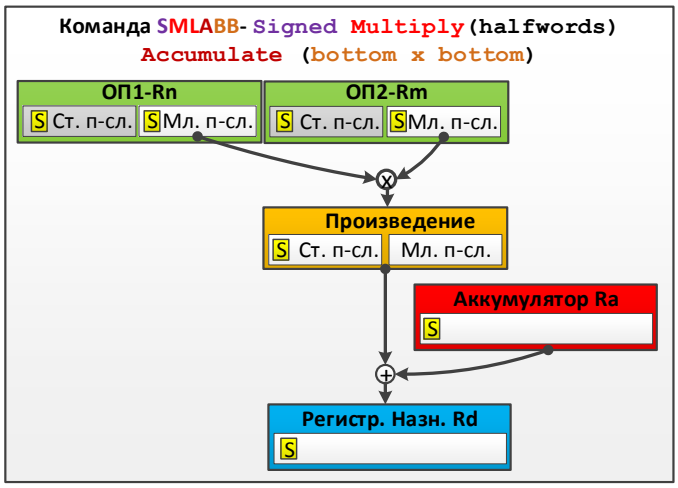


Рис. 17.2 Алгоритм выполнения команды SMLABB

При выполнении множественных операций умножения с накоплением в качестве имени регистра-приемника Rd и имени регистра-аккумулятора Ra должно быть указано имя одного и того же регистра ЦПУ:

Ra = (Ra) + [(полуслово из Rn)*(полуслово из Rm)]

1) Модифицируйте основную программу проекта CPU_22 так, чтобы она обрабатывала 16-разрядные коэффициенты и выборки в формате (S4.12). Используйте те же значения коэффициентов и выборок, что и ранее (+7.0).

2) Проверьте факт формирования флага насыщения Q уже на третьей операции умножения с накоплением.

3) В чем же основное отличие этих двух реализаций регулятора/фильтра?

1) В первую очередь измените формат коэффициентов в ПЗУ с 32-разрядного на 16-разрядный. При доступе к коэффициентам измените величину авто-смещения указателя с +4 на +2 (число байт в памяти, занимаемых полусловом). Используйте команду умножения с накоплением SMLABB. Если не справились, подключите к проекту файл MyProg_22_1.s. Изучите его.

2) На одном и том же наборе входных данных, представленных только целыми числами, обе программы работают одинаково, что и должно быть.

3) В первом варианте точность представления коэффициентов выше, во втором – ниже. Поэтому, в общем случае, когда в коэффициентах будет не только целая, но и дробная часть, предпочтение следует отдать первой реализации.

17.5 Цифровой фильтр «скользящего среднего» с кольцевым буфером выборок

17.5.1 Назначение и принцип действия

Одно из основных применений цифровых фильтров – выделение из зашумленного помехами аналогового сигнала полезной составляющей, которую можно использовать при расчете управляющих воздействий в системах цифрового управления. Наиболее простой фильтр – это *фильтр*

«скользящего среднего». Истинное значение входной переменной определяется как среднее значение нескольких, подряд сделанных выборок:

$$y[k] = \frac{x[k]+x[k-1]+x[k-2]+\cdots+x[k-n]}{(n+1)} = b_0 \cdot x[k] + b_1 \cdot x[k-1] + \cdots + b_n \cdot x[k-n] \qquad (17.1)$$

Выражение (17.1) полностью соответствует уравнению КИХ фильтра, в котором все коэффициенты фильтра одинаковы и равны:

$$b_i = {}^{1}/_{(n+1)}. \qquad (17.2)$$

Наиболее часто используется усреднение по четному числу выборок, например, по 4 или 8. Если одна или несколько выборок существенно отличаются от среднего значения, то выходное значение фильтра изменяется незначительно. Это эквивалентно операции «сглаживания» входной переменной, когда отдельные девиации переменной (выбросы) нивелируются.

17.5.2 Директива Ассемблера FILL заполнения памяти константой

ARM ASM

Часто при программировании возникает ситуация, когда необходимо заполнить целую область памяти заданной константой, например, проинициализировать несколько одинаковых коэффициентов регулятора/фильтра (как в фильтре «скользящего среднего»). Если директива Ассемблера SPACE только выделяет определенную область памяти под переменные, не инициализируя ее, то директива FILL дополнительно заполняет эту область памяти шаблоном (константой). Напомним синтаксис этой директивы:

`{label} FILL expr{,value{,valuesize}}`

где:

- `label` – необязательная метка (опция);
- `expr` - число байт в инициализируемой памяти;
- `value` – значение, которым должна заполняться резервируемая область памяти. Если опция не указана, то область памяти заполняется нулями.
- `valuesize` – размер константы заполнения в байтах. Может быть равным: 1 (байт), 2 (полуслово), 4 (слово). Если опция не указана, то принимается равной 1 (байт).

Будьте внимательны: если память заполняется полусловами, то число резервируемых байт должно быть удвоено по сравнению с числом слов, если полными словами, то увеличено в 4 раза. Как и после директивы SPACE, после директивы заполнения памяти константой FILL рекомендуется использовать директиву выравнивания памяти по полному слову ALIGN.

Пример:

```
AREA MyCode, CODE, ReadOnly
Koeff_b FILL  50, 0xAF, 1     ; Инициализация 50 байтовых
                              ; констант 0xAF
Koeff_H FILL 10, 35, 2        ; Инициализация 5 полуслов 35
```

17.5.3 Программная реализация фильтра «скользящего среднего»

В предыдущей главе (см. 16.7) мы подробно исследовали комплект подпрограмм для работы с кольцевым буфером 32-разрядных данных. Так как разрядность данных после преобразования в АЦП обычно равна 10-12 разрядам, то для представления выборок сигнала точности 16-разрядного формата S4.12 может быть достаточно. Преобразуем библиотеку подпрограмм работы с кольцевым буфером для 16-разрядных чисел с фиксированной точкой, добавив в нее процедуру преобразования входных данных, считанных с 10-разрядного АЦП. Для реализации фильтра «скользящего среднего» будем использовать коэффициенты в 16-разрядном формате (S4.12), предполагая, что усреднение аналогового сигнала выполняется по 8 выборкам (коэффициенты фильтра одинаковы и равны 1/8). Для реализации фильтра используем команду умножения с накоплением SMLABB Ra, Rn, Rm, Ra, в которой первый сомножитель (коэффициент фильтра) загружается в регистр Rn (в младшее слово), а второй – выборка – считывается из кольцевого буфера в регистр Rm (также в младшее слово). Для расчета выходного управляющего воздействия потребуется цикл из 8-и команд умножения с накоплением. Обратимся к проекту CPU_23. В его составе три программных модуля: стартовый файл StartUp_1.s, основная программа приложения MyProg_23.s и файл с библиотекой обслуживания кольцевого буфера 16-разрядных данных Circ_Buffer.s (несколько модернизированной).

```
MyProg
; Основная программа
; Обявить переменную - число полуслов в кольцевом буфере
N       EQU  8
; Объявить точки входа в подпрограммы обслуживания
; кольцевого буфера выборок-полуслов
        IMPORT Inicy_Circ_Buffer_16
        IMPORT Convert_X_16
        IMPORT Write_Circ_Buffer_16
        IMPORT Read_Circ_Buffer_16
        IMPORT Output_Control_32
; Инициализация параметров кольцевого буфера,
; предварительная очистка кольцевого буфера
        BL Inicy_Circ_Buffer_16
; Задать число циклов работы фильтра
        MOV r10, #20
```

В начале основной программы объявляется переменная, определяющая порядок фильтра N, и делаются ссылки на внешние переменные – точки входа в подпрограммы обслуживания кольцевого буфера, расположенные в файле Circ_Buffer.s. Далее вызывается подпрограмма очистки и инициализации параметров кольцевого буфера. Задается число циклов вызова основной процедуры фильтрации, с учетом того, что в процессе отладки на вход фильтра будет подано *скачкообразное входное воздействие* (константа) и графически будет исследоваться реакция фильтра на это воздействие.

```
;*********************************************************
; Зарезервировать в кодовой памяти одинаковые коэффициенты
; фильтра "скользящего среднего" в формате (S4.12) полуслов,
; соответствующие значению 1/N
k       EQU 4096/N
Koeff   FILL (N*2), k, 2
        EXPORT Koeff
;*********************************************************
```

В конце основной программы зарезервируем в ПЗУ коэффициенты фильтра. Сделаем это с помощью директивы заполнения памяти константой FILL. Напомним, что относительному значению переменной 1 в формате (S4.12) будет соответствовать целочисленный код 2^{12}=4096. Следовательно, нужно зарезервировать коэффициенты, равные 4096/k.

```
;*********************************************************
; Объявить секцию данных в ОЗУ для размещения кольцевого
; буфера выборок (полуслов в формате S4.12)
          ALIGN
          AREA MyData, DATA, ReadWrite
ADR_0
          SPACE N*2
ADR_1
          EXPORT ADR_0  ; Начальный адрес буфера
          EXPORT ADR_1  ; Конечный адрес буфера
; Входная и выходная переменные фильтра - глобальные
; для наблюдения в логическом анализаторе
; Зарезервировать одно слово в памяти для имитации
; порта ввода данных (выходной регистр АЦП -
; расположен в младшем полуслове)
Port_ADC_32
          SPACE 4
          EXPORT Port_ADC_32
; Зарезервировать одно слово в памяти для имитации
; порта вывода управляющего воздействия (выход фильтра)
; в формате S4.28
Port_Out_32
          SPACE 4
          EXPORT Port_Out_32
;*********************************************************
```

Как обычно, в конце основной программы объявим секцию данных, в которой будут размещаться переменные проекта. В первую очередь – это кольцевой буфер 16-разрядных выборок входных переменных с начальным адресом ADR_0. После него зарезервируем место под две глобальные переменные: порт ввода данных из ячейки памяти, на которую отображен выходной регистр АЦП Port_ADC_32,

и порт вывода, рассчитанного в процедуре фильтрации, выходного управляющего воздействия Port_Out_32. Несмотря на то, что порт ввода данных с АЦП может быть всего лишь 16-разрядным (полезная информация в младшем слове), мы объявляем его 32-разрядным для возможности графического отображения входного сигнала в окне логического анализатора. Выходное воздействие – 32-разрядный код в формате (S8.24) – также глобальная переменная (см. директивы EXPORT).

```
24  ; Тело цифрового фильтра "скользящего среднего"
25  Filter
26  ; Получить очередную выборку данных из АЦП
27  ; Конвертироать выборку в формат (S4.12)
28          BL Convert_X_16
29  ; Сохранить выборку в кольцевом буфере
30          BL  Write_Circ_Buffer_16
31  ; Рассчитать выходное управляющее воздействие
32    ; Очистить регистр-аккумулятор r5
33          MOV r5,#0
34    ; Инициализировать счетчик числа умножений/накоплений
35          MOV r8,#N
36    ; Установить указатель в r9 на начало таблицы коэффициентов
37          ADR r9,Koeff
38  Loop_Filter
39    ; Считать очередной коэффициент в регистр r6 с пост-авто
40    ; инкрементированием указателя
41          LDRH r6,[r9],#2
42    ; Считать очередную выборку в регистр r7 с авто-модификацией
43    ; указателя в кольцевом буфере выборок
44          BL Read_Circ_Buffer_16
45    ; Умножение с накоплением коэффициента на выборку
46    ; [(r6)*(r7)+r5] -> r5
47          SMLABB r5,r6,r7,r5
48    ; Декрементировать счетчик числа циклов
49          SUBS r8,#1
50    ; Если не все операции выполнены, повторить
51          BNE Loop_Filter
52  ; Выдать управляющее воздействие в порт
53  ; Port_Out_Control_32 в формате слова (S8.24)
54          BL Output_Control_32
55  ; Декрементировать счетчик числа циклов работы фильтра
56          SUBS r10,#1
57  ; Если не все циклы фильтрации выполнены,
58  ; повторить для новой выборки
59          BNE Filter
60  ; Фильтрация заданного числа выборок выполнена
61  ; Предотвратить выполнение несуществующего кода
62          B .
```

Перейдем к телу основной программы. Сначала получим из порта ввода очередное значение входной переменной и вызовем подпрограмму конвертации исходного числа (U10.0) в формат (S4.12). Сохраним полученное значение в кольцевом буфере. Предварительно очистим регистр-аккумулятор r5, зададим число циклов умножения с накоплением в r8 и установим указатель r9 на начало коэффициентов фильтра в ПЗУ.

Теперь можно последовательно загрузить очередной коэффициент и очередную выборку из кольцевого буфера, - выполнить их умножение с накоплением. Затем, повторить эту операцию для всех коэффициентов и выборок, имеющихся в буфере.

В завершение – выдать рассчитанное значение управляющего воздействия в порт вывода. Новую процедуру фильтрации начнем с получения очередного значения входной переменной, если не все входные данные считаны. В нашем примере фильтр вызывается 20 раз (см. строку 23 программы выше).

В самом конце – традиционная команда предотвращения выполнения несуществующего кода, на которой можно поставить точку останова.

Теперь – некоторые особенности библиотеки обслуживания кольцевого буфера. Мы уже отмечали, что для удобства работы с кольцевым буфером за ним «жестко» закреплены определенные регистры ЦПУ. Это позволяет оперативно контролировать границы кольцевого буфера и работать с текущим указателем данных в буфере.

```
;*************************************************************
; Комплект подпрограмм с унифицированным распределением
; регистров ЦПУ для цифровой фильтрации с
; использованием кольцевого буфера данных в формате S4.12:
; r0 - содержит начальный адрес кольцевого буфера
; r1 - указатель адреса текущей ячейки кольцевого буфера
;   ( пост-инкрементируется при записи данных в буфер,
;     пре-декрементируется перед чтением данных из буфера)
; r2 - содержит конечный адрес кольцевого буфера
; r3 - содержит адрес порта ввода входной переменной x[k]
; r4 - содержит адрес порта вывода управляющего воздействия y[k]
;*************************************************************
; Объявить внешние переменные, используемые в подпрограммах
        IMPORT ADR_0    ; Начальный адрес кольцевого буфера
        IMPORT ADR_1    ; Конечный адрес кольцевого буфера
        IMPORT Port_ADC_32  ; Выходной порт АЦП
        IMPORT Port_Out_32  ; Порт вывода управляющего
                            ; воздействия
; Обявить точки входа в подпрограммы - глобальными переменными
        EXPORT Inicy_Circ_Buffer_16
        EXPORT Convert_X_16
        EXPORT Read_Circ_Buffer_16
        EXPORT Write_Circ_Buffer_16
        EXPORT Output_Control_32

; Объявить кодовую секцию MyCode для размещения
; подпрограмм обслуживания кольцевого буфера.
        AREA MyCode, CODE, ReadOnly

```

```
;*************************************************************
; Подпрограмма инициализации параметров кольцевого буфера
; с автоматической очисткой буфера как линейного буфера
; Используемые регистры: r4,r5
; Выход: очищенный кольцевой буфер данных в ОЗУ
;   r0 - минимально допустимый адрес кольцевого буфера
;   r1 - указатель текущей ячейки кольцевого буфера
;   r2 - максимально допустимый адрес кольцевого буфера
;   r3 - адрес порта ввода входной переменной x[k]
;   r4 - адрес порта вывода управляющего воздействия y[k]
;*************************************************************
Inicy_Circ_Buffer_16
; Загрузить в r0 начальный адрес кольцевого буфера
        LDR r0,=ADR_0
; Скопировать в регистры r1 и r4
        MOV r1, r0
        MOV r4, r0
; Загрузить в r2 конечный адрес кольцевого буфера
        LDR r2,=ADR_1
        SUB r2,#2
; Очистить все ячейки кольцевого буфера, как линейного
; Загрузить "шаблон" заполнения буфера данными
        MOV r3,#0
Loop
; Очистить текущую ячейку с пост-авто-смещением указателя
        STRH r3, [r4],#2
; Сравнить текущий адрес с последним адресом буфера
        CMP r4,r2
; Пока буфер не очищен полностью, продолжать
        BLS Loop  ; "Меньше или то же самое"
; Установить указатель r3 на адрес порта Port_ADC_32
        LDR r3,=Port_ADC_32
; Установить указатель r4 на адрес порта вывода
; управляющего воздействия Port_Out_Control_32
        LDR r4,=Port_Out_32
; Выход из подпрограммы
        BX lr
;*************************************************************
```

Так как все переменные объявлены в основном программном модуле, для работы с ними требуется ссылка на них, как на внешние (импортируемые). Напротив, все точки входа в подпрограммы объявляются общедоступными. Все подпрограммы располагаются в той же кодовой секции, что и основная программа.

Процедура инициализации, как и ранее (см. 16.7), не только предварительно очищает буфер выборок, готовя его к приему входных данных, но и инициализирует регистры ЦПУ, обслуживающие кольцевой буфер, а также регистры-указатели на порт ввода входной переменной и порт выдачи управляющего воздействия. Очистка буфера выполняется, как линейного.

```
;*************************************************************
; Подпрограмма преобразования входной переменной X,
; считанной из порта Port_ADC, в число с фиксированной
; точкой в формате S4.12 (X в относительных единицах)
; Коэффициент возможного превышения номинального значения 2
; Вход: Данные в порту Port_ADC_32 (10 разр. код без знака)
; Выход: Переменная X в формате числа со знаком S4.12
;        в регистре ЦПУ r7
;*************************************************************
Convert_X_16
; Считать исходные данные - 10 разрядный код без знака
; в младших разрядах порта Port_ADC (выход АЦП) с
; расширением нулями в область старших разрядов
        LDRH r7,[r3]
; Преобразовать в целое число со знаком в формате S10.0
; (эквивалентные форматы S6.10 и S22.10)
        SUB r7, #512
; Формат относительного значение входной переменной с учетом
; возможного двойного превышения номинального значения
; будет соответствовать формату S2.8
; Конвертировать в формат S4.12
        LSL r7,r7,#4
; Результат - в младшем полуслове регистра r7
; Выход из подпрограммы
        BX lr
;*************************************************************
```

Технология конвертации данных, полученных с АЦП, подробно рассмотрена нами ранее для разных типов сигналов (см. 16.9). Предположим, что в нашем случае используется разнополярный входной сигнал.

Заметим, что исходное число является числом без знака в формате U10.0. Сначала оно преобразуется в знаковый формат S1.9, который для относительных выходных переменных с коэффициентом возможного превышения номинального значения 2 является форматом (S2.8). Для перехода к формату (S4.12) – выполняется дополнительный логический сдвиг влево на 4 разряда. Полученное значение будет знаковой входной переменной в относительных единицах в формате (S4.12), которая и подлежит записи в кольцевой буфер выборок (более подробно в 16.9).

Остальные подпрограммы нам уже знакомы (см. 16.7). Их отличительная особенность состоит в том, что теперь все данные 16-разрядные. Следовательно, при работе с указателями величины авто-смещений уменьшаются в два раза (вместо 4-х – на 2 байта).

```
;*************************************************************
; Подпрограмма записи новой выборки в кольцевой буфер
; Входы: новое значение входной переменной x[k] в формате
;        (S4.12) в относительных единицах в регистре r7
;        (младшее полуслово)
; Выход: модифицированный кольцевой буфер
;*************************************************************
Write_Circ_Buffer_16
; Записать новую выбооку по текущему указателю адреса
; "свободной" ячейки в кольцевом буфере с авто-смещением
; указателя кольцевого буфера
        STRH r7, [r1],#2
; Проверить превышение максимально допустимого адреса
        CMP r1, r2
; Если да, пере-инициализировать указатель кольцевого
; буфера начальным адресом буфера
        MOVHI r1,r0
; Выход из подпрограммы
        BX lr
;*************************************************************
```

```
;*************************************************************
; Подпрограмма чтения очередной выборки из
; кольцевого буфера с предварительным декрементированием
; указателя и проверкой выхода за минимально допустимый
; адрес
; Выход: r7 - считанные из буфера данные
;*************************************************************
Read_Circ_Buffer_16
; Декрементировать на 2 текущее значение в указателе
; кольцевого буфера
        SUB r1,#2
; Сравнить с минимально допустимым адресом
        CMP r1, r0
; Если "строго ниже", заменить максимальным адресом
        MOVLO r1,r2
; Считать выборку в регистр r7
        LDRH r7, [r1]
; Выход из подпрограммы
        BX lr
;*************************************************************
```

Еще раз обратите внимание на то, как при записи данных в кольцевой буфер контролируется верхняя граница буфера, а при извлечении из него данных – нижняя граница.

Последней в наборе подпрограмм является процедура выдачи управляющего воздействия: результат расчета выдается в порт вывода с использованием обычной косвенной адресации.

```
;*************************************************************
; Подпрограмма выдачи управляющего воздействия
; Вход: Накопленное произведение в регистре-аккумуляторе r5
;       в формате (S4.28)
; Выход: 32-разрядное слово в формате (S4.28) в порту
;        Port_Out_Control_32
; Используемый регистр: r4
;*************************************************************
Output_Control_32
; Выдать рассчитанное управляющее воздействие
        STR r5, [r4]
; Завершить подпрограмму
        BX lr
;*************************************************************
```

Выполните трансляцию файлов проекта и его сборку. Перед отладкой программы исследуйте файл с картой загрузки CPU_23.map на предмет фактического размещения в оперативной памяти переменных. Вам придется при отладке программы открыть окно дампа памяти, чтобы, с одной стороны, знать, в какую ячейку памяти можно ввести входную переменную (оцифрованный сигнал с АЦП) – Port_ADC_32, а с другой – просматривать текущее содержимое кольцевого буфера (с начального адреса ADR0) и порта вывода управляющего воздействия – Port_Out_32. Для текущего наблюдения за входной и выходной переменной целесообразно открыть окно логического анализатора и указать имена переменных Port_ADC_32 и Port_Out_32 для включения их в механизм трассировки данных и вывода графиков на экран.

ADR_0	0x20000000	Data	0	myprog_23.o(MyData)
ADR_1	0x20000010	Data	0	myprog_23.o(MyData)
Port_ADC_32	0x20000010	Data	0	myprog_23.o(MyData)
Port_Out_32	0x20000014	Data	0	myprog_23.o(MyData)

Настройте логический анализатор, как подробно описано в 16.7. Входное воздействие считается целым числом без знака в формате U10.0 (0÷1023). В меню логического анализатора можно задать для этой переменной маску 0x3FF и нулевой сдвиг вправо. Выходное воздействие является вещественным числом со знаком в формате S8.24. Целесообразно вывести на экран только его старшее полуслово (S8.8) со сдвигом вправо на 16 разрядов.

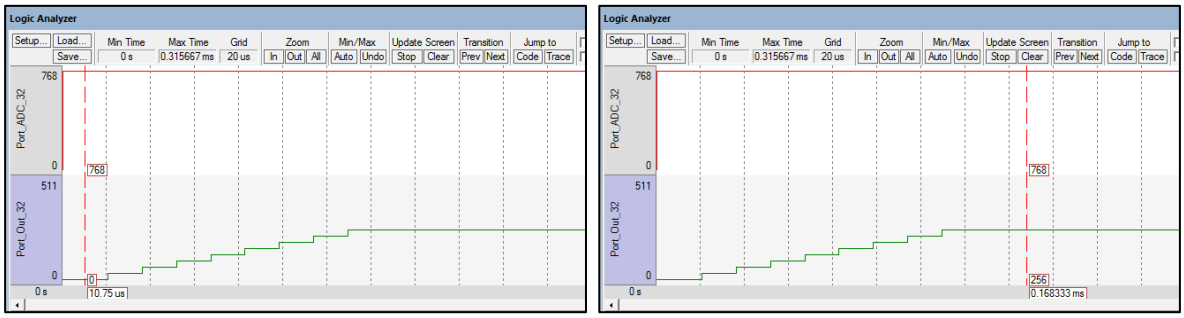
Рассмотрим поведение программы при двух входных воздействиях (пока только положительных):

1) U10.0=768, что соответствует относительному значению x*=+1;
2) U10.0=1020, что соответствует относительному значению x*=≈+2.

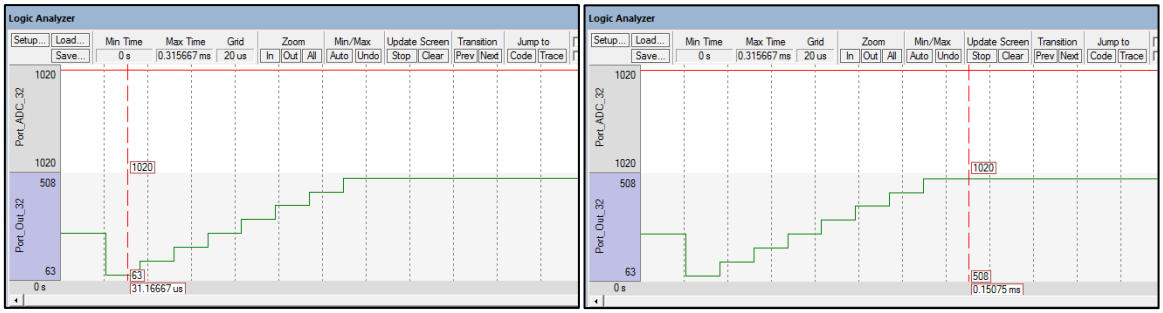
Далее мы покажем, как можно использовать логический анализатор и для наблюдения за знаковыми переменными (см. 22.7).

Введите нужное входное управляющее воздействие в соответствующую ячейку памяти (модифицируйте память). После этого запустите программу в режиме RUN до точки останова.

В первом случае выходной сигнал за восемь тактов работы фильтра «скользящего среднего» линейно возрастает с 0 до числа 256, которое в формате S8.8 (только старшее полуслово) представляет собой именно относительную 1. Величина приращения 32 соответствует 1/8 значения очередной выборки, записанной в буфер. После того, как все выборки заполнят буфер, выходной сигнал меняться не будет, что и должно быть:



Во втором случае входное воздействие увеличилось почти вдвое. Выходной установившийся сигнал будет равен 508/256=+1.984375≈+2. Цифровой фильтр правильно отрабатывает задание:



1) Выполните отладку фильтра «скользящего среднего» при других значениях входной переменной. Для корректного отображения результатов в логическом анализаторе используйте пока положительные входные сигналы. В главе 22 показано, как можно использовать его для удобного отображения и знакопеременных сигналов.

2) Комплект подпрограмм обслуживания кольцевого буфера работает так, что при записи очередной выборки сначала выполняется запись в буфер, а затем – указатель текущей ячейки перемещается на одну позицию в направлении возрастающих адресов. Казалось бы, можно было этого не делать, так как дальше все равно потребуется извлечение данных из кольцевого буфера, но уже в направлении убывающих адресов. Ваше мнение?

2) Да, но при извлечении данных из кольцевого буфера выполняется пост-авто-смещение указателя в направлении убывающих адресов с контролем пересечения нижней границы буфера. Следовательно, после извлечения всех данных из буфера указатель будет показывать на исходную ячейку памяти. Все равно, перед записью новой выборки придется сместить указатель на одну позицию в направлении возрастающих адресов, причем с контролем пересечения границы буфера.

Процессоры Cortex-M4 имеют широкий спектр команд умножения с накоплением, позволяющий эффективно реализовывать любые цифровые регуляторы и фильтры в формате чисел с фиксированной точкой. При этом можно выбирать как команды, в которых контролируется «насыщение», так и команды «длинного» умножения с накоплением, в которых переполнения автоматически исключаются.

Среда µVision имеет все необходимые инструменты для эффективной отладки таких приложений. Однако, нужно иметь в виду, что работа с числами с фиксированной точкой требует от программиста *особой внимательности*, в том числе при интерпретации данных в окнах памяти и в окнах логического анализатора – Вы всегда должны помнить о формате данных и месте расположения десятичной точки. К сожалению, логический анализатор по умолчанию работает только с переменными без знака определенной разрядности, что затрудняет отладку приложений с плавающей точкой.

Для ускорения отладки рекомендуется вести подобные разработки на языках высокого уровня с использованием *специализированных библиотек поддержки вычислений в форматах с фиксированной точкой (I.Q)*. Они позволяют выбрать нужный формат представления чисел и поддерживают любые вычисления в этом формате, начиная от сложения, вычитания, умножения, умножения с накоплением, – до вычисления тригонометрических функций. Недостающие функции Вы можете разработать сами, в том числе, с учетом знаний, полученных в этой главе.

В главах 19, 20, 21 книги описывается технология разработки программного обеспечения с использованием чисел в формате с плавающей точкой, которая позволяет быстро создать и отладить любой цифровой регулятор/фильтр. Более того, команды сопроцессора можно использовать и для ускорения отладки программ, написанных в формате чисел с фиксированной точкой. Для этого применяются команды преобразования чисел из формата с фиксированной точкой в формат с плавающей точкой и специальные методы отображения в логическом анализаторе графиков переменных в формате с плавающей точкой (см. 22.7).

17.6 Команды насыщения

17.6.1 Назначение

Наряду с командами умножения с накоплением *команды насыщения* относятся к командам поддержки цифровой обработки сигналов. Они используются, прежде всего, для ограничения результатов вычислений на допустимом уровне, позволяя избежать переполнений и связанных с этим скачков разрывности управляющих воздействий, приводящих к автоколебаниям и даже к потере устойчивости систем управления.

В цифровых системах управления с интегральными регуляторами имеется еще одна опасность – выход интегрального регулятора может надолго остаться в зоне предельно больших или предельно низких значений («как бы залипнуть»), и процесс уменьшения накопленной «интегральной составляющей» может затянуться, что опять же приведет к автоколебаниям и, возможно, к потере устойчивости системы управления. В таком случае авто-ограничение выхода регулятора на допустимом уровне эквивалентно исключению его «инерционности». Такой цифровой регулятор становится аналогичным аналоговому на базе операционного усилителя с ограниченным выходным сигналом.

17.6.2 Путеводитель по командам насыщения

Команды насыщения работают только с числами со знаком в дополнительном коде. Они выполняют либо знаковое насыщение (Signed Saturate), ограничивая входное значение максимально возможным диапазоном представления знаковых чисел заданной разрядности, либо беззнаковое насыщение (Unsigned Saturate), ограничивая входное значение максимально возможным диапазоном чисел без знака заданной разрядности и нулем. Префикс «S» в кодировке команды означает знаковое насыщение, а префикс «U» – беззнаковое насыщение.

Во всех командах насыщения исходные операнды считаются числами со знаком в дополнительном коде (целыми, дробными или вещественными с фиксированной точкой, положение которой знает только программист). Поэтому, исходное 32-разрядное слово со знаком можно ограничить числом без знака максимум на уровне 31-го разряда – числом (2^{31}-1). Теоретически допустимое в рамках 32-х разрядов максимальное слово без знака 2^{32} просто не имеет аналога среди знаковых 32-разрядных чисел, поэтому – не используется.

Факт «срабатывания» насыщения сопровождается установкой флага «насыщения» Q. Он «защелкивается» (фиксируется) и сбрасывается исключительно программно, когда регистр состояния процессора считывается в регистр общего назначения командой MRS и далее очищается командой MSR (см. 17.4.2).

Имеется группа команд, которые выполняют *параллельное сложение или вычитание* операндов-полуслов или байт, автоматически ограничивая значения двух или четырех сумм или разностей допустимым диапазоном представления чисел со знаком или без знака соответствующей разрядности (16 или 8). Для идентификации этих команд используется специальный префикс «Q» перед мнемокодом команды:

Q – параллельное сложение или вычитание байт/полуслов со знаком и знаковым ограничением;

UQ – параллельное сложение или вычитание байт/полуслов со знаком с беззнаковым ограничением;

Напомним, что в 32-разрядных процессорах ARM обычные команды сложения и вычитания слов не выполняют авто-ограничения результата. При выходе результата за пределы разрядной сетки могут формироваться лишь флаги «C» (**C**arry) – переполнение при выполнении операций с числами без знака и «V» (O**v**erflow) – переполнение при выполнении операций с числами со знаком. При параллельных операциях сложения/вычитания полуслов или байт с ограничением результата на уровне максимально/минимально допустимых значений для 16- или 8-разрядного формата эти флаги не формируются. Не формируется и флаг насыщения Q, так как непонятно, при выполнении какой из двух или четырех параллельных операций он возник.

Таким образом, флаг Q формируется только при выполнении операций с полными 32-разрядными словами – ограничения или сложения/вычитания с попутным ограничением. Исключением являются команды параллельного знакового или беззнакового насыщения двух полуслов, входящих в слово, которые также формируют флаг насыщения, если оно имело место при выполнении любой из двух операций насыщения.

Команды работы с отдельными полусловами или байтами, входящими в слова, применяются для обработки так называемых *упакованных* в слова байтов или полуслов, – при работе с массивами данных, полученных, обычно, по каналам связи. В большинстве систем управления реального времени требуется настоящая 32-разрядная обработка данных. Соответствующие команды, как наиболее важные, выделены в табл. 17.11 красными прямоугольниками в левой части таблицы.

Таблица 17.11 Справочник по командам насыщения

	Мнемоника	Операнды	Краткое описание	Флаги
Команды насыщения знаковых слов диапазоном чисел со знаком заданной разрядности или диапазоном чисел без знака заданной разрядности				
	SSAT	Rd, #n, Rm{, shift #s}	Ограничение слова, предварительно сдвинутого на #s разрядов диапазоном n-разрядного числа со знаком $-2^{(n-1)} \le x \le 2^{(n-1)}-1$	Q
	USAT	Rd, #n, Rm{, shift #s}	Ограничение слова, предварительно сдвинутого на #s разрядов диапазоном n-разрядного числа без знака $0 \le x \le 2^n-1$	Q
Команды параллельного насыщения знаковых полуслов диапазоном чисел со знаком заданной разрядности или диапазоном чисел без знака заданной разрядности				
	SSAT16	Rd, #n, Rm	Ограничение двух полуслов диапазоном n-разрядного числа со знаком $-2^{(n-1)} \le x \le 2^{(n-1)}-1$	Q
	USAT16	Rd, #n, Rm	Ограничение двух полуслов диапазоном n-разрядного числа без знака $0 \le x \le 2^n-1$	Q
Команды параллельного знакового насыщения сумм или разностей слов, одноименных полуслов или байт стандартными диапазонами чисел со знаком в дополнительном коде				
	QADD	{Rd,} Rn, Rm	Ограничение суммы слов диапазоном $-2^{31} \le x \le 2^{31}-1$	Q
	QADD16	{Rd,} Rn, Rm	Ограничение двух сумм одноименных полуслов диапазоном $-2^{15} \le x \le 2^{15}-1$	-
	QADD8	{Rd,} Rn, Rm	Ограничение четырех сумм одноименных байт диапазоном $-2^7 \le x \le 2^7-1$	-
	QSUB	{Rd,} Rn, Rm	Ограничение разности слов диапазоном $-2^{31} \le x \le 2^{31}-1$	Q
	QSUB16	{Rd,} Rn, Rm	Ограничение двух разностей одноименных полуслов диапазоном $-2^{15} \le x \le 2^{15}-1$	-
	QSUB8	{Rd,} Rn, Rm	Ограничение четырех разностей одноименных байт диапазоном $-2^7 \le x \le 2^7-1$	-
Команды параллельного вычисления суммы и разности полуслов с «перекрещиванием» операндов и знаковым насыщением				
	QASX	{Rd,} Rn, Rm	Ограничение суммы и разности «перекрещенных» полуслов в регистрах-источниках диапазоном $-2^{15} \le x \le 2^{15}-1$	-
	QSAX	{Rd,} Rn, Rm	Ограничение разности и суммы «перекрещенных» полуслов в регистрах-источниках диапазоном $-2^{15} \le x \le 2^{15}-1$	-
Команды параллельного беззнакового насыщения сумм или разностей одноименных полуслов или байт стандартным диапазоном чисел без знака				
	UQADD16	{Rd,} Rn, Rm	Ограничение двух сумм одноименных полуслов диапазоном $0 \le x \le 2^{16}-1$	-
	UQADD8	{Rd,} Rn, Rm	Ограничение четырех сумм одноименных байт диапазоном $0 \le x \le 2^8-1$	-
	UQSUB16	{Rd,} Rn, Rm	Ограничение двух разностей одноименных полуслов диапазоном $0 \le x \le 2^{16}-1$	-
	UQSUB8	{Rd,} Rn, Rm	Ограничение четырех разностей одноименных байт диапазоном $0 \le x \le 2^8-1$	-

Команды параллельного вычисления суммы и разности полуслов с «перекрещиванием» операндов и беззнаковым насыщением				
	UQASX	**{Rd,} Rn, Rm**	**Параллельное вычисление суммы и разности полуслов с «перекрещиванием» операндов и беззнаковым насыщением $0 \le x \le 2^{16}-1$**	**-**
	UQSAX	**{Rd,} Rn, Rm**	**Параллельное вычисление разности и суммы полуслов с «перекрещиванием» операндов и беззнаковым насыщением $0 \le x \le 2^{16}-1$**	**-**

17.6.3 Команды насыщения слов со знаком

Это главные команды «насыщения», которые предназначены для ограничения 32-разрядного результата вычислений диапазоном чисел со знаком заданной разрядности (знаковое насыщение) или диапазоном чисел без знака заданной разрядности:

```
SSAT{cond} Rd,#n,Rm {,shift #s}
USAT{cond} Rd,#n,Rm {,shift #s}
```

Предварительный сдвиг исходного операнда-слова в Rm:
ASR #s - Арифметический сдвиг вправо;
LSL #s - Логический сдвиг влево,
где s – число разрядов сдвига (1÷31)

Регистр-источник данных – слова со знаком в дополнительном коде

#n – Номер бита, к которому должно быть выполнено ограничение в диапазоне (1÷32) для знакового насыщения и (0-31) для беззнакового

Регистр назначения для сохранения результата насыщения

Код условного выполнения команды (опция)

SAT – (Saturation) – Насыщение, ограничение знаковых слов допустимым диапазоном чисел
со знаком: $-2^{(n-1)} \le x \le 2^{(n-1)}-1$ или чисел без знака: $0 \le x \le 2^n-1$

S - Signed – Ограничение диапазоном чисел со знаком
U – Unsigned – Ограничение диапазоном чисел без знака

Это мощные команды, так как позволяют *предварительно* выполнить *сдвиг* исходного операнда-слова на требуемое число разрядов перед выполнением операции насыщения. Ограничение выполняется к диапазону знакового слова или к диапазону слова без знака требуемой разрядности n:

Диапазоны «знаковых» ограничений для разных n:
n=8: $(-2^{8-1}) \le x \le (2^{8-1}-1)$ или **$-128 \le x \le +127$**;
n=16: $(-2^{16-1}) \le x \le (2^{16-1}-1)$ или **$-32\,768 \le x \le +32\,767$**;
n=32: $(-2^{32-1}) \le x \le (2^{32-1}-1)$ или **$-2\,147\,483\,648 \le x \le +2\,147\,483\,647$**;

Диапазоны «беззнаковых» ограничений для разных n:
n=8: $(0 \le x \le 2^8-1)$ или **$0 \le x \le 255$**
n=16: $(0 \le x \le 2^{16}-1)$ или **$0 \le x \le 65535$**
n=31: $(0 \le x \le 2^{31}-1)$ или **$0 \le x \le 2\,147\,483\,647$**

Обратите внимание, что для беззнакового ограничения максимальное значение n на единицу меньше, чем для знакового. Это позволяет корректно выполнить беззнаковое ограничение любого исходного числа со знаком.

Графическая иллюстрация двух возможных вариантов ограничения с программно устанавливаемым уровнем ограничения показана на рис. 17.3. Первое ограничение является симметричным, как в аналоговых операционных усилителях. Второе – **несимметричным**, все отрицательные числа автоматически заменяются нулями.

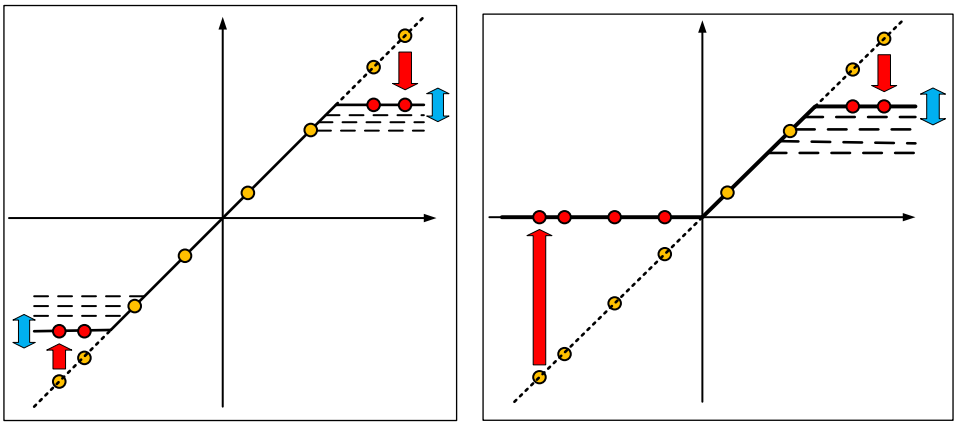


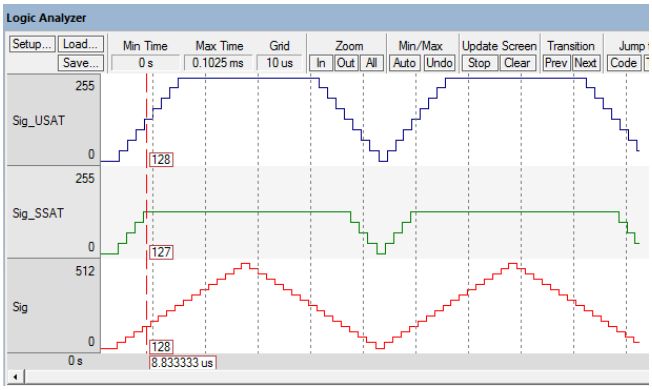
Рис. 17.3 Алгоритм знакового и беззнакового ограничения

Любые числа, которые выходят за требуемый диапазон, – ограничиваются, а те, которые находятся внутри него, – остаются неизменными.

17.6.4 Пример ограничения выходного сигнала генератора

Обратимся к проекту CPU_24. Программа приложения MyPog_24.s демонстрирует особенности использования команд знакового и беззнакового ограничения на примере цифрового генератора периодического сигнала. *Задача*: сгенерировать треугольный периодический сигнал Sig (опорного генератора) с нарастающим фронтом от минимального значения MIN до максимального MAX с приращением DELTA на каждом такте работы генератора и со спадающим фронтом от максимального значения до минимального с таким же приращением; повторить генерацию опорного сигнала на заданном числе периодов; ограничить выходной сигнал опорного генератора диапазоном 8-разрядных чисел со знаком (-128 ÷ +127) и выдать в качестве сигнала Sig_SSAT; ограничить выходной сигнал опорного генератора диапазоном 8-разрядных чисел без знака (0 ÷ 255) и выдать в качестве сигнала Sig_USAT.

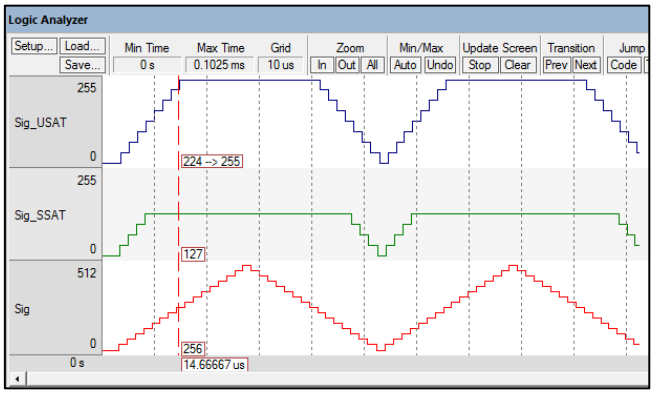
Предваряя текст программы, покажем результат ее отладки в окне логического анализатора. Треугольный сигнал опорного генератора Sig возрастает от 0 (MIN) до 512 (MAX) с дискретностью 32 (16 приращений), после чего уменьшается с 512 до 0 с той же дискретностью. Генерируется заданное число периодов опорного сигнала (в данном случае 2).



Сигнал опорного генератора приводится с использованием команды знакового ограничения SSAT к диапазону 8-разрядных чисел со знаком Sig_SSAT, а с помощью команды беззнакового ограничения USAT – к диапазону 8-разрядных чисел без знака – Sig_USAT.

Вы видите, как уже первое число опорного генератора 128, превышающее максимальное положительное число со знаком $S8_{max}$= +127, ограничивается значением +127. При дальнейшем росте опорного сигнала его значение (+256) начинает превышать

уже максимально возможное 8-разрядное число без знака $U8_{max}$=255, и выходной сигнал Sig_USAT ограничивается на уровне 255. Мы использовали начальную часть диапазона целых чисел со знаком для наглядности представления данных в логическом анализаторе.




```
; Основная программа
; Число генерируемых периодов сигнала
N       EQU 2
; Определение максимального и минимального
; значений опорного сигнала
MAX     EQU 512
MIN     EQU 0
; Определение величины приращения опорного сигнала
DELTA   EQU 32
; Установить значения указателей r5,r6,r7 для доступа
; к выходным переменным Sig, Sig_SSAT, Sig_USAT
        LDR r5,=Sig
        LDR r6,=Sig_SSAT
        LDR r7,=Sig_USAT
```

```
; Объявить секцию данных в ОЗУ для размещения основного
; сигнала, знаково и без знаково ограниченного сигналов
        ALIGN
        AREA MyData, DATA, ReadWrite
Sig
        SPACE 4
Sig_SSAT
        SPACE 4
Sig_USAT
        SPACE 4
; Объявить все переменные глобальными для наблюдения
; в логическом анализаторе
        EXPORT Sig
        EXPORT Sig_SSAT
        EXPORT Sig_USAT
```

```
;*****************************************************************
; Подпрограмма выдачи текущего значения опорного сигнала без
; ограничений, сигнала со знаковым и без знаковым ограничением в
; ячейки "наблюдаемых" переменных в памяти
;*****************************************************************
Out_Signals
; Выдать текущее значение опорного сигнала без ограничения
        STR r0,[r5]
; Ограничить опорный сигнал знаково на уровне 8-и разрядов
; и сохранить в переменной Sig_SSAT
        SSAT r1, #8, r0
        STR r1,[r6]
; Ограничить опорный сигнал без знаково на уровне 8-и разрядов
; и сохранить в переменной Sig_USAT
        USAT r2, #8, r0
        STR r2,[r7]
; Возврат в основную программу
        BX lr
;*****************************************************************
```

Обратите внимание на вторую контрольную линию на графиках. Сигнал опорного генератора получил приращение +224+(+32)=+256. Знаковое ограничение дает результат +127, а беззнаковое – 255, что полностью соответствует алгоритму работы команд SSAT и USAT.

В начале программы задаются параметры генератора опорного сигнала, а также инициализируются указатели (r5, r6, r7) для доступа к глобальным переменным в памяти Sig, Sig_SSAT, Sig_USAT, имитирующим три возможных выхода генератора. Именно эти переменные используются в качестве «наблюдаемых» для вывода информации в окно логического анализатора.

Сами глобальные переменные, резервируются в секции данных (в ОЗУ) в конце основной программы. Напоминаем, что для представления в логическом анализаторе они должны быть 32-разрядными словами, объявленными общедоступными с помощью директивы Ассемблера EXPORT.

Операции модификации сигнала опорного генератора (ограничения) и сохранения результатов в соответствующих глобальных переменных программы будут использоваться как на нарастающем, так и на спадающем фронте генератора. Целесообразно оформить их в виде отдельной подпрограммы Out_Signals.

Значения указателей r5, r6 и r7 проинициализированы в начале основной программы и больше не меняются. Это позволяет использовать их для сохранения выходов генераторов в глобальных переменных. В заголовке подпрограммы можно было бы дополнительно отметить, что входными переменными являются содержимое регистра r0 – значение опорного сигнала, и содержимое регистров r5, r6 и r7 – указателей на место расположения в памяти выходов генератора.

Перейдем к генерации сигналов, пока без задания точного времени такта его работы, который в реальных системах управления определяется с помощью периферийных устройств: интервального таймера и контроллера прерываний. Инициализируется счетчик числа периодов и начинается генерация переднего фронта

сигнала с установки минимального значения сигнала MIN. Вызывается подпрограмма ограничения и выдачи трех сигналов Out_Signals. Текущее значение опорного сигнала в регистре r0 увеличивается на величину DELTA и сравнивается с максимальным MAX. Пока оно не превысит MAX – цикл ступенчатого увеличения опорного значения – продолжается. После выхода из цикла подтверждается начальное значение сигнала для спадающего фронта генератора и начинает формироваться спадающий фронт (по тем же принципам).

```
; Сгенерировать периодические сигналы на заданном
; числе периодов
; Установить счетчик числа периодов
        MOV r8,#N
Gener
; Нарастающий фронт опорного сигнала
; Установить начальное значение опорного сигнала
        MOV r0,#MIN
UP
; Подпрограмма выдачи текущих значений всех сигналов
; в том числе ограниченных
        BL Out_Signals
; Инкрементировать значение опорного сигнала
        ADD r0,#DELTA
; Сравнить с максимальным значением
        CMP r0,#MAX
; Если "Строго меньше" продолжить формирование нарастающего
; фронта опорного сигнала и ограниченных сигналов
        BLT UP
; Установить точное максимальное значение опорного сигнала
        MOV r0,#MAX
```

Когда полный период отработан, декрементируется счетчик числа периодов и, если он не равен нулю, выполняется повторная генерация очередного периода.

```
DN
; Выдать текущие значения всех сигналов в том числе
; ограниченных
        BL Out_Signals
; Декрементировать значение опорного сигнала
        SUB r0,#DELTA
; Сравнить с минимальным значением
        CMP r0,#MIN
; Если "Строго больше" продолжить формирование спадающего фр
; опорного сигнала и ограниченных сигналов
        BGT DN

; Очередной период отработан
; Декрементировать счетчик числа периодов
        SUBS r8,#1
; Если не все периоды сгенерированы – повторить
        BNE Gener
; Предотвратить выполнение несуществующего кода
        B .
```

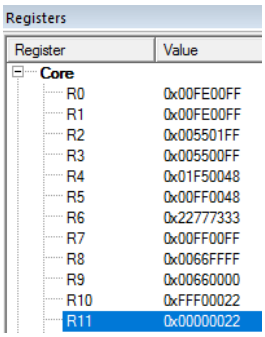
Последней командой «B.» (переход по текущему адресу) предотвращается выполнение несуществующего кода. В этом месте можете поставить точку останова для отладки программы.

Мы на простом примере показали принцип работы команд ограничения значения полного 32-разрядного слова. Работу команд ограничения полуслов и байт предлагаем Вам исследовать самостоятельно.

1) Выполните отладку программы при различных значениях параметров генератора. Помните, что не все значения непосредственных операндов могут кодироваться в 32-разрядном формате. Если Вы встретитесь с такой ситуацией, придется предварительно загрузить константу в один из свободных регистров.
2) Создайте программу для одновременного ограничения на уровне байта без знака U8 нескольких пар полуслов, которые Вы будете последовательно загружать в регистры ЦПУ, выполнять ограничение и сохранять результаты для анализа в регистрах с большими на единицу номерами. Проследите за выработкой флага насыщения S в окне регистров (в xPSR). Сформулируйте правило формирования флага S для этой команды.
3) Замените в предыдущем задании команду без знакового ограничения на команду знакового ограничения SSAT16. Выполните отладку. Объясните полученный результат.
4) Команду QADD можно использовать, например, для суммирования элементов массива слов с получением 32-разрядной суммы и последующего контроля ее достоверности (возможного выхода за диапазон чисел со знаком в формате S32). Преимущество такого решения – отсутствие необходимости контроля переполнений при операциях сложения и учета этих переполнений в старшем слове суммы. Попробуйте разработать соответствующую подпрограмму. Ей будут передаваться следующие параметры: начальный адрес массива слов S32 в памяти, длина массива,

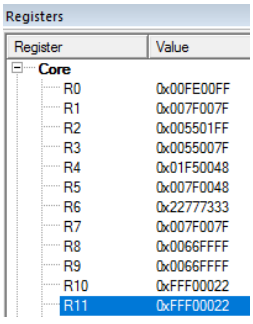
адрес сохранения суммы в памяти. Проверьте работу подпрограммы на нескольких наборах слов в памяти.

2) Используйте для загрузки младшего полуслова команду MOVW r0, #Const, для загрузки старшего полуслова – команду MOVT r0, #Const. Затем выполните параллельное ограничение полуслов с сохранением результата в очередном по номеру регистре ЦПУ, например: USAT16, r1,#8,r0. Для примера, ниже представлен результат работы программы MyProg_24_1.s. Результат в зависимости от исходного содержимого регистров: R0 (R1) – нет ограничения; R2 (R3) – ограничение сверху младшего полуслова; R4 (R5) – ограничение сверху старшего полуслова; R6 (R7) – ограничение сверху обоих слов; R8 (R9) – ограничение снизу младшего полуслова; R10 (R11) – ограничение снизу старшего полуслова.



Правило формирования флага S: «Исходные полуслова рассматриваются как числа со знаком S16 (в диапазоне -32768 ÷ +32767). Если ни одно из полуслов не ограничивается, то флаг S не формируется. Если хотя бы одно из двух слов ограничивается – возникает флаг насыщения. Если обе операции генерируют флаг S, то определить, какая именно операция вызвала насыщение – невозможно. Флаг насыщения формируется при ограничении чисел как сверху, так и снизу».

3) Исходные 16-разрядные числа со знаком S16 приводятся к формату 8-разрядных чисел со знаком S8. Результат ограничения, в зависимости от исходного содержимого регистров, следующий:



R0 (R1) – оба полуслова сверху; R2 (R3) – первое полуслово сверху; R4 (R5) – второе полуслово сверху; R6 (R7) – оба полуслова сверху; R8 (R9) – нет никаких ограничений; R10 (R11) – нет никаких ограничений (см. MyProg_24_2.s).

4) Подключите к проекту CPU_24 программу MyProg_24_3.s. В ней содержится пример реализации подпрограммы сложения заданного числа слов массива с одновременным насыщением. В конце подпрограммы контролируется возникновение флага насыщения. Если он есть, то – результат сложения обнуляется, а сам флаг сбрасывается. Убедитесь в правильности работы программы, например, задайте все числа массива равными 400 000 000 (не будет насыщения) и 500 000 000 (насыщение возникнет).

Команды насыщения дополняют набор команд цифровой обработки сигналов процессоров Cortex-M4, позволяя исключить переполнения при реализации цифровых регуляторов и фильтров при работе с числами в формате с фиксированной точкой.

18 КОМАНДЫ ПРЕДВАРИТЕЛЬНОЙ ОБРАБОТКИ ДАННЫХ

Оглавление

18.1 Назначение. Область применения

Название этой главы условно. Речь идет об обработке входных данных, полученных по каналам связи или из регистров периферийных устройств, которая должна быть выполнена до основного вычислительного процесса, или об обработке результатов основного процесса перед выводом данных. Конечно, эта обработка может быть выполнена с использованием основных логических и арифметических команд процессора, в том числе, с попутными операциями в кольцевом сдвиговом регистре ЦПУ. Тем не менее, разработчики системы команд ARM посчитали необходимым включить в нее ряд специальных операций, существенно облегчающих решение подобных задач. В этой главе мы рассмотрим некоторые из них – наиболее употребительные в системах цифрового управления оборудованием.

Речь идет, прежде всего, о командах *распаковки* данных, полученных по каналам связи от интеллектуальных датчиков или удаленных контроллеров сбора информации и управления оборудованием. Такие данные обычно упаковываются в последовательности байт или полуслов и должны быть распакованы перед обработкой. Напротив, после завершения обработки результаты, при необходимости их передачи, должны быть вновь *упакованы* в слова и переданы внешним устройствам по каналам связи.

Часто возникает ситуация, когда полезные данные находятся в определенных битовых полях (как, например, в регистрах АЦП) и возникает задача их извлечения перед обработкой. Для этой цели предусмотрены *команды работы с битовыми полями*.

Процессор работает, в основном, со словами. Для обработки байт или полуслов разрабатываются отдельные процедуры. Для того, чтобы ими можно было пользоваться для всех байт/полуслов в слове, предусматриваются команды *реверсирования данных в слове*. Они используются и тогда, когда по каналам связи получены данные с другим порядком размещения полуслов/байт/бит в слове.

18.2 Команды распаковки и упаковки данных

Ядро процессоров Cortex-M3/M4/M4F является 32-разрядным и только полные 32-разрядные слова могут обрабатываться в нем. Напротив, память является байтовой и в ней могут храниться данные разной длины: 8-разрядные (байты), 16-разрядные (полуслова) и 32-разрядные (слова). При больших объемах 8- или 16-разрядных данных нет необходимости хранить их в виде слов, для этого достаточно более короткого формата. Такой подход позволяет существенно экономить память микропроцессорной системы, однако требует особой технологии обработки данных внутри процессора – специальных *операций распаковки данных*. Когда данные распакованы, выполняется их обработка, а затем – сохранение в памяти результатов или отдельными байтами/полусловами, или полными словами. В последнем случае потребуется дополнительная операция, но уже по *упаковке данных* в слова.

Разумеется, Вы можете воспользоваться уже знакомыми нам командами загрузки регистров LDR из памяти байтами или полусловами *с автоматическим расширением формата* байта или полуслова до формата полного слова с заполнением старших разрядов слова нулями или знаковым разрядом числа. При этом к основному мнемокоду команды LDR добавляются суффиксы, специфицирующие тип выполняемой операции (см. 10.7.2):

Таблица 18.1 Синтаксис команд загрузки регистров

Команда	Операция
`LDRB Rt, [Rn{,#offset}]`	**Загрузка байта без знака (B) с расширением нулем**
`LDRSB Rt, [Rn{,#offset}]`	**Загрузка байта со знаком (SB) с расширением знаковым разрядом**
`LDRH Rt, [Rn{,#offset}]`	**Загрузка полуслова без знака (H) с расширением нулем**
`LDRSH Rt, [Rn{,#offset}]`	**Загрузка полуслова со знаком (SH) с расширением знаковым разрядом**

Преимущество такого подхода – возможность использования всего арсенала методов косвенной адресации памяти. Как мы уже отмечали, хотя данные и считываются из памяти словами, на стадии загрузки в регистры ЦПУ ненужная информация в старших разрядах автоматически отсекается.

Второй вариант решения: загрузить в регистр ЦПУ данные из памяти «как есть» – в виде 32-разрядного слова и затем выполнить распаковку этого слова по байтам или полусловам. Рассмотрим этот вариант подробнее.

18.2.1 Синтаксис команд распаковки данных

Имеется значительная группа команд распаковки данных с унифицированным синтаксисом:

```
Op {Rd}, {Rn}, Rm {, ROR #n}
```

Таблица 18.2 Синтаксис команд распаковки данных

Op	Оптокод операции
Rm	**Регистр ЦПУ, содержащий «упакованные» в 32-разрядное слово полуслова или байты, подлежащие распаковке.**
ROR #n	**Опция предварительной обработки содержимого регистра источника Rm в кольцевом сдвиговом регистре – сдвиг на 0, 8, 16, или 24 разряда с помощью операции «Циклического сдвига вправо» ROR.**
Rn	**Опция регистра-дополнительного слагаемого (аккумулятора), к которому должно быть прибавлено распакованное слово. При этом в мнемонику команды обязательно добавляется суффикс A (от ADD – сложить).**
Rd	**Опция регистра-приемника результата операции**

Как и для всех команд, если регистр-приемник Rd не указан, его функцию выполняет регистр-источник данных Rm. Если требуется только распаковка, без дополнительной операции сложения с содержимым регистра Rn, то имя регистра-слагаемого опускается. Если попутный циклический сдвиг вправо не требуется, то, опускается и опция циклического сдвига ROR #n. Будьте внимательны! Если имя регистра-приемника опущено, то распаковка выполняется в тот же самый регистр-источник Rm и его прежнее содержимое («упакованное» 32-разрядное слово) будет потеряно.

В основе символики этих команд лежит операция расширения формата (e**XT**end) байта или полуслова до формата полного 32-разрядного слова. При этом префиксы **S** и **U** используются для спецификации знакового или беззнакового расширения, а суффиксы **B** и **H** – для спецификации источника данных, байта или полуслова. Суффикс **16** означает, что выполняется расширение до полуслова одновременно обоих младших байт в младшем и старшем полуслове регистра-источника Rm:

Значение символов в мнемонике команд распаковки байта или полуслова из полного слова и расширения формата до слова, возможно, с опциональным сложением				
S/U	**XT**	**A**	**B/H**	**16**
				Расширить оба младших байта из полуслов регистра-источника Rm до полуслова
			B - Byte - Расширить байт **H - Halfword - Расширить полуслово**	
		A - После расширения выполнить операцию сложения (Add) полученного слова с содержимым дополнительного операнда-слагаемого Rn. Опция может отсутствовать		
	XT - eXTend - Расширить формат байта или полуслова (корень оптокода)			
S –Signed Extend – Расширение формата числа со знаком в дополнительном коде **U – Unsigned Extend – Расширение формата числа без знака**				

Графическая иллюстрация алгоритма выполнения команд распаковки байта со знаком SXTB и байта без знака UXTB представлена на рис. 18.1. На этапе 1 за счет циклического сдвига вправо на число разрядов 0, 8, 16 или 24 на месте младшего байта может оказаться байт 0, 1, 2 или 3 исходного слова. На втором этапе младший байт расширяется до слова с учетом знака или как беззнаковый. Полученное слово сохраняется в регистре назначения Rd.

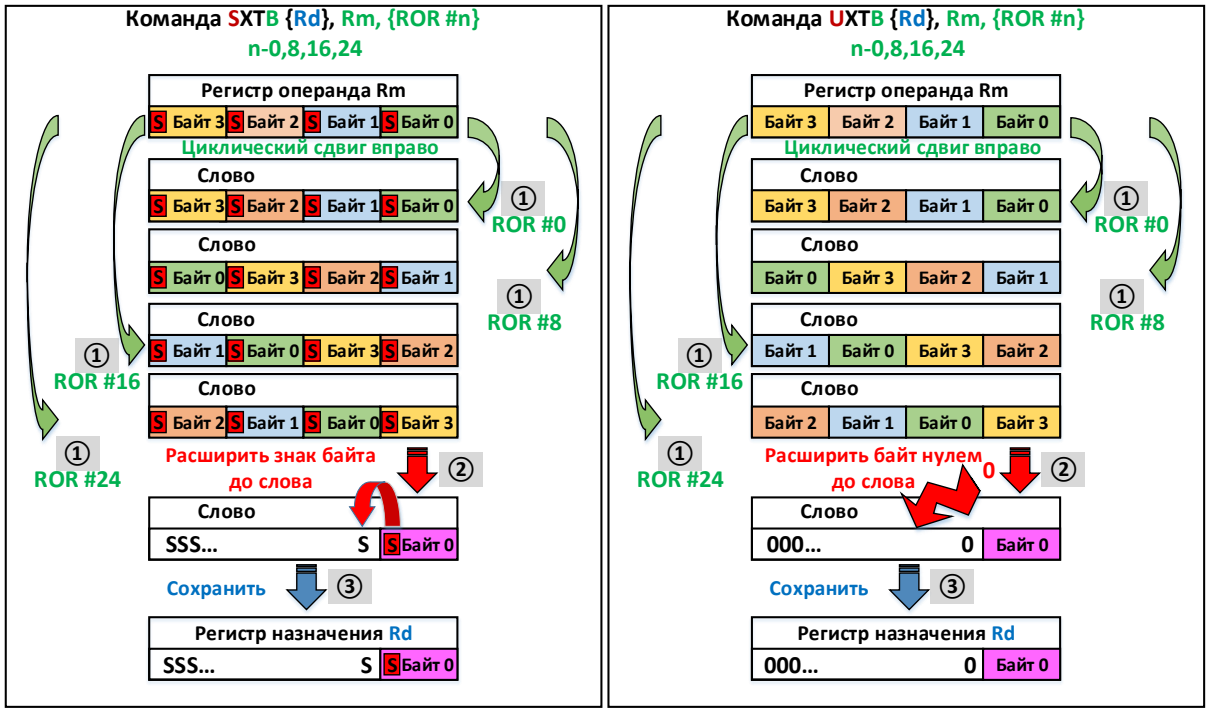


Рис. 18.1 Алгоритм выполнения команд распаковки байт SXTB и UXTB

Данные должны распаковываться последовательно (байт за байтом или полуслово за полусловом). При этом источник «упакованных» данных не должен меняться, а регистр-приемник, напротив, каждый раз должен быть другим.

Операция распаковки с дополнительной опцией сложения может использоваться как операция накопления распакованных данных в регистре-аккумуляторе. При этом имена регистра-слагаемого Rn и регистра-приемника Rd должны быть одинаковы (это – имя регистра-аккумулятора).

18.2.2 Примеры использования команд распаковки слов

В проекте CPU_25 (программа MyProg_25.s) рассматриваются два возможных варианта решения задачи предварительной обработки массива байт без знака в памяти для преобразования данных к формату U32 – слова без знака. Предполагается, что массив данных находится в ОЗУ и перед началом отладки программы может инициализироваться (в окне дампа памяти) любыми исходными данными формата U8.

```
63  ; Объявить секцию данных в ОЗУ, подлежащих распаковке
64            ALIGN
65            AREA MyData, DATA, ReadWrite
66  ; Массив байтовых данных в памяти
67  Array_8   SPACE N
```

```
10  ; Основная программа
11  ; Число байт в массиве
12  N         EQU 4
13  ; Загрузить указатель r0 начальным адресом массива
14  ; байт в памяти данных
15            LDR r0,=Array_8
16  ; Вызвать первую подпрограмму
17            BL Un_Pack_1
18  ; Перезагрузить указатель
19            LDR r0,=Array_8
20  ; Вызвать вторую подпрограмму
21            BL Un_Pack_2
22  ; Ограничить выполнение несуществующего кода
23            B .
```

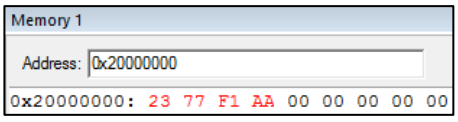
В основной программе в указатель r0 загружается адрес начала массива и последовательно вызываются две подпрограммы распаковки байт без знака в регистры r5, r6, r7, r8 и r9, r10, r11, r12, соответственно.

Первая подпрограмма использует стандартные команды загрузки регистров из памяти байтами без знака с авто-расширением нулями в сторону старших разрядов. Вторая – вначале копирует все 4 байта в регистр r1 целиком, а затем выполняет распаковку слова в регистре r1 побайтно, считая все байты целыми числами без знака, также выполняя авто-расширение нулями в область старших разрядов.

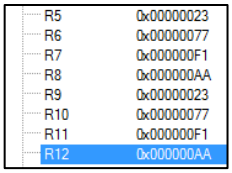
```
25  ;*****************************************************
26  ; Подпрограмма распаковки 4-х байт без знака из памяти
27  ; с использованием команды загрузки байта как числа
28  ; без знака и без знаковым расширением
29  ; Входы: r0 - указатель начала массива байт в памяти
30  ; Выходы: r5,r6,r7,r8
31  ;*****************************************************
32  Un_Pack_1
33  ; Загрузить три байта с пост-авто инкрементированием
34  ; указателя и без знаковым расширением
35            LDRB r5,[r0],#1
36            LDRB r6,[r0],#1
37            LDRB r7,[r0],#1
38  ; Загрузить последний байт
39            LDRB r8,[r0]
40  ; Возврат в основную программу
41            BX lr
42  ;*****************************************************
```

```
44  ;*****************************************************
45  ; Подпрограмма распаковки 4-х байт без знака из памяти
46  ; с использованием команды распаковки слова в регистре
47  ; Входы: r0 - указатель начала массива байт в памяти
48  ; Используемые регистры: r1 - временного хранения
49  ; Выходы: r9,r10,r11,r12
50  ;*****************************************************
51  Un_Pack_2
52  ; Загрузить полное слово в регистр временного хранения
53            LDR r1,[r0]
54  ; Распаковать последовательно в регистры r9,r10,r11,r12
55            UXTB r9,r1
56            UXTB r10,r1,ROR #8
57            UXTB r11,r1,ROR #16
58            UXTB r12,r1,ROR #24
59  ; Возврат в основную программу
60            BX lr
61  ;*****************************************************
```

Приведем содержимое памяти перед выполнением программы результаты работы обеих подпрограмм:



Видно, что содержимое четырех регистров r5, r6, r7, r8 полностью совпадает с содержимым регистров r9, r10, r11, r12.



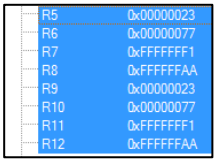
Обе процедуры работают корректно и какое решение предпочесть – выбирает программист. Может оказаться более удобным сначала загрузить в один из регистров «упакованные» данные, а затем, по мере необходимости, извлекать из него нужные данные на обработку.

1) Модифицируйте обе подпрограммы для работы с байтами со знаком. Проверьте работу на том же исходном наборе данных в памяти.
2) Модифицируйте обе подпрограммы для работы с полусловами без знака. Проверьте на том же наборе данных в памяти.
3) Еще раз внесите изменения для работы с полусловами со знаком.

Убедитесь в правильности работы обеих процедур.

4) Разработайте два варианта подпрограммы расчета контрольной суммы массива байт без знака, число которых кратно 4-м (размеру полного слова): первый – с использованием команды загрузки LDRB; второй – с использованим команды распаковки слова с одновременным сложением UXTAB. Оцените эффективность обоих решений по быстродействию.

1) Используйте в первой подпрограмме команду загрузки регистров байтами со знаком LDRSB, а во втрой подпрограмме – команду распаковки байт со знаком SXTB, предварительно загруженных в регистр r1 (см. MyProg_25_1.s). Результаты отладки на том же наборе данных подтверждают, что знаковые числа расширяются старшим знаковым разрядом исходного числа-байта.



2) В первой подпрограмме используйте команду загрузки регистра полусловом без знака LDRH, а во второй – команду распаковки полуслов без знака UXTH, загруженных в регистр r1 (см. MyProg_25_2.s). Теперь число выходных регистров сокращается в два раза (для первой подрограммы - r5, r6, для второй – r7, r8). Результаты отладки на том же наборе данных подтверждают правильность обоих решений.

3) В первой подпрограмме используйте команду LDRSH, а во второй – команду SXTH (см. MyProg_25_3.s). Результаты отладки подтверждают работоспособность обоих решений.

4) Варианты обеих подпрограмм представлены ниже (см. MyProg_25_4.s):

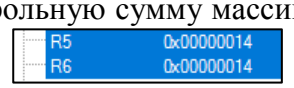
```
36  ;*********************************************************
37  ; Подпрограмма расчета контрольной суммы массива байт
38  ; без знака с использованием команды LDRB загрузки
39  ; байта с одновременным без знаковым расширением
40  ; и команды обычного сложения (накопления)
41  ; Входы: r0 - указатель начала массива байт в памяти
42  ;         r1 - число байт в массиве
43  ; Используемые регистры: r2 - временного хранения
44  ; Выход: r5 - контрольная сумма массива байт без знака
45  ;*********************************************************
46  SUMMA_1
47  ; Очистить регистр контрольной суммы
48            MOV r5,#0
49  Loop_1
50  ; Загрузить очередной байт из памяти в регистр r2
51  ; с без знаковым расширением и авто-смещением
52  ; указателя в регистре r0 на 1
53            LDRB r2,[r0],#1
54  ; Добавить к текущей контрольной сумме
55            ADD r5,r2,r5
56  ; Декрементировать счетчик числа считанных байт
57            SUBS r1,#1
58  ; Если не все байты в массиве обработаны, повторить
59            BNE Loop_1
60  ; Возврат в основную программу
61            BX lr
62  ;*********************************************************
```

```
64  ;*********************************************************
65  ; Подпрограмма расчета контрольной суммы массива байт
66  ; без знака со считыванием полного 32-разрядного слова
67  ; и распаковкой байт с одновременным расширением до слова
68  ; и сложением с содержимым регистра-аккумулятора UXTAB
69  ; Входы: r0 - указатель начала массива слов в памяти
70  ;         r1 - число слов в массиве байт
71  ; Используемые регистры: r2 - временного хранения
72  ; Выходы: r6 - контрольная сумма массива байт без знака
73  ;*********************************************************
74  SUMMA_2
75  ; Очистить регистр контрольной суммы
76            MOV r6,#0
77  Loop_2
78  ; Загрузить очередное слово из памяти в регистр r2
79  ; ("упакованное" 4-мя байтами)
80  ; с авто-смещением указателя в регистре r0 на 4
81            LDR r2,[r0],#4
82  ; Последовательно распаковать все 4-е байта с добавлением
83  ; результата распаковки к содержимому аккумулятора r6
84            UXTAB r6,r6,r2,ROR #0
85            UXTAB r6,r6,r2,ROR #8
86            UXTAB r6,r6,r2,ROR #16
87            UXTAB r6,r6,r2,ROR #24
88  ; Декрементировать счетчик числа считанных полных слов
89            SUBS r1,#1
90  ; Если не все слова в массиве байт обработаны, повторить
91            BNE Loop_2
92  ; Возврат в основную программу
93            BX lr
94  ;*********************************************************
```

Первая подпрограмма на первый взгляд кажется более простой и короткой. Она, действительно, более простая, но при обработке больших массивов данных менее производительная. Сравните: в первом случае тело цикла из двух команд (строки 53, 55) выполняется $2*N_{байт}=2*4*N_{слов}=8*N_{слов}$ тактов; во втором случае тело цикла из 5-и команд (строки 81, 84, 85, 86, 87) выполняется $(1+4)*N_{слов}=5*N_{слов}$ тактов. Так что более длинная программа оказывается более производительной.

Следовательно, нужно хорошо предствлять возможности всего набора команд процессора, чтобы для каждого конкретного случая оптимизировать решение либо по

объему требуемой кодовой памяти, либо по быстродействию. Разумеется, обе подпрограммы позволяют точно рассчитать контрольную сумму массива. Так, для набора из 8-и байт 2, 2, 2, 2, 3, 3, 3, 3 она равна 20d=0x14.



Применение команд распаковки байт или полуслов с одновременным сложением расширенного до формата слова результата с текущим содержимым регистра-аккумулятора, который одновременно может выполнять и функцию регистра назначения, эквивалентно операции *накопления* (рис. 18.2). Однако имейте в виду, что все команды распаковки не меняют состояния флагов процессора, следовательно, контроль переполнений при дополнительном сложении отсутствует.

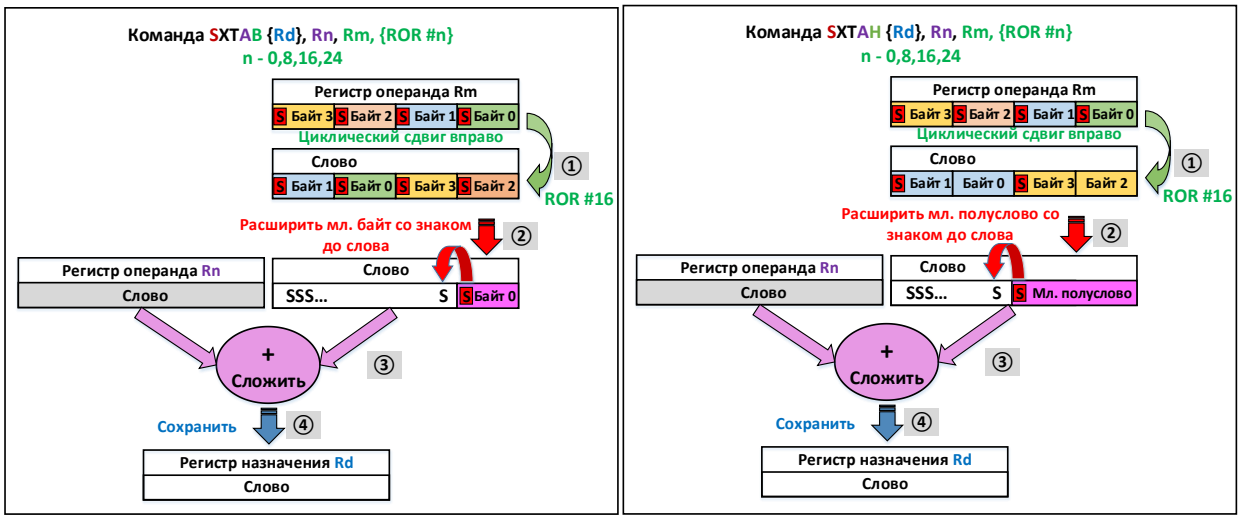


Рис. 18.2 Алгоритм выполнения команд SXTAB и SXTAH

18.2.3 Команды упаковки полуслов в слова

Процессор поддерживает операции сохранения в памяти полных слов, полуслов и байт. Для упаковки байт в слово специальные команды отсутствуют, но можно воспользоваться для этой цели обычными логическими и арифметическими командами, а также командами вставки в слово битовых полей, начиная с заданного разряда (рассмотрим в этой главе).

Если Вы выполняли обработку 16-разрядных данных и уверены, что в процессе обработки не возникли переполнения, то можете воспользоваться командами *упаковки полуслов* в полные 32-разрядные слова в нужном для Вас порядке и затем сохранить эти слова целиком в памяти. Преимущество таких команд: они допускают попутный логический или арифметический сдвиг одного из полуслов на заданное число разрядов. Первое полуслово загружается без каких-либо изменений, а второе – после предварительной операции сдвига на указанное число разрядов. У программиста появляется возможность комбинировать битовые поля исходных полуслов для получения нужного ему формата полного слова.

Имеются две команды упаковки полуслов **PKHBT** и **PKHTB**. Первые два символа в коде операции **PK** (от англ. Pack) означают – Упаковать. Суффикс **H** – упаковать полуслова (Halfword). Следующие два суффикса «**BT**» или «**TB**» определяют порядок извлечения полуслов из первого и второго операндов-источников (регистров Rn и Rm соответственно).

Если из первого регистра-источника Rn должно быть извлечено младшее полуслово, то первым в команде указывается суффикс B (Bottom), если старшее, то суффикс T (Top). Первое извлекаемое из регистра Rn полуслово всегда размещается в регистре-приемнике без какого-либо сдвига (B – в младшем полуслове, T – в старшем).

Второе полуслово извлекается из второго регистра-источника, причем *после* опционально указанного сдвига всех бит в этом слове на заданное число разрядов. Синтаксис команд упаковки полуслов.

```
PKHBT{cond}{Rd},Rn,Rm{,LSL #n}
PKHTB{cond}{Rd},Rn,Rm{,ASR #n}
```

LSL #n – Логический сдвиг второго операнда влево на n р-ов
ASR #n – Арифм-ский сдвиг второго операнда вправо на n р-ов

Rm – второй регистр-источник

Rn – первый регистр-источник

Rd – регистр-приемник

cond – Код условного выполнения команды

BT – Bottom-Top – Сначала младшее полуслово из первого регистра-источника Rn, затем старшее полуслово из второго регистра-источника Rm после опционального логического сдвига влево LSL на заданное число разрядов #n.

TB – Top-Bottom – Сначала старшее полуслово из первого регистра-источника Rn, затем младшее полуслово из второго регистра источника Rm после опционального арифметического сдвига вправо ASR на заданное число разрядов #n.

H – Halfword – Полуслово

PK – Pack – Упаковать два полуслова в слово

Если из второго регистра-источника извлекается старшее полуслово, то после предварительного логического сдвига влево на нужное число разрядов, если младшее полуслово, то после предварительного арифметического сдвига вправо на нужное число разрядов – рис. 18.3.

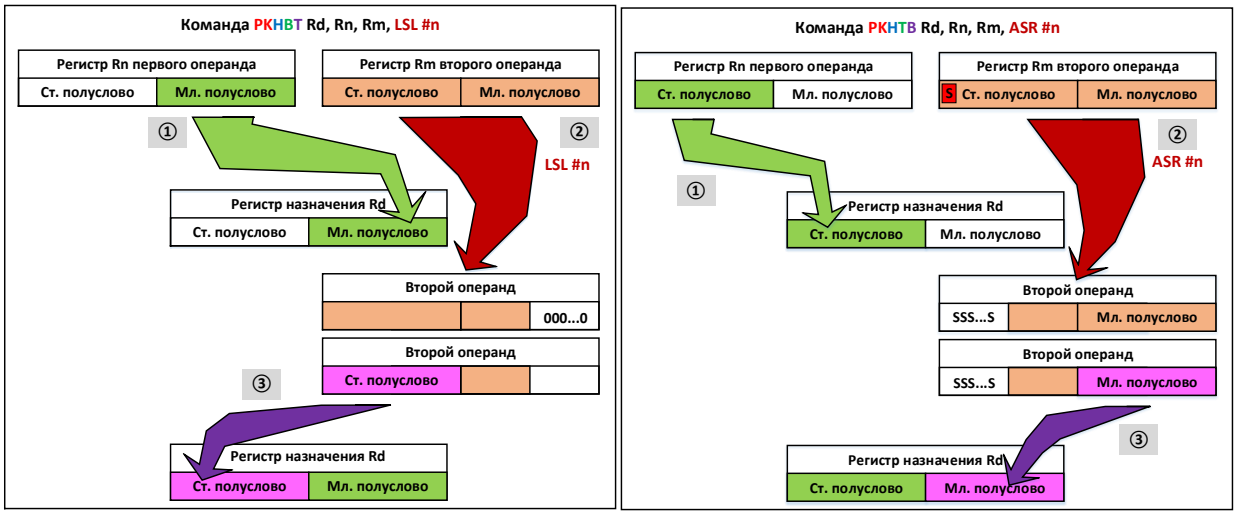


Рис. 18.3 Алгоритм выполнения команд PKHBT и PKHTB

Если опция сдвига опущена, то полуслова извлекаются в указанном в коде операции порядке и загружаются в регистр назначения без изменений.

1) Какие биты «вдвигаются» справа при выполнении логического сдвига влево?

2) В чем особенность операции арифметического сдвига вправо?

3) Выберите команды, которые позволят упаковать исходные полуслова в полные слова, в соответствии с алгоритмами, представленными на рис. 18.4. Не забудьте указать тип и величину сдвига содержимого второго регистра-источника.

4) Разработайте программу, позволяющую проверить правильность сделанного Вами выбора.

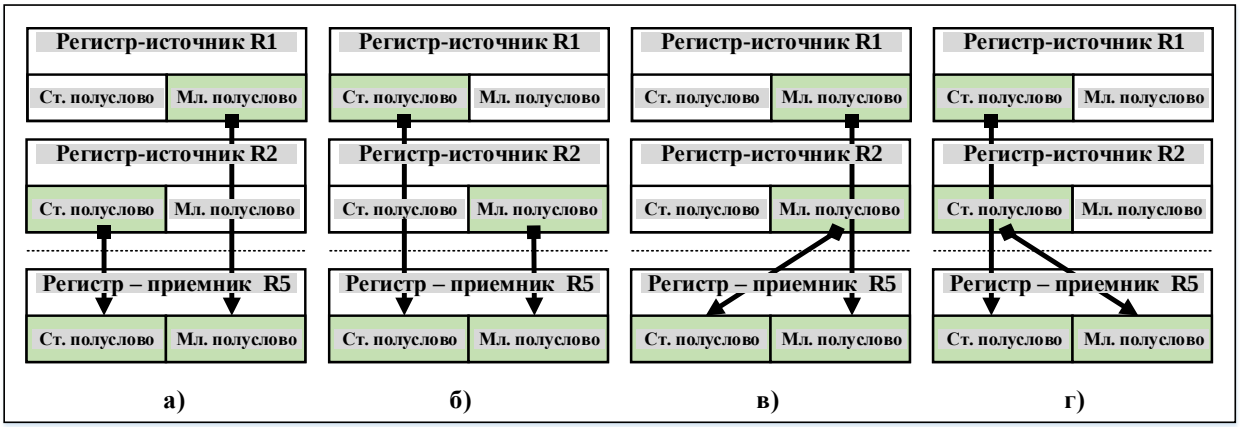


Рис. 18.4 Упакуйте полуслова в соответствии с представленным алгоритмом

1) Нулевые.

2) Старший разряд исходного слова рассматривается как знаковый разряд и автоматически распространяется вправо при сдвиге. Это позволяет сохранить знак исходного числа неизменным.

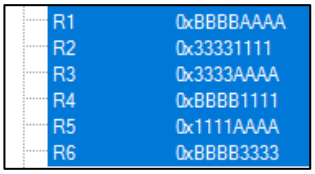
3) Подойдут следующие команды:

а)	**PKHBT r5, r1, r2**
б)	**PKHTB r5, r1, r2**
в)	**PKHBT r5, r1, r2, LSL #16**
г)	**PKHTB r5, r1, r2, ASR #16**

4) Вариант программной реализации задания см. в проекте CPU_25 (MyProg_25_5.s).

```
11  Pack
12  ; Выполнить загрузку двух регистров ЦПУ r1,r2
13  ; полусловами, - имитация выполненных расчетов
14  ; над 16-разрядными данными
15          MOVW r1,#0xAAAA
16          MOVT r1,#0xBBBB
17          MOVW r2,#0x1111
18          MOVT r2,#0x3333
19  ; Загрузить в регистр r3 младшее полуслово
20  ; из регистра r1 и старшее из регистра r2
21          PKHBT r3,r1,r2
22  ; Загрузить в регистр r4 старшее полуслово
23  ; из регистра r1 и младшее из регистра r2
24          PKHTB r4, r1,r2
25  ; Загрузить в регистр r5 младшее полуслово
26  ; из регистра r1 и старшее из регистра r2
27  ; после логического сдвига на 16 разрядов влево
28          PKHBT r5,r1,r2, LSL #16
29  ; Загрузить в регистр r6 старшее полуслово
30  ; из регистра r1 и младшее из регистра r2
31  ; после арифметического сдвига на 16 разрядов вправо
32          PKHTB r6, r1,r2, ASR #16
```

Выполняется инициализация регистров r1, r2 полусловами, а затем – 4 операции упаковки полуслов в слова. Результат упаковки полуслов полностью соответствует ожидаемому:



18.3 Команды работы с битовыми полями

18.3.1 Назначение команд работы с битовыми полями

В состав любого микроконтроллера входит большое число как *системных регистров*, управляющих режимами работы процессорного ядра, так и *регистров периферийных устройств*, управляющих работой периферии. Регистры всех встроенных устройств микроконтроллера отображены на память, то есть для доступа к ним пригоден весь арсенал средств процессора, который обычно используется для доступа к ячейкам памяти. Большинство системных и периферийных регистров являются 32-разрядными.

Модернизация содержимого любого ***регистра встроенного устройства*** обычно выполняется (за исключением начальной инициализации) в режиме **«Чтение-Модификация-Запись»**:

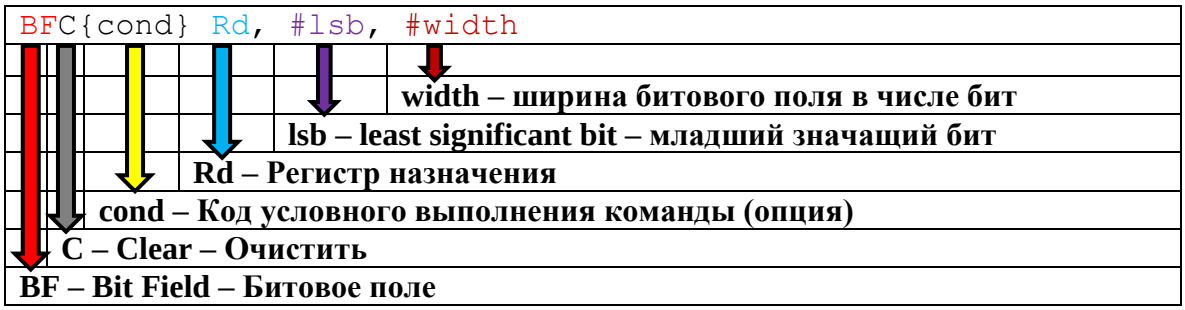
1) Сначала исходное содержимое регистра встроенного устройства считывается в один из регистров общего назначения процессора;
2) Затем выполняется модификация тех бит или битовых полей регистра ЦПУ, которые необходимы для перенастройки режима работы периферии. Все остальные битовые поля должны остаться прежними, во избежание случайного изменения состояния других битовых полей, отвечающих за другие функции устройства (не изменяемые в данный момент);
3) Наконец, новое содержимое регистра ЦПУ сохраняется по адресу регистра встроенного устройства.

На первом этапе используются обычные команды чтения содержимого памяти в регистр процессора, на третьем – обычные команды сохранения содержимого регистра процессора в памяти. Специальные команды предусмотрены для реализации этапа 2 – обработки (модификации) битовых полей регистров общего назначения:

1) Прежде всего – это полный набор логических команд, которые выполняют логические операции над данными в регистре процессора и непосредственными данными (маской): И, ИЛИ, Исключающее ИЛИ, и т.д. Маска может быть не только непосредственным операндом, но и содержимым одного из регистров процессора (при многократном использовании). Более того, в этом случае она может предварительно обрабатываться в кольцевом сдвиговом регистре процессора – сдвигаться на заданное число разрядов. Эти возможности позволяют сбросить в 0 или установить в 1 любой нужный бит или любое битовое поле регистра общего назначения.
2) Для работы с битовыми полями в систему команд включены и несколько специальных команд, существенно упрощающих и ускоряющих эту работу.

18.3.2 Очистка битового поля заданной длины

Наиболее часто требуется очистить один или несколько битов регистра ЦПУ или регистра, встроенного в микроконтроллер устройства. Если данные уже находятся в регистре ЦПУ, воспользуйтесь командой BFC:



Она выполняет очистку (обнуление) битового поля длиной width бит (целое число от 1 до 32), начиная с младшего значащего бита, номер которого lsb (целое число от 0 до 31). Регистром назначения Rd может быть любой из регистров центрального процессора. Графическая иллюстрация алгоритма выполнения команды показана на рис. 18.5.

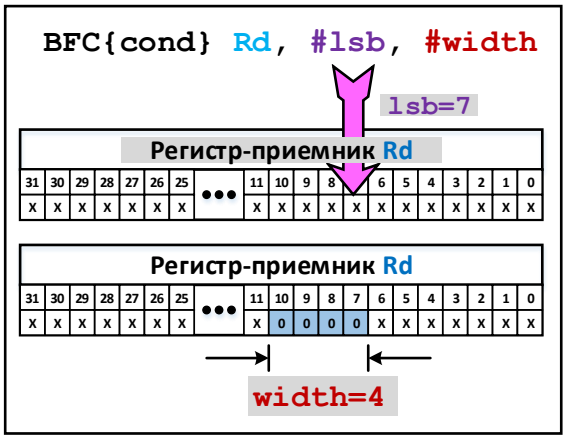
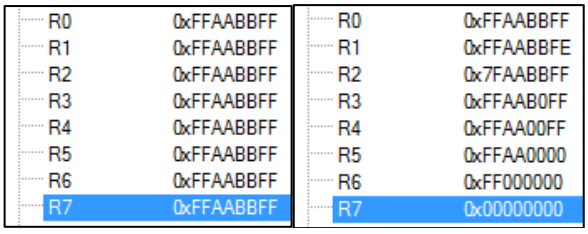


Рис. 18.5 Алгоритм выполнения команды BFC – очистка битового поля заданной длины

Пример использования команды очистки битовых полей находится в файле MyPorg_25_6.s. Одна и та же константа загружается в регистры r0-r7 ЦПУ. С помощью команды BFC выполняется очистка младшего бита регистра r1, старшего бита регистра r2, младшего нибла второго байта регистра r3, второго байта регистра r4, младшего полуслова регистра r5, трех младших байт регистра r6 и всего регистра r7 целиком.

```
12  Clear_Bits
13  ; Выполнить загрузку регистра ЦПУ r0
14  ; исходными данными
15          LDR r0,=0xFFAABBFF
16  ; Скопировать в регистры r1-r7
17          MOV r1,r0
18          MOV r2,r0
19          MOV r3,r0
20          MOV r4,r0
21          MOV r5,r0
22          MOV r6,r0
23          MOV r7,r0
```

```
24  ; Очистить нулевой бит регистра r1
25          BFC r1,#0,#1
26  ; Очистить старший бит регистра r2
27          BFC r2,#31,#1
28  ; Очистить младший нибл второго байта r3
29          BFC r3,#8,#4
30  ; Очистить второй байт регистра r4
31          BFC r4,#8,#8
32  ; Очистить младшее полуслово регистра r5
33          BFC r5,#0,#16
34  ; Очистить младшие три байта регистра r6
35          BFC r6,#0,#24
36  ; Очистить все биты слова в регистре r7
37          BFC r7,#0,#32
```

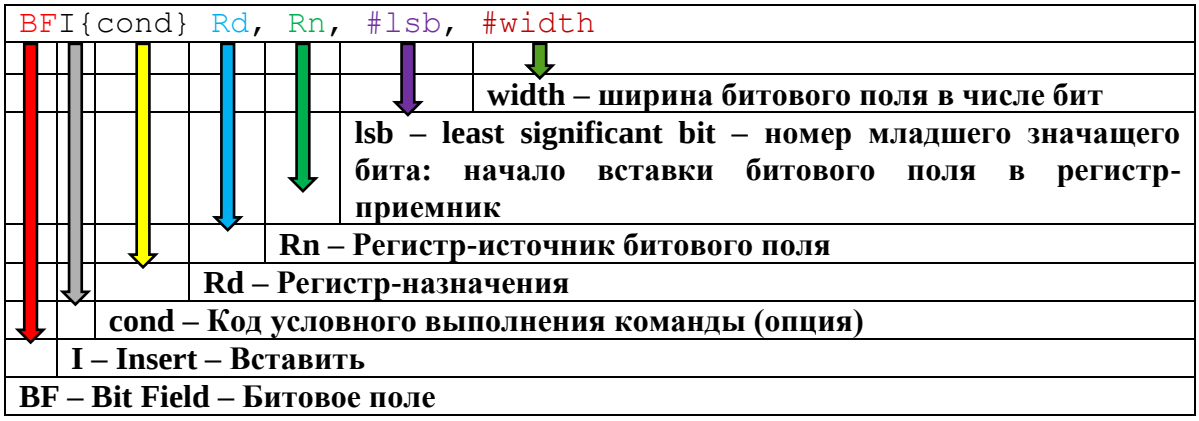


Убедитесь сами, что результаты отладки программы, представленные ниже, соответствуют истине:

Вместо логической операции очистки битов BIC, которая фактически является операцией «Логического И-НЕ» с маской, возможно попутно сдвинутой на заданное число разрядов, используйте для этой цели команду очистки битового поля BFC. Она проще и удобнее.

18.3.3 Команда вставки битового поля

Задача вставки битовых полей из одного регистра ЦПУ в другой часто возникает в процессе преобразования данных, например, при упаковке данных в слова перед выводом во внешние периферийные устройства, а также по каналам связи. Имеется специальная команда, которая позволяет скопировать битовое поле из младшей части регистра-источника в любую область регистра-приемника. Фактически эта команда позволяет сформировать любую битовую последовательность из битовых полей, уже имеющихся в регистрах ЦПУ:



Команда извлекает из младшей части регистра-источника Rn, начиная с нулевого бита, битовое поле заданной ширины width, специфицированное параметром «ширина битового поля в числе бит» и загружает извлеченную битовую последовательность в регистр-приемник Rd, начиная с указанного номера младшего значащего бита lsb – рис. 18.6. Все остальные биты регистра-приемника остаются в прежнем состоянии. Так как данные извлекаются исключительно из младшей части регистра-источника, может потребоваться предварительная операция сдвига данных вправо, чтобы нужное битовое поле оказалось в младшей части регистра (это может быть циклический сдвиг вправо ROR на требуемое число разрядов).

Загрузка в регистр-приемник Rd определенной последовательности битовых полей соответствует операции *упаковки битовых полей в слово*. При этом исходные битовые поля могут быть результатами предыдущей обработки данных, сдвинутыми вправо в область младших разрядов или предварительно загруженными в регистры непосредственными константами. Порядок загрузки битовых полей в регистр-приемник теоретически может быть любым, главное – чтобы они не накладывались друг на друга. Лучше выполнять загрузку последовательно, начиная с младших разрядов.

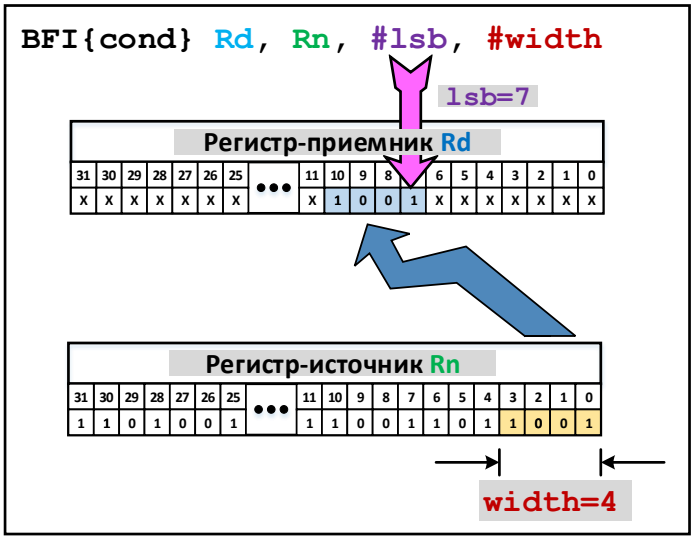


Рис. 18.6 Алгоритм выполнения команды вставки битового поля BFI

Пример упаковки битовых полей в 32-разрядное слово Вы найдете в файле MyProg_25_7.s. В его начале имитируется вычислительный процесс, результатом которого является загрузка выходных данных в регистры r0-r3 ЦПУ. Нужные для упаковки битовые поля находятся в младшей части этих регистров. Командой BFI они последовательно извлекаются и вставляются в поля регистра назначения r4, начиная с младшего бита. Итог упаковки:

```
12  Pack_Bit_Field
13  ; Выполнить инициализацию нескольких регистров
14  ; ЦПУ непосредственными константами
15  ; (имитация выполненнных расчетов)
16          MOVW r0,#0xFF22
17          MOVW r1,#0x7755
18          MOVW r2,#0x55DD
19          MOVW r3,#0x5F7A
20  ; Упаковать данные в слово в регистре r4 в
21  ; следующем порядке:
22  ; bit[0:3]r0; bit[0:8]r1; bit[0:4]r2; bit[0:15]r3;
23          BFI r4,r0,#0,#4
24          BFI r4,r1,#4,#8
25          BFI r4,r2,#12,#4
26          BFI r4,r3,#16,#16
```

R0	0x0000FF22
R1	0x00007755
R2	0x000055DD
R3	0x00005F7A
R4	0x5F7AD552

1) Проследите за выполнением программы в пошаговом режиме. Обратите внимание на то, как последовательно битовые поля исходных регистров модифицируют содержимое регистра назначения R4.
2) Требуется модифицировать содержимое регистров назначения r1, r2, r3, начиная с 4-го разряда, записав в него двоичный код 2_1001 (0x09). Все остальные биты регистров должны остаться неизменными. Решить задачу тремя способами: с использованием логических команд процессора и масок, сдвинутых влево на стадии ассемблирования программы; с использованием второго универсального операнда с попутным сдвигом его содержимого влево на нужное число разрядов; с использованием команды вставки битового поля в регистр-приемник, начиная с указанного разряда.
3) Выполните отладку программы, созданной в соответствии с предыдущим заданием.
4) Выберите оптимальный, на Ваш взгляд, вариант решения. Обоснуйте свой выбор.

2) Варианты решения задачи приведены в программе MyProg_25_8.s. Устанавливается одинаковое исходное содержимое всех 4-х регистров ЦПУ r0, r1, r2, r3 для удобства контроля результатов преобразования данных в регистровом окне среды µVision при отладке программы.

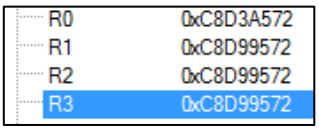
В первом варианте решения используются непосредственные константы-маски, которые автоматически сдвигаются влево на 12 разрядов с помощью логической операции Ассемблера «:SHL:» (выполняется на стадии ассемблирования программы). Первая команда BIC вначале очищает нужное нам битовое поле от предыдущих данных, а вторая команда ORR – записывает в него требуемую битовую последовательность.

```
12  Modify_Bit_Field
13  ; Установить начальное содержимое регистра r0
14          MOVW r0,#0xA572
15          MOVT r0,#0xC8D3
16  ; Скопировать в r1,r2,r3
17          MOV r1,r0
18          MOV r2,r0
19          MOV r3,r0
20  ; Модификация bit[19:12] регистров r1,r2,r3 значением 0x99
21  ; Вариант №1: с использованием логических команд процессора
22  ;             и масок, сдвинутых на нужное число разрядов
23  ;             с помощью логических операций Ассемблера
24          BIC r1,#0xFF:SHL:12
25          ORR r1,#0x99:SHL:12
26  ; Вариант №2: с использованием второго универсального операнда
27  ;             и попутным сдвигом масок в кольцевом
28  ;             сдвиговом регистре процессора
29          MOV r4,#0xFF
30          BIC r2,r4,LSL #12
31          MOV r4,#0x99
32          ORR r2,r4,LSL #12
33  ; Вариант №3: с использованием команды вставки битового поля
34          MOV r4,#0x99
35          BFI r3,r4,#12,#8
```

Второй вариант предполагает использование универсального операнда-маски в одном из регистров процессора (в данном случае в r4), с попутным сдвигом маски в кольцевом сдвиговом регистре, но уже на стадии выполнения программы. Естественно, что в этом случае потребуется предварительная загрузка нужной маски в регистр r4.

В третьем варианте можно ограничиться только загрузкой требуемой битовой последовательности в один из свободных регистров ЦПУ и последующей вставкой этой последовательности в регистр назначения с заданного разряда.

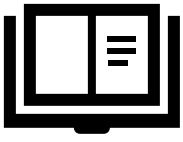
3) Мы специально оставляем содержимое регистра r0 неизменным, чтобы можно было видеть, как модифицируются значения в регистрах r1, r2, r3 по мере пошагового выполнения программы. Приведем итоговое содержимое регистрового окна. Оно свидетельствует о том, что все три возможных решения дают правильный результат.



4) Выбор очевиден. Первый и третий варианты требуют всего лишь двух команд процессора и, с точки зрения использования ресурсов процессора, эквивалентны. Второе решение сопровождается дополнительными операциями загрузки масок в регистры ЦПУ и проигрывает предыдущим, как по объему используемой памяти, так и по быстродействию. Третий вариант с использованием вставки битового поля наиболее прост и понятен – рекомендуется для применения.

Команда вставки битовых полей BFI очень эффективна для решения не только задач упаковки данных, но и модификации нужных битовых полей обычных регистров ЦПУ, регистров специального назначения и регистров периферийных устройств. Ее использование позволяет уменьшить число программных ошибок, связанных с неправильным выбором маски или числа разрядов сдвига маски при использовании классических решений. Использование команды BFI повышает надежность разрабатываемого программного обеспечения.

18.3.4 Команды распаковки данных в битовых полях любой размерности



Начиная с версии процессора Cortex-M4, система команд расширена двумя командами работы с битовыми полями, которые позволяют распаковывать не только байты, полуслова, но и данные произвольного формата (например, 24-битные, 12-битные и т.д.). Эти команды выполняют извлечение данных из слова в регистре-источнике и

авто-расширение формата до 32-разрядного полного слова, как с учетом знака, так и без (считая исходную битовую последовательность целым числом без знака). Они имеют следующий синтаксис:

```
SBFX{cond} Rd, Rn, #lsb, #width
UBFX{cond} Rd, Rn, #lsb, #width
```

width – ширина битового поля в числе бит

lsb – least significant bit – младший значащий бит (начальный бит битового поля)

Rn – Регистр-источник (битового поля)

Rd – Регистр назначения

cond – Код условного выполнения команды (опция)

BFX – Bit Field eXtract – Битовое поле извлечь (распаковать)

S – Signed – Расширить битовое поле как число со знаком в дополнительном коде
U – Unsigned – Расширить битовое поле как число без знака

Из регистра-источника Rn с заданного начального разряда lsb извлекается битовая последовательность (битовое поле) указанной длины width, которая расширяется до 32-х бит либо как число со знаком, либо как число без знака. Полученное слово сохраняется в регистре-приемнике Rd – рис. 18.7.

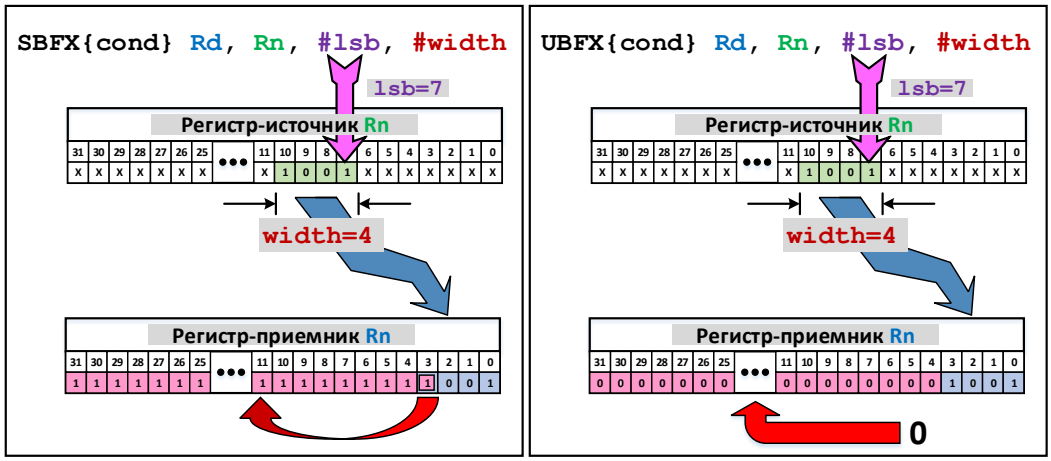


Рис. 18.7 Алгоритм распаковки битовых полей

Очевидно, что эти команды можно рассматривать как *универсальные* для распаковки данных в любом формате. Данные могут быть получены из портов ввода, из регистров периферийных устройств или по каналам связи с другими устройствами.

1) Предложите альтернативу рассмотренным ранее командам распаковки байт и расширения результата до полного слова:
   a. SXTB Rd, Rm, ROR #8
   b. UXTB Rd, Rm, ROR #16
2) Предложите альтернативу рассмотренным ранее командам распаковки полуслов и расширения до полного слова:
   a. SXTH Rd, Rm
   b. UXTH Rd, Rm, ROR #16

1) В командах-аналогах первый непосредственный операнд задает номер начального бита, а второй – число бит в битовом поле
   a. SBFX Rd, Rn, #8, #8
   b. UBFX Rd, Rn, #16, #8
2) Вначале распаковывается младшее полуслово как число со знаком, а затем старшее, как число без знака:
   a. SBFX Rd, Rn, #0, #16
   b. UBFX Rd, Rn, #16, #16

1) Создайте программу для проверки возможности замены команд распаковки байт и полуслов командами-аналогами, работающими с битовыми полями. Можете использовать примеры из контрольных вопросов выше.
2) Убедитесь в отладчике, что оба решения одинаково работоспособны.

3) Данные аналого-цифрового преобразования сигнала, считываемые из выходного порта АЦП, представляют собой 12-разрядный код без знака, расположенный в старших разрядах младшего полуслова. Какой командой можно преобразовать эти данные в 32-разрядное слово без знака?

1) Один из вариантов решения представлен в файле MyPorg_25_9.s. Для имитации порта ввода резервируется одна ячейка памяти в кодовой памяти Port_In. В начале программы содержимое порта ввода считывается в регистр r1 и далее распаковывается побайтно и пословно. В отладчике Вы можете ввести в память любые данные и проследить за их распаковкой в регистровом окне. Для каждого из заданий распаковка выполняется двумя способами:

```
10  ; Основная программа
11  Un_Pack_8_16
12  ; Загрузить регистр ЦПУ r1 исходными данными
13  ; (имитация ввода из порта периферийного устройства)
14          LDR r0,=Port_In
15          LDR r1,[r0]
16  ; Извлечь из полученного слова 2-й байт
17  ; расширить знаково до слова и сохранить
18  ; в регистрах r2,r3
19    ; Первый способ (распаковать байт)
20          SXTB r2,r1,ROR #8
21    ; Второй способ (распаковать битовое поле)
22          SBFX r3,r1,#8,#8
23
24  ; Извлечь из полученного слова 3-й байт
25  ; расширить без знаково до слова и сохранить
26  ; в регистрах r4,r5
27    ; Первый способ (распаковать байт)
28          UXTB r4,r1,ROR #16
29    ; Второй способ (распаковать битовое поле)
30          UBFX r5,r1,#16,#8
```

```
32  ; Извлечь из полученного слова младшее полуслово
33  ; расширить знаково до слова и сохранить
34  ; в регистрах r6,r7
35    ; Первый способ (распаковать полуслово)
36          SXTH r6,r1
37    ; Второй способ (распаковать битовое поле)
38          SBFX r7,r1,#0,#16
39
40  ; Извлечь из полученного слова старшее полуслово
41  ; расширить без знаково до слова и сохранить
42  ; в регистрах r8,r9
43    ; Первый способ (распаковать полуслово)
44          UXTH r8,r1,ROR #16
45    ; Второй способ (распаковать битовое поле)
46          UBFX r9,r1,#16,#16
47  ; Повторить распаковку для других входных данных
48          B Un_Pack_8_16
```

2) Оба варианта распаковки данных дают правильные результаты. Ниже представлено исходное содержимое регистра r1 и результаты его распаковки в регистры (r2, r3); (r4, r5); (r6, r7); (r8, r9).

R1	0xA5CFF1FD
R2	0xFFFFFFF1
R3	0xFFFFFFF1
R4	0x000000CF
R5	0x000000CF
R6	0xFFFFF1FD
R7	0xFFFFF1FD
R8	0x0000A5CF
R9	0x0000A5CF

3) После считывания содержимого порта в один из регистров ЦПУ Rn достаточно всего одной команды извлечения из регистра-источника Rn битового поля, начиная с 4-го разряда, длиной 12 разрядов, с авто-расширением нулями в область старших разрядов: UBFX Rd, Rn, #4, #12.

18.4 Команды реверсирования данных

Данные в памяти микропроцессорной системы могут располагаться по-разному. Используется как *прямой*, так и *обратный порядок* расположения байт в слове и полуслове, бит в слове, полуслове или байте. Если данные получены от другой микропроцессорной системы по каналу связи и имеют другой порядок упаковки в слово полуслов или байт, или другой порядок упаковки бит, потребуется дополнительная процедура приведения данных к внутреннему формату, применяемому в ARM-процессорах (обычно – к обратному порядку, когда младшие данные располагаются вначале).

Для поддержки этих операций в систему команд включены несколько команд *реверсирования* данных – рис. 18.8.

1) Команда **REV** реверсирует порядок расположения байт в слове, то есть меняет прямой порядок расположения байт на обратный, и наоборот.
2) Команда **REV16** реверсирует порядок расположения байт в каждом отдельном полуслове – в младшем и старшем.
3) Команда **REVSH** реверсирует порядок расположения байт исключительно в младшем полуслове и дополнительно выполняет знаковое расширение (суффикс «S») полученного полуслова до 32-разрядного слова – заполняет все старшие разряды знаковым разрядом полученного после операции реверсирования байт младшего полуслова.
4) Команда **RBIT** реверсирует порядок расположения бит в слове (с прямого на обратный, и наоборот).

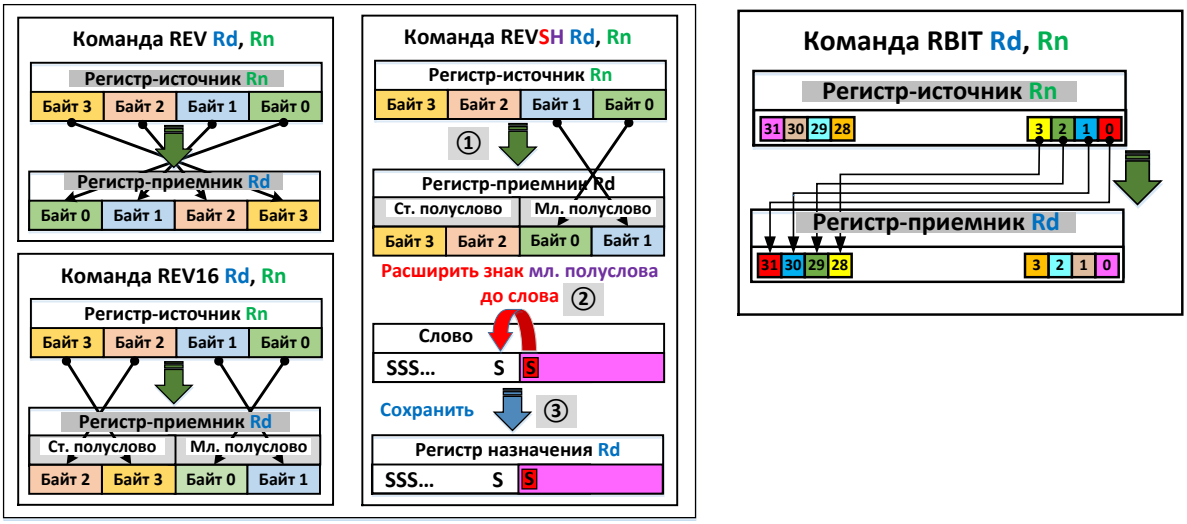


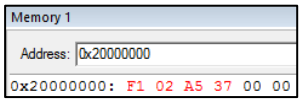
Рис. 18.8 Алгоритм работы команд реверсирования данных

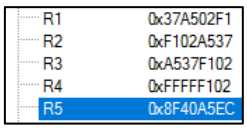
1) Какой порядок расположения байт используется в ARM-процессорах – прямой или обратный? Полуслов в слове? Бит в байте?
2) Создайте программу для изучения алгоритма работы команд реверсирования данных.
3) Проверьте работу команд реверсирования данных в отладчике.

1) Во всех трех случаях стандартным для ARM (устанавливаемым «по умолчанию») является обратный порядок расположения полуслов, байт и бит в слове – 32-bit big-endian data. При переинициализации процессора обратный порядок можно изменить на прямой. Без крайней необходимости лучше не пробовать!

2) Программа MyProg_25_10 содержит пример использования команд реверсирования данных. Данные вводятся из порта ввода и обрабатываются с сохранением результатов обработки в разных регистрах ЦПУ (r2, r3, r4, r5) для удобства оценки результата обработки. Байты в памяти расположены в обратном порядке. В таком же порядке они загружаются в регистр r1.

```
16  ; Изменить порядок расположения байт в слове с
17  ; обратного на прямой и сохранить результат в r2
18          REV r2,r1
19
20  ; Изменить порядок расположения байт в каждом
21  ; полуслове и сохранить результат в регистре r3
22          REV16 r3,r1
23
24  ; Изменить порядок расположения байт только в
25  ; младшем полуслове, расширить младшее слово
26  ; как число со знаком до формата полного слова
27  ; и сохранить в регистре r4
28          REVSH r4,r1
29
30  ; Изменить порядок расположения бит в слове и
31  ; сохранить результат в регистре r5
32          RBIT r5,r1
```





3) Проследите, как последовательно изменяются данные в процессе обработки: в r2 – обратный порядок расположения байт в слове заменяется прямым; в r3 – обратный порядок расположения байт в каждом полуслове заменяется прямым; в r4 – меняется порядок расположения байт в младшем полуслове, и оно расширяется знаковым разрядом до полного слова; r5 – меняется порядок всех бит в слове.

Система команд ARM-процессоров содержит специальные команды, облегчающие преобразование данных после ввода из портов или регистров периферийных устройств, а также специальные команды преобразования данных перед выводом результатов расчета в порты вывода или для передачи по интерфейсам каналов связи. Особенно эффективны команды работы с битовыми полями, позволяющие во многих случаях отказаться от типовых операций маскирования данных с использованием логических команд.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер.с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
3) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
4) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
5) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

19 ВВЕДЕНИЕ В АРИФМЕТИКУ ЧИСЕЛ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Оглавление

Арифметика чисел с плавающей точкой является одним из самых сложных разделов микропроцессорной техники. Специалисты, которые разрабатывают программное обеспечение на языках высокого уровня, обычно не вникают в детали этой технологии, справедливо считая, что корректность работы программы в любом случае должны обеспечить те, кто разработали и оттестировали компилятор с языка высокого уровня. Иная ситуация при программировании на языке низкого уровня - Ассемблере. Здесь Вы будете иметь дело с сопроцессором как таковым, с его режимами работы, его системой команд и особенностями обработки чисел с плавающей точкой, в том числе с возможными исключениями (некорректностями), возникающими при такой обработке.

Требования к квалификации разработчика существенно возрастают – он должен знать не только основы арифметики чисел с плавающей точкой, архитектуру и систему команд сопроцессора, но и возможные последствия выполнения каждой команды, способы обнаружения так называемых исключительных ситуаций и выхода из них.

Более того, разработчик должен уметь оценить точность выполненных расчетов с числами с плавающей точкой и, при необходимости, либо выбрать процессор с большей разрядностью, либо вернуться к расчетам в формате с фиксированной точкой.

Для разработчиков на C/C++ знание основ обработки чисел с плавающей точкой также необходимо, по крайней мере, для правильной настройки опций компилятора, выбора методов оптимизации программы, оценки точности реализованного алгоритма.

Текущая и следующая главы содержат основы теории обработки чисел в формате с плавающей точкой. В них введены основные понятия и определения, описаны возможные режимы работы сопроцессора и исключительные ситуации, с которыми может столкнуться программист. Сложные разделы, которые при первом чтении можно просмотреть бегло, помечены пиктограммой «Для любознательных». В последующих главах на конкретных примерах мы будем учиться разрабатывать и отлаживать программы с использованием системы команд сопроцессора с плавающей точкой.

19.1 Международный стандарт двоичной арифметики с плавающей точкой

Двоичные числа в формате с фиксированной точкой даже в 32-разрядных процессорах не всегда обеспечивают требуемую точность расчетов, особенно в тех случаях, когда диапазон изменения значений переменных значителен – от очень малых до очень больших (даже при использовании относительных единиц). Поэтому, специалисты в области компьютерной индустрии разработали специальные форматы представления двоичных чисел с так называемой *плавающей точкой* и методы обработки этих чисел (арифметику двоичных чисел с плавающей точкой). Этому направлению компьютерных вычислений уже более полувека. До утверждения международного стандарта каждый производитель компьютеров и программного обеспечения реализовывал вычисления с плавающей точкой в соответствии со своим собственным опытом. При этом зачастую использовались разные режимы округления, разные обработчики нестандартных ситуаций (исключений), что приводило к неоднозначности в реализации одних и тех же алгоритмов.

Международный стандарт IEEE-754 создан *ассоциацией инженеров электриков и электронщиков* IEEE (Institute of Electrical and Electronics Engineers) в 1985 г. и уточнен в 2008 г. Он обязывает всех производителей поддерживать одни и те же форматы и методы обработки данных с плавающей точкой. Различают числа с плавающей точкой:

1) 16-разрядные – половинной точности (**half-precision**);
2) 32-разрядные – одинарной точности (**single-precision**) – тип **float**в C/C++;

3) 64-разрядные – двойной точности (**double-precision**) – тип **double**в C/C++;
4) 128-разрядные – четверной точности (**quad-precision**).

Процессорные ядра Cortex-M4F поддерживают первый формат исключительно для более компактного хранения данных в памяти, второй формат – для хранения данных и вычислений. Форматы большой разрядности на аппаратном уровне не поддерживаются.

Числа с плавающей точкой двойной точности (double-precision) могут обрабатываться программно при использовании компиляторов с языков высокого уровня и соответствующих библиотек. Считается, что в микроконтроллерных применениях использование арифметики с плавающей точкой расширенной и двойной точности (43, 64 разряда) с аппаратной поддержкой нецелесообразно из-за чрезмерного удорожания процессорного ядра.

В этой главе мы рассмотрим базовые положения арифметики с плавающей точкой, поясняющие принцип действия и устройство сопроцессора (модуля FPU). Наша задача – понять, в каких случаях программист может использовать арифметику чисел с плавающей точкой вместо обычной арифметики чисел с фиксированной точкой. Какие преимущества дает эта технология и какие ограничения имеет?

19.2 Краткая историческая справка

Вычисления с плавающей точкой выполнялись раньше исключительно на суперкомпьютерах, а по мере развития микропроцессорной техники стали доступны сначала в миникомпьютерах, а затем – в персональных компьютерах. В микроконтроллерах, предназначенных для встроенных применений, раньше использовалась в основном арифметика с фиксированной точкой. Только с начала нового века (2000 г.) сопроцессоры поддержки вычислений с плавающей точкой стали встраиваться в микроконтроллеры. Так, в 1999 г. опциональный сопроцессор появился в ARM10. Сегодня процессоры, предназначенные для встраивания в оборудование, например, на базе Cortex-M4F, уже имеют на борту модуль поддержки вычислений с плавающей точкой FPU.

Интересно, что первый в мире процессор с плавающей точкой был разработан в Германии еще в 1938-41 гг. и обрабатывал 22-разрядные числа. В 1964 г. фирма IBM выпустила компьютеры с обработкой 32-разрядных чисел в формате с плавающей точкой. В современных суперкомпьютерах разрядность чисел с плавающей точкой превышает 64, а скорость вычислений достигает 10^{15} = 1 квадрильона операций с плавающей точкой в секунду.

Международный стандарт IEEE 754 был разработан в начале 80-х годов под руководством специалистов Intel, создавших сопроцессор i8087 для персональных компьютеров. Окончательная редакция стандарта выпущена в 2008 г.

19.3 Зачем нужны числа с плавающей точкой в микроконтроллерах?

Аппаратная реализация вычислений с плавающей точкой достаточно сложна и существенно удорожает стоимость кристалла микроконтроллера. Возникает естественный вопрос – стоит ли игра свеч? Что получает за дополнительные деньги разработчик, покупая микроконтроллер с сопроцессором? Чем вычисления в формате с плавающей точкой отличаются от вычислений с фиксированной точкой?

До сих пор мы пользовались 8-, 16- и 32-разрядными форматами целых чисел и чисел с фиксированной точкой (байтами, полусловами, словами). В этих форматах представлялись как числа без знака, так и числа со знаком в дополнительном коде.

Естественно, наибольший диапазон имеют 32-разрядные числа: целые без знака – от 0 до 4,294,967,295 (2^{32}–1); целые со знаком от −2,147,483,648 (-2^{31}) до 2,147,483,647 (2^{31}−1).

В чем недостаток таких форматов?

- Кажущийся большим, диапазон представления целых чисел на практике оказывается *недостаточным*. Приходится пользоваться арифметикой, так называемых, многобайтовых чисел (64-, 128-разрядных).
- Кажущийся достаточным диапазон представления малых чисел, даже при переходе к форматам с фиксированной точкой (в том числе к дробным числам), все равно оказывается недостаточным. Например, заряд электрона 1.6×10^{-19} кулон даже не может быть представлен в расчетах.
- Проблема представления «сверхбольших» и «сверхмалых» чисел несколько нивелируется при переходе к расчетам в относительных единицах. Но, такой переход требует дополнительных преобразований данных, затрудняет визуализацию процессов управления и отладку.
- Одновременное использование как больших, так и малых чисел приводит к необходимости пользоваться разными форматами чисел с фиксированной точкой, выполняя переключения между ними непосредственно в программе. Это существенно усложняет алгоритмы и, кроме того, требует унифицированных библиотек поддержки вычислений в любых I.Q-форматах.

Главное преимущество чисел в формате с плавающей точкой состоит в том, что они могут представлять и «сверхбольшие» и «сверхмалые» числа. Недостаток – такое представление не всегда является точным.

Формат плавающей точки обеспечивает некий компромисс – возможность представлять в рамках имеющегося числа двоичных разрядов ЦПУ и очень большие, и очень маленькие числа. Экспоненциальная форма числа предполагает, что указываются и *мантисса* и *порядок* числа. При этом вся числовая шкала делится на множество поддиапазонов, число которых равно числу возможных значений порядка. В каждом из таких поддиапазонов мантисса может изменяться на единицу, обеспечивая одинаковый интервал между соседними числами. Однако, величина этого интервала, т.е. абсолютная точность представления чисел, меняется по мере перехода от одного диапазона к другому. Это означает, что абсолютная точность падает по мере роста значения числа.

Так как внутри каждого поддиапазона можно представить только конечное число значений, то любое вещественное число будет приблизительным – *округленным до ближайшего возможного значения*, которое называется *репрезентабельным* (*представимым*) значением (от англ. representable). При выполнении арифметических операций с числами в формате с плавающей точкой обязательно потребуется один из методов округления. Просто отбрасывание незначимого остатка (усечение) может привести к большим погрешностям в вычислениях. Поэтому, все сопроцессоры имеют средства выбора оптимального для данной прикладной задачи метода округления.

1) С каким интервалом на числовой оси располагаются целые числа?
2) Чему равна абсолютная погрешность представления целых чисел?
3) Чему равна абсолютная погрешность представления дробных чисел в формате S1.32?
4) Одинаково ли число так называемых битовых шаблонов (битовых комбинаций) которые возможны для представления 32-разрядных целых чисел, чисел в формате с фиксированной точкой I.Q, чисел с плавающей точкой (с мантиссой и порядком)?

1) С интервалом в 1. Они имеют одинаковое «цифровое разрешение».
2) Абсолютная погрешность равна 1.
3) Весу младшего разряда дробной части числа, то есть 2^{-31}.
4) Да, оно равно 2^{32} для любых чисел.

Программист должен четко понимать преимущества и недостатки работы с целыми числами, вещественными с фиксированной точкой и вещественными с плавающей точкой. Для каждого типа чисел имеются свои оптимальные области применения. Например, вряд ли кто-то будет подсчитывать число деталей на конвейере с использованием чисел в формате с плавающей точкой, когда для этого более удобно использовать целые числа, более того – целые числа без знака.

19.4 Представление двоичного числа в нормализованном экспоненциальном виде

Для облегчения понимания, дальнейшее изложение будем вести на конкретных примерах. Так, десятичное число +105.5625 можно представить в *нормализованном* виде, если десятичную точку сдвинуть в положение сразу после первого ненулевого разряда числа $+105.5625=+1.055625*10^2$. Такой сдвиг должен сопровождаться умножением числа на основание системы счисления (в данном случае 10) в степени, равной числу разрядов сдвига, если сдвиг десятичной точки выполняется влево. Если сдвиг десятичной точки выполняется вправо, то степень будет иметь отрицательное значение, например, $+0.0000584=+5.84*10^{-5}$.

Нормализованным называется представление числа в виде *мантиссы* с одним значащим разрядом до точки и *порядка* – степени основания системы счисления, которая определяет число, на которое должна быть умножена мантисса.

Наше первое число можно записать так: $1.055625*exp_{10}^{+2}$. При этом мантисса M=1.055625, а экспонента представляется основанием системы счисления (10) и порядком +2. Во втором примере порядок экспоненты будет отрицательным (-5). В десятичной системе счисления мантисса нормализованного числа находится в диапазоне $1 \leq M < 10$, так как первая значащая цифра числа может быть любой от 1 до 9. Поэтому, значение мантиссы 9.9999 вполне допустимо.

Таким образом, в общем случае для представления числа с плавающей точкой требуется место под *знак числа*, *мантиссу*, *знак порядка* и, собственно, *порядок*.

Нормализация двоичных чисел в формате с фиксированной точкой выполняется так же, как десятичных: двоичная точка переносится на такое число разрядов влево или вправо, чтобы в целой части мантиссы оказался только один значащий разряд, а это в двоичной системе счисления – единица. При этом мантисса должна быть умножена на экспоненту, – основание системы счисления (2) в степени, равной числу разрядов, на которое перенесена двоичная точка. Если сдвиг точки выполняется влево, то порядок положительный, вправо – отрицательный, например,

$$11110000.1010 = 1.11100001010* exp_2^{+111}$$
$$0.00000011011 = 1.1011*exp_2^{-111}$$

Мантисса нормализованного двоичного числа находится в диапазоне $\mathbf{1 \leq M < 2}$, то есть всегда больше или равна 1, но строго меньше 2. Именно поэтому разработчики международного стандарта предложили не отводить под единицу мантиссы отдельный двоичный разряд – считать, что единица как бы присутствует в двоичной мантиссе «по

умолчанию». Это позволяет сэкономить один разряд в формате числа и представлять в нем только дробную часть мантиссы. Если для отображения дробной части мантиссы выделить 23 разряда, то для представленных выше двух примеров, получим:

1) $М_{др}$=11100001010000000000000
2) $М_{др}$=10110000000000000000000.

Так как мантисса *нормализованного двоичного числа* всегда больше или равна единице, то ее целая часть (единица) по умолчанию опускается, – предполагается всегда равной 1, а в двоичном представлении мантиссы присутствует только ее дробная часть.

19.5 Формат 32-разрядного двоичного числа с плавающей точкой однократной точности

Итак, для отображения мантиссы разработчики сэкономили один разряд, считая, что 1 присутствует в мантиссе «по умолчанию», а сама мантисса представлена в числе только своей дробной частью $М_{др}$.

Как быть со знаком числа и знаком порядка? Для знака числа выделяется отдельный, старший разряд числа (31-й), который называется «знаковым» и обозначается буквой S (0 – число положительное, 1-отрицательное).

Со знаком порядка поступают так. Для нормализованных чисел однократной точности ограничивают сдвиг двоичной точки влево или вправо (экспоненту exp) допустимым диапазоном (**-126** ÷ +**127**) разрядов. Если добавить к числу разрядов сдвига (к экспоненте exp) число +127, то полученное значение (1 ÷ +254) размещается в одном байте при любом допустимом значении порядка числа.

Полученное значение называется *смещенной экспонентой* E и не содержит отдельного знакового разряда E = exp +127.

При этом появляется возможность использования двух оставшихся значений экспоненты для представления *особых чисел* с плавающей точкой:

Exp = -127 (E = -127+127=0) – нулей (имеющих минимальный порядок);
Exp = +128 (E = +128 +127 = 255) – бесконечностей (максимальный порядок).

Для размещения смещенной экспоненты E выделяется 8 разрядов 32-разрядного числа после знакового разряда. Оставшиеся 23 бита используются для размещения мантиссы, точнее – ее дробной части $М_{др}$ (единица целой части мантиссы как бы присутствует«по умолчанию»).

Таким образом, *двоичные числа в формате с плавающей точкой однократной точности* в соответствии с международным стандартом IEEE754 имеют следующий формат:

Разряды 32-разрядного числа с плавающей точкой однократной точности		
1р	**8р**	**23р**
31	**30 23**	**22 0**
		$М_{др}$ – Мантисса (только ее дробная часть)
	Смещенная экспонента E= exp + 127 (байт без знака)	
S – Знак числа: 0 – положительное, 1 – отрицательное		

19.6 Как преобразовать десятичное число в формате с фиксированной точкой в двоичное число с плавающей точкой?

1-й шаг – классическая операция преобразования десятичного числа с фиксированной точкой в двоичное число с фиксированной точкой. Это обычная операция разложения десятичного числа по двоичным разрядам, с учетом того, что двоичные разряды слева от двоичной точки имеют веса 2^0, 2^1, 2^2, … в соответствии с номерами двоичных разрядов, начинающихся с 0, а справа от двоичной точки – 2^{-1}, 2^{-2}, 2^{-3}, … в соответствии с номерами двоичных разрядов дробной части, начинающихся с 1.

Целая часть десятичного числа должна быть разложена на составляющие 1, 2, 4, 8, 16, ..., а дробная – на составляющие 0.5; 0.25; 0.125; … Коэффициенты перед этими составляющими (1 или 0) и дадут двоичный код десятичного числа, например

105.5625_{10}= 1*64+1*32+0*16+1*8+0*4+0*2+1*1+1*0.5+0*0.25+0*0.125+1*0.0625

или в двоичном коде 1101001.1001_2.

2-й шаг – это представление двоичного числа в экспоненциальной форме:

$1101001.1001 = 1.1010011001*exp_2^{+110}$

Нам пришлось сдвинуть двоичную точку на 6 разрядов влево, следовательно, порядок числа равен $+6_{10} = +110_2$. Таким образом, мантисса нашего двоичного числа M=1.1010011001, а экспонента exp_2^{+110}, где $+110_2$ – порядок.

3-й шаг – получение кода только дробной части мантиссы для записи в 23 разряда, выделенных для представления мантиссы $M_{др}$= 10100110010000000000000.

4-й шаг – получение кода смещенной экспоненты для записи в поле порядка:

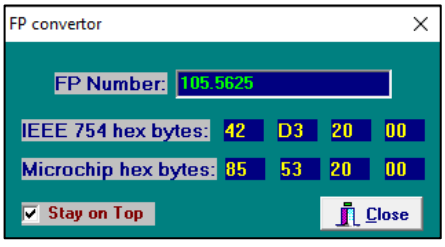
E= +6 +127 = +133 = +(128 +4+1) = $1000\ 0101_2$.

Теперь остается только заполнить соответствующие поля шаблона двоичными кодами и получить представление десятичного числа 105.5625_{10} в формате с плавающей точкой:

Разряды 32-разрядного числа с плавающей точкой однократной точности							
1р	**8р**			**23р**			
Зн.	**Смещ. эксп. E**			**Мантисса (только дробная часть) – $M_{др}$**			
0	**1 0 0 0 0 1 0 1**			**1 0 1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0**			
0100	**0010**	**1101**	**0011**	**0010**	**0000**	**0000**	**0000**
42		**D3**		**20**		**00**	

Зеленым фоном выделен двоичный код числа, серым – выполнена разбивка двоичного кода на ниблы (4-битовые) коды, каждый из которых соответствует одной шестнадцатеричной цифре, а в нижней строке – число представлено в шестнадцатеричном формате $42D32000_{16}$ = 0x42D32000, который обычно используется в процессорной технике для компактности представления двоичного кода.

В Интернете, в открытом доступе, имеется большое число специальных *калькуляторов для программистов* (так называемых *конверторов форматов*), которые позволяют ввести десятичное число в формате с фиксированной точкой и получить его *компьютерное представление* в формате с плавающей точкой однократной точности. Результат работы одного из таких калькуляторов FPconvrt для нашего примера подтверждает правильность выполненного нами вручную преобразования числа 105.5625.



При программировании на Ассемблере или Си преобразование чисел из *стандартной десятичной экспоненциальной формы* в формат двоичного числа с плавающей точкой однократной точности (float) автоматически выполняется транслятором с Ассемблера или компилятором с языка Си. Более того, интегрированные среды отладки допускают ввод таких чисел в удобном для программиста формате с авто-преобразованием в соответствующий двоичный код. Поддерживается также отображение результатов расчетов над числами в формате float в удобной для восприятия программистом и отладки форме.

19.7 Как ввести число в формате с плавающей точкой в программе на Ассемблере?

ARM ASM

Для ввода чисел с плавающей точкой можно использовать одну из стандартных *десятичных экспоненциальных форм* представления таких чисел, используемую в математике и в технике:

- {-}цифрыE{-}цифры
- {-}{цифры}.цифры
- {-}{цифры}.цифрыE{-}цифры,

где: «цифры» – обычные десятичные цифры (0,1,…9), а «E» – символ порядка, который можно набирать как в верхнем, так и в нижнем регистре.

Знаки числа и порядка являются опциями. Если знак опущен, то он «по умолчанию» считается положительным.

Можно выполнить предварительное преобразование нужного числа в двоичную форму (например, с использованием специальных программ-конверторов) и при написании программы ввести число в шестнадцатеричной системе счисления

- 0xhexdigits,

где hexdigits- шестнадцатеричные цифры (0,1, 2,…A,B,C,D,E,F).

Для процессоров, обрабатывающих числа с плавающей точкой однократной точности:

- Максимально возможное число по модулю 3.40282347e+38.
- Минимально возможное число по модулю 1.17549435e–38.

Для резервирования констант с плавающей точкой в кодовой памяти используется директива Ассемблера **DCFS** (определить код в формате с плавающей точкой однократной точности) – зарезервировать место в памяти и проинициализировать константу в формате с плавающей точкой однократной точности.

Примеры:

```
DCFS    1.5E-31,-4E+15
DCFS    1.75
DCFS    0.028
DCFS    5.725e15
DCFS    0x7FC00000  ; Ввод «Не числа» QuietNaN в Hex-форме
```

19.8 Как преобразовать двоичное число в формате с плавающей точкой однократной точности в десятичное число?

Для этого необходимо в двоичном коде числа с плавающей точкой IEE 754 выделить и записать в виде целых чисел без знака три параметра:

- Бит знака числа S (31-й бит);
- Смещенную экспоненту E (30-23-й биты);
- Дробную часть мантиссы $M_{др}$ (22-0-й биты).

Эквивалентное десятичное число рассчитывается по следующей формуле:

$$F_{10}= (-1)^{S}*2^{(E-127)}*(1+M_{др}/2^{23}) \quad (19.1)$$

Для рассматриваемого нами примера десятичного числа 105.5625_{10}:
S=0;
E=133;
$M_{др}$ = 101 0011 0010 0000 0000 0000 = 532000_{16} = 5447680_{10}

Подставим эти значения в формулу (19.1):
$F_{10} = (-1)^{0}*2^{(133-127)}*(1+5447680/2^{23}) = 2^{6}*(1+0.649414062) = 64*1.649414062$
F_{10} = 105.5625
Результат абсолютно правильный!

19.9 Как представляются числа с плавающей точкой двойной точности?

Стандарт IEE 574 регламентирует представление чисел с плавающей точкой пяти типов:

- с половинной точностью (half-precision) – **16** бит;
- с одинарной точностью (single-precision) – **32** бита;
- с двойной точностью (double-precision) – **64** бита;
- с одинарной расширенной точностью (single-extended precision) >=**43** бит (редко используется);
- с двойной расширенной точностью (double-extended precision) >= **79** бит (обычно используется 80 бит);

При программировании на Ассемблере мы будем иметь дело *только с числами одинарной точности*. Числа половинной точности поддерживаются исключительно в операциях преобразования и пересылки данных, но не в вычислительных операциях. При программировании на Си, Вы можете использовать также числа с плавающей точкой двойной точности. Работа с ними поддерживается не на аппаратном уровне, а на уровне специализированных библиотек, поставляемых вместе с компилятором языка Си. Разные форматы отличаются только числом разрядов, выделенных для представления порядка (смещенной экспоненты) и мантиссы. На рис. 19.1 дано сравнение форматов чисел одинарной и двойной точности:

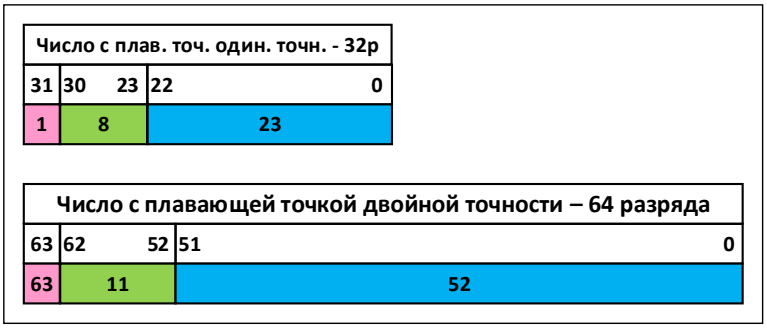


Рис. 19.1 Формат чисел с плавающей точкой одинарной и двойной точности

19.10 Шкала и диапазон представления чисел с плавающей точкой однократной точности

Не все возможные 32-разрядные битовые последовательности представляют собой числа в формате с плавающей точкой. Некоторые последовательности имеют особое значение и обрабатываются сопроцессором специальным образом.

Различают:

1. *Нормальные* (*нормализованные*) числа – normalnumbers;
2. *Субнормальные* (*денормализованные*) числа – subnormalnumbers;
3. *Нули* – zero;
4. *Бесконечности* – infinities;
5. «*Не числа*» – NaNs.

Ниже в табл. 19.1 представлен весь *диапазон возможных значений чисел* с плавающей точкой однократной точности и «специальных чисел», указаны особенности их представления с точки зрения значений порядка и мантиссы, приведен диапазон возможных шестнадцатеричных значений каждого числа. Самые важные положения:

1) Нулем считается число, которое имеет минимальное значение порядка **(-127**) и минимальное значение мантиссы (**1.0**) – с дробной частью, равной нулю.
2) Для симметричности представления вещественных чисел различают два разных нуля (**+0**) и (**-0**), которые отличаются только знаковым разрядом. Для нулей все разряды числа, кроме знакового (смещенной экспоненты и дробной части мантиссы) – нулевые.
3) Нормированные вещественные числа имеют порядок в диапазоне от (**-126**) до (**+127**) и любое значение мантиссы (в рамках 23-х разрядов). При этом диапазоны представления положительных и отрицательных чисел симметричны и равны **$1.17549435 \cdot e^{-38} <$ Нормированные числа $< 3.40282347 \cdot e^{+38}$**
4) Числа, имеющие максимально возможное значение порядка (**+128**) и минимально возможное значение мантиссы (**1.0**) с нулевой дробной частью считаются бесконечно большими. Для симметричности числовой шкалы предусматривается как положительная (+∞), так и отрицательная (-∞) бесконечности.
5) Все числа, имеющие максимальное значение порядка (**+128**) и любое значение мантиссы, кроме минимально-возможного (**>1.0**), считаются «Не числами» (NaN – NotaNumber), выходящими за пределы возможного представления чисел с плавающей точкой однократной точности.

6) Между нулями и минимально-возможными нормализованными числами на числовой шкале образуются *разрывы непрерывности* (показаны желтым цветом в табл.19.1). Для уменьшения разрывов в зоне чисел, близких к нулю, стандарт предполагает использование *денормализованных* чисел, *исключающих эти разрывы*. Денормализованными считаются все числа, имеющие минимально возможный порядок нормализованного числа (**-127**) и мантиссу, меньшую единицы (**<1.0**).

Таблица 19.1 Числа с плавающей точкой однократной точности

Hex- формат				**Десятичное значение**	**Норм./ Денорм.**	**Особенности представления**
7F	**FF**	**FF**	**FF**	**NANs – Не числа**		**Порядок ←Max (+128); Мантисса ← [Любая >Min (1.0)]**
7F	**80**	**00**	**01**			
7F	**80**	**00**	**00**	**+∞**		**Порядок ←Max (+128); Мантисса ←Min (1.0)**
7F	**7F**	**FF**	**FF**	**+3.40282347e+38**	**Норм.**	**Порядок ← [(-126) <Любой < (+127)]; Мантисса ← Любая**
00	**80**	**00**	**00**	**+1.17549435e-38**		
00	**7F**	**FF**	**FF**	**+1.17549421e-38**		**Порядок ←Min (-127); Мантисса ← [Любая >Min (1.0)]**
					Денорм.	
00	**00**	**00**	**01**	**+1.40129846e-45**		
00	**00**	**00**	**00**	**+0**	**Норм.**	**Порядок ←Min (-127); Мантисса ← Min (1.0)**
80	**00**	**00**	**00**	**-0**		**Порядок ←Min (-127); Мантисса ← Min (1.0)**
80	**00**	**00**	**01**	**-1.40129846e-45**		**Порядок ←Min (-127); Мантисса ← [Любая >Min (1.0)]**
					Денорм.	
80	**7F**	**FF**	**FF**	**+1.17549421e-38**		
80	**80**	**00**	**00**	**-1.17549435e-38**	**Норм.**	**Порядок ← [(-126) <Любой < (+127)]; Мантисса ← Любая**
FF	**7F**	**FF**	**FF**	**-3.40282347e+38**		
FF	**80**	**00**	**00**	**-∞**		**Порядок ←Max (+128); Мантисса ←Min (1.0)**
FF	**80**	**00**	**01**	**NANs – Не числа**		**Порядок ←Max (+128); Мантисса ← [Любая >Min (1.0)]**
FF	**FF**	**FF**	**FF**			

19.11 Денормализованные числа

19.11.1 Представление денормализованных чисел

Числа с плавающей точкой могут быть представлены в денормализованном виде, когда точка сдвигается на такое число разрядов, чтобы в целой части числа оказался ноль. Для примера: +105.5625 = $0.1055625*10^3$=0.1055625*$exp_{10}3$. Таким образом, мантисса денормализованного десятичного числа всегда представляет собой дробь (0<M≤1), а экспонента является основанием системы счисления в, равной порядку числа. Для чисел в двоичной системе счисления двоичная точка сдвигается на такое число разрядов, чтобы первая значащая цифра после нее была равна единице, например:

$$11110000.1010 = 0.111100001010* exp_2^{+1000}$$
$$0.00000011011 = 0.11011*exp_2^{-110}$$

В этом случае мантисса всегда находится в диапазоне двоичных чисел **0.1≤M<1** или в десятичной форме **0.5≤M**<1.

Это означает, что все денормализованные двоичные числа имеют не единицу «по умолчанию» в поле представления мантиссы, а ноль.

Стандарт IEEE 754 предполагает использование денормализованных чисел *для расширения диапазона нормализованных чисел* в области малых значений (вблизи нуля). Такие числа будут иметь самый маленький порядок нормализованного числа (-127) и мантиссу, которая представлена исключительно дробной частью («единица по умолчанию» будет отсутствовать). Сводка признаков, по которым различаются числа с плавающей точкой однократной точности, приведена в табл. 19.2. Зеленым фоном выделены числа, являющиеся основными, нормализованными числами, используемыми при большинстве вычислений. Денормализованные числа, называемые иногда *субнормальными*, также поддерживаются процессорным ядром Cortex-M4F и позволяют поднять точность вычислений в области нуля («сверхмалых» чисел).

Таблица 19.2 Признаки классификации чисел в формате с плавающей точкой

Экспонента exp	**Др. часть мантиссыf (fraction)**	**Число**	**Комментарий**
-127	**f=0**	**±0**	**Ноль**
-127	**f≠0**	**$0.f \times 2^{-127}$**	**Денормализованные**
-126≤ exp ≤ +127	**f – любое**	**$1.f \times 2^{exp}$**	**Нормализованные**
+128	**f=0**	∞	**Бесконечность**
+128	**f≠0**	**NaN**	**Не числа**

1. Какие изменения нужно внести в формулу (19.1) для расчета десятичных значений денормализованных чисел?
2. Воспользуйтесь этими формулами для оценки значений чисел на границе нормализованных и денормализованных чисел (проверьте данные в табл. 19.1).
3. Сравните диапазон представления вещественных чисел в формате нормализованных чисел с плавающей точкой с диапазоном вещественных чисел в формате с фиксированной точкой.
4. Какое минимальное число можно представить в формате с фиксированной точкой? Сравните с минимальным числом в формате нормализованного числа с плавающей точкой.
5. Сравните с минимальным денормализованным числом.
6. Является ли «бесконечность» следующим большим репрезентабельным числом (которое может быть представлено в рамках данной разрядной сетки), по сравнению с максимально возможным числом в формате с плавающей точкой?
7. Можно ли считать числовую шкалу чисел в формате с плавающей точкой практически непрерывной?

1. Нужно убрать из значения мантиссы «1» по умолчанию.
3. Максимальные 32-разрядные числа со знаком в формате с фиксированной точкой (S32.0): + 2 147 483 647 и - 2 147 483 648. Диапазон возможных значений этих чисел в экспоненциальной форме при любых форматах с фиксированной точкой не превосходит ($-2.147483648e^{9} \div +2.147483647e^{9}$). Диапазон нормализованных чисел с плавающей точкой ($-3.40282347 \cdot e^{+38} \div +3.40282347 \cdot e^{+38}$) превышает его на 29 порядков.
4. Минимальное 32-разрядное число со знаком в формате с фиксированной точкой – это дробь в формате (S1.31). Минимальное значение дроби соответствует $\pm 4.6566128e^{-10}$, предельной дискретности представления чисел в формате с фиксированной точкой. С ростом числа разрядов целой части дискретность представления чисел падает. Минимальное нормализованное число в формате с плавающей точкой ±1.17549435e-38 (меньше на 28 порядков).
5. Минимальное денормализованное число ±1.40129846e-45 еще на 7 порядков расширяет диапазон представления «сверхмалых» чисел.
6. Да, это так. Более того, в расчетах при переполнении результат может заменяться бесконечностью (+∞, -∞) без прерывания вычислительного процесса.
7. Да, если использовать в расчетах и денормализованные числа.

19.11.2 Графическое представление шкалы чисел в формате с плавающей точкой

Представим для простоты, что число с плавающей точкой всего лишь 6-разрядное. При этом старший разряд – знаковый, далее следуют три разряда смещенной экспоненты и два разряда мантиссы. Максимальное значение смещенной экспоненты равно 7 (в 3-битовом поле). Величина смещения – половина этого значения +3. Возможные значения смещенной экспоненты E, экспоненты exp и множителя, на который должна быть умножена мантисса, приведены в табл. 19.3.

Таблица 19.3 Возможные значения экспоненты и множителя

Смещенная Экспонента E			**Экспонента (exp) = E - 3**	**Множитель**	**Значение**
0	**0**	**0**	**-3 (Субнормальные числа)**	**2^{exp+1}**	**1/4**
0	**0**	**1**	**-2**	**2^{exp}**	**1/4**
0	**1**	**0**	**-1**		**1/2**
0	**1**	**1**	**0**		**1**
1	**0**	**0**	**+1**		**2**
1	**0**	**1**	**+2**		**4**
1	**1**	**0**	**+3**		**8**
1	**1**	**1**	**+4 (Бесконечность)**		

Нулевое значение смещенной экспоненты используется для представления субнормальных чисел (Subnormals) – «сверхмаленьких». Для них значение числа равно множителю 2^{exp+1}, умноженному на мантиссу в формате дроби 0.f. (единица «по умолчанию» опускается).

Мы ограничили поле мантиссы всего лишь двумя двоичными разрядами. При этом для каждого значения экспоненты получим только 4 возможных значения мантиссы: для субнормальных чисел в формате 0.f, а для нормализованных числе в формате 1.f – табл. 19.4. Это означает, что в каждом поддиапазоне чисел с плавающей точкой будут представлены только 4 возможных числа. Это будут *репрезентабельные* числа – *представимые* в рамках выделенного для данного числа разрядов.

Таблица 19.4 Возможные значения мантисс

Дробная часть .f		Значение мантиссы субнормального числа 0.f (денормализованного)	Значение мантиссы нормального числа 1.f (нормализованного)
0	**0**	**0.00**	**1.00**
0	**1**	**0.25**	**1.25**
1	**0**	**0.50**	**1.50**
1	**1**	**0.75**	**1.75**

Обратите особое внимание на то, что множители для денормализованных чисел 2^{exp+1} и начального диапазона нормализованных чисел 2^{exp} одинаковы. Это означает, что *интервал* между ближайшими денормализованными числами и нормализованными числами в самом начале числовой шкалы *один и тот же* (в нашем примере 0.0625).

Убедитесь сами, что для рассматриваемого формата существуют следующие поддиапазоны репрезентебельных (представимых) чисел (табл. 19.5).

Таблица 19.5 Диапазоны репрезентебельных чисел

Поддиапазон	Репрезентативные числа
Субнормальные:	**0.0000; 0.0625; 0.1250; 0.1875**
Нормализованные, 1-й	**0.2500; 0.3125; 0.3750; 0.4375**
Нормализованные, 2-й	**0.5000; 0.6250; 0.7500; 0.8750**
Нормализованные, 3-й	**1.00; 1.25; 1.50; 1.75**
Нормализованные, 4-й	**2.00; 2.50; 3.00; 3.50**
Нормализованные, 5-й	**4.00; 5.00; 6.00; 7.00**
Нормализованные, 6-й	**8.00; 10.0; 12.0; 14.0**

С увеличением числа разрядов мантиссы на один разряд число репрезентабельных значений чисел в каждом поддиапазоне (для каждого значения экспоненты) удваивается. Оно равно 2^n, где n – число двоичных разрядов, выделенных для представления мантиссы. В нашем простом примере (два разряда мантиссы) таких чисел всего 4. Графическая иллюстрация числовой шкалы чисел для нашего примера представлена на рис. 19.2.

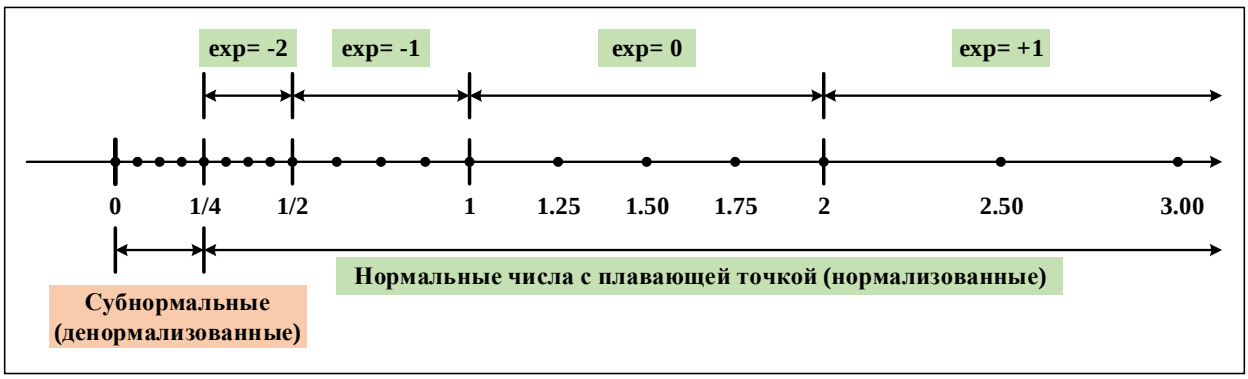


Рис. 19.2 Пример числовой шкалы 6-битовых чисел с плавающей точкой

Для чисел с плавающей точкой однократной точности при 23-х разрядах мантиссы это будет уже 2^{23}=8388608 чисел. Однако, плотность их расположения на числовой оси с ростом значения числа будет падать (с каждым новым интервалом в два раза). Следовательно, абсолютная погрешность представления больших чисел будет увеличиваться.

Примеры:

1) При нулевом значении экспоненты exp=0 внутри диапазона возможны следующие значения чисел с плавающей точкой однократной точности:

   Минимальное: $2^0 = 1.0$

   Максимальное: $2^0 \times (2^1 - 2^{-23}) = 1.99999998808$

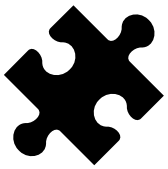
   Интервал между соседними числами $2^{-23} = 0.0000001192 \approx 1.192 \times 10^{-7}$

2) При максимально возможном значении экспоненты +127, получим:

   Минимальное: $2^{127} = 1.70141118 \times 10^{38}$

   Максимальное $2^{127} \times (2^1 - 2^{-23}) = 3.4028234 \times 10^{38}$

   Интервал между соседними числами $2^{127} \times 2^{-23} = 2.0282409 \times 10^{31}$



1. Абсолютная точность представления чисел в формате с плавающей точкой зависит от значения экспоненты и резко падает с увеличением значения числа.
2. При выполнении инженерных расчетов точности в несколько десятичных цифр до и после десятичной точки вполне достаточно. Ее может не хватить только в финансовых и бухгалтерских расчетах, которые и не предполагается выполнять на встраиваемых микроконтроллерах, – для этого существуют компьютеры.
3. Для инженерных расчетов, когда в результате должны быть получены только несколько значащих цифр (одна – до точки и три, четыре – после точки), формат чисел однократной точности более чем достаточен.
4. Если представлять переменные управляемого процесса в относительных единицах, то точность вычислений будет оптимальной - погрешность в районе 7-го знака после десятичной точки.
5. Денормализованные (субнормальные) числа с плавающей точкой расширяют числовую шкалу *вплоть до нуля*, полностью исключая разрыв непрерывности в области «околонулевых» чисел.

На рис. 19.3 для сравнения показаны диапазоны целых 32-разрядных чисел, 16-разрядных чисел с плавающей точкой половинной точности и 32-разрядных чисел с плавающей точкой однократной точности (в положительной части числовой шкалы).

Диапазон чисел с плавающей точкой однократной точности, расширенный в области «околонулевых» чисел «субнормальными» числами, непрерывен и оптимален по точности для использования в системах встроенного управления оборудованием.

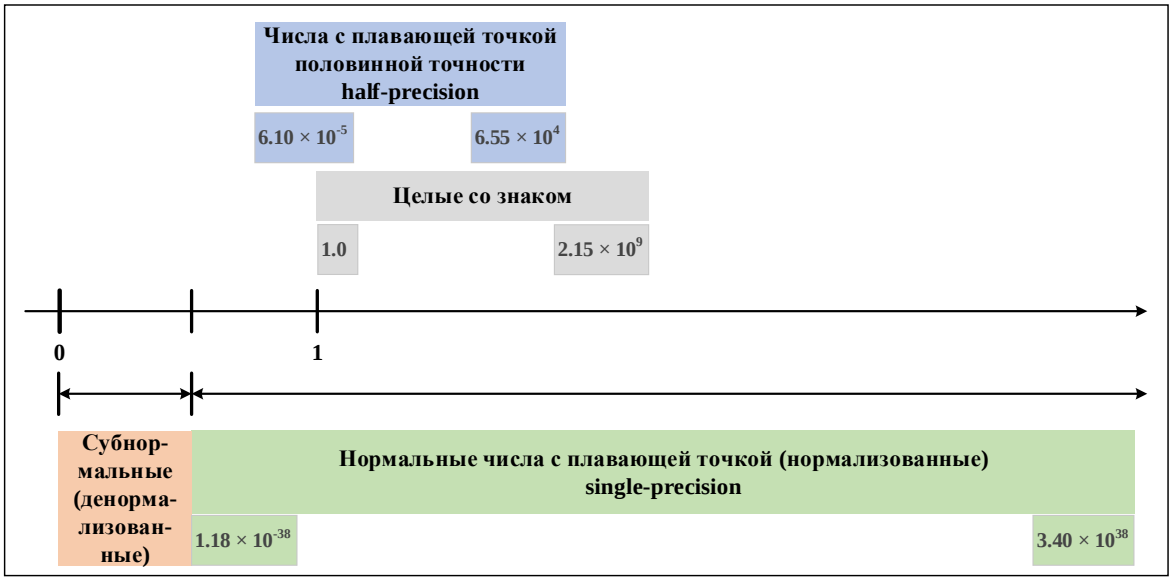


Рис. 19.3 Сравнение диапазонов разных чисел

19.12 Технология выполнения арифметических операций над числами с плавающей точкой

Приведем краткую справку по технологии аппаратного выполнения арифметических операций над числами с плавающей точкой для понимания устройства сопроцессора и особенностей набора его команд, поддерживающего такие вычисления.

19.12.1 Сложение

Каждое слагаемое представлено в виде мантиссы и порядка. Складывать можно только числа, имеющие одинаковый порядок:

$$m1 \cdot e^{n1} + m2 \cdot e^{n2} = m1 \cdot e^{n1} + m2' \cdot e^{n1} = (m1 + m2') \cdot e^{n1}$$

1) Если порядки чисел одинаковы, то складываются мантиссы чисел.
2) Если порядки слагаемых разные (предположим порядок первого числа больше порядка второго, n1 >n2), то числа приводятся к одному порядку (большему):
   a. Определяется разность порядков чисел (n1-n2).
   b. Мантисса меньшего числа сдвигается вправо на число разрядов (n1-n2).
3) Приведенное значение мантиссы меньшего числа $m2'$ складывается с мантиссой большего числа $m1$.
4) Выполняется нормализация полученной суммы и коррекция порядка суммы.

19.12.2 Вычитание

Вычитание чисел выполняется аналогично сложению. Число с меньшим порядком приводится к числу с большим порядком. Из мантиссы большего числа вычитается мантисса меньшего. Выполняется нормализация полученной разности.

19.12.3 Умножение

При умножении чисел с плавающей точкой мантиссы множителей перемножаются, а порядки – складываются:

$$(m1 \cdot e^{n1}) \cdot (m2 \cdot e^{n2}) = (m1 \cdot m2) \cdot e^{n1+n2}$$

19.12.4 Деление

При делении чисел с плавающей точкой мантисса делимого делится на мантиссу делителя, а из порядка делимого вычитается порядок делителя:

$$(m1 \cdot e^{n1})/(m2 \cdot e^{n2}) = (m1/m2) \cdot e^{n1-n2}$$

19.13 Способы округления чисел с плавающей точкой

Все арифметические операции над числами с плавающей точкой выполняются в модуле поддержки вычислений с плавающей точкой FPU аппаратно с *резервированием числа разрядов для последующего округления* (расширением разрядной сетки в сторону меньших разрядов). Стандарт IEEE754 предусматривает четыре способа округления:

1. Округление в направлении к ближайшему значению числа в рамках разрядной сетки. (**Roundtothenearest**) – 32 разряда для чисел однократной точности.
2. Округление в направлении нуля (**Roundtozero**) – нужное число разрядов остается, а остальные просто отбрасываются – эквивалентно усечению (truncation)
3. Округление в направлении к +∞ (**Roundto +∞**) – в направлении положительной бесконечности до числа, ближайшего большего в рамках данной разрядной сетки.
4. Округление в направлении к -∞ (**Roundto -∞**) – в направлении отрицательной бесконечности, до числа, ближайшего меньшего в рамках данной разрядной сетки.

Первый вариант округления используется в арифметике чисел с плавающей точкой наиболее часто. Это округление либо вверх, либо вниз, которое дает максимально возможную точность при заданном числе бит для представления числа. По существу, – это округление к ближайшему «репрезентабельному» числу (которое может быть представлено в рамках имеющегося формата). При инициализации модуля FPU программист должен выбрать один из режимов округления. Для правильного выбора режима округления приведем наглядный пример возможных вариантов округления десятичных чисел до сотых, поясняющий принцип выполнения этих операций (см. табл. 19.6).

Таблица 19.6 Пример использования разных режимов округления

Исходное значение числа	**Вариант округления (до сотых)**			
	(Roundtothenearest) Кближайшему	**(Roundtozero Кнулю**	**(Roundto +∞) К +∞**	**(Roundto -∞) К -∞**
+1.433	**+1.43**	**+1.43**	**+1.44**	**+1.43**
-1.433	**-1.43**	**-1.43**	**-1.43**	**-1.44**
+1.778	**+1.78**	**+1.77**	**+1.78**	**+1.77**
-1.778	**-1.78**	**-1.77**	**-1.77**	**-1.78**

Программист *должен выбрать и проинициализировать* один из четырех возможных режимов округления для работы с числами в формате с плавающей точкой. По умолчанию это первый способ – округление к **ближайшему возможному значению** в рамках выделенной для числа разрядной сетки.

19.14 Когда результатом операции является «Не Число»? Типы «Не чисел»

Когда с математической точки зрения результат операции является *неопределенным* (примеры в табл. 19.7), сопроцессор в качестве результата операции возвращает «Не число» NaN (от англ. Not-a-Number) и формирует флаг исключительной ситуации InvalidOperation («Некорректная операция»). Этот флаг может вызвать запрос обработки исключения UsageFault («Ошибка использования») и, если обработка разрешена, переход на соответствующую подпрограмму – обработчик исключения. Только программист, зная специфику своей задачи, может разработать подпрограмму «обработки» исключения: заменить результат приемлемым и продолжить вычисления, или остановить вычислительный процесс с выдачей соответствующего уведомления на один из приборов индикации системы (для доработки алгоритма и программы).

Типовыми операциями, генерирующими «Не числа», являются операции деления на ноль, извлечения квадратного корня из отрицательного числа, умножения нуля на бесконечность и им подобные, когда с математической точки зрения операция просто бессмысленна.

Второй вариант генерации «Не чисел», когда один или более операндов в арифметической операции с числами с плавающей точкой *сами являются* «Не числами». В этом случае в качестве результата операции также возвращается «Не число».

Таблица 19.7 Примеры операций, возвращающих «Не Числа» (NaN)

Операция	**Значения операндов**
+	**(-∞) + (+ ∞)**
×	**0 × (±∞)**
/	**0/0;** ∞/∞
√	**√x, при x<0**

Особенность представления «Не чисел» **NaN**в том, что все они имеют максимально возможную экспоненту exp = +128: E= 255, все биты которой равны 1, (как для бесконечностей) и мантиссу, отличную от нуля (единицу, по крайней мере, в одном из 23-х разрядов). Напомним, что обе бесконечности кодируются нулевым значением мантиссы.

«Не числа» NaN имеют еще несколько возможных применений.

1) Могут использоваться в качестве «значений по умолчанию» в регистрах процессора или в структурах данных, расположенных в памяти. Если такое значение будет считано до записи корректного начального значения (до инициализации переменной или регистра), то любая операция с плавающей точкой возвратит в качестве результата NaN, что будет сигнализировать о некорректности вычислений.
2) Программист может сам проанализировать поведение программы в какой-то точке. Если результат не соответствует ожидаемому (например, больше допустимого числа 35.78), он может вернуть принудительно в качестве результата «Не число».

Дальнейшие операции с «Не числами» могут вызвать исключение, т.е. прерывание нормального хода выполнения программы. Это упрощает отладку, позволяя найти тот фрагмент кода, действия в котором выполнены некорректно.

«Не числа» NaN, специально предназначенные для принудительной *сигнализации о некорректности программы или алгоритма,* называются «Сигнализирующими Не числами» - *SingalingNaNs* (**sNaN**). Факт их получения в качестве аргумента любой арифметической командой может свидетельствовать о выполненной ранее *некорректной операции,* о том, что программист желает прервать обычный ход вычислений, искусственно вызвав исключение. Поэтому, любая арифметическая команда (кроме команды пересылки данных) вызовет генерацию флага исключения *InvalidOperation* («Недопустимая операция») и вызов соответствующего обработчика исключения, если обработка исключений разрешена. Это дополнительная возможность отладки программы в сложных проектах. Получение «Не числа» в качестве аргумента команды пересылки – допустимо, и не вызывает генерации исключения по «Недопустимой операции». Такие числа могут использоваться, например, для передачи символов в программах символьной обработки данных. Сам символ может располагаться в поле мантиссы, например, в младшем байте.

Второй возможный тип «Не чисел» можно назвать «Несигнализирующими или «тихими Не числами» *QuietNaNs* (**qNaN**). При их получении не будет автоматически формироваться флаг «Недопустимой операции».

Итак, все «Не числа» делятся на два типа:

- Signaling NaNs (**sNaN**) – «Сигнализирующие Не числа»;
- Non-signaling, QuietNaNs (**qNaN**) – Обычные, несигнализирующие, «тихие Не числа».

Примеры «Не чисел», несущих дополнительную информацию:

Таблица 19.8 «Не числа»

Не число с дополн. информацией	**В формате Single-Precision**
Quiet NaN, 0x01	**0x7FC00001**
Quiet NaN, 0x55	**0x7FC00055**
Signalling NaN, 0x01	**0x7F800001**
Signalling NaN, 0x55	**0x7F800055**

Вы видите, что для обычных «Не чисел» (Не сигнализирующих) самый старший разряд мантиссы равен 1, а для Сигнализирующих «Не чисел» – нулю. Младший байт используется для передачи кода символа (см. рис. 19.4).

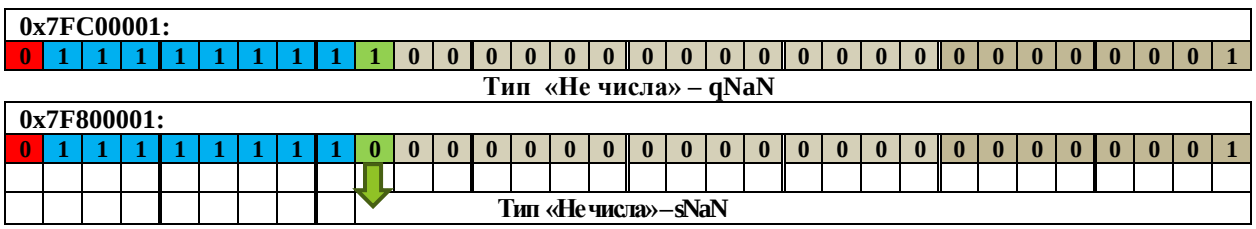


Рис. 19.4 «Не числа»

Знаковый бит в «Не числах» не является значащим. Он также может использоваться для передачи дополнительной полезной информации.

Среди всех «Не чисел» различают «Не числа по умолчанию» DefaultNaN. Их представление не регламентируется стандартом, а зависит от разработчика сопроцессора. Как правило, это те значения, которые возвращаются, если один из операндов

арифметической команды сам является «Не числом». В процессорах Cortex-M4Fв качестве «Не числа по умолчанию» принимается «тихое Не число» qNaN, со всеми нулевыми разрядами мантиссы, кроме самого старшего. Число «DefaultNaN» в Cortex-M4F не используется для передачи полезной (символьной информации).

19.15 Исключения при работе с числами с плавающей точкой

Исключением называется прерывание программы пользователя при получении предположительно недостоверного результата операции и передача управления соответствующей подпрограмме – обработчику исключения. По умолчанию процессор должен выставить соответствующий флаг исключительной ситуации, сформировать результат, например, NaN и продолжить выполнение программы. Если обработка возникшего исключения разрешена, то управление будет автоматически передано соответствующей программе – обработчику исключения, в противном случае – программа будет просто выполняться дальше. В первом случае программа-обработчик исключения решает, как информировать программиста о некорректно выполненной команде, или какой результат вернуть, во втором случае (обработка исключений запрещена) – всю ответственность за дальнейшую работу программы берет на себя программист.

Он может в «подозрительных» местах программы выполнить «программный поллинг» флагов исключительных ситуаций и принять решение, как о продолжении выполнения программы, так и о результате операции, который должен быть возвращен.

Типовая исключительная ситуация – «*Деление на ноль*» или «*Переполнение*», – выход за пределы допустимых чисел. При этом различают как «*Переполнение вверх*» – превышение результатом операции максимально возможного числа в формате с плавающей точкой, так и «*Переполнение вниз*» – попадание результата в зону «околонулевых», субнормальных чисел или обнуление. В первом случае в качестве результата обычно возвращается бесконечность, а во втором – субнормальное число или ноль.

Важное преимущество работы с числами в формате с плавающей точкой состоит в том, что сам формат обеспечивает как бы *автонасыщение при переполнениях вверх* и *автосброс в ноль* или *в диапазон субнормальных чисел* при переполнениях вниз (зависит от режима работы сопроцессора). При этом в большинстве случаев работа программы по управлению объектом может быть успешно продолжена без опасности возникновения автоколебаний, как это обычно бывает при работе с числами в формате с фиксированной точкой. Более того, числа в субнормальном диапазоне будут обрабатываться наравне с нормальными, а на флаг «Переполнение вниз» можно просто не обращать внимание. Он будет свидетельствовать только о том, что результат попал в диапазон субнормальных чисел.

В случае возникновения исключения в регистре состояния сопроцессора (подробно в след. главе) устанавливаются соответствующие флаги, которые обладают свойством «защелкивания» (могут быть сброшены только при считывании). Это позволяет программным способом определять, был ли полученный результат целой серии вычислений корректным и можно ли его использовать дальше или необходимо сбросить.

Всего имеется 5 типов исключений, для каждого из которых предусмотрен свой отдельный флаг:

1) divisionbyzero – деление на ноль;
2) overflow – переполнение вверх;

3) underflow – переполнение вниз;
4) invalid operation – недопустимая операция;
5) inexact – неточность.

«Недопустимые операции» возникают тогда, когда исходные операнды таковы, что результат операции не определен – табл. 19.7. Это относится также к операциям сравнения, если один из операндов не является числом NaN, а также к любым арифметическим операциям, когда один или более операндов являются не числами NaN. В любом их этих случаев формируется флаг «Недопустимая операция», и в качестве результата возвращается «Не число».

Обычно при «Переполнении вверх» в качестве результата возвращается $\pm\infty$ или максимально возможное число $\pm x_{max}$, а при «Переполнениях вниз» – 0 или денормализованное число. При этом максимальное значение нормализованного числа $x_{max} = 1.11\ldots11\times2^{+127}$.

Исключение inexact («*Неточность*) свидетельствует о том, что результат операции не может быть представлен одним из «репрезентабельных» чисел, входящих в допустимый диапазон чисел с плавающей точкой. Такая ситуация может возникнуть, например, при преобразовании целого числа в формат числа с плавающей точкой. При этом автоматически выполняется округление в сторону возможного «представимого» числа.

Для квалифицированных программистов *настоятельно рекомендуется* использовать подпрограммы – обработчики исключений, работающие совместно с важнейшим периферийным устройством – контроллером прерываний. В этом случае подпрограмма-обработчик принимает решение, какой результат возвращать в основную программу для продолжения вычислений.

В этой главе мы привели лишь наиболее общие сведения о числах в формате с плавающей точкой и о технологии их обработки, регламентируемые международным стандартом. При рассмотрении аппаратной структуры сопроцессора (модуля FPU) и изучении системы команд поддержки вычислений с плавающей точкой эти сведения будут детализироваться, в том числе, на примерах реальных задач. Все проекты – доступны для читателя и могут быть протестированы в отладчике среды µVision. Обращайтесь к материалам текущей главы по мере изучения последующих глав.

Список рекомендуемой литературы

1) Интернет-ресурс habr.com. Статья «Что нужно знать про арифметику с плавающей запятой».
2) Интернет-ресурс ru.wikipedia.org. Статья «Число_с_плавающей_запятой».
3) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
4) Встраиваемые высокопроизводительные цифровые системы управления. Практический курс разработки и отладки программного обеспечения сигнальных микроконтроллеров TMS320x28xxx в интегрированной среде Code Composer Studio: учеб. пособие / А.С. Анучин, Д.И. Алямкин, А.В. Дроздов и др.; под общ. ред. В.Ф. Козаченко, – М.: Издательский дом МЭИ, 2010. – 270 с.

20 МОДУЛЬ ПОДДЕРЖКИ ВЫЧИСЛЕНИЙ С ПЛАВАЮЩЕЙ ТОЧКОЙ FPU

Оглавление

20.1 Назначение

Модуль *поддержки вычислений с плавающей точкой* Floating Point Unit (FPU) интегрирован в состав центрального процессора Cortex-M4F. Он выполняет функцию сопроцессора при работе с операндами в формате с плавающей точкой *однократной точности*, обеспечивая совместимость с общепризнанным международным стандартом арифметики чисел с плавающей точкой IEEE 754-2008 (глава 19). На аппаратном уровне поддерживаются все 32-разрядные вычисления однократной точности и типы данных, описанные в стандарте.

В полном объеме (аппаратно и на уровне системы команд) поддерживаются операции:

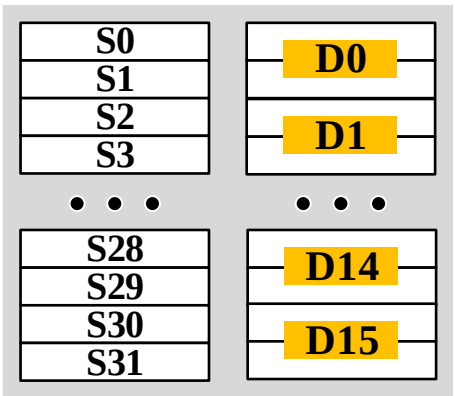
- Загрузки констант в формате с плавающей точкой в регистры модуля FPU;
- Сложения, вычитания, умножения, деления;
- Умножения с накоплением;
- Извлечения квадратного корня;
- Преобразования данных из формата с фиксированной точкой (fixed-point) в формат с плавающей точкой (floating-point) и обратно.

Модуль FPU является сопроцессором, который существенно расширяет диапазон чисел, с которыми может работать процессор, предоставляя разработчикам систем управления реального времени (в том числе на языке ассемблер) принципиально новые возможности создания и отладки программного обеспечения непосредственно в физических координатах переменных, что делает этот процесс более удобным и наглядным. Разработка микроконтроллерных систем управления все больше приближается к современным технологиям компьютерного моделирования сложных систем на персональных компьютерах.

20.2 Регистры модуля поддержки вычислений с плавающей точкой

Модуль плавающей точки FPU содержит **расширенный регистровый файл** (extension register file), специально предназначенный для хранения чисел в формате с плавающей точкой и обработки этих чисел в модуле FPU:

- 32 регистра однократной точности S0-S31 (single-word);
- 16 регистров двукратной точности D0-D15 (double-word).



Каждая пара 32-разрядных регистров однократной точности образует один 64-разрядный регистр двойной точности.

Перед выполнением любой операции над числами в формате с плавающей точкой операнды должны быть предварительно загружены в регистровый файл модуля FPU. Туда же будет сохраняться результат операции (в один из регистров файла). Каждый регистр двойной точности состоит из двух регистров однократной точности:

- Младшее 32-разрядное слово регистра D<n> имеет символическое имя S<2n>;

- Старшее 32-разрядное слово регистра D<n> имеет символическое имя S<2n+1>.

Например, Вы можете обратиться к младшему значащему слову регистра двойной точности D14 по имени S28, а к старшему значащему слову регистра D14 – по имени S29. Напомним, что в процессоре Cortex-M4F операции с числами двойной точности поддерживаются только с помощью дополнительных библиотек, хотя загрузка в регистры D0-D15 и сохранение данных из них, а также пересылка данных двойной точности (одновременно двух регистров однократной точности), поддерживаются как на аппаратном уровне, так и на уровне системы команд.

20.3 Регистры управления модулем FPU

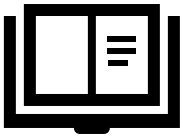
Сопроцессор является сложным устройством, причем *программно настраиваемым*. Прежде чем его использовать, программист должен не только разрешить работу модуля FPU, но и задать нужный для конкретного приложения режим работы. Имеется несколько *системных регистров*, предназначенных для управления модулем FPU, настройки режимов его работы и контроля текущего состояния модуля – табл. 20.1. Два самых важных регистра, которые чаще всего используются, выделены в табл. 20.1 фоном. Их назначение будет рассмотрено подробно.

Таблица 20.1 Системные регистры модуля FPU

Адрес	Имя	Тип	Сост. при сбросе	Название
0xE000ED88	**CPACR**	**RW**	**0x00000000**	***Coprocessor Access Control Register*** **Регистр управления доступом к сопроцессору**
0xE000EF34	**FPCCR**	**RW**	**0xC0000000**	***Floating-point Context Control Register*** **Регистр управления контекстом**
0xE000EF38	**FPCAR**	**RW**	**-**	***Floating-point Context Address Register*** **Регистр адреса контекста**
-	**FPSCR**	**RW**	**-**	***Floating-point Status Control Register*** **Регистр статуса и управления**
0xE000EF3C	**FPDSCR**	**RW**	**0x00000000**	***Floating-point Default Status Control Register*** **Регистр управления статусом по умолчанию**

Специальные регистры FPCCR и FPCAR предназначены для сохранения и восстановления контекста модуля FPU при входе и выходе из процедуры обслуживания исключительных ситуаций и здесь не рассматриваются. Последний регистр FPDSCR в таблице содержит установки «по умолчанию», которые можно использовать для загрузки или перезагрузки основного регистра статуса и управления FPSCR.

20.4 Регистр управления доступом к сопроцессору CPACR



Данный регистр позволяет установить привилегию доступа к сопроцессору, то есть, по сути, разрешить или запретить доступ к нему. Из множества полей, которые введены для совместимости с другими ARM-процессорами для Cortex-M4F используются только два битовых поля с символическими именами CP11 и CP10, которые как раз и отвечают за подключение модуля поддержки вычислений с плавающей точкой FPU, состоящего из двух сопроцессоров CP11 и CP10:

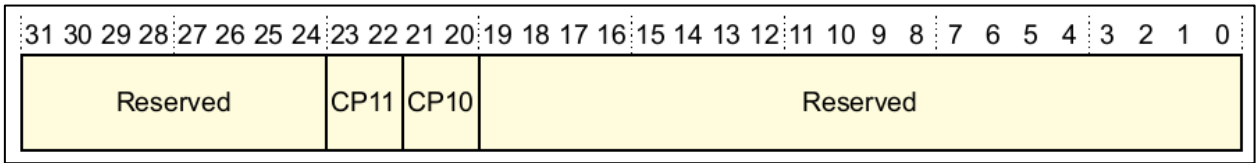


Рис. 20.1 Регистр CPACR FPU

Таблица 20.2 Битовые поля регистр CPACR

Биты	Имя	Функция
[31:24]	**-**	**Зарезервированы. При чтении – 0. Запись – игнорируется.**
[23:22]	**CP11**	**Биты управления привилегией доступа к сопроцессору 11**
[21:20]	**CP10**	**Биты управления привилегией доступа к сопроцессору 10**
[19:0]	**-**	**Зарезервированы. При чтении – 0. Запись – игнорируется.**

Код в этих битовых полях соответствует следующим возможным режимам доступа (см. табл. 20.3).

Таблица 20.3 Режимы доступа к сопроцессору FPU

Код в поле CP	Режим доступа к сопроцессору
00	**Access denied – Отказ в доступе. При любой попытке доступа будет генерироваться ошибка использования UsageFault – NOCP (нет сопроцессора)**
01	**Priveleged access only. Только привилегированный доступ. Попытка доступа в непривилегированном режиме генерирует ошибку использования UsageFault – NOCP (нет сопроцессора)**
10	**Reserved. Код зарезервирован. Результат доступа непредсказуем.**
11	**Full access. Полный доступ.**

Для того, чтобы разрешить работу модуля поддержки вычислений с плавающей точкой FPU, выполните инициализацию каждого битового поля CP11, CP10 в регистре управления доступом к сопроцессору CPACR по рассмотренной в 17.4.1 технологии «Чтение, модификация, запись» регистров специального назначения. Используйте на начальном этапе освоения процессора Full access (Полный доступ). Если инициализация модуля FPU не выполнена, а к нему имеется обращение, то автоматически генерируется запрос на обработку исключения Usage Fault (Ошибка использования). Если обработка этого исключения разрешена, выполняется переход на соответствующую подпрограмму-обработчик (см. 2.7.2). Пример инициализации модуля FPU мы рассмотрим в конце этой главы.

20.5 Регистр статуса и управления модулем вычислений с плавающей точкой FPSCR

Этот регистр обеспечивает полный контроль программиста за режимом работы и текущим состоянием модуля FPU. Назначение его битовых полей рассмотрим подробно (см. табл. 20.4), так как именно он позволяет установить нужный для приложения пользователя режим работы, проверить текущее состояние вычислительного процесса по результатам сравнения переменных в формате с плавающей точкой или на предмет возникновения исключительной ситуации (возможной некорректности выполненных вычислений с операндами в формате с плавающей точкой).

В соответствии с этими тремя важнейшими функциями в регистре имеются три группы битовых полей: (31-28) – флаги результатов операций сравнения чисел с

плавающей точкой; (26-22) – биты управления режимом работы сопроцессора; (7-0) – биты флагов исключительных ситуаций, возникающих при обработке данных:

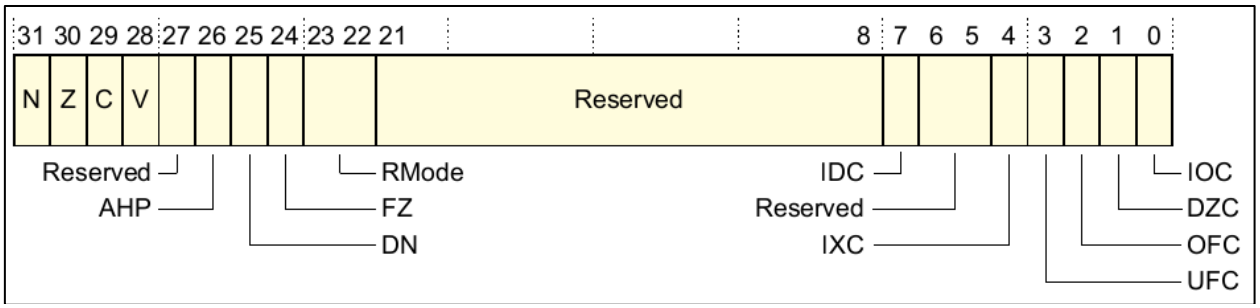


Рис. 20.2 Регистр FPSCR FPU

Таблица 20.4 Значения битовых полей регистра статуса и управления FPSCR

Биты	Имя	Функция	Возможные значения
Флаги результатов операций сравнения чисел с плавающей точкой			
31	**N**	**Negative – Флаг отрицательного результата**	**Обновляются только операциями сравнения чисел с плавающей точкой VCMP**
30	**Z**	**Zero – Флаг нуля**	
29	**C**	**Carry – Флаг переноса**	
28	**V**	**Overflow – Флаг переполнения**	
27	**-**	**Reserved – Зарезервировано**	
Биты установки альтернативных режимов работы модуля FPU			
26	**AHP**	**Alternative half-precision control bit Бит управления альтернативным режимом половинной точности**	**0 – Формат половинной точности по стандарту IEEE-754 1 - Альтернативный формат половинной точности фирмы ARM**
25	**DN**	**Default NaN mode control bit Бит управления режимом генерации «Не чисел по умолчанию»**	**0 – Обработка «Не чисел» (NaN) по стандарту IEEE-754 1 - Любая операция с участием операндов «Не чисел» возвращает результат «Не число по умолчанию» (Default NaN)**
24	**FZ**	**Flush-to-zero mode control bit Бит управления режимом «Приближения» к нулю (замены малых чисел нулем)**	**0 – Режим Flush-to-zero запрещен. В соответствии с IEEE 754 возможна работа с субнормальными числами. 1 - Разрешение режима работы Flush-to-zero**

Биты установки режима округления.				
23:22	**RMode**	**Rounding Mode control field Битовое поле управления режимом округления.**	**00**	**Round to Nearest: RN - mode К ближайшему возможному представимому числу**
			01	**Round towards Plus Infinity: RP - mode В направлении + ∞**
			10	**Round towards Minus Infinity: RM - mode В направлении - ∞**
			11	**Round towards Zero: RZ - mode В направлении нуля. Усечение, отбрасывание остатка**
21:8	**-**	**Reserved - Зарезервировано**		

Кумулятивные (накопительные) флаги исключений cumulative exception bit - устанавливаются в 1, если соответствующее исключение произошло.			
7	**IDC**	**Input Denormal cumulative**	**Ввод денормированного числа**
6:5	**-**	**Reserved - Зарезервировано**	
4	**IXC**	**Inexact cumulative**	**Неточность**
3	**UFC**	**Underflow cumulative**	**Переполнение вниз**
2	**OFC**	**Overflow cumulative**	**Переполнение вверх**
1	**DZC**	**Division by Zero cumulative**	**Деление на ноль**
0	**IOC**	**Invalid Operation cumulative**	**Недопустимая операция**

20.5.1 Флаги результатов сравнения чисел с плавающей точкой

Числа с плавающей точкой можно сравнивать между собой. Для сравнения используется команда VCMP. Все остальные команды не оказывают никакого влияния на эти флаги. Но и эта команда не обновляет флаги в регистре статуса приложения APSR непосредственно, а только в регистре статуса и управления модуля плавающей точки FPSCR. Для того, чтобы эти флаги использовать для выработки кодов условий ветвления и кодов условного выполнения команд (в том числе, команд обработки чисел с плавающей точкой), необходимо скопировать их из регистра FPSCR в регистр APSR с помощью специальной команды сопроцессора VMRS (пересылки из регистра специального назначения модуля FPU в регистр ЦПУ):

```
VMRS APSR_nzcv, FPSCR
```

Будут скопированы только четыре бита: (N, Z, C, V). В результате появится *возможность условного ветвления программы* или *условного выполнения* последующих *команд по стандартным кодам условий*, которые применяются и при работе с числами в формате с фиксированной точкой.

Смысл флагов (N, Z, C, V) зависит от того, в каком месте они были выработаны: при выполнении обычных команд ЦПУ или команд сравнения чисел с плавающей точкой. Это накладывает некоторые ограничения на использование кодов условного ветвления и условного выполнения команд:

- Числа в формате с плавающей точкой всегда являются знаковыми. Поэтому, коды условий, используемые для сравнения чисел без знака (выше, ниже, выше или тоже самое и т.д.), *не могут применяться* при работе с числами с плавающей точкой;
- «Не числа» (NaN) не могут иметь никаких операций отношения с числами или с другими «Не числами» (друг с другом). Такое сравнение будет неупорядоченным (unordered compare), то есть некорректным. Попытка выполнения такого сравнения вызовет генерацию флага исключения IOC («Недопустимая операция»);
- Для условного выполнения команд обработки чисел с плавающей точкой используйте коды условий, предназначенные для сравнения знаковых чисел: EQ, NE, GT, GE, LT, LE. Их вполне достаточно для любых алгоритмов.

Коды условного ветвления программы и условного выполнения команд являются *общими* для всех команд процессора, в том числе для команд, обрабатывающих числа в формате с плавающей точкой. Следовательно, технология условной передачи управления и условного выполнения команд, используемая в процессорах ARM, распространяется и на команды модуля FPU. Единственное, что требуется, – после операции сравнения чисел с плавающей точкой, скопировать содержимое статусного регистра сопроцессора FPSCR в регистр статуса программы пользователя APSR.

20.5.2 Установка альтернативных режимов работы сопроцессора

Три бита регистра FPSCR (26-24) позволяют задать, при необходимости, *специальные режимы работы* модуля FPU, отличающиеся от режима, регламентируемого международным стандартом. Вы вряд ли будете их использовать. Тем не менее, для интересующихся, приведем их краткое описание:

- **AHP** – Alternative half-precision Format – Альтернативный режим обработки чисел в формате половинной точности (16 разрядов). Отличается от регламентируемого стандартом IEEE 754 тем, что особые числа (бесконечности и «Не числа» NaN) не поддерживаются, а формат экспоненты увеличен на один разряд для расширения диапазона чисел половинной точности – это собственный формат фирмы ARM. В режиме половинной точности, в том числе альтернативном, в процессорах Cortex-M4F, можно только хранить числа с плавающей точкой, а обрабатывать – нельзя. Режим половинной точности применяется обычно для экономии памяти в системе. В современных микроконтроллерах памяти достаточно, и преобразование чисел из обычного формата в формат половинной точности и обратно вряд ли будет востребован.
- **DN** – Default NaN mode – Установить режим «Не числа по умолчанию» (Default NaN). В соответствии со стандартом «Не числа» могут быть нескольких типов и содержать дополнительную полезную информацию (например, символьную). Если установить режим DN, то любая операция с плавающей точкой, в которой, по крайней мере, один из входных операндов является «Не числом», будет возвращать в регистр назначения специальное число – «Не число по умолчанию» Default NaN. Находящаяся в исходных операндах «Не числах» любого типа полезная информация (например, символьная) будет утеряна.
- **FZ** – Flush-to-zero mode – Режим приближения к нулю, когда маленькие значения чисел, меньшие минимально возможных для стандарта чисел с плавающей точкой однократной точности (субнормальные), автоматически заменяются нулями при выполнении арифметических операций. Если факт замены «крошечных» чисел на ноль произошел, то выставляется флаг исключения Underflow cumulative («Переполнение вниз») UFC (FPSCR[3]). В стандартном режиме работы «крошечные» числа представляются в виде денормализованных чисел, расширяя числовую шкалу в область «околонулевых» чисел и повышая точность расчетов. При этом округление до нуля не делается.

При обнулении флагов разрешения специальных режимов работы сопроцессор будет работать в режиме полной совместимости со стандартом IEEE 754, что и рекомендуется для большинства приложений.

20.5.3 Биты задания режима округления

Последующие два бита регистра FPSCR (23-22) задают *режим округления* при выполнении операций над числами в формате с плавающей точкой, подробно рассмотренные в предыдущей главе (см. 19.3). Отметим, что для большинства приложений наиболее подходящим является режим округления RN – до ближайшего *репрезентабельного* (representable) числа, которое может быть представлено в рамках 32-разрядного формата числа с плавающей точкой однократной точности (до ближайшего *представимого* числа). Остальные режимы округления применяются, главным образом, для тестирования устойчивости алгоритмов в условиях *однонаправленного округления* (усечения результатов или их округления в направлении только положительной или только отрицательной бесконечности). Это позволит оценить возможность самых пессимистических результатов работы программы и исключить неожиданности.

Внутри сопроцессора вычисления с плавающей точкой *выполняются с повышенной разрядностью*. Это означает, что при разрядности дробной части мантиссы 23 бита и всей мантиссы 24 бита (плюс бит «по умолчанию», равный 1), разрядность вычислителя *расширена в сторону младших разрядов*. Предположим, что вычисления выполняются только для чисел в формате однократной точности и разрядность мантиссы расширена вправо до 40 бит (точное значение знает только разработчик сопроцессора). При этом старшие 24 бита мантиссы будут всегда выдаваться в качестве результата операции после завершения процедуры округления, а оставшиеся младшие 16 бит (остаток) будут использоваться исключительно в процедуре округления и после ее завершения отбрасываться.

Если выполняется округление в сторону ближайшего репрезентабельного числа в формате с плавающей точкой, то выдаваемое значение мантиссы может быть либо оставлено неизменным, либо увеличено на единицу младшего разряда (инкрементировано). Блок округления сопроцессора выполняет анализ: превышает ли отбрасываемый остаток половину младшего разряда мантиссы (1/2 LSB)? Если да, то

округление выполняется в сторону следующего репрезентабельного числа путем инкрементирования значение мантиссы. В противном случае значение мантиссы не меняется.

В этом алгоритме есть одна неопределенность. Что делать, если остаток в точности равен половине младшего разряда мантиссы 1/2 LSB? В соответствии с международным стандартом округление выполняется к ближайшему возможному четному значению мантиссы Round to Nearest Even (RNE). Если младший разряд L был равен нулю – он не меняется, если 1, то мантисса – инкрементируется. Иллюстрация этого, наиболее часто используемого, алгоритма округления представлена на рис. 20.3.

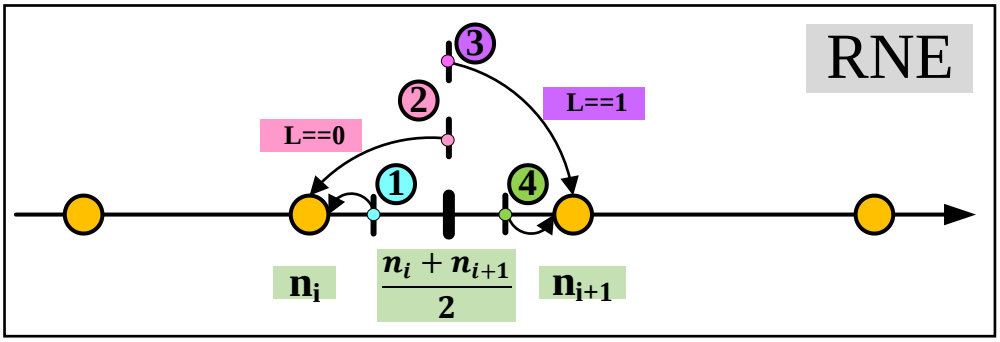


Рис. 20.3 Технология округления к ближайшему четному значению мантиссы RNE

В режиме округления в направлении положительной бесконечности RP мантисса любого положительного результата *с ненулевым остатком* инкрементируется на 1. Точный и отрицательный результат не округляется. Для отрицательных чисел это округление эквивалентно просто отбрасыванию остатка – рис. 20.4.

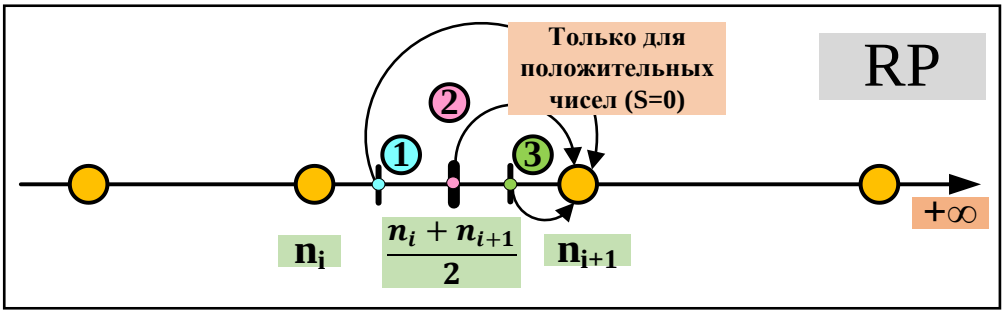


Рис. 20.4 Режим округления в направлении положительной бесконечности

В режиме округления в направлении отрицательной бесконечности RM мантисса любого отрицательного результата с ненулевым остатком инкрементируется на 1. Точные и положительные результаты не округляются. Для положительных чисел это округление эквивалентно просто отбрасыванию остатка – рис. 20.5.

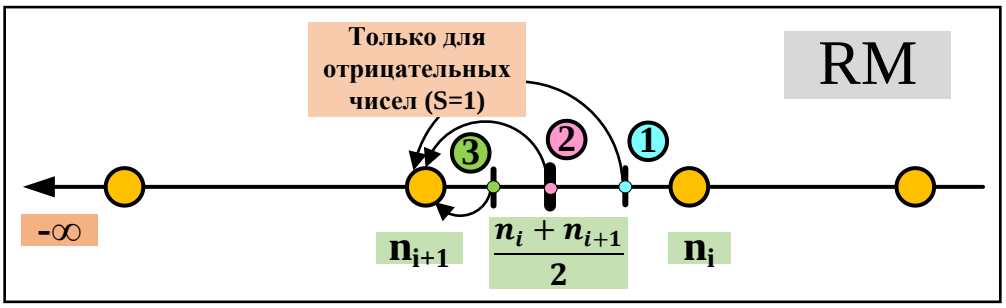


Рис. 20.5 Режим округления в направлении отрицательной бесконечности

Режим округления RZ эквивалентен режиму усечения – остаток просто отбрасывается. При этом любые неточные числа заменяются репрезентабельными числами, максимально приближенными к нулю – рис. 20.6.

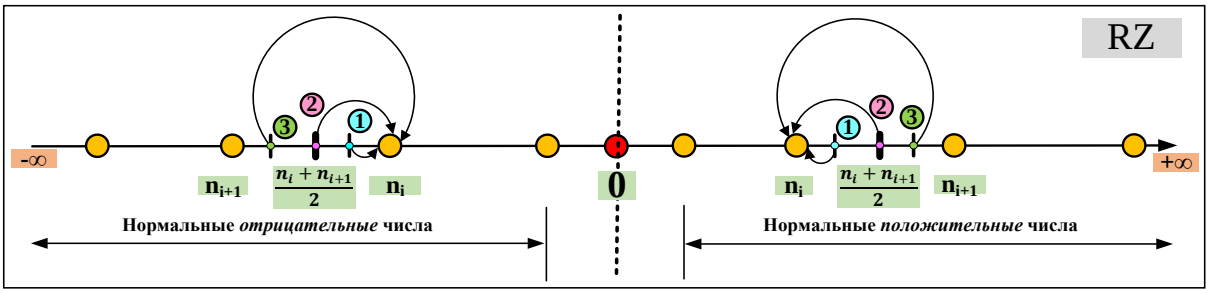


Рис. 20.6 Режим округления RZ

Для большинства применений используйте режим округления к ближайшему репрезентабельному числу RN. Остальные режимы применяйте для оценки возможных погрешностей расчетов. Так, если выполнять вычисления для некоторой задачи, округляя результаты все время в меньшую сторону, а затем – в большую сторону, можно оценить

диапазон возможных значений результата, связанный с ограниченной точностью представления данных в процессоре с текущей разрядной сеткой. Это позволит или обосновать необходимость перехода к вычислениям с плавающей точкой двойной точности, или наоборот, подтвердить достаточную точность вычислений в формате чисел с одинарной точностью. Понятно, что цена вопроса – стоимость микроконтроллера с соответствующими возможностями и всей системы на его основе.

20.5.4 Флаги исключений

В начальных битах регистра FPSCR расположены *флаги кумулятивных (накопительных) исключений*. Это флаги, которые могут сопровождать *возможно некорректно выполненные операции* над числами в формате с плавающей точкой. Судить о корректности или некорректности соответствующей операции может только программист, знающий специфику своей задачи. Поэтому сопроцессор работает следующим образом:

1) Выставляет флаг возможной некорректности и «защелкивет» его – фиксирует. Такие флаги получили название «флагов-липучек». Состояние «флага-липучки» остается неизменным даже при дальнейшем выполнении программы – запоминается.
2) Если обработка исключения по «Ошибке использования» Usage Fault запрещена, то процессор просто продолжает выполнение программы.
3) Если обработка исключения Usage Fault разрешена, то выполняется передача управления соответствующей подпрограмме-обработчику исключения, в которой принимается решение остановить или продолжить ход выполнения программы и, если продолжить, то какой результат некорректной операции вернуть.
4) Если обработка исключений запрещена, то работа программы продолжается несмотря на установленный флаг возможной некорректности.
5) После завершения выполнения определенного модуля программист может проверить наличие флагов возможной некорректности и, либо заблокировать выдачу результатов расчета (в частности, управляющих воздействий), либо – нет.

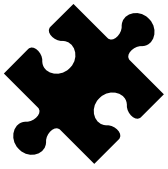
Мы уже отмечали, что особенность обработки чисел в формате с плавающей точкой состоит в том, что выход за допустимый диапазон представления чисел *автоматически ограничивается*. Так, при выходе за диапазон вверх (переполнение вверх), результату может быть присвоено значение +∞ или -∞. Более того, при этом расчеты регулятора или цифрового фильтра могут быть продолжены, причем без каких-либо серьезных последствий, типа потери устойчивости системы управления или автоколебаний. Это связано с технологией «автонасыщения» и «автоограничения» результата, присущей арифметике чисел с плавающей точкой.

Флаги кумулятивных исключений обладают свойством фиксации, как в триггере-защелке. Один раз возникнув, они не сбрасываются до тех пор, пока не последует обращения к регистру FPSCR по чтению. Это гарантирует, что при выполнении нескольких последовательных операций факт исключения (некорректности выполнения какой-либо операции) будет обнаружен. Наиболее важные условия формирования флагов исключений перечислены в табл. 20.5.

Таблица 20.5 Флаги кумулятивных исключений

Название флага	Условие возникновения
IDC	**Input Denormal – Ввод денормализованного числа. Слишком маленький входной операнд был автоматически округлен до нуля.**
IXC	**Inexact – Неточность. Результат вычисления не мог быть представлен одним из репрезентабельных чисел и был округлен до ближайшего возможного (например, при преобразовании целых чисел в числа с плавающей точкой).**
UFC	**Underflow – Переполнение вниз. Результат операции оказался меньше минимально возможного в формате чисел с плавающей точкой однократной точности, попал в диапазон субнормальных чисел и округлен до репрезентабельного числа в этом диапазоне или до 0.**
OFC	**Overflow – Переполнение вверх. Результат превысил максимально допустимое число в формате с плавающей точкой. Заменен максимально возможным числом или бесконечностью.**
DZC	**Division by Zero – Деление на ноль**
IOC	**Invalid Operation – Недопустимая операция. Корректный результат не может быть получен в принципе. Возвращено «Не число».**

При выполнении операций с числами в формате с плавающей точкой одновременно могут возникнуть и два флага исключения. Так, при переполнении вверх (выходе за диапазон возможных чисел с плавающей точкой в сторону бесконечности) или при переполнении вниз (выходе за диапазон возможных чисел в сторону нуля), если субнормальное число пришлось округлить до репрезентабельного числа, будет возникать и флаг «Неточность».



Флаг наличия исключения может быть непосредственно связан с одной из линий запроса прерывания процессора. В этом случае исключение будет восприниматься как прерывание с переходом на подпрограмму обслуживания прерывания. В ней решается, при каких условиях и как продолжить выполнение программы или прервать ее. Процессор имеет шесть выходных контактов FPIXC, FPUFC, FPOFC, FPDZC, FPIDC, каждый из которых отражает состояние одного из флагов исключений.

20.6 Исключение «Деление на ноль» - Division by Zero

Флаг DZC возникает, если делитель является нулем (+0) или (-0), а делимое является нормальным или субнормальным числом. При этом возвращается результат равный +∞ (если знаки делимого и делителя одинаковы) или -∞ (если знаки разные).

Если и делимое, и делитель оба равны нулю *одновременно*, то результат операции будет *неопределенным*. Как всегда, в таких случаях формируется флаг IOC «Недопустимой операции», и в регистр назначения в качестве результата записывается «Не число» NaN.

20.7 Исключение «Недопустимая операция» – Invalid Operation

Имеется несколько условий, при которых формируется флаг исключения «Недопустимая операция» IOC. В этом случае всегда с математической точки зрения результат операции не может быть вычислен – он *неопределен*, как в операции деления нуля на ноль:

1. Арифметические операции, операндами которых являются «Сигнализирующие Не числа» (sNaN). Мы уже отмечали, что это специальный формат представления «Не числа», который принудительно может использовать программист, заменяя им результат любой из «подозрительных» операций с плавающей точкой, которую он решил проверить на корректность. После такой замены любая последующая команда с операндом sNaN вызовет генерацию исключения IOC (Недопустимая операция), возможно с переходом к процедуре обслуживания исключения (если обработка исключений разрешена).
2. Если один или оба операнда арифметической операции или операции сравнения являются «Не числами» NaN (любого типа), то такая операция не может быть выполнена, – выставляется флаг IOC. Это правило не распространяется на операции пересылки данных. «Не числа» могут нести дополнительную символьную информацию, которая после пересылки данных может быть выделена и нужным образом интерпретирована.
3. Арифметические операции, результат которых не может быть определен, такие как
    a. Сложение бесконечностей разного знака: (+∞) + (-∞);
    b. Вычитание бесконечностей одного знака: (+∞) - (+∞);
    c. Умножение бесконечности на ноль: (±∞)*(±0);
    d. Деление нуля на ноль: (±0)/(±0);
    e. Деление бесконечности на бесконечность: (±∞)/(±∞);
    f. Извлечение квадратного корня из отрицательного числа.
4. Операции преобразования форматов чисел, если исходный операнд является «Не числом».
5. Некоторые операции преобразования форматов чисел. Например, преобразования числа в формате с фиксированной точкой F32 в целочисленный формат S32, когда исходное число выходит за диапазон возможных целых чисел со знаком (слишком большое). Так, число 5.0×10^{15} существенно больше максимально возможного целого положительного числа $+(2^{31}-1) = 2.1475\times10^{9}$. Преобразование невозможно. В этом случае в регистр назначения записывается самое большое возможное целое число и формируются два флага исключения: «Недопустимая операция» (IOC) и «Неточность» (IXC). Флаг «Переполнения» (OFC) в таких случаях не формируется, так как операция преобразования формата не относится к арифметическим операциям.

В случае любой «Недопустимой операции» в регистр назначения команды записывается «Не число по умолчанию» – Default NaN. Создатели Cortex-M4F в качестве «Не числа по умолчанию» выбрали так называемое «Тихое Не число» (Quiet NaN – qNaN) – *см. 19.5*.

20.8 Исключение «Переполнение вверх» – Overflow

Данное исключение возникает в том случае, когда результат арифметической операции по абсолютному значению превышает максимально возможное значение в рамках формата числа с плавающей точкой и не может быть представлен. Переполнение возможно как при положительном, так и при отрицательном результате.

При возникновении «Переполнения вверх» устанавливаются сразу два флага: флаг переполнения (OFC) и флаг неточности (IXC). Последний свидетельствует о том, что результат представлен слишком «грубо».

Для режима округления до ближайшего репрезентабельного числа RN в качестве результата возвращается либо положительная, либо отрицательная бесконечность. Для других режимов округления возвращаемое значение зависит от знака числа – табл. 20.6.

Таблица 20.6 Значения, возвращаемые при «Переполнении вверх» Overflow

Режим округления	**При положительном результате**	**При отрицательном результате**
RNE	$+\infty$	$-\infty$
RP	$+\infty$	$-MAX_{normal}$
RM	MAX_{normal}	$-\infty$
RZ	MAX_{normal}	$-MAX_{normal}$

Так, при режимах округления в сторону минус бесконечности или нуля и положительном результате вместо $+\infty$ возвращается MAX_{normal} – максимально возможное нормальное число (минимальное из двух возможных значений в рамках формата чисел однократной точности MAX_{normal} и $+\infty$).

С точки зрения качества алгоритмов управления реального времени замена очень больших чисел на бесконечность или максимально возможное нормальное число аналогична операции «насыщения», которая исключительно важна в цифровой обработке сигналов. Если «насыщения» нет, то при работе с числами в форматах с фиксированной точкой возможна резкая смена знака управляющего воздействия и потеря устойчивости системы управления, что исключается при работе с числами в формате с плавающей точкой.

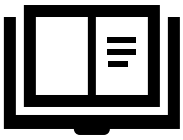
1) Выполняется операция деления (+0/+0) или (-0/+0). Какой флаг исключения будет выработан?
2) Будет ли этом случае обязательно остановлено выполнение программы?
3) Являются ли флаги исключений фатальными ошибками?
4) Какой из флагов исключений является наиболее «опасным» и его желательно проверять?
5) Почему программы управления, реализованные в формате чисел с плавающей точкой, часто называют «интеллектуальными», «разумными», «робастными»?

1) «Недопустимой операции».
2) Нет. Флаги исключений лишь информируют о том, что результат, возможно, некорректный. Дело программиста – обращать на их возникновение внимание или нет. Так, при делении на ноль возможен возврат особого числа – положительной или отрицательной бесконечности. При этом расчет может быть продолжен. Операции с бесконечностями допускаются стандартом.
3) Нет. При их появлении сопроцессор возвращает значение «по умолчанию», и вычислительный процесс может быть продолжен. Так, при «Переполнении вверх», будет возвращен результат в виде положительной или отрицательной бесконечности, что полностью соответствует смыслу – результат вышел за максимально допустимое значение числа с плавающей точкой. Это может быть, например, при неправильном

наборе входных данных. В этом случае коррекция алгоритма может и не потребоваться. Сопроцессор выполнит как бы автоограничение.

4) Флаг «Недопустимая операция». Он может потребовать коррекции программы, алгоритма или диапазона допустимых входных данных.
5) При выходе за допустимый диапазон вычисления могут быть продолжены, и может быть получен если не точный, то, по крайней мере, оценочный выходной результат, не приводящий к катастрофическим сбоям, таким, например, как потеря устойчивости системы. Робастность (robust) предполагает нечувствительность системы к широкому диапазону изменения переменных или параметров. Программа может оказаться работоспособной в широком диапазоне изменения переменных.

20.9 Исключение «Переполнение вниз» - Underflow



Операции с числами в формате с плавающей точкой в модуле FPU процессоров Cortex-M4F в соответствии со стандартом IEEE 754 выполняются не только с нормальными числами, но с субнормальными (денормализованными). Фактически шкала чисел с плавающей точкой в зоне нуля является непрерывный. При этом возможны следующие ситуации:

1) Результат расчета является репрезентабельным (представимым) субнормальным числом, то есть определен точно (знак результата значения не имеет). В этом случае флаг переполнения вниз UFC не вырабатывается. Полученный результат используется в дальнейших расчетах без каких-либо ограничений.
2) Результат расчета оказался субнормальным числом и не может быть представлен точно (округляется до ближайшего репрезентабельного субнормального числа): одновременно выставляются оба флага UFC (Переполнение вниз) и IXC (Неточность).
3) Если результат операции оказался столь мал, что вышел вниз даже за диапазон субнормальных чисел, то он округляется до нуля и выставляются оба флага UFC и IXC.

Итак, исключение «Переполнение вниз» возникает только тогда, когда результат операции оказался в субнормальном диапазоне и является *неточным*. При этом выставляются сразу оба флага UFC и IXC.

В некоторых микропроцессорных системах результаты расчетов, попадающие в субнормальный диапазон, автоматически округляются до чистого нуля со знаком. В Cortex-M4F *гарантируется корректная работа и с очень маленькими числами*. Числовая шкала становится непрерывной от -∞ до +∞. Это очень важное преимущество для систем управления реального времени, так как исключает резкие изменения управляющих воздействий, связанные с дискретным переходом от одного поддиапазону чисел к другому.

При необходимости, в процессорах Cortex-M4F Вы можете полностью исключить поддиапазон субнормальных чисел из числовой шкалы, установив режим работы сопроцессора Flush-to-zero (Округление к нулю). При этом все субнормальные числа будут автоматически округляться до знаковых нулей. Это позволит сравнить точность расчетов для Вашего приложения в двух случаях: с использованием поддиапазона субнормальных чисел и без его использования.

1) Можно ли понимать исключение «Переполнение вниз» как потерю значимости результата операции?
2) Можно ли в этом случае продолжать выполнение расчета?
3) Может ли в режиме сопроцессора Flush-to-zero (Округление к нулю) появиться опасность возникновения автоколебаний в системе управления?

1) Да, в том смысле, что результат вышел вниз за пределы допустимых нормальных чисел в формате с плавающей точкой. Он стал либо субнормальным числом, либо нулем.
2) Да, субнормальные числа могут использоваться в качестве операндов во всех командах сопроцессора.
3) Да, из-за нарушения «непрерывности» числовой шкалы.

20.10 Условия совместимости модуля FPU со стандартом IEEE 754 и ограничения

Если флаги трех специальных режимов работы сопроцессора (формата половинной точности AHP, режима «Не чисел по умолчанию» DN и «Округления к нулю малых чисел» FZ сброшены, то модуль FPU функционально полностью соответствует стандарту IEEE 754 на аппаратном уровне в части обработки 32-разрядных чисел с плавающей точкой однократной точности.

Некоторые операции, описанные в стандарте IEEE 754-2008, не поддерживаются:

- Вычисления с числами с плавающей точкой двойной точности. Их обработка возможна с помощью специальных библиотек.
- Получение остатка (Remainder).
- Округление числа с плавающей точкой к целочисленному числу с плавающей точкой (Round floating-point number to integer-valued floating-point number).
- Преобразование двоичного числа в десятичное (Binary-to-decimal conversions).
- Прямое сравнение чисел с плавающей точкой однократной и двукратной точности (Direct comparison of single-precision and double-precision values).

Отметим особо, что специальный тип очень эффективных операций умножения с накоплением, особенно нужных для реализации систем цифрового регулирования и фильтрации, когда оба действия (умножение и накопление) выполняются одновременно (fused MAC), поддерживается и полностью соответствует стандарту.

20.11 Время выполнения команд сопроцессора

Процессоры ARM-Cortex-M4F являются RISC-процессорами, в которых время выполнения большинства команд фиксировано – один цикл. При реализации аппаратной части сопроцессора разработчики стремились к тому, чтобы команды сопроцессора также выполнялись за один цикл. Однако, сложность обработки чисел в формате с плавающей точкой не позволила достичь предельного быстродействия для всех команд. Для выполнения некоторых сложных команд требуется несколько циклов – табл. 20.7.

Таблица 20.7 Время выполнения операций в модуле плавающей точки FPU

Типы операций	Время выполнения, циклов
Сложение, вычитание, отрицание, умножение, умножение с отрицанием, сравнение, получение абсолютного значения числа, конвертирование между целыми числами или числами с фиксированной точкой и числами с плавающей точкой, пересылка между регистрами ЦПУ и регистрами сопроцессора, загрузка непосредственных констант.	**1**
Загрузка регистров сопроцессора из памяти и сохранение в памяти	**2**
Умножение с накоплением нескольких типов	**3**
Деление	**14**
Извлечение квадратного корня	**14**

Большинство команд сопроцессора выполняется за один цикл. Важная для разработчиков систем цифрового управления и цифровой фильтрации группа команд умножения с накоплением выполняется всего за 3 цикла. И только сложные операции деления и извлечения квадратного корня требуют 14-ти процессорных циклов.

20.12 Как разрешить работу модуля плавающей точки?

Сразу после сброса процессора работа модуля FPU запрещена (регистр CPACR обнулен). Поэтому, прежде чем использовать команды обработки чисел с плавающей точкой, необходимо «включить» модуль FPU в работу. Обычно это делается в процедуре начальной инициализации процессора после подачи питания или сброса Reset. Ниже приводится фрагмент кода, который выполняет это подключение как в привилегированном (privileged), так и в пользовательском (user) режиме работы процессора. Сразу после включения процессор переводится в привилегированный режим работы. Доступ ко всем регистрам управления ресурсами процессора возможен только в этом режиме.

Разрешить работу модуля плавающей точки FPU:

```
; Регистр CPACR находится по адресу 0xE000ED88 (см. табл. 20.1)
; Загрузить этот адрес в регистр R0 (псевдо-команда)
    LDR.W R0, =0xE000ED88
; Считать текущее содержимое регистра CPACR в R1
; с использованием косвенной адресации по адресу в R0
    LDR R1, [R0]
; Установить биты 23-20 для разрешения полного доступа к
; сопроцессорам CP10 и CP11 (именно они являются модулем FPU)
    ORR R1, R1, #(0xF << 20)
; Записать модифицированное значение обратно в CPACR
    STR R1, [R0]
; Ожидание до полного завершения операции записи в память
    DSB
; Очистить («смыть») конвейер команд (на всякий случай)
    ISB
; Начиная с этого места, работа модуля FPU возможна
; Будет выполняться следующая после ISB команда
```

Дадим некоторые комментарии по поводу команд DSB и ISB. Команда DSB устанавливает так называемый *барьер синхронизации данных*. Никакие последующие команды доступа к памяти не будут выполняться до тех пор, пока предыдущие команды доступа к памяти не будут полностью завершены. В нашем случае: пока не будет полностью выполнена команда записи в специальный регистр управления подключением сопроцессоров CPACR.

Команда ISB устанавливает так называемый *барьер синхронизации инструкций*. Все ранее попавшие на конвейер команды «смываются» с конвейера, и начинается новая выборка команд из памяти, начиная с той команды, которая в программе расположена сразу за командой ISB. Если на конвейере ранее были какие-то команды, то их выполнение отменяется. Это гарантия того, что любая команда, даже если это будет команда обработки данных в сопроцессоре, будет выполнена корректно.

20.13 Как обрабатываются «Не числа» NaN

Ниже приводится дополнительная информация, поясняющая технологию обработки так называемых «Не чисел». Напомним (см. главу 19), что операнды в формате плавающей точки однократной точности, которые имеют максимальное значение экспоненты +128 и нулевую мантиссу, считаются положительной или отрицательной бесконечностью (в зависимости от содержимого знакового разряда). Если экспонента имеет максимальное значение +128, а мантисса является ненулевой, то все такие коды считаются «Не числами» (NaN). Среди множества «Не чисел» выделяют так называемые «Сигнализирующие не числа» – Signaling NaN (sNaN), которые в старшем значащем бите мантиссы (точнее ее дробной части) имеют ноль.

Таким образом, различают два типа «Не чисел», которые имеют разное значение самого старшего 22-го бита мантиссы – табл. 20.8. «Не число по умолчанию» Default NaN отличается тем, что оно имеет единичный старший бит мантиссы (22-й разряд) и все остальные разряды, равные нулю. «Сигнализирующее не число», напротив, имеет нулевой старший бит мантиссы и произвольное значение всех остальных разрядов.

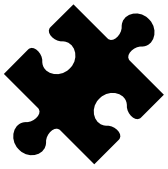
Таблица 20.8 Представление «Не чисел» в модуле плавающей точки FPU

Знак (S) **31**	**Смещенная экспонента** **30 23**	**Мантисса (Только дробная часть)** **22**	**0**	**Что представляет**
0	**0xFF**	**1**	**00000… 0**	**Default NaN Quiet NaN (qNaN)**
0	**0xFF**	**0**	**xxxxx… x**	**Signaling NaN (sNaN)**

Обработка входных операндов, которые являются не числами (NaN), в модуле FPU и специализированных библиотеках может быть двух типов:

- В режиме ***полной совместимости full-compliance mode*** в соответствии со стандартом IEEE 754:
  - Для операций пересылки данных «Не числа» (sNaN) перемещаются без каких-либо изменений и без генерации флага исключения по «Недопустимой операции» Invalid Operation (IOC).
  - В операциях с одним операндом, таких как VABS (абсолютное значение), VNEG (отрицание) и VMOV (пересылка), «Не числа» sNaN просто копируется и, если это требуется в команде (абсолютное значение,

отрицание), изменяется знаковый разряд. При этом генерации флага исключения по «Недопустимой операции» IOC не происходит.
    - Во всех арифметических операциях наличие «Не числа» в любом из операндов вызывает генерацию исключения «Недопустимая операция» и возврат в качестве результата «Не числа по умолчанию».
- Если установлен ***режим генерации «Не числа по умолчанию» Default NaN***, то:
    - *любая операция* с участием операнда, представляющего собой «Не число» NaN любого типа будет возвращать в качестве результата так называемое «Не число по умолчанию» Default NaN, независимо от значения дробной части мантиссы в любом из исходных операндов «Не чисел» NaN.
    - Сигнализирующие «Не числа» sNaNs в арифметических операциях будут вызывать установку флага IOC (FPSCR[0]) – «Недопустимая операция».



Основное отличие «Сигнализирующих не чисел» (signaling NaN) от «Не чисел по умолчанию» Default NaN («Тихих не чисел» Quiet NaNs) состоит в том, что получение «Сигнализирующего не числа» в качестве операнда арифметической операции всегда будет вызывать генерацию флага исключения по недопустимой операции IOC. Сам операнд сигнализирует о необходимости формирования флага исключения.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User's Manual. Texas Instruments Inc. – 2011.
3) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
4) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
5) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

21 КОМАНДЫ РАБОТЫ С ЧИСЛАМИ В ФОРМАТЕ С ПЛАВАЮЩЕЙ ТОЧКОЙ

Оглавление

21.1 Основные соглашения

Напомним, что командами сопроцессора можно пользоваться только при выполнении двух условий:

1) Модуль обработки чисел в формате с плавающей точкой FPU физически присутствует в составе процессорного ядра (используется микроконтроллер с ядром Cortex-M4F.
2) Выполнена программная инициализация режима работы модуля FPU, в том числе дано разрешение на его работу.

Ранее мы отмечали, что синтаксис всех команд, в том числе команд сопроцессора, унифицирован (см. главу 7). Тем не менее, команды поддержки операций с плавающей точкой имеют некоторые особенности:

1) Сопроцессор может обрабатывать только числа, расположенные в его регистрах окружения S0 – S31 или D0 – D15 (в регистровом файле сопроцессора). Регистры S0-S31 – это 32-разрядные регистры однократной точности, к которым при необходимости можно обратиться как к паре регистров, представляющих собой один 64-разрядный регистр двойной точности или два регистра однократной точности.
2) Обращения к регистру двойной точности по именам D[n] или S[2n], где n – номер регистра, – эквивалентны. Поддерживаются только операции загрузки/сохранения данных в регистры двойной точности. Операции обработки чисел с плавающей точкой двойной точности аппаратно не поддерживаются.
3) Данные в регистрах двойной точности располагаются аналогично данным в памяти: младшее 32-разрядное слово в регистре S[2n], а старшее – в регистре S[2n+1].
4) Каждый из регистров однократной точности может использоваться в качестве регистра-источника данных Sn или регистра-приемника данных Sd без каких-либо ограничений, как это было для регистров общего назначения ЦПУ: ограничения накладывались на использование регистров R13 (LR), R14 (SP) и R15 (PC).
5) Все регистры S0 – S31 сопроцессора могут использоваться в качестве регистров временного хранения данных и при обычных вычислениях в ЦПУ.
6) В арифметических операциях над числами с плавающей точкой используются регистр-приемник Sd и два регистра-источника, первый Sn и второй – Sm. Именно в таком порядке имена регистров указываются в поле операндов команды: Sd, Sn, Sm.
7) Если имя регистра-приемника Sd опущено, то им является первый регистр-источник Sn.
8) Все команды сопроцессора в мнемонике команды имеют префикс V.
9) Все команды сопроцессора в мнемокоде могут иметь опциональный суффикс {.F32}, {.F64} или {.32}, {.64}, специфицирующий размер обрабатываемых данных в битах {.size}. Опускать этот суффикс *не рекомендуется*.

10) В командах преобразования форматов чисел используются одновременно два суффикса, первый – специфицирует формат регистра-приемника данных, а второй – формат регистра-источника данных, например: «.F32.U32» – преобразовать целое число без знака U32 в число с плавающей точкой однократной точности; «.S32.F32» – преобразовать число с плавающей точкой однократной точности в 32-разрядное целое со знаком в дополнительном коде.
11) В командах преобразования форматов чисел и исходное число, и результат его преобразования должны находиться в одном из регистров окружения сопроцессора.
12) Никакие команды сопроцессора, кроме команд сравнения, не модифицируют флаги в регистре статуса и управления сопроцессора FPSCR.
13) Все команды обработки чисел с плавающей точкой могут быть условно-выполняемыми – иметь суффикс cond. Коды условий становятся актуальными только после операции сравнения чисел с плавающей точкой и копирования флагов из регистра статуса сопроцессора FPSCR в регистр статуса программы приложения APSR.
14) Флаги исключительных ситуаций формируются только арифметическими командами и командами преобразования форматов чисел. Команды загрузки/сохранения просто копируют данные «как есть», не вырабатывая флагов исключений.

21.2 Путеводитель по командам работы с числами в формате с плавающей точкой

В табл. 21.1 дана классификация команд работы с числами в формате с плавающей точкой. Вы можете использовать ее в качестве справочника для выбора нужной команды. Далее в книге дается краткая характеристика команд каждой группы с примерами применения. Полное описание команд с указанием ограничений по их использованию и примерами находится в электронном приложении (см. приложение 1, «Команды модуля FPU»).

В табл. 21.1 не приведены команды преобразования форматов чисел с плавающей точкой однократной точности в формат половинной точности и обратно, из-за их ограниченного применения на практике.

Дополнительно в путеводитель включены псевдокоманды Ассемблера, обеспечивающие эффективную загрузку данных в регистры сопроцессора. Использование в них суффикса, определяющего формат «.F32» или «.F64», *обязательно*.

Для удобства читателей применяется цветовое выделение операндов команд, суффиксов и префиксов, такое же, как и во всей книге.

Таблица 21.1 Путеводитель по командам работы с числами в формате с плавающей точкой

Мнемоника	Операнды	Краткое описание	Фл	Ис
Команда загрузки в регистры модуля FPU непосредственных операндов однок. точности FS				
VMOV{cond}.F32	Sd, #imm	из оптокода команды	-	-
Команды загрузки в регистры модуля FPU операндов однократной точности FS из памяти				
VLDR{cond}{.F32} VLDR{cond}{.F32}	Sd, [Rn {,#imm}] Sd, [PC, #imm]	из памяти (базовая ад. со смещ.) из памяти (относ. ад. по сч. ком.)	- -	- -
Псевдо-команды Ассемблера для загрузки непосредственных операндов однок. точности FS				
VLDR{cond}.F32	Sd, label	из памяти (по метке)	-	-
VLDR{cond}.F32	Sd,=Constant	из «литерального пула»	-	-
Команды загрузки в регистры модуля FPU операндов двукратой точности FD из памяти				
VLDR{cond}{.F64} VLDR{cond}{.F64}	Dd, [Rn {,#imm}] Dd, [PC, #imm]	из памяти (базовая ад. со смещ.) из памяти (относ. ад. по сч. ком.)	- -	- -
Псевдо-команды Ассемблера для загрузки непосредственных операндов двукрат. точности FD				
VLDR{cond}.F64	Dd, label	из памяти (по метке)	-	-
VLDR{cond}.F64	Dd,= Constant	из «литерального пула»	-	-
Команда множественной загрузки нескольких регистров S или D из памяти				
VLDM{mode}{cond}{.size}	Rn{!}, list	Mode = IA – инкрементировать адрес после доступа к данным; Mode = DB – декрементировать адрес перед доступом к данным; ! – обновить базовый регистр после множественного доступа	-	-
Команда множественного сохранения содержимого нескольких регистров S или D в памяти				
VSTM{mode}{cond}{.size}	Rn{!}, list	Mode = IA – инкрементировать адрес после доступа к данным; Mode = DB – декрементировать адрес перед доступом к данным; ! – обновить базовый регистр после множественного доступа	-	-
Псевдо-команды Ассемблера для однократной загрузки регистров S или D в режиме пост-инкрементирования и пре-декрементирования базового адреса (кодируются как VLDM)				
VLDR{cond}{.size} VLDR{cond}{.size}	Fd,[Rn],#offset Fd,[Rn, #-offset]!	Пост-инкрементная Пре-декрементная Fd = Sd|Dd	-	-
Псевдо-команды Ассемблера для однократного сохранения регистров S или D в режиме пост-инкрементирования и пре-декрементирования базового адреса (кодируются как VSTM)				
VSTR{cond}{.size} VSTR{cond}{.size}	Fd,[Rn],#offset Fd,[Rn, #-offset]!	Пост-инкрементная Пре-декрементная	-	-
Команды пересылки данных между регистрами сопроцессора однокр. и двойной точности				
VMOV{cond}.F32	Sd, Sm	(Sd) ← (Sm)	-	-
VMOV{cond}.F64	Dd, Dm	(Dd) ← (Dm)	-	-
Команды пересылки данных между регистрами сопроцессора однократной точности и ЦПУ				
VMOV{cond}	Sd, Rm	(Sd) ← (Rm)	-	-
VMOV{cond}	Rd, Sm	(Rd) ← (Sm)	-	-
Команды двойной пересылки данных между регистрами сопроцессора однок. точности и ЦПУ				
VMOV{cond}	Sd, Sd1, Rn, Rm	(Sd) ← (Rn); (Sd1) ← (Rm);	-	-
VMOV{cond}	Rd1, Rd2, Sm, Sm1	(Rd1) ← (Sm); (Rd2) ← (Sm1);	-	-
Команды копирования одного из слов регистра сопроцессора двойной точности в регистр ЦПУ				
VMOV{cond}	Rd, Dm[x]	X=0 : (Rd) ← (Dm[0]); X=1 : (Rd) ← (Dm[1]);	-	-
Команды пересылки данных из регистра статуса и управления сопроцессором в регистры ЦПУ и обратно (инициализация, смена режима работы FPU)				
VMRS{cond} VMRS{cond} VMSR{cond}	Rd, FPSCR APSR_nzcv, FPSCR FPSCR, Rm	(Rd) ← (FPSCR); (APSR_nzcv) ← (FPSCR); (FPSCR) ← (Rm); Иниц.-я FPU	-	-

Команды множественного сохранения/восстановления содержимого регистров сопроцессора в стеке/из стека				
VPUSH{cond}{.size}	list	(Стек) ← (Fi)…(Fk)	-	-
VPOP{cond}{.size}	list	(Fk)…(Fi) ← (Стек)	-	-
Команды обработки одного операнда с плавающей точкой				
VABS {cond}.F32 VNEG {cond}.F32 VSQRT{cond}.F32	Sd, Sm Sd, Sm Sd, Sm	(Sd) ← Abs (Sm) - Абсол. значен. (Sd) ← - (Sm) - Инверсия (Sd) ← √(Sm) - Квадр. корень	- - -	- - Да
Арифметические операции с двумя операндами с плавающей точкой				
VADD{cond}.F32 VSUB{cond}.F32 VMUL{cond}.F32 VDIV{cond}.F32	{Sd,} Sn, Sm {Sd,} Sn, Sm {Sd,} Sn, Sm {Sd,} Sn, Sm	(Sd) ← (Sn) + (Sm) - Сложение (Sd) ← (Sn) - (Sm) - Вычитание (Sd) ← (Sn) • (Sm) - Умножение (Sd) ← (Sn) / (Sm) - Деление	- - - -	- - - Да
Команды умножения с округлением произведения, накоплением и вторым округлением				
VMLA{cond}.F32 VMLS{cond}.F32	{Sd,} Sn, Sm {Sd,} Sn, Sm		- -	Да Да
Команды умножения с инверсией знака произведения и накоплением				
VNMLA{cond}.F32 VNMLS{cond}.F32	{Sd,} Sn, Sm {Sd,} Sn, Sm		- -	Да Да
Команды умножения, совмещенного с накоплением, округлением суммы/разности				
VFMA{cond}.F32 VFMS{cond}.F32	{Sd,} Sn, Sm {Sd,} Sn, Sm		- -	Да Да
Команды умножения, совмещенного с накоплением, округлением суммы/разности и инверсией				
VFNMA{cond}.F32 VFNMS{cond}.F32	{Sd,} Sn, Sm {Sd,} Sn, Sm		- -	Да Да
Команды сравнения двух операндов с плавающей точкой				
VCMP{cond}.F32 VCMP{cond}.F32	Sn, Sm Sn, #0.0	(Sn) - (Sm) - Сравнить (Sn) - #0.0 - Сравнить с нулем	Да, в FPSCR	-
Команды сравнения двух операндов с проверкой допустимости форматов операндов				
VCMP{E}{cond}.F32 VCMP{E}{cond}.F32	Sn, Sm Sn, #0.0		Да, в FPSCR	Да
Преобразование целого 32-разрядного числа в формат числа с плавающей точкой				
VCVT{cond}.F32.S32 VCVT{cond}.F32.U32	Sd, Sm Sd, Sm	[F32 в (Sd)] ← [S32 из (Sm)] [F32 в (Sd)] ← [U32 из (Sm)]	-	Да
Преобразование числа в формате с плавающей точкой в целое 32-разрядное число				
VCVT{R}{cond}.S32.F32 VCVT{R}{cond}.U32.F32	Sd, Sm Sd, Sm	[S32 в (Sd)]← [F32 из (Sm)] [U32 в (Sd)]← [F32 из (Sm)] R – использовать режим округления модуля FPU вместо усечения	-	Да
Преобразование числа с фиксированной точкой в число с плавающей точкой				
VCVT{cond}.F32.Td Td = S32, U32, S16, U16	Sd, Sm, #fbits	[F32 в (Sd)] ← [Td из (Sm)] #fbits = Q (для I.Q)	-	Да
Преобразование числа с плавающей точкой в число с фиксированной точкой				
VCVT{cond}.Td.F32 Td = S32, U32, S16, U16	Sd, Sm, #fbits	[Td в (Sd)]← [F32 из (Sm)] #fbits = Q (для I.Q)	-	Да

21.3 Загрузка регистров модуля FPU исходными данными

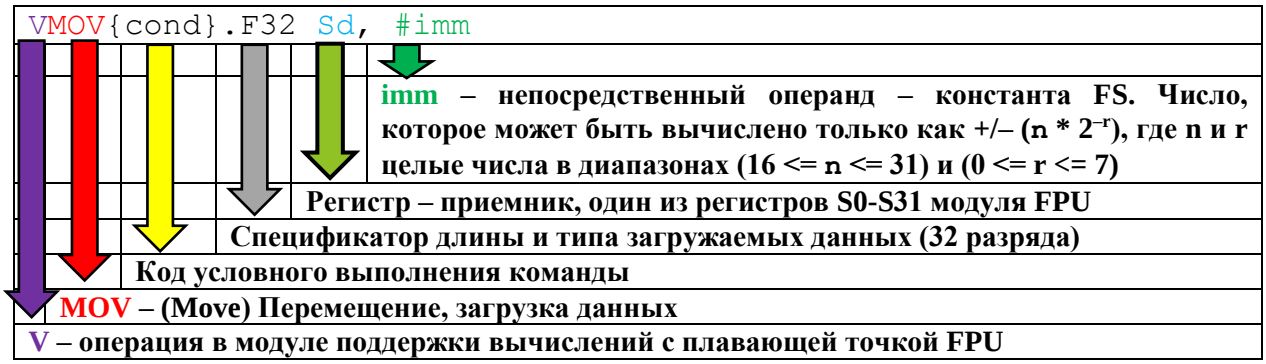
Обработка чисел с плавающей точкой в сопроцессоре возможна только в том случае, когда они загружены в регистры окружения сопроцессора. Технология загрузки данных в регистры модуля FPU подобна загрузке данных в регистры ЦПУ. Поддерживаются почти все возможности, которые были рассмотрены ранее применительно к регистрам ЦПУ (см. главу 10).

21.3.1 Команда непосредственной загрузки

Регистры сопроцессора 32-разрядные, как и команды. Следовательно, по-прежнему возникает проблема: как в формате 32-разрядной команды разместить 32-разрядные данные – числа с плавающей точкой однократной точности?

Очевидно, что в общем случае это невозможно. Разработчики Cortex-M4F предложили свою технологию сжатия констант в формате с плавающей точкой, которая не является универсальной, но может быть использована для некоторых наиболее часто используемых констант.

Если вводимая Вами константа не входит в число допустимых констант (не может быть упакована в короткий формат), размещаемый в самой команде, то транслятор с Ассемблера выведет на экран предупреждение об ошибке. Придется воспользоваться уже известным нам методом размещения констант в кодовой памяти в так называемых «литеральных пулах» с помощью псевдокоманды загрузки VLDR Sd,=Const (см. далее), которая будет заменена транслятором на команду относительной адресации по счетчику команд и рассчитанному смещению до места расположения константы. Синтаксис команды непосредственной загрузки VMOV:



Примеры допустимых для ввода непосредственных констант в формате F32 (FS):

При n=16: ±16.0; ±8.0; ±4.0; … ±0.5; ±0.25; ±0.125
При n=31: ± 31; ±15.5; ±7.75; ±3.875; ±1.9375; ±0.96875; …

21.3.2 Загрузка регистров сопроцессора из памяти

Если исходные данные для обработки в сопроцессоре находятся в памяти (оперативной или кодовой), то они могут быть загружены как в регистры однократной точности (S0-S31), так и в регистры двукратной точности (D0-D15).

Данные могут быть либо предварительно размещены в памяти с помощью специальных директив Ассемблера, либо считаны из портов ввода или периферийных устройств и преобразованы в формат чисел F32 (как это делается, покажем далее).

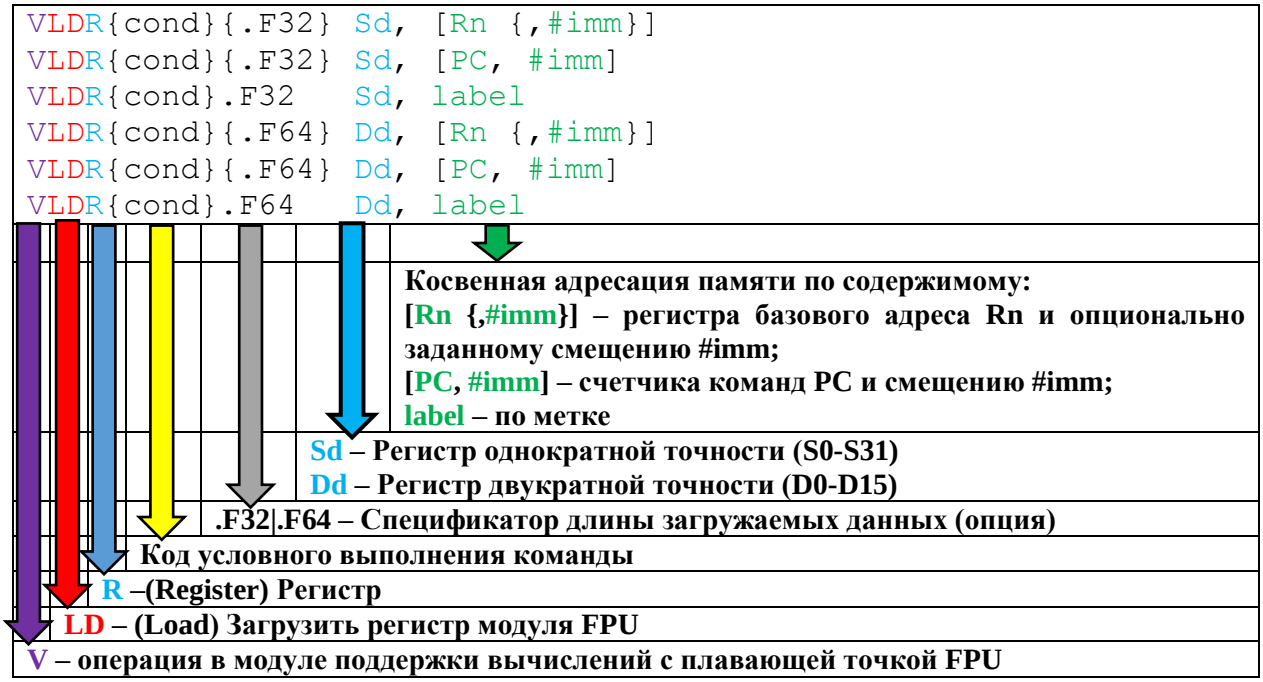
В этих командах можно использовать *обычную косвенную адресацию памяти* по содержимому базового регистра Rn и опционально заданному смещению #imm, которое может быть как положительным, так и отрицательным, а также *относительную адресацию по текущему значению счетчика команд PC и смещению*.

В качестве базового регистра Rn можно использовать любой регистр общего назначения процессора, в том числе указатель стека SP. Величина смещения должна быть кратна 4 (выровнена по слову) и находиться в диапазоне ±1020.

Если данные в памяти имеют имя (метку), то метка может быть указана в качестве операнда – источника данных. В этом случае транслятор с Ассемблера автоматически

рассчитает величину смещения от места расположения текущей команды до метки и сгенерирует команду загрузки регистра из ячейки памяти, адрес которой равен текущему содержимому счетчика команд плюс смещение. Эта технология подробно рассмотрена нами ранеее в 10.3. Метка не может располагаться относительно команды загрузки на расстоянии большем ±1 Кбайт.

Синтаксис команд загрузки регистров сопроцессора из памяти:



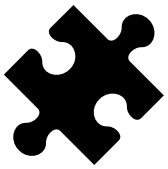
Если в команде специфицируются 64-битовые данные, то они извлекаются из двух последовательно расположенных в памяти 32-разрядных слов. Напомним, что работа с числами с плавающей точкой двойной точности на аппаратном уровне сопроцессора не поддерживается. Для этой цели требуется подключение к проекту одной из специализированных библиотек.

21.3.3 Инициализация памяти константами с плавающей точкой

Напомним (см. 19.7), что константы в формате с плавающей точкой для ввода на Ассемблере и C/C++ записываются в общепринятой нотации десятичного числа с мантиссой и порядком:

№	Формат ввода F-константы	Рекомендуется использовать
1	{-}{цифры}.цифрыE{-}цифры {-}{цифры}.цифрыe{-}цифры	В большинстве случаев
2	0xHex-цифры или &Hex-цифры	Для ввода особых чисел с плавающей точкой (бесконечности, «Не числа»)

Не рекомендуется опускать в числе «десятичную точку», хотя и такая форма записи допустима – в этом случае число до порядка считается целым.



Если Вы копируете какие-либо константы или таблицы констант из компьютерных программ, например, из Электронных таблиц, в программу на Ассемблере, обращайте внимание на разделитель целой и дробной части числа. В ряде случаев для этой цели используется запятая. В Ассемблере разделителем может быть *только точка!*

Имеются две директивы Ассемблера DCFS (Define Code – Определить код) в формате плавающей точки однократной точности (FS – Floating Point Single) и DCFD – определить код в формате плавающей точки двойной точности (FD - Floating Point Double). Константы размещаются в текущей секции, обычно кодовой, начиная с текущего адреса. Примеры:

```
; Резервирование констант однократной точности FS
    DCFS    1.2E-3
    DCFS    0.038
    DCFS    0x7FC00000              ; «Не число» NaN
; Резервирование констант двукратной точности FD
    DCFD    3.725e15
    DCFD    &FFF0000000000000       ; Минус бесконечность
```

21.4 Команды обработки одного операнда в формате с плавающей точкой

Имеются три команды такого типа: получение абсолютного значения числа, инверсии знака числа и извлечения квадратного корня. Исходное число в формате с плавающей точкой однократной точности FS должно находиться в регистре-источнике Sm (S0-S31). Результат операции сохраняется в регистре-приемнике Sd (S0-S31). Опускать имя регистра приемника нельзя. Синтаксис этих команд:

```
VABS  {cond}.F32 Sd, Sm
VNEG  {cond}.F32 Sd, Sm
VSQRT {cond}.F32 Sd, Sm
```

Регистр-источник операнда в формате с плавающей точкой

Регистр – приемник результата операции

F32 – Формат: 32- разрядный с плавающей точкой

Код условного выполнения команды (опция)

ABS – Получение абсолютное значения числа
NEG – Получение отрицательное значения числа
SQRT – Извлечение квадратного корня из числа

V – операция в модуле поддержки вычислений с плавающей точкой FPU

21.5 Учимся отлаживать программы обработки чисел с плавающей точкой в симуляторе µVision

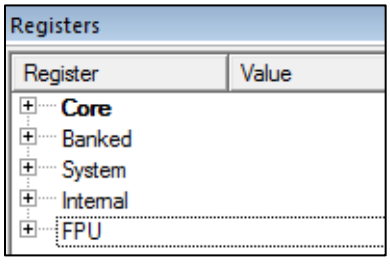
21.5.1 Модернизированный стартовый файл

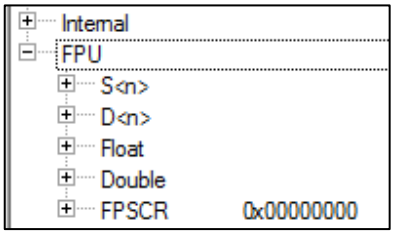
Создадим первый простой проект для отладки программы обработки чисел в формате с плавающей точкой FPU_1. Вы знаете, что прежде чем обращаться к сопроцессору, его необходимо проинициализировать. Если нас устраивают режимы работы «по умолчанию», то нужно только разрешить работу модуля FPU (см. 20.12). Обратимся к проекту FPU_1. Он содержит модифицированный стартовый файл StartUp_2.s: в процедуру инициализации процессора по сбросу включен программный блок, разрешающий работу сопроцессора с настройками «по умолчанию». Сопроцессор будет работать в строгом соответствии со стандартом обработки чисел с плавающей точкой IEEE 754. Сразу после его «активизации» управление будет передано программе пользователя, как это делалось и ранее.

```
61  ; Разрешить работу модуля обработки чисел с плавающей точкой FPU
62  ; в режиме "по умолчанию" (полное соответствие IEEE 754)
63    ; Обращение к регистру управления доступом к сопроцессорам
64    ; по технологии "Чтение- модификация - запись"
65                  ldr R0, =0xE000ED88
66                  ldr R1, [R0]
67    ; Установить биты 23-20 - полный доступ к FPU
68                  orr R1, R1, #(0xF << 20)
69    ; Сохранить новую настройку
70                  str R1, [R0]
71    ; Команды, гарантирующие переинициализацию процессора
72                  dsb
73                  isb
74  ; Передать управление пользовательской программе MyProg
75  ; Объявление точки входа в программу пользователя
76  ; (внешняя метка)
77                  IMPORT  MyProg
78  ; Псевдо-команда: загрузить адрес MyProg в регистр r0
79                  LDR     r0, =MyProg   ; r0 <- адрес точки входа
80  ; Косвенная передача управления программе пользователя
81                  BX      r0            ; PC <- (r0)
```

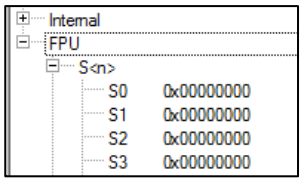
21.5.2 Доступ к регистрам сопроцессора в процессе отладки

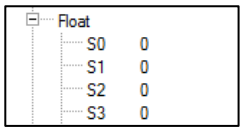
При отладке любой программы, обрабатывающей числа с плавающей точкой, исследуется текущее содержимое регистров сопроцессора и при необходимости выполняется их модификация. Выполним трансляцию двух файлов, входящих в проект FPU_1: StartUp_2.s и программы приложения MyProg.s. Убедимся, что нет синтаксических ошибок, создадим выходной файл проекта и загрузим его на отладку в симулятор. Откроем *окно регистров*. В списке доступных для исследования регистров имеются не только регистры процессорного ядра – позиция **Core** (с ними мы имели дело до сих пор), но и регистры сопроцессора (позиция **FPU**). Щелкните клавишей мыши на «крестике» рядом с обозначением «FPU». Содержимое этой позиции раскроется. Оказывается, Вы можете наблюдать в регистровом окне текущее содержимое регистров с плавающей точкой однократной S<n> и двойной D<n> точности, причем в двух вариантах: в шестнадцатеричном коде, то есть в *компьютерном представлении* числа с плавающей точкой однократной FS и двойной точности FD (позиции S<n> и D<n>) и в виде соответствующих этому представлению *десятичных чисел с плавающей точкой* (позиции Float и Double). Кроме того, будет доступно текущее содержимое регистра статуса и управления модулем плавающей точки FPSCR.





21.5.2.1 Просмотр и модификация регистров данных сопроцессора





Откройте банк регистров однократной точности S<n> в двух вариантах: S<n> и Float, щелкнув на соответствующих «крестиках». Будет выведено текущее содержимое всех 32-х регистров сопроцессора (показано только начало списков). Теперь Вы можете инициализировать любыми константами или модифицировать содержимое регистров FPU. Доступны несколько способов:

1) Дважды щелкните правой клавишей мыши в окне S<n> на содержимом регистра, которое требует изменения. Оно будет выделено пунктирной рамкой и дополнительно подсвечено синим цветом. Введите требуемое значение в виде целого числа в десятичной системе счисления (например, «100»), или в шестнадцатеричной системе обязательно с префиксом «0x» (например, «0xAF5»). Закончите ввод нажатием клавиши <Enter>. Содержимое регистра S<n> изменится и отобразится в окне в Hex-формате числа с плавающей точкой однократной точности FS. Одновременно в окне «Float» появится эквивалентное значение в формате десятичного числа с плавающей точкой (в общепринятой математической нотации).

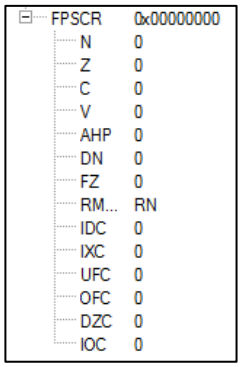
2) Таким же двойным щелчком мыши выделите содержимое нужного регистра в окне «Float». Введите новое значение в виде десятичного числа с плавающей точкой (например, 2.756 или 45.55E-7). Завершите ввод клавишей <Enter>. Новое значение в этом формате отобразится в окне «Float». Одновременно в окне S<n> будет представлено его компьютерное представление в формате FS – 32-разрядного числа с плавающей точкой однократной точности.

3) Наберите имя нужного регистра сопроцессора в командном окне «Command», затем знак равенства и введите новое значение либо в десятичной, либо в шестнадцатеричной системе счисления. Нажмите <Enter>. Это значение отобразится в окне S<n> в Hex-формате, а в окне Float – в формате десятичного числа с плавающей точкой. В окне «Command» прямой ввод десятичного числа в формате с плавающей точкой не поддерживается.

Для модификации содержимого любого из регистров сопроцессора можно «щелкнуть» левой клавишей мыши в поле значения регистра – оно будет подсвечено синим цветом. Затем, не нажимая клавишу «Back Space», ввести новое значение.

1) Поэкспериментируйте в окне регистров среды µVision с просмотром и модификацией содержимого регистров окружения сопроцессора путем ввода целых чисел в десятичной, шестнадцатеричной системах счисления и в формате десятичного числа с плавающей точкой. Оцените результат ввода.
2) Если Вы не хотите редактировать старое значение, а сразу ввести новое, то после выделения содержимого регистра двойным щелчком мыши нажмите клавишу «Back Space» («←») и вводите новое значение.
3) Проверьте, доступно ли отображение содержимого любого регистра сопроцессора после набора в командном окне его имени, например, S2<Enter>.

21.5.2.2 Просмотр и модификация регистра статуса и управления сопроцессора



Откройте позицию FPSCR в регистровом окне, и перед Вами появится текущее содержимое регистра статуса и управления модулем FPU (см. 20.5). В соответствии с описанным ранее назначением битовых полей этого регистра, в начальной части списка выводится текущее содержимое флагов, вырабатываемых операцией сравнения чисел с плавающей точкой VCMP (N ,Z, C, V), затем – содержимое трех битовых полей, управляющих специальными режимами работы сопроцессора (AHP, DN, FZ) и отдельно – состояние поля управления текущим режимом округления (в данном случае – режима округления «по умолчанию» – к ближайшему представимому числу с плавающей точкой RN). В конце списка выводятся текущие значения всех флагов исключений. Напомним: IDC – ввод денормированного числа; IXC – неточность; UFC – переполнение вниз; OFC – переполнение вверх; DZC – деление на ноль; IOC – некорректная операция.

Вы можете не только просмотреть содержимое регистра статуса и управления сопроцессором, но и изменить состояние любого бита. Так, при отладке программы с точками останова или в пошаговом режиме работы при выполнении какой-нибудь команды может возникнуть флаг исключительной ситуации: исследуйте возвращаемое этой командой значение (в регистре назначения), затем сбросьте флаг исключения и продолжайте отлаживать очередной фрагмент программы.

Среда µVision предоставляет широкие возможности просмотра текущего содержимого регистров сопроцессора и модификации этого содержимого. Встроенная система *автоконвертации форматов* позволяет одновременно наблюдать это содержимое в формате компьютерного представления числа с плавающей точкой (FS или FD) и в обычном формате десятичного числа с плавающей точкой. Программист имеет доступ ко всем битам регистра состояния и управления сопроцессором: может прямо в процессе отладки изменить режим работы сопроцессора и сбросить флаг любой исключительной ситуации.

21.5.3 Приступаем к отладке

Откройте файл SturtUp_2.s и исследуйте его содержимое. Найдите фрагмент стартового файла, который выполняет инициализацию сопроцессора. Мы уже давали подробные комментарии по структуре этого модуля (см. 20.12). Просмотрите эту информацию еще раз. Сразу после завершения инициализации сопроцессора управление передается пользовательской программе

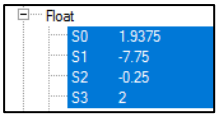
```
4  ; Точка входа в программу пользователя
5  MyProg
6  ; Объявить основную программу myProg общедоступной
7    EXPORT MyProg
8
9  ; Загрузить первые 4-е регистра модуля FPU константами
10   VMOV.F32 s0, #1.9375  ;+31e-4
11   VMOV.F32 s1, #-7.75   ;-31e-2
12   VMOV.F32 s2, #-0.25   ;-16e-6
13   VMOV.F32 s3, #+2.0    ;+16e-3
14 ; Установите здесь точку останова для загрузки регистров
15 ; любыми исходными данными в процессе отладки программы
16 Loop
17 ; Вычислить абсолютные значения чисел и записать результат
18 ; в регистры s4-s7
19   VABS.F32 s4, s0
20   VABS.F32 s5, s1
21   VABS.F32 s6, s2
22   VABS.F32 s7, s3
```

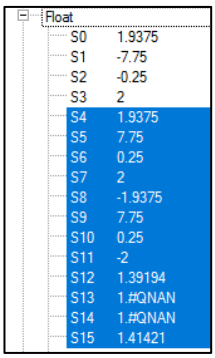
MyProg.s, которая выполняет инициализацию четырех регистров сопроцессора S0-S3 непосредственными константами. Константы выбраны такими, которые могут быть сжаты и закодированы в формате 32-разрядной команды VMOV. Программа выполняет расчет абсолютных значений чисел с плавающей точкой, их инверсных значений, а также квадратного корня из этих чисел.

```
23  ; Проинвертировать значения чисел и сохранить результат
24  ; в регистрах s8-s11
25    VNEG.F32 s8,s0
26    VNEG.F32 s9,s1
27    VNEG.F32 s10,s2
28    VNEG.F32 s11,s3
29  ; Извлечь из исходных чисел квадратный корень и сохранить
30  ; в регистрах s12-s15
31    VSQRT.F32 s12,s0
32    VSQRT.F32 s13,s1
33    VSQRT.F32 s14,s2
34    VSQRT.F32 s15,s3
35  ; Повторить для другого набора входных данных
36    B Loop
```

Результаты расчетов последовательно сохраняются в регистрах сопроцессора S4-S7, S8-S11, S12-15. После завершения всех операций управление вновь передается в начало программы. Если поставить в этом месте точку останова, то можно переинициализировать регистры сопроцессора другими данными и проверить работу программы на любом другом наборе переменных, продолжив выполнение программы еще раз до точки останова или в пошаговом режиме.

Поставим точку останова на команде VABS.F32 s4,s0 и выполним программу до точки останова. В результате первые четыре регистра сопроцессора будут проинициализированы начальными значениями. Еще раз выполним программу до точки останова. Проанализируем результат в регистровом окне. Команда VABS возвращает абсолютное значение исходного числа, а команд VNEG – меняет знак числа на обратный.

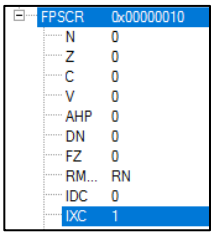


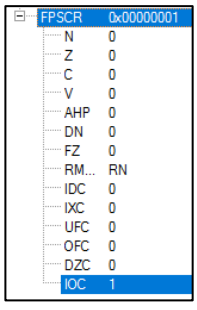


Несколько другая ситуация возникает при работе команды извлечения квадратного корня из числа в формате с плавающей точкой. В первом и последнем случае (аргумент положительный) результат похож на правду, но вычислен немного неточно. Проверьте на калькуляторе! Во втором и третьем случаях (аргумент отрицательный) в качестве результата операции мы получили 1.#QNAN. Это так называемое «тихое Не число» (см. 19.14). Сопроцессор всегда возвращает «Не число», если математическая операция не может быть выполнена, когда ее результат неопределен (см. 20.7).

Попытка извлечения квадратного корня из отрицательного числа рассматривается сопроцессором как некорректная, и в качестве результата операции возвращается «тихое Не число».

Сбросим процессор в исходное состояние, откроем в регистровом окне содержимое регистра статуса и управления модулем плавающей точки FPSCR и проследим за его состоянием в процессе пошагового выполнения программы. Как только будет выполнена команда извлечения квадратного корня из числа 1.9375, состояние этого регистра изменится. Установится флаг исключения IXC «Неточность». О чем он свидетельствует? Он сообщает программисту, что полученный результат $\sqrt{1.9375} \approx 1.39194$ не является точным, а лишь *приближенным*, который сопроцессор смог представить в формате 32-разрядного числа с плавающей точкой однократной точности FS. Вы можете продолжить вычисления, не обращая внимания на этот флаг. И только в том случае, если Вас не устроит точность всего расчета, можете перейти к обработке чисел двойной точности (выбрав другой процессор или написав программу на C/C++ с программной поддержкой вычислений с двойной точностью).





Мы остановили выполнение программы. Сбросим флаг неточности IXC. Дважды щелкните мышью в поле значения этого флага, чтобы перейти в режим редактирования, и установите 0 вместо 1. Можно продолжать пошаговое выполнение программы. Очередная команда должна извлечь квадратный корень из отрицательного числа. Посмотрим,

что будет при ее выполнении. Сопроцессор формирует флаг исключения IOC – «Некорректная операция». Так и должно быть, поскольку мы попытались извлечь квадратный корень из отрицательного числа. Следующая команда сделает то же самое – уведомит программиста о некорректности выполненных вычислений флагом исключения IOC. Несмотря на то, что при выполнении последней команды проблем не возникает, выставленные ранее флаги исключительных ситуаций IXC и IOC не исчезают, – раз возникнув, они фиксируются (защелкиваются). Это позволяет после выполнении всего фрагмента программы проверить, возникали ли исключительные ситуации.

Выставляя флаги исключительных ситуаций, сопроцессор «предупреждает» о возможных некорректностях в работе программы обработки чисел с плавающей точкой. Программист может разрешить обработку всех или некоторых исключений, но должен при этом написать соответствующий обработчик исключения – подпрограмму, которая проверит, какое именно исключение возникло и можно ли продолжать выполнение программы. Если да, то – какой результат нужно вернуть вместо результата, возвращенного сопроцессором по умолчанию.

Проверьте работу программы на других наборах входных данных в режиме ее выполнения до точки останова. Загружайте регистры сопроцессора S0-S3 исходными данными сразу после точки останова и следите за результатами их обработки и возможными исключениями.

1) Проверьте, что число 0 является допустимым для всех операций, выполняемых в программе.
2) Введите в качестве начального исходного значения «Не число», например 0x7FC00000. Какой результат будут возвращать команды в этом случае?
3) Можно ли в одно-операндных командах опускать имя регистра-приемника, например, так VABS.F32 s0?
4) Попробуйте в командах непосредственной загрузки регистров сопроцессора использовать любые константы. Что за сообщения выдает транслятор?

2) Да. Команда VNEG при этом возвращает отрицательный ноль.

3) Если операнд команды является «Не числом», то в качестве результата возвращается то же «Не число» без какой-либо обработки, кроме, возможно, изменения знакового разряда, как это происходит при выполнении команды VNEG (вместо 0x7FC00000 возвращается 0xFFC00000 с инвертированным знаковым разрядом). Будьте осторожны – обработка «Не чисел» не производится!

4) Нет. Будет синтаксическая ошибка. Допустимо только: VABS.F32 s0, s0. Имена регистра-источника и приемника указывать обязательно, даже если они одинаковы. Проверьте, как работает программа MyProg_1.s в том же проекте.

```
 9  ; Загрузить первые 4-е регистра модуля FPU константами
10    VMOV.F32 s0, #1.999
11    VMOV.F32 s1, #-7.7803
12    VMOV.F32 s2, #-0.2335
13    VMOV.F32 s3, #+2.0E23
```

5) Мы изменили программу – MyProg_2.s в попытке загрузить другие нужные нам константы. Их значения таковы, что не могут быть сжаты для включения в формат команды VMOV. Транслятор выдает сообщения «Непосредственное значение не может быть использовано в операции». Придется воспользоваться псевдо-командой загрузки регистров.

```
MyProg_2.s(10): error: A1240E: Immediate value cannot be used with this operation
MyProg_2.s(11): error: A1240E: Immediate value cannot be used with this operation
MyProg_2.s(12): error: A1240E: Immediate value cannot be used with this operation
MyProg_2.s(13): error: A1240E: Immediate value cannot be used with this operation
```

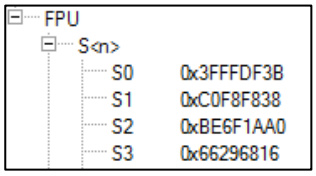
21.5.4 Псевдокоманды загрузки регистров сопроцессора

Имеются две специальные псевдокоманды загрузки регистров сопроцессора константами: **VLDR.F32 Sd,=Const** и **VLDR.F64 Dd,=Const**. Они позволяют инициализировать регистры любыми значениями чисел в формате с плавающей точкой без каких-либо ограничений. Обратите особое внимане на синтаксис псевдо-команд: после имени регистра приемника через запятую указывается знак равенства, а после него *без символа непосредственного операнда «#»* – константа в формате десятичного числа с плавающей точкой – см. MyProg_3.s.

```
 9  ; Загрузить первые 4-е регистра модуля FPU константами
10    VLDR.F32 s0,=1.999
11    VLDR.F32 s1,=-7.7803
12    VLDR.F32 s2,=-0.2335
13    VLDR.F32 s3,=+2.0E23
```

В псевдокомандах используется уже описанная нами ранее для загрузки обычных регистров ЦПУ (см. 10.3) технология создания «литеральных пулов» в кодовой памяти. Транслятор сам создает нужную константу в памяти и автоматически генерирует команду доступа к ней с использованием относительной адресации по счетчику команд PC и непосредственному смещению до места расположения константы (до «литерального пула»). Нужное смещение также рассчитывается транслятором автоматически – VLDR Sd, [PC+#offset].

Выполните трансляцию файла MyProg_3.s и сборку всего проекта. Отладьте проект в симуляторе и убедитесь, что все нужные константы загружаются в регистры-источники и обрабатываются последущими командами правильно. Следите за возвращаемыми данными в регистры назначения, а также за состояним флагов исключительных ситуаций.



Сравните содержимое регистров однократной точности S0-S3, после загрузки с использованием псевдо-команд, с содержимым памяти в конце текущей кодовой секции (после команды передачи управления на себя b Stop). Действительно, с помощью двух команд резервирования полуслов DCW транслятор автоматически размещает в кодовой памяти сначала младшее полуслово каждой константы с плавающей точкой, а затем – старшее полуслово, то есть создает в конце программы «литеральный пул» – таблицу констант, к которым будут обращения.

```
Disassembly
    39:          b Stop
0x0000006E E7FE      B          0x0000006E
0x00000070 DF3B      DCW        0xDF3B
0x00000072 3FFF      DCW        0x3FFF
0x00000074 F838      DCW        0xF838
0x00000076 C0F8      DCW        0xC0F8
0x00000078 1AA0      DCW        0x1AA0
0x0000007A BE6F      DCW        0xBE6F
0x0000007C 6816      DCW        0x6816
0x0000007E 6629      DCW        0x6629
```

В окне дизассемблера проследите также за тем, как транслятор заменяет каждую псевдо-команду реальной командой загрузки данных из кодовой помяти по адресу относительно текущего содержимого счетчика команд с дополнительным смещением. Действительно, для первой команды адрес следующей, подлежащей выполнению команды, (PC)=0x30. Если добавить к нему рассчитанное транслятором смещение 0x40 – получим адрес 0x70, где и располагается младшее слово

```
    10:          VLDR.F32 s0,=1.999
0x0000002C ED9F0A10  VLDR          s0,[pc,#0x40]
    11:          VLDR.F32 s1,=-7.7803
0x00000030 EDDF0A10  VLDR          s1,[pc,#0x40]
    12:          VLDR.F32 s2,=-0.2335
0x00000034 ED9F1A10  VLDR          s2,[pc,#0x40]
    13:          VLDR.F32 s3,=+2.0E23
```

соответствующей константы с плавающей точкой.

При использовании псевдо-команд загрузки констант с плавающей точкой в регистры модуля FPU все ограничения на значение константы снимаются. Загрузка данных выполняется за один цикл с одновременным использованим как шины ICODE, так и шины DCODE, подключенной к кодовой памяти.

21.5.5 Косвенная загрузка регистров сопроцессора из памяти и загрузка по метке

В разделе 21.3.3 мы рассмотрели директивы Ассемблера для размещения констант с плавающей точкой в кодовой памяти. Выполним с их помощью размещение в конце нашей программы (перед директивой END), констант, подлежащих загрузке в регистры с плавающей точкой однократной точности. Пометим каждую константу меткой Const_0, Const_1,… Никаких ограничений на значения констант, размещенных в памяти – нет.

```
38  ; Резервирование и инициализация
39  ; констант FS в кодовой памяти
40  Const_0 DCFS 1.999
41  Const_1 DCFS -7.7803
42  Const_2 DCFS -0.2335
43  Const_3 DCFS +2.0E23
```

Обратимся к программе MyProg_4.s в том же проекте FPU_1. В ней использованы два разных способа загрузки регистров сопроцессора:

1) с применением классической косвенной адресации по содержимому указателя в базовом регистре r0;

2) с применением псевдо-команды загрузки регистра сопроцессора данными из памяти по их адресу.

```
 9  ; Загрузка регистра сопроцессора из памяти с использование
10  ; обычной косвенной адресации
11        LDR r0,=Const_0
12        VLDR s0,[r0]
13  ; Загрузка регистров сопроцессора по метке данных в памяти
14        VLDR s1,Const_1
15        VLDR s2,Const_2
16  ; Еще одна загрузка с использованием косвенной адресации
17        ADD r0,#4
18        VLDR s3,[r0]
```

Естественно, что в первом случае требуется предварительная инициализации базового регистра r0 адресом расположения данных – для этого использована псевдо-команда загрузки обычного регистра ЦПУ адресом Const_0.

Во втором случае достаточно указать адрес данных в качеств операнда псевдо-команды VLDR Sd,Label. Транслятор заменит псевдо-команду операцией доступа к данным по текущему содержимому счетчика команд и непосредственному смещению (командой относительной адресации). Убедитесь в этом сами, исследуя программу в окне дизассемблера. Второй метод проще и понятнее и поэтому рекомендуется к применению.

```
    14:                             VLDR s1,Const_1
0x00000032 EDDF0A11  VLDR             s1,[pc,#0x44]
    15:                             VLDR s2,Const_2
```

```
0x00000036 ED9F1A11  VLDR             s2,[pc,#0x44]
```

Если данные в памяти расположены последовательно, как в нашем случае, рационально использовать более мощные команды множественной загрузки регистров сопроцессора (см. ниже).

21.6 Команды множественной загрузки и сохранения регистров сопроцессора

Число регистров сопроцессора для хранения чисел в формате с плавающей точкой однократной точности равно 32 (S0-S31). Их достаточно много для того, чтобы решать большинство практических задач, вообще не пользуясь для временного хранения данных памятью. В

начале расчета регистры сопроцессора загружаются исходными данными, а по завершении расчета их содержимое выдается во вне (сохраняется в памяти или регистрах периферийных устройств). Разработчики системы команд ARM Cortex-M4F предусмотрели команды, как для *множественной (групповой) загрузки* регистров сопроцессора, так и для *множественного сохранения* содержимого регистров в памяти.

Для доступа к данным в памяти используется *базовая адресация с авто-смещением* (рассчитывается автоматически в процессе выполнения команды). Начальный адрес области памяти предварительно записывается в базовой регистр Rn. В процессе пересылки данных эффективный адрес ячейки памяти определяется как содержимое базового адреса плюс/минус смещение. Величина смещения с каждой пересылкой данных изменяется на 4 (размер слова формата FS в байтах):

- В режиме IA (инкрементировать адрес после доступа) смещение добавляется к эффективному адресу *после завершения пересылки данных;*
- В режиме DB (декрементировать перед доступом) смещение вычитается из эффективного адреса *перед каждым доступом к данным*.

Синтаксис команд множественной загрузки/сохранения следующий:

```
VLDM{mode}{cond}{.size} Rn{!}, list
VSTM{mode}{cond}{.size} Rn{!}, list
```

Листинг регистров сопроцессора, подлежащих загрузке/сохранению. Имена регистров разделяются запятой и помещаются в фигурные скобки { }

Суффикс обратной перезаписи содержимого базового регистра Rn (авто-обновления после множественной пересылки данных). Если обновление требуется, то указывается обязательно и для режима IA, и для DB.

Rn – базовый регистр, содержащий начальный адрес памяти

.size – Спецификатор длины загружаемых данных (опция):
.32– для регистров однократной точности; .64 – двойной точности

cond – Код условного выполнения команды

mode – режим: IA – Инкрементировать адрес памяти после каждого доступа (на 4).
Режим по умолчанию (суффикс может быть опущен)
DB – Декрементировать адрес памяти перед каждым доступом (на 4)

M – (Multiple) Множественная загрузка/сохранение регистров сопроцессора

LD – (Load) Загрузить регистры модуля FPU

ST – (Store) Сохранить содержимое регистров модуля FPU

V – операция в модуле поддержки вычислений с плавающей точкой FPU

Содержимое базового регистра в общем случае остается неизменным. С помощью *суффикса обратной перезаписи* «!» можно задать режим *автообновления содержимого базового регистра* Rn по последнему содержимому, после выполнения всех пересылок данных. Эта опция позволяет организовать доступ к нескольким последовательным блокам данных в памяти, без необходимости переинициализации базового регистра. Рекомендуется для доступа к массивам данных.

Наиболее часто используется режим IA (инкрементирования адреса после доступа). Он считается для этих команд режимом «по умолчанию», и для него в мнемокоде команды можно опустить суффикс IA, но не суффикс автообновления содержимого базового регистра «!», если это требуется.

Для режима DB (декрементирования адреса перед доступом) опция обратной перезаписи также должна быть включена, если необходима.

Эти команды могут работать с данными не только в произвольной области памяти, но и в стеке. При этом в качестве базового регистра Rn должен использоваться указатель стека SP. Такой способ адресации удобен при работе с подпрограммами: для сохранения контекста сопроцессора в стеке в начале подпрограммы и для его восстановления – в конце, перед возвратом в основную программу.

Напомним, что система команд ARM-процессоров допускает организацию стеков двух типов (см. 14.1). По умолчанию используется стек типа EA, когда запись в стек выполняется в режиме «декрементирования указателя перед записью» DB, а извлечение данных из стека – в режиме «инкрементирования указателя после чтения» IA. Соответствие суффиксов в командах множественной загрузки/сохранения типу стека приведено в табл. 21.2. Зеленым фоном в таблице выделен стек «по умолчанию», который используется большинством программистов.

Таблица 21.2 Соответствие типов команд множественной загрузки/сохранения типам стеков

Опция команды mode	**Соответствующий тип стека**	**Комментарий**
DB/IA	**EA**	**Декрементировать указатель стека перед записью, инкрементировать после чтения. Освобождение стека в направлении возрастающих адресов при чтении – Empty Ascending Stack.**
IA/DB	**FD**	**Инкрементировать указатель стека после записи, декрементировать после чтения. Освобождение стека в направлении убывающих адресов при чтении – Full Descending Stack.**

Список регистров сопроцессора должен содержать имена регистров, разделенные запятой, заключенный в фигурные скобки, например, {s0, s1, s2}. Список может содержать и диапазон регистров, подлежащих загрузке/сохранению, например, {s0-s5}. Могут загружаться как регистры однократной точности Sd, так и регистры двойной точности Dd. В первом случае можно использовать дополнительный спецификатор длины данных «.32», а во втором – «.64». В списке должен быть по крайней мере один регистр. Нельзя смешивать регистры однократной и двойной точности в одном списке. Регистры перечисляются *обязательно в порядке возрастающих номеров*, причем номера должны быть последовательными, – разрывы не допускаются.

Порядок загрузки регистров: первым загружается регистр с наименьшим номером; далее – по возрастанию номеров регистров сопроцессора.

Проверим, как работает команда множественной загрузки, модифицировав программу MyProg_4.s. Новый файл MyProg.s Вы найдете в проекте FPU_2. Заменим несколько команд загрузки регистров сопроцессора однократной точности одной командой множественной загрузки. Используем также команду загрузки двух регистров двойной точности D3, D4 теми же константами из памяти. Все возможные опции в командах множественной загрузки регистров сопроцессора опущены. Загрузка по умолчанию выполняется с пост-авто-инкрементированием смещения после очередной записи, без автообновления базового регистра после завершения всех операций.

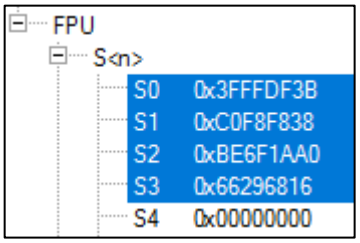
```
 9  ; Демонстрация эффективности команд множественной загрузки
10  ; регистров сопроцесора однократной и дройной точности
11  ; Инициализация базового регистра r0 начальным адресом
12  ; констант в памяти
13        LDR r0,=Const_0
14  ; Загрузка сразу 4-х регистров сопроцессора s0-s3
15        VLDM r0,{s0-s3}
16  ; Загрузка 2-х регистров сопроцессора d2-d3
17        VLDM r0,{d2,d3}
```

1) При выполнении первой команды загрузки регистров однократной точности s0-s3 будут ли одновременно загружены регистры двойной точности d0,d1?

2) Какое содержимое будет в регистрах d0,d1?

3) Являются ли команды VLDM r0,{s0-s3} и VLDM r0,{d0,d1} полностью эквивалентными?

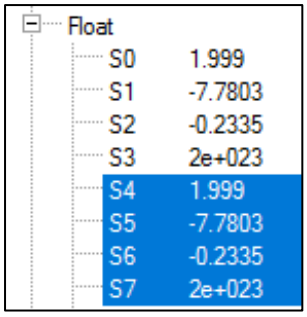
1) Да, так как на самом деле регистры двойной точности состоят из пар регистров однократной точности d0=(s1,s0); d1=(s3,s2). Младшее слово содержит данные в четном регистре, а старшее – в нечетном. Вот результат выполнения первой команды множественной загрузки:



Действительно, любая запись в регистры однократной точности приводит к модификации значений в соответствующих регистрах двойной точности. D0=(S1,S0); D1=(S3,S2).

2) D0=(S1,S0); D1=(S3,S2).

3) Да, множественная загрузка первых 4-х регистров однократной точности полностью эквивалентна загрузке первых 2-х регистров двойной точности, если иметь в виду, что она выполняется «как есть» – загрузкой 32-разрядной или 64-разрядной битовой последовательности без какой-либо интерпретации значения числа. Можете убедиться в этом, отлаживая программу MyProg_1.s, в которой команда VLDM r0,{d0,d1} заменена на VLDM r0,{d2,d3} для удобства сравнения результатов.



Для того, чтобы сохранить результаты обработки данных в памяти, необходимо прежде всего зарезервировать память для этой цели. Еще раз модифицируем исходную программу (см. MyProg_2.s), добавив в нее секцию данных под начальные значения переменных в формате FS в регистрах s0-s3 и результаты их обработки в регистрах s4-s7 (VABS); s8-s11 (VNEG); s12-s15 (VSQRT). Всего – 16 слов.

```
52  ; Секция данных в ОЗУ для прима результатов расчетов
53          AREA MyData, DATA, ReadWrite
54  ; Зарезервировать место под 16 числе в формате FS
55  RAM     SPACE 4*16
```

```
44          ALIGN
45  ; Резервирование и инициализация
46  ; констант FS в кодовой памяти
47  Const_0 DCFS 1.999
48  Const_1 DCFS -7.7803
49  Const_2 DCFS -0.2335
50  Const_3 DCFS +2.0E23
```

В конце программы, как и в проекте FPU_1, выделим место в кодовой памяти под константы в формате FS, подлежащие загрузке и последующей обработке. Директива ALIGN применена для того, чтобы выровнять кодовую память по дважды четному адресу, обеспечивающему оптимальный доступ к константам – словам.

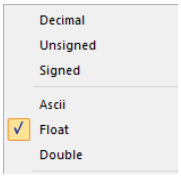
```
11  ; Инициализация базового регистра r0 начальным адресом
12  ; констант в памяти
13      LDR r0,=Const_0
14  ; Загрузка первых 4-х регистров сопроцессора s0-s3
15  ; константами из кодовой памяти
16      VLDM r0,{s0-s3}
17  ; Вычислить абсолютные значения чисел и записать результат
```

Выполним множественную загрузку исходных данных из кодовой памяти в регистры s0-s3 с использованием команды VLDM. Дальше используем тот же блок обработки, позволяющий получать абсолютные значения, инверсии и квадратные корни исходных чисел с плавающей точкой.

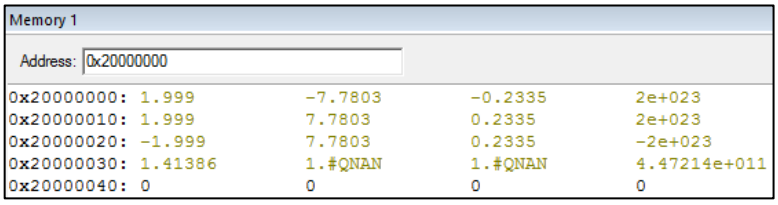
После завершения обработки воспользуемся командой множественного сохранения регистров сопроцессора в оперативной памяти VSTM. При ее выполнении будут сохранены 4 группы данных: исходные числа; их абсолютные значения; их инверсии; результаты вычисления квадратного корня из исходных чисел.

```
35  ; Сохранение результатов вычислений
36  ; Инициализация базового регистра r0 начальным адресом
37  ; области ОЗУ для хранения результатов
38      LDR r0,=RAM
39  ; Сохранить результаты расчетов в памяти
40      VSTM r0,{s0-s15}
```

Вы уже знаете, как по таблице символов определить начальный адрес области ОЗУ – RAM. Он равен 0x20000000. Откройте окно памяти, начиная с этого адреса, и с помощью контексно-зависимого меню установите в нем режим отображения данных в формате чисел с плавающей точкой однократной точности Float.



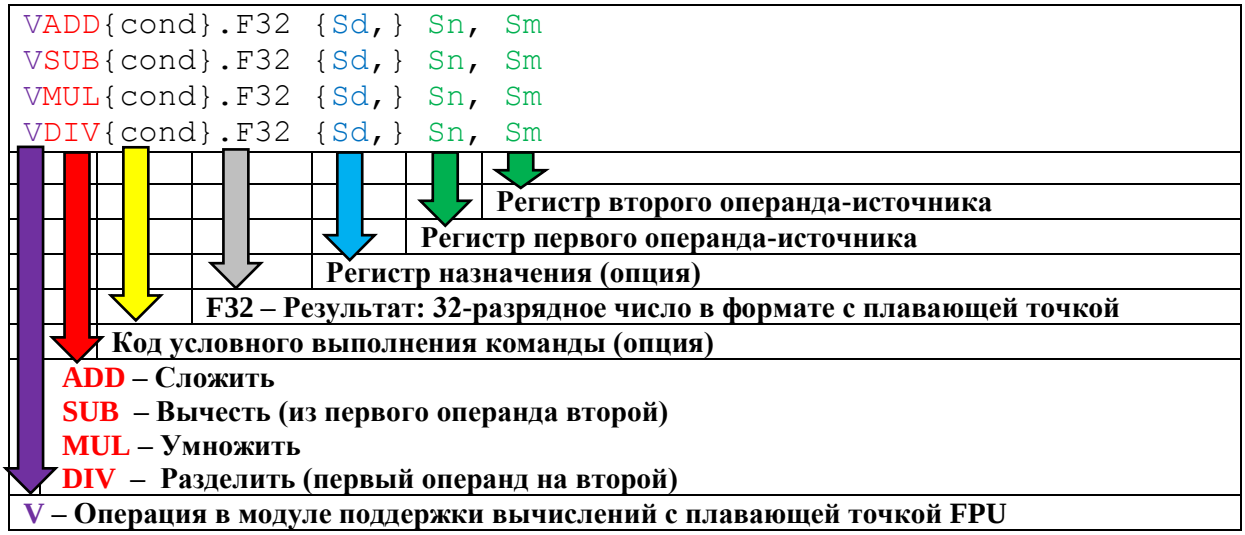
Теперь можно выполнить программу либо в пошаговом режиме, либо до точки останова (команды передачи управления на себя b Stop). Ниже приведено окно дампа памяти, которое свидетельствует о правильности выполнения всех команд. Обратите внимание на то, что результаты извлечения квадратного корня из отрицательных чисел представлены «тихими Не числами», как и должно быть ввиду некорректности этих вычислений.



Используйте эффективные команды множественной загрузки регистров сопроцессора данными из памяти и множественного сохранения результатов обработки данных в памяти. Помните, что список регистров для загрузки/сохранения должен состоять из последовательных номеров регистров, начиная с самого младшего. Именно в этой последовательности данные будут загружаться или сохраняться в памяти. Рассмотренные команды являются прерываемыми. При возникновении прерывания контекст команды сохраняется и следует передача управления подпрограмме обслуживания прерывания. После возврата из процедуры обслуживания прерывания выполнение операции множественной загрузки/сохранения продолжается.

21.7 Арифметические операции поддержки вычислений с плавающей точкой

Вместе с представленной ранее группой однооперандных команд (VABS, VNEG, VSQRT) эта группа двухоперандных команд обеспечивает *весь спектр необходимых вычислительных операций* с числами в формате с плавающей точкой однократной точности:



Операции выполняются над двумя числами в формате FS, находящимися в любых двух регистрах S0-S31 модуля плавающей точки – Sn и Sm. Результат операции сохраняется в одном из регистров S0-S31 модуля – Sd. Если регистр назначения не специфицирован в команде, то результат операции сохраняется в первом регистре-источнике Sn.

В табл. 21.3 приведен список исключений, которые могут возникать при выполнении этих арифметических операций.

Таблица 21.3 Возможные исключения при выполнении арифметических операций

Операция	Возможные исключения				
	Неточ.-ть Inexact	Перен. вверх Overflow	Перен. вниз Underfow	Делен. на ноль Division by Zero	Недоп. опер.-я Invalid Operation
ADD	V	V	V		V
SUB	V	V	V		V
MUL	V	V	V		V
DIV	V	V	V	V	V

Особенности использования арифметических команд рассмотрим на простом примере. Необходимо вычислить корни квадратного уравнения $ax^2+bx+c=0$, коэффициенты которого a, b, c – произвольные числа с плавающей точкой однократной точности, расположенные в оперативной памяти и доступные для модификации в процессе отладки программы. Рассчитанные значения корней уравнения x_1, x_2 вывести в ячейки памяти, расположенные вслед за переменными a, b, c. Выполнить дополнительно проверку вычислений путем определения значений двух выражений $z_1=ax_1^2+bx_1+c$; $z_2=ax_2^2+bx_2+c$, которые должны быть нулевыми. Полученные значения z_1, z_2 также вывести в ячейки памяти вслед за значениями корней уравнения.

```
62  ; Секция исходных данных в ОЗУ и результатов расчета корней
63          AREA MyData, DATA, ReadWrite
64  ; Коэффициенты квадратного уравнения
65  a       SPACE 4
66  b       SPACE 4
67  c       SPACE 4
68  ; Корни квадратного уравнения
69  x1      SPACE 4
70  x2      SPACE 4
71  ; Результаты проверки правильности вычислений
72  z1      SPACE 4
73  z2      SPACE 4
74    END       ; Конец программы
```

В этой простой задаче будут использованы все, представленные выше арифметические команды (см. файл MyProg_3.s в проекте FPU_2). Прежде всего, в самом конце программы объявим секцию данных и зарезервируем в ней место для: коэффициентов квадратного уравнения a, b, c; рассчитанных значений

корней x_1, x_2; проверочных значений квадратного трехчлена при значениях аргумента x, соответствующих корням z_1, z_2.

```
 9  ; Расчет корней квадратного уравнения
10  ; ax^2+bx+c=0
11  ; Загрузить коэффициенты квадратного уравнения
12  ; из памяти в регистры сопроцессора s0-s2
13        LDR r0,=a
14        VLDM r0,{s0-s2}
```

Программа начинается с загрузки коэффициентов a, b, c в регистры сопроцессора. Наиболее просто и удобно использовать для этого команду множественной загрузки сразу трех регистров сопроцессора s0-s2. Базовый регистр r0 нужно предварительно загрузить начальным адресом расположения коэффициентов в памяти. Так как ОЗУ начинается с адреса 0x20000000, придется использовать псевдокоманду загрузки, которая допускает любое значение адреса и предварительно размещает его в «литеральном пуле».

```
15  ; Рассчитать дискреминант квадратного уравнения
16  ; D=(b^2-4ac) и квадратный корень из него √D
17  ; a=(s0); b=(s1); c=(s2); √D=(s3)
18        VMOV.F32 s4,#4     ; s4 <- 4
19        VMUL.F32 s4,s0     ; s4 <- 4a
20        VMUL.F32 s4,s2     ; s4 <- 4ac
21        VMUL.F32 s3,s1,s1  ; s3 <- b^2
22        VSUB.F32 s3,s4     ; s3 <- b^2-4ac
23        VSQRT.F32 s3, s3   ; s3 <- √(b^2-4ac)
```

Расчет начинается с определения дискриминанта квадратного уравнения D и квадратного корня из него. Все действия, выполняемые командами, очевидны и сопровождаются краткими комментариями справа. Еще из школьного курса математики вы знаете, что корни уравнения существуют только в том случае, когда дискриминант – положительный (D>0).

Для их расчета понадобится вспомогательная переменная 2a. Как и в фрагменте программы выше, загрузка коэффициентов 4 и 2 возможна по команде VMOV.F32 с непосредственным операндом в формате самой команды.

```
24  ; Рассчитать 2a
25        VMOV.F32 s4,#2     ; s4 <- 2
26        VMUL.F32 s4,s0     ; s4 <- 2a
```

```
27  ; Рассчитать первый корень x1=(-b+√D)/2a
28        VNEG.F32 s5,s1     ; s5 <- (-b)
29        VADD.F32 s6,s5,s3  ; s6 <- (-b+√D)
30        VDIV.F32 s6,s4     ; s6 <= (-b+√D)/2a
31  ; Рассчитать второй корень x2=(-b-√D)/2a
32        VSUB.F32 s7,s5,s3
33        VDIV.F32 s7,s4
34  ; Проверить правильность расчета для x1
35  ; z1=a(x1^2)+bx1+c
36  ; x1=(s6)
37        VMUL.F32 s8,s6,s6  ; s8 <- x1^2
38        VMUL.F32 s8,s0     ; s8 <- a(x1^2)
39        VMUL.F32 s9,s1,s6  ; s9 <- bx1
40        VADD.F32 s8,s9     ; s8 <- a(x1^2)+bx1
41        VADD.F32 s8,s2     ; s8 <- a(x1^2)+bx1+c
42  ; Проверить правильность расчета для x2
43  ; z2=ax2^2+bx2+c
44  ; x2=(s7)
45        VMUL.F32 s9,s7,s7
46        VMUL.F32 s9,s0
47        VMUL.F32 s10,s1,s7
48        VADD.F32 s9,s10
49        VADD.F32 s9,s2
```

Теперь все готово для расчета корней квадратного уравнения. Значение первого корня сохраняется в регистре s6, а второго – в регистре s7.

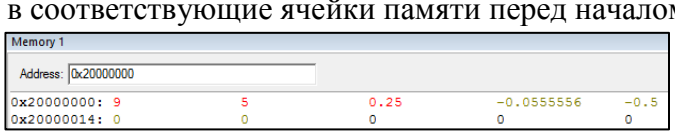
Мы учимся и, поэтому, пока не доверяем «сопроцессору». Проверим, правильно ли будут рассчитаны корни, вычислив значения правых частей уравнения z_1 и z_2 при двух значениях аргументов, равных корням уравнения: $x=x_1$ и $x=x_2$.

Сохраним контрольные значения z_1 и z_2 в регистрах s8, s9, соответственно.

Теперь осталось только вывести рассчитанные значения в регистрах s6-s9 в память, начиная с адреса x1.

```
50  ; Сохраненить результаты вычислений в памяти
51  ; Инициализация базового регистра r0 начальным адресом
52  ; области ОЗУ для хранения результатов
53        LDR r0,=x1
54  ; Сохранить результаты расчетов в памяти
55        VSTM r0,{s6-s9}
56  ; Предотвратить выполнение "случайного" кода
57  Stop
58        b Stop
59        ALIGN
```

Выполним это с использованием команды множественного сохранения содержимого регистров сопроцессора в памяти VSTM. Расчет завершен. На команде «зацикливания» программы можно поставить точку останова, чтобы выполнять программу целиком при любых значениях исходных данных a, b, c, вводимых в соответствующие ячейки памяти перед началом ее выполнения.



«Постройте» проект целиком и загрузите его на отладку в симулятор. Откройте окно дампа памяти, начиная с

адреса 0x20000000 (адрес первого коэффициента), и установите режим отображения в памяти чисел в формате с плавающей точкой однократной точности float. Введите любые значения коэффициентов уравнения и выполните программу до точки останова. Исследуйте результат в окне дампа памяти. Например, для коэффициентов a=9.0; b=5.0; c=0.25 получим x_1=-0.0555556; x_2=-0.5. Расчет правильный. Он подтверждается нулевыми значениями контрольных параметров z_1 и z_2.

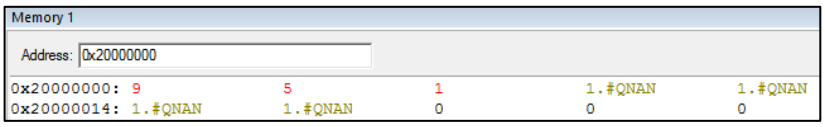
1) Проверьте, были ли в процессе расчета установлены флаги исключительных ситуаций? Какие? При выполнении каких команд? Что они означают?

2) Измените значения коэффициентов так, чтобы дискриминант квадратного уравнения стал отрицательным. Объясните полученный результат расчета.

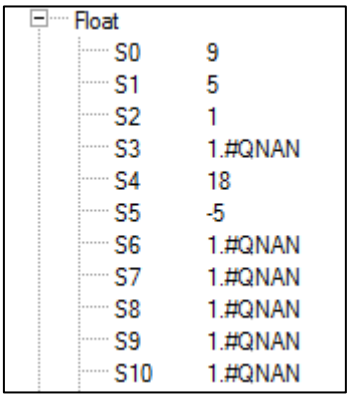
3) Как будет выполняться любая арифметическая команда, если один или более из ее операндов будут «Не числами»?

1) Был установлен единственный флаг IXC (Неточность). При выполнении программы в пошаговом режиме заметим, что первый раз флаг был установлен при расчете корня x1= -0.0555556 (операцией деления VDIV). Это означает, что сопроцессор округлил точное значение частного до ближайшего представимого в рамках 32-разрядного числа с плавающей точкой. Если сбросить этот флаг и продолжить выполнение программы, то увидим, что флаг неточности выставлялся еще несколько раз при расчете контрольного значения переменной z1. Нулевые значения z1 и z2 подтверждают точность итоговых результатов, а наличие флага IXC свидетельствует лишь о том, что сопроцессору пришлось выполнять округление результатов до ближайших репрезентабельных чисел в формате FS. Так что флаг IXC – довольно частое явление при обработке чисел с плавающей точкой и «самый безопасный».

2) Один из вариантов параметров и результат расчета показан ниже. Ясно, что извлечение квадратного корня из отрицательного числа – недопустимая операция, — возникает исключение IOC (Некорректная операция). В качестве результата такой операции будет возвращено так называемое «тихое Не число» – qNaN. Оно же окажется и во всех итоговых результатах.



3) Любые арифметические операции с «Не числами» – *не выполняются*. В качестве результата в регистр-приемник возвращается исходное «Не число». Обратите внимание, что как только в регистр s3 в результате некорректной операции извлечения квадратного корня было записано «тихое Не число» qNaN, результатом всех последующих арифметических операций, в которых исходным операндом было содержимое регистра s3, так же стало «Не число» qNaN (s6-s10). Таким образом, «Не число» в итоговом результате свидетельствует о некорректности всего расчета и о невозможности его использования.



Флаг исключения IOC (Недопустимая операция) свидетельствует о том, что соответствующий фрагмент программы работает неправильно: либо исходные данные находятся вне допустимого диапазона, либо сама программа некорректна. Наличие «Не числа» в качестве результата расчета говорит о том же: результат расчета фактически не получен. Требуется либо коррекция алгоритма/программы, либо диапазона используемых переменных.

Недопустимые операции должны либо обрабатываться в соответствующих подпрограммах-обработчиках, либо не допускаться вообще. Для этого можно использовать операции сравнения чисел с плавающей точкой и команды условного ветвления/условного выполнения.

21.8 Как проверить наличие флага исключительной ситуации?

Флаги исключительных ситуаций один раз возникнув, «защелкиваются» в регистре статуса и управления модуля FPSCR (см. 20.5). Особенность этого регистра в том, что он относится к регистрам *специального назначения*, прямой доступ к которому невозможен. Поэтому, протестировать наличие какого-либо флага программным путем можно только одним способом – считать содержимое регистра FPSCR в один из регистров общего назначения ЦПУ и протестировать состояние нужного бита/бит уже в этом регистре.

Конечно, остается и другая возможность – написать подпрограмму-обработчик исключения и разрешить в контроллере прерываний обработку этого исключения. При этом, сразу после возникновения исключительной ситуации, управление будет передано обработчику. И уже в нем решаются вопросы «Что делать» и «Какой результат вернуть»?

Напомним расположение основных флагов исключений в регистре FPSCR.

Таблица 21.4 Флаги исключительных ситуаций в регистре FPSCR

Флаги исключительных ситуаций в регистре статуса и управления FPSCR		
Бит	**Имя**	**Назначение**
4	**IXC**	**Inexact cumulative – Неточность**
3	**UFC**	**Underflow cumulative – Переполнение вниз**
2	**OFC**	**Overflow cumulative – Переполнение вверх**
1	**DZC**	**Division by Zero cumulative – Деление на ноль**
0	**IOC**	**Invalid Operation cumulative – Недопустимая операция**

В систему команд сопроцессора включены две операции, которые позволяют считать текущее содержимое регистра FPSCR в один из регистров ЦПУ (VMRS) и, после возможной модификации содержимого этого регистра (например, для установки другого режима работы сопроцессора или сброса флага исключительной ситуации), сохранить новое содержимое регистра ЦПУ в регистре FPSCR:

```
VMRS{<cond>} Rd, FPSCR
VMSR{<cond>} FPSCR, Rm
```

В мнемонике этих команд первый символ «**V**» означает команду сопроцессора, второй «**M**» (от MOVE) – пересылку данных. Суффикс «**R**» обозначает регистр центрального процессора, а суффикс «**S**» – регистр специального назначения сопроцессора. В расположении суффиксов и операндов соблюдается общее правило

кодирования команд: первым указывается регистр-приемник, вторым – регистр-источник данных. Данные пересылаются «справа – налево».

Наличие двух команд поддерживает технологию «Чтение – Модификация – Запись», которая рекомендуется для работы со всеми регистрами специального назначения.

Команды допускают использование любых регистров ЦПУ в качестве операндов, однако применение специальных регистров R12 (LR), R13 (SP), R14 (PC) не рекомендуется.

После копирования флагов в регистр ЦПУ, они могут быть протестированы с использованием обычных логических операций и очищены.

Модернизируем программу расчета корней квадратного уравнения так, чтобы в самом ее конце проверить возможное наличие исключительной ситуации IOC («Недопустимая операция») – MyProg_4.s в том же проекте FPU_2.

```
58  ; Проверка флага исключительной ситуации
59  ; IOC ("Недопустимая операция")
60  ; Считать состояние регистра статаса и управления FPU
61  ; в регистр ЦПУ r1
62        VMRS r1, FPSCR
63  ; Протестировать нулевой бит
64        TST r1,#1
65  ; Если бит равен 0, результат корректный, продолжить
66        BEQ Next
67  ; Зафиксировано исключение "Недопустимая операция"
68  ; Сбросить бит IOC в регистре FPSCR
69        BIC r1,#1
70        VMSR FPSCR,r1
71  ; Обработка исключительной ситуации
72  Error
73        B Error
74  Next
```

Сначала содержимое регистра FPSCR считывается в свободный регистр ЦПУ (в r1), а затем тестируется с использованием команды TST. Проверяется состояние нулевого бита регистра. Если он равен 0, значит исключения не было. Можно продолжить программу (BEQ Next). В противном случае, сбрасываем флаг исключения IOC и записываем результат обратно в регистр FPSCR. Дальше следует фрагмент программы, который выполняет обработку исключительной ситуации (например, сигнализирует пользователю о некорректной операции и останавливает выполнение кода).

1) Задайте коэффициенты уравнения так, чтобы дискриминант был отрицательным. Вы должны получить флаг IOC (Недопустимая операция) при извлечении квадратного корня из отрицательного числа. Проследите в регистровом окне, что этот флаг формируется, а в процессе тестирования – автоматически сбрасывается.

2) Задайте нулевое значение коэффициента a=0. Какое/какие исключения возникнут теперь? Как выполнить программный поллинг нового исключения?

1) Например, a=9, b=5, c=1. Очистка флага IOC выполняется.

2) При вычислении первого корня будет выполняться операция деления нуля на ноль (0/0). Это недопустимая операция, так как ее результат не может быть определен. Сопроцессор формирует флаг IOC (Недопустимая операция) и в качестве результата x_1 возвращает положительное «тихое Не число» 1.#QNAN. Некорректной будет и операция деления при вычислении второго корня x_2. Однако, ее отличие от первой операции деления в том, что делимое (-5-5= -10) не равно нулю. Сопроцессор выставит флаг исключения DZC (Деление на ноль), а в качестве результата вернет -∞ (кодируется как -1.#INF). Для проверки флага исключения DZC измените маску в операции тестирования: TST r1, #2.

В среде µVision и в C++ особые числа формата с плавающей точкой имеют специальные символические обозначения. При этом первая цифра (1 или -1) отражает состояние знакового разряда особого числа, которое условно может считаться положительным или отрицательным, а далее после символа # приводится буквенное обозначение числа: qNaN – «тихое Не число»; sNaN – «сигнальное Не число»; Inf – «бесконечность» (подробно см. в 19.14). Получение такого результата вычислений свидетельствует о его выходе за пределы допустимого диапазона нормальных и субнормальных чисел с однократной точностью. Нужно корректировать алгоритм, программу или диапазон используемых переменных. Лучше всего заранее предвидеть возможные осложнения и постараться избежать их программным путем, вообще не прибегая к анализу флагов исключений (см. 21.9).

21.9 Как сравнивать числа с плавающей точкой?

21.9.1 Операции сравнения

Как мы уже отмечали, рассматривая устройство регистра статуса и управления модуля плавающей точки FPSCR (см. 20.5), при выполнении операции сравнения чисел с плавающей точкой с помощью команды сравнения CMP, в регистре FPSCR устанавливаются флаги результата сравнения N, Z, C, V, которые можно использовать для последующего условного выполнения команд или условного ветвления программы.

Поддерживаются команды сравнения двух чисел с плавающей точкой в регистрах сопроцессора между собой или одного числа с плавающей точкой с нулем (с непосредственной константой):

```
VCMP{E}{<cond>}.F32   Sn,   Sm
VCMP{E}{<cond>}.F32   Sn,   #0.0
```

Обе команды VCMP и VCMPE вычитают из первого операнда второй и устанавливают флаги в регистре FPSCR. Результат вычитания нигде не сохраняется, как и в случае операций сравнения чисел с фиксированной точкой. Отличие команд VCMP и VCMPE друг от друга состоит в том, как они обрабатывают операнды «Не числа» (NaN) – табл. 21.5.

Таблица 21.5 Обработка «Не чисел» в операциях сравнения

Команда	Как обрабатываются операнды «Не числа» NaN
VCMP	**Только тогда, когда один или оба операнда являются «Сигнальными не числами» sNaN, устанавливается флаг «Недопустимая операция» Invalid Operation (IOC).**
VCMPE	**Если один или оба операнда являются «Не числом» NaN *любого типа*, то устанавливается флаг «Недопустимая операция» Invalid Operation (IOC)**

Таким образом, вторая команда проверяет входные операнды более тщательно. Любые операнды «Не числа», в том числе «Сигнальные» (sNaN) и «Не сигнальные», так называемые «тихие Не числа» (qNaN), вызывают формирование флага недопустимой операции. Обратите внимание: «Сигнальные не числа» (sNaN) всегда вызывают исключение «Недопустимая операция». Этот факт может использоваться программистами для принудительного вызова обработчика исключения IOC (см. 19.14).

Для того, чтобы случайно не выполнить сравнение числа с плавающей точкой с «Не числом» любого типа, пользуйтесь более «жесткой» командой VCMPE. Если такой «казус» произойдет, – возникнет исключение по «Недопустимой операции».

Итак, операция сравнения VCMPE будет выполняться корректно только в том случае, когда исходные операнды являются числами в формате с плавающей точкой однократной точности в диапазоне от -∞ до +∞ включительно. Напомним, что бесконечности являются допустимыми числами и могут участвовать в операциях сравнения. Положительная бесконечность больше любого числа с плавающей точкой, а отрицательная – меньше любого числа с плавающей точкой. При сравнении положительного нуля +0 с отрицательным -0, они считаются эквивалентными.

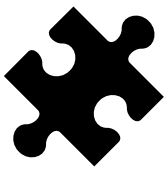
21.9.2 Как сформировать коды условного выполнения по результату сравнения?

Для того, чтобы выработать коды условного выполнения, которые *являются общими* для команд центрального процессора и сопроцессора, необходимо выполнить копирование флагов N, Z, C, V из специального регистра сопроцессора FPSCR в регистр статуса программы пользователя APSR:

```
VMRS{<cond>} APSR_nzcv, FPSCR
```

В мнемонике команды после имени регистра статуса приложения пользователя APSR через символ подчеркивания указывается список пересылаемых в него флагов nzcv. Остальные флаги регистра FPSCR (в том числе флаги исключений) – не копируются.

В обычных операциях с фиксированной точкой флаг V сигнализирует о выходе результата за возможный диапазон представления чисел (переполнении). В операциях с плавающей точкой для этой цели служат флаги исключений – «Переполнение вверх» (Overflow) и «Переполнение вниз» (Underflow). Необходимость в использовании флага V отпадает. Поэтому, смысл флагов C и V отличается от смысла этих флагов в целочисленной арифметике. Они, в том числе, сигнализируют о том, что один или оба операнда в операции сравнения являются «Не числами» (NaN), то есть сравнение – некорректно. Для исключения двусмысленностей копируйте все 4 флага. Их значения позволят точно установить коды условного выполнения **EQ, NE, GT, GE, LT, LE**, которые могут быть результатом сравнения любых знаковых чисел (как с фиксированной, так и с плавающей точкой).



При сравнении чисел с плавающей точкой не используйте коды, которые применяются для сравнения чисел без знака (HI, HS, LO, LS), а также коды наличия или отсутствия флага переноса C или переполнения V. Используйте только условия сравнения знаковых чисел. Вы можете применять их с любыми командами ЦПУ или сопроцессора. Не забывайте про преимущества условного выполнения группы команд по альтернативным условиям. Транслятор будет формировать блоки условного выполнения автоматически.

Модернизируем предыдущую программу так, чтобы избежать исключений «Деление на ноль» и «Недопустимая операция» при извлечении квадратного корня из отрицательного числа – MyProg_5.s. Сразу после загрузки коэффициентов уравнения a,b,c и расчета дискриминанта выполним проверку на наличие двух действительных корней уравнения. Вначале проверим на отличие коэффициента «a»(s0) от нуля, а затем на положительное значение дискриминанта уравнения d(s3) > 0.

```
26  ; Проверить на наличие двух корней уравления
27  ; Проверить коэффициент "a" на ноль
28  ; Если равен нулю - решения нет (один корень)
29  ; Перейти к блоку обработки ошибки Error
30        VCMPE.F32 s0,#0.0
31        VMRS APSR_nzcv, FPSCR
32        BEQ Error
33  ; Проверить значение дискриминанта на минус
34        VCMPE.F32 s3,#0.0
35        VMRS APSR_nzcv, FPSCR
36        BLT Error
37  ; Корни существуют, можно вычислять
```

```
68  ; Сохранить результаты расчетов в памяти
69        VSTM r0,{s6-s9}
70  ; Завершить программу
71        B Stop
72  Error
73  ; Здесь будет располагаться программа-обработчик
74  ; ошибки
75        B Error
76  ; Предотвратить выполнение "случайного" кода
77  Stop
78        b Stop
```

Если оба условия выполняются одновременно, то корни вычисляются, в противном случае управление передается обработчику ошибки Error. В нашем примере обработчик ошибок не разрабатывается, только обозначается его расположение в программе.

Если поставить точки останова в указанном месте, программа легко отлаживается при любых значениях коэффициентов.

Задайте последовательно коэффициенты уравнения, для которых мы ранее уже получили результаты: (9, 5, 0.25); (9, 5, 1.0); (0, 5, 1). В режиме выполнения до точки останова убедитесь в правильности работы программы.

1) Когда при сравнении чисел с плавающей точкой установится флаг Z?
2) Можно ли сравнивать число в формате FS с «Не числом»?
3) Можно ли по результату сравнения чисел с плавающей точкой выполнить ветвление по условию HI?

1) Тогда и только тогда, когда оба операнда будут числами с плавающей точкой с тождественно равными знаком, порядком и мантиссой.
2) Нет, за исключением сравнения с бесконечностями.
3) Нет. Это условие вырабатывается при сравнении чисел без знака.

21.9.3 Примеры условного выполнения команд сопроцессора

21.9.3.1 Поиск максимального числа с плавающей точкой в массиве данных в памяти

Требуется найти максимальное число с плавающей точкой в массиве данных, расположенных в памяти.

Пример 1. Вначале предположим, что объем массива невелик (всего 4 элемента) и данные расположены в кодовой памяти. На самом деле место расположения данных значения не имеет, – главное, чтобы массив был проинициализирован числами с плавающей точкой в формате FS.

При маленьком объеме массива одно из самых простых решений состоит в том, чтобы скопировать все элементы массива в регистры сопроцессора и выполнить сравнение чисел непосредственно в этих регистрах. Обратимся к проекту FPU_3. В конце программы пользователя MyProg.s зарезервируем в кодовой памяти несколько констант в формате чисел с плавающей точкой однократной точности – массив исходных данных.

```
48  ; Резервирование и инициализация
49  ; констант FS в кодовой памяти
50  Const_0 DCFS -1.9E12
51  Const_1 DCFS -7.8888
52  Const_2 DCFS +0.3377
53  Const_3 DCFS -2.0E23
```

```
11  ; Инициализация базового регистра r0 начальным адресом
12  ; массива констант в памяти
13        LDR r0,=Const_0
14  ; Загрузка 4-х регистров сопроцессора s0-s3
15        VLDM r0,{s1-s4}
16  ; Вызвать подпрограмму поиска максимального числа
17  ; из 4-х в регистрах s1-s4
18        BL Max_Float
```

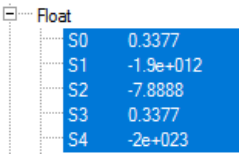
```
23  ;*****************************************************
24  ; Подпрограмма поиска максимального из 4-х чисел
25  ; с плавающей точкой
26  ; Входы: s1-s4
27  ; Выход: s0 - максимальное значение
28  ;*****************************************************
29  Max_Float
30  ; Сохранить s1 в качестве текущего максимума
31          VMOV.F32 s0,s1
32  ; Сравнить следующее число в s2 с текущим максимумом
33          VCMPE.F32 s2,s0
34          VMRS APSR_nzcv, FPSCR
35          VMOVGT.F32 s0,s2
36  ; Сравнить следующее число в s3 с текущим максимумом
37          VCMPE.F32 s3,s0
38          VMRS APSR_nzcv, FPSCR
39          VMOVGT.F32 s0,s3
40  ; Сравнить следующее число в s4 с текущим максимумом
41          VCMPE.F32 s4,s0
42          VMRS APSR_nzcv, FPSCR
43          VMOVGT.F32 s0,s4
44  ; Максимальное значение найдено в регистре s0
45  ; Возврат в основную программу
46          BX lr
47  ;*****************************************************
```

Скопируем элементы массива в регистры сопроцессора s1-s4 с использованием операции множественной загрузки с авто-инкрементированием указателя r0, предварительно загруженного начальным адресом массива.

Вызовем подпрограмму Max_Float поиска максимального значения 4-х чисел в формате FS в регистрах s1-s4.

В подпрограмме текущему максимуму в регистре s0 присваивается значение первого числа в регистре s1. Все последующие числа сравниваются с текущим максимумом и в том случае, если новое значение превышает максимум, значение максимума заменяется на большее.

Как видите, для выполнения операции обновления применяется команда VMOVGT.F32 с суффиксом условного выполнения «GT» – Строго больше. Если условие не выполняется, команда просто пропускается. Результат работы программы подтверждает ее корректность. Действительно, максимальным из 4-х представленных чисел является число 0.3377.



Аппарат условного выполнения команд ARM-процессоров эффективен и работает со всеми командами, в том числе – обработки чисел с плавающей точкой. Транслятор автоматически заключает такие команды в блоки условного выполнения IT.

Пример 2. Рассмотрим второй вариант решения той же задачи (см. MyProg_1), когда число элементов массива велико и имеет смысл последовательно считывать данные из массива, определяя каждый раз текущий максимум. Нужно просканировать элементы всего массива в цикле, контролируя число проходов цикла. В начале программы объявим общее число элементов массива,

```
 9  ; Поиск максимального значения числа с плавающей точкой
10  ; в массиве констант в ПЗУ заданной длины
11  ; Число элемнтов массива
12  N     EQU 4
13  ; Инициализация базового регистра r0 начальным адресом
14  ; массива констант в памяти
15        LDR r0,=Const_0
16  ; Инициализация регистра r1 - числом элементов массива
17        MOV r1,#N
18  ; Вызвать подпрограмму поиска максимального числа
19  ; в массиве чисел с плавающей точкой
20        BL Max_Float_1
```

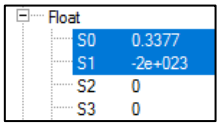
установим указатель r0 на начальный адрес массива и загрузим в счетчик числа циклов r1 число элементов массива N. Это будут параметры, передаваемые в подпрограмму Max_Float_1.

Для считывания первого элемента массива можно использовать ту же команду множественной загрузки регистров сопроцессора VLDM, указав в списке регистров только один регистр {s0}. Это значение и будет текущим максимумом. Переместим указатель r0 на адрес следующего элемента массива и декрементируем счетчик числа обработанных элементов массива r1.

```
25  ;*************************************************
26  ; Подпрограмма поиска максимального числа
27  ; с плавающей точкой в массиве заданной длины
28  ; Входы: r0 - указатель начального адреса массива
29  ;         r1 - число элементов массива
30  ; Выход: s0 - максимальное значение
31  ;*************************************************
32  Max_Float_1
33  ; Считать первый элемент массива из памяти и загрузить
34  ; в регистр текущего максимума s0
35          VLDM r0,{s0}
36  ; Указатель r0 - на следующий элемент массива
37          ADD r0,#4
38  ; Декрементировать счетчик числа элементов массива
39          SUB r1,#1
40  Loop
41  ; Сравнить значение очередного элемента массива с
42  ; текущем максимумом и при превышении - заменить
43  ; максимум
44          VLDM r0,{s1}
45          VCMPE.F32 s1,s0
46          VMRS APSR_nzcv, FPSCR
47          VMOVGT.F32 s0,s1
48  ; Указатель r0 - на следующий элемент массива
49          ADD r0,#4
50  ; Декрементировтаь счетчик числа обработаных элементов
51          SUBS r1,#1     ; Выставить флаги
52  ; Если не все элементы массива обработаны - повторить
53          BNE Loop
54  ; Максимальное значение найдено - в регистре s0
55  ; Возврат в основную программу
56          BX lr
57  ;*************************************************
```

Расположим далее тело цикла, которое будет выполняться нужное число раз. В нем считывается очередной элемент массива и проверяется, превышает ли его значение текущий максимум. Если да, то текущий максимум обновляется.

В конце цикла выполняется переход к следующему элементу массива, и декрементируется счетчик числа циклов. Подпрограмма завершается, если все элементы массива обработаны. Естественно, при ее выполнении будет получен тот же результат на том же наборе исходных данных.



1) Зачем в команде SUB (строка 51) добавлен суффикс S?
2) Почему нам пришлось программным путем увеличивать смещение в указателе r0 на +4?
3) Сколько всего регистров сопроцессора используется в этом варианте решения задачи?
4) Команда множественной загрузки регистров VLDM по умолчанию работает в режиме IA (Инкрементировать смещение после доступа к данным). Если после имени базового регистра указать суффикс «!», то после завершения доступа к данным указатель будет автоматически обновлен. При этом его дополнительного смещения отдельной командной не потребуется. Это заметно упрощает программу. Попробуйте выполнить такую модификацию.

1) Для выработки флагов в операции декрементирования r1. Напомним, что флаги в ЦПУ вырабатываются только «по требованию» программиста.
2) Команда множественной загрузки VLDM в данном варианте использования обращается к одному элементу массива, но суффикс «!» автоинкрементирования указателя после обращения к данным отсутствует.
3) Всего два s0, s1. Это очень экономично.

4) Используйте команду VLDMEA r0!, {s0}. Суффикс EA работает по умолчанию. Его можно опустить, а вот суффикс автомодификации содержимого указателя «!» нужен обязательно – см. MyProg_2.s.

21.9.3.2 Сортировка массива чисел с плавающей точкой по убыванию значений

Ранее мы подробно рассмотрели алгоритм сортировки массива целых чисел по методу так называемого «*всплывающего пузырька*» (см. 14.7.2). Этот же алгоритм можно использовать и для работы с числами в формате с плавающей точкой: попарно сравниваются два элемента массива, и элемент с большим значением «поднимается» вверх, а с меньшим – «опускается» вниз. Вариант решения задачи представлен в проекте FPU_4. В основной программе вызывается подпрограмма сортировки массива чисел с плавающей точкой в ОЗУ. В качестве параметров подпрограмме передаются начальный адрес массива и его длина.

```
 9  ; Программа сортировки массива чисел с плавающей точкой,
10  ; расположенного в ОЗУ
11  ; Число чисел в массиве
12  N      EQU 8
13
14  ; Передать два параметра в подпрограмму сортировки массива
15  ; содержимым регистров ЦПУ r8, r9
16         MOV r8, #(N-1)     ; Число пар чисел в массиве
17         LDR r9, =Array_FS ; Начальный адрес массива
18  ; Вызвать подпрограмму сортировки
19         BL Ordering_Max_Min
```

```
24  ;*****************************************************************
25  ; Подпрограмма упорядочивания массива чисел с плавающей точкой
26  ; по убывающим значениям MAX -> MIN
27  ; Входы:
28  ; (r8) - Число пар элементов массива, подлежащего обработке
29  ; (r9) - Начальный адрес массива в памяти
30  ; Выходы: Массив в памяти, упорядоченный по убыванию чисел
31  ; Используемые регистры:
32  ; r1 - регистр-указатель адреса первого числа сравниваемой пары
33  ; r2 - регистр-указатель адреса второго числа сравниваемой пары
34  ; s1 - регистр сопроцессора для хранения первого числа пары
35  ; s2 - регистр сопроцессора для хранения второго числа пары
36  ; r3 - счетчик числа обработанных пар чисел в массиве
37  ; r4 - счетчик числа проходов массива
38  ;*****************************************************************
39  Ordering_Max_Min
40  ; Инициализация счетчика числа повторных проходов массива
41         MOV r4, r8
42  ; Внешний цикл (несколько сканов массива)
43  Outer_Loop
44  ; Инициализация регистров-указателей адресами
45  ; первой пары чисел в массиве
46         MOV r1, r9
47         MOV r2, r9
48         ADD r2, #4
49  ; Инициализация счетчика числа пар чисел в массиве
50  ; для одного прохода массива
51         MOV r3, r8
```

```
52  ; Внутренний цикл (текущий проход массива)
53  Inner_Loop
54  ; Загрузить очередную пару чисел в регистры сопроцессора
55         VLDR s1, [r1]
56         VLDR s2, [r2]
57  ; Сравнить второе число с первым и переслать
58  ; флаги результатов операции сравнения
59  ; в регистр статуса программы приложения APSR
60         VCMPE.F32 s2, s1
61         VMRS APSR_nzcv, FPSCR
62  ; Если второе число «Строго больше» первого, поменять их местами
63         VSTRGT s1, [r2]
64         VSTRGT s2, [r1]
65  ; Модифицировать содержимое регистров-указателей для доступа
66  ; к следующей паре чисел
67         ADD r1, #4
68         ADD r2, #4
69  ; Декрементировать счетчик числа обработанных пар чисел
70  ; и если он не равен нулю, то продолжить текущий скан массива
71  ; (упорядочить очередную пару)
72         SUBS r3, #1
73         BNE Inner_Loop
74  ; Внутренний цикл (текущий проход) завершен
75
76  ; Декрементировать счетчик числа проходов массива и
77  ; если он не равен нулю, то выполнить очередной проход
78       SUBS r4, #1
79       BNE Outer_Loop
80  ; Массив упорядочен по убыванию Max→Min, что и требовалось
81  ; Возврат в основную программу
82       BX LR
```

```
85  ; Объявление секции даных в ОЗУ
86        ALIGN
87        AREA MyData, DATA, ReadWrite
88  ; Резервирование памяти под массив чисел с плавающей точкой
89  Array_FS  SPACE N*4
```

Подпрограмма реализована в виде двух вложенных друг в друга циклов: внешнего, выполняющего несколько сканов массива, и внутреннего, выполняющего один просмотр массива, в котором попарно сравниваются элементы массива, и элемент, значение которого больше, перемещается вверх.

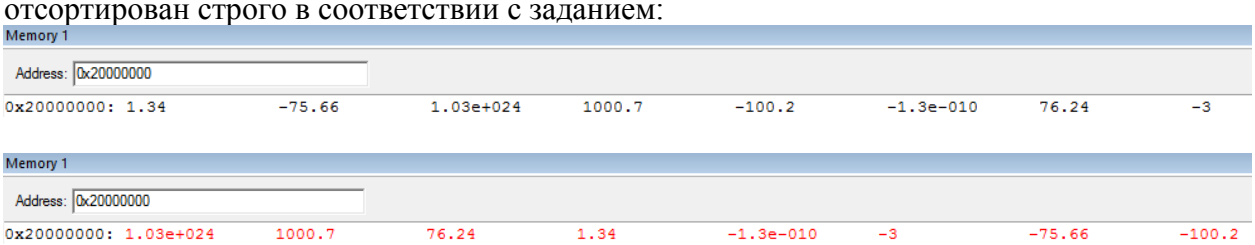
Для того, чтобы максимальное значение, если оно находится в самом конце массива, «добралось» до самого верха, должно быть выполнено число сканов массива, равное числу пар входящих в него чисел.

Обратите внимание не только на механизм сравнения чисел со знаком, но и на использование команд сохранения данных в массиве с кодом условного выполнения «GT». Он позволяет поменять пару элементов массива местами, без дополнительных команд условного ветвления программы.

Как обычно, сам массив располагается в секции данных, где под него резервируется место.

Инициализация элементов массива выполняется в процессе отладки программы путем ввода в окно дампа памяти произвольных чисел в формате с плавающей точкой.

Надеемся, что Вы сможете поставить точку останова в нужном месте программы, выполнить ее и проверить содержимое массива до и после его обработки. Один из вариантов исходных данных и результатов их обработки показан ниже. Массив отсортирован строго в соответствии с заданием:



1) Вспомните, что такое блок условного выполнения и команда процессора, его организующая? Кто ее генерирует?
2) Откройте окно дизассемблера и найдите блоки условного выполнения в нашей программе. Что означают команды начала блоков?
3) Какое максимальное количество команд условного выполнения может быть размещено в блоке?
4) Могут ли в командах блоков условного выполнения IT использоваться не альтернативные условия?

1) Все команды с кодами условного выполнения автоматически помещаются транслятором в блоки условного выполнения, начинающиеся с команды IT. Команда IT выполняет функцию переключателя, задающего порядок выполнения всех команд в блоке. Основное условие выполнения указывается в качестве операнда команды IT и совпадает с условием выполнения первой команды блока. Остальные три команды могут быть выполнены по тому же условию – дополнительный суффикс T (True) в команде IT или по альтернативному E (Else).
2) В блоке условного выполнения две команды VSTRGT с одним и тем же кодом условного выполнения GT. Поэтому, для первой команды в качестве условия выполнения указывается GT (операнд команды начала блока IT), а для второй команды с тем же условием выполнения к коду операции начала блока IT добавляется суффикс «T» (True). Все это транслятор делает автоматически.

```
0x00000052 EEF1FA10  VMRS          APSR_nzcv,FPSCR
    63:                                VSTRGT s1, [r2]
0x00000056 BFC4      ITT           GT
0x00000058 EDC20A00  VSTRGT        s1,[r2,#0x00]
    64:                                VSTRGT s2, [r1]
```

```
0x0000005C ED811A00  VSTRGT         s2,[r1,#0x00]
    67:                                ADD r1, #4
```

3) Не более 4-х. Но, если их больше, то транслятор просто последовательно создает нужное число блоков.
4) Нет, любой блок условного выполнения «окружает» только команды с противоположными (альтернативными) условиями, например, GT/LE, что и требуется в большинстве практических случаев. Если последовательно записаны две команды с разными, но не альтернативными кодами, например, GT, EQ, то транслятор последовательно создает два разных блока условного выполнения.

Наличие команд сравнения чисел с плавающей точкой и команд их арифметической обработки позволяет реализовывать практически любые вычислительные алгоритмы. Возможность условного выполнения операций

с плавающей точкой заметно упрощает программу, исключая условные ветвления, делая ее «линейной», доступной для качественного документирования и дальнейшего удобного сопровождения.

21.10 Псевдокоманды автоинкрементной и автодекрементной загрузки/сохранения регистров сопроцессора

Разработчики Ассемблера предусмотрели по две псевдокоманды, с относительно простым синтаксисом для *однократной* автоинкрементной и автодекрементной загрузки/сохранения регистров сопроцессора данными из памяти, похожие на аналогичные команды ЦПУ. Фактически транслятор заменяет их описанными выше командами множественной загрузки/сохранения регистров, включая в список регистров только один регистр назначения. Синтаксис этих псевдокоманд следующий:

```
VLDR{cond}{.size} Fd, [Rn], #offset   ; Пост-инкрементная
VLDR{cond}{.size} Fd, [Rn, #-offset]! ; Пре-декрементная

VSTR{cond}{.size} Fd, [Rn], #offset   ; Пост-инкрементная
VSTR{cond}{.size} Fd, [Rn, #-offset]! ; Пре-декрементная
```

где:

cond – опциональный код условного выполнения

.size – спецификатор размера данных (data size specifier) – **.32**, если в качестве регистра назначения Fd указан регистр однократной точности S, или **.64**, если регистром назначения является регистр двойной точности D.

Fd – регистр сопроцессора, в который загружаются данные. Это может быть либо регистр однократной точности Sd, либо регистр двукратной точности Dd.

Rn – регистр ЦПУ, содержащий базовый адрес для доступа к данным в памяти.

offset – смещение, числовое выражение, которое рассчитывается на этапе трансляции программы. Его значение должно быть равно 4, если в качестве регистра сопроцессора используется регистр однократной точности Sd, и 8 – если регистр двойной точности Dd.

Как видите, синтаксис псевдокоманд полностью соответствует аналогичным командам загрузки/сохранения регистров общего назначения данными из памяти, что удобно. Нужно только перед мнемокодом команды загрузки регистра LDR не забыть префикс «V», указывающий на работу с регистрами сопроцессора, а также вместо имени регистра ЦПУ указать имя одного из регистров сопроцессора s0-s31 или d0-d15. Для постинкрементной адресации необходимо после имени базового регистра в квадратных скобках [Rn], через запятую, указать величину смещения (4 или 8), а для пре-декрементной адресации – указать такое же отрицательное смещение внутри квадратных скобок и не забыть восклицательный знак «!» после закрывающей квадратной скобки, символизирующий пре-декрементную адресацию.

Пример 1. Модифицируем программу сортировки массива по убыванию (FPU_4) с использованием рассмотренных выше псевдокоманд (MyProg_1.s). Фрагмент подпрограммы, в котором выполнены изменения

вряд ли можно назвать более простым и понятным. Дело в том, что обе команды сохранения должны быть условными (выполняемыми по коду GT – строго больше). Но, в этом случае условными будут и операции пост-авто-инкрементирования базовых регистров r1, r2 на 4. Для того, чтобы перезапись не выполнялась, а базовые регистры инкрементировались и при альтернативном значении условия, пришлось ввести дополнительно команды ADDLE r2, #4 и ADDLE r1, #4. Будьте внимательны! Если эти команды исключить, – работа с указателями будет некорректной. Убедитесь в этом самостоятельно.

```
59  ; Сравнить второе число с первым и переслать
60  ; флаги результатов операции сравнения
61  ; в регистр статуса программы приложения APSR
62          VCMPE.F32 s2, s1
63          VMRS APSR_nzcv, FPSCR
64  ; Если второе число «Строго больше» первого, поменять их местами
65  ; с пост- авто смещением указателей
66          VSTRGT s1, [r2],#4
67          ADDLE r2,#4
68          VSTRGT s2, [r1],#4
69          ADDLE r1,#4
```

Пример 2. Рассматриваемые псевдокоманды эффектны при обработке массивов чисел с плавающей точкой для автоматического перехода от текущего элемента массива к следующему. При этом команда загрузки/сохранения будет безусловной, и опасности некорректной работы с указателями не возникнет. Предположим, требуется найти контрольную сумма массива чисел с плавающей точкой по известному начальному адресу массива и его длине и оформить этот алгоритм в виде подпрограммы. Пример программы - MyProg_2.s в проекте FPU_4. Объявим переменную числа элементов массива N и передадим параметры в подпрограмму (число элементов массива и начальный адрес).

```
12  ; Основная программа
13  ; Число чисел в массиве
14  N       EQU 8
15  ; Передать два параметра в подпрограмму расчета контрольной
16  ; суммы массива чисел в формате FS
17
18          MOV r0,#N           ; Число чисел в массиве
19          LDR r1,=Array_FS    ; Начальный адрес массива
20  ; Вызвать подпрограмму расчета контрольной суммы
21          BL Summa_FP
```

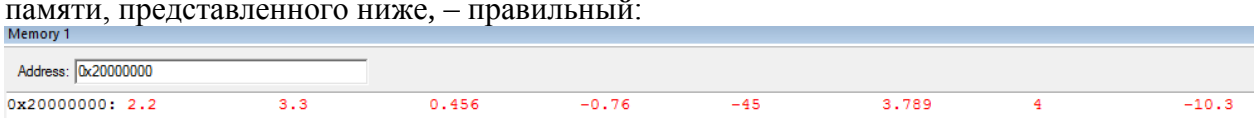
```
26  ;*************************************************
27  ; Подпрограмма расчета контрольной суммы массива
28  ; чисел с плавающей точкой
29  ; Входы:
30  ; (r0) - Число чисел в массиве
31  ; (r1) - Начальный адрес массива в памяти
32  ; Выходы:
33  ; (s0) - Контрольная сумма массива
34  ; Используемые регистры:
35  ; (s1) - Копия текущего элемента массива
36  ;*************************************************
37  Summa_FP
38  ; Обнулить регистр текущей контрольной суммы
39          VLDR.F32 s0,=0
40  Loop
41  ; Считать очередное число из массива в регистр s1
42  ; с пост-авто-смещением указателя (в регистре r1)
43          VLDR s1,[r1],#4
44  ; Накопить в регистре контрольной суммы
45          VADD.F32 s0, s1
46  ; Декрементировать счетчик числа циклов и если
47  ; не равен нулю, повторить
48          SUBS r0,#1
49          BNE Loop
50  ; Возврат в основную программу
51          BX LR
52  ;*************************************************
```

В подпрограмме обнулим регистр s0 текущей контрольной суммы и будем добавлять к ней в цикле по одному элементу массива, считанному из памяти в регистр сопроцессора s1.

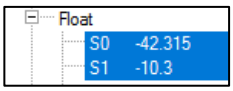
Считывание выполняем псевдокомандой загрузки регистра s1 с пост-авто-смещением указателя на 4.

Для накопления используем обычную команду сложения VADD.

Выполним инициализацию элементов массива в окне дампа памяти и запустим программу на выполнение до точки останова в конце основной программы. Результат, полученный для набора данных в памяти, представленного ниже, – правильный:



1) Убедитесь, что программа работает при любом наборе исходных данных.



2) Исследуйте программу в окне дизассемблера. Действительно ли псевдокоманда заменяется транслятором на команду

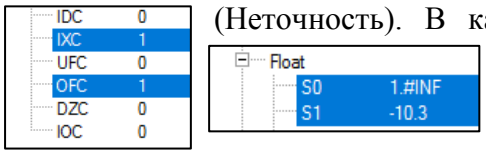
множественной загрузки регистров сопроцессора с указанием в списке регистров только одного регистра?

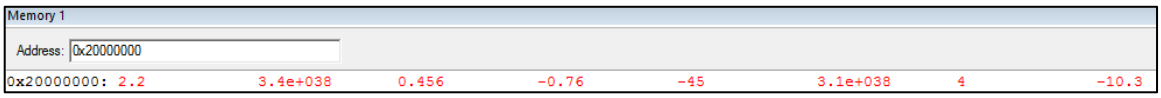
3) Проверьте, не возникает ли при сложении элементов массива исключений? Включите в массив числа, приближающиеся к максимально возможным положительным в формате с плавающей точкой. Какое при этом возникнет исключение?
4) Протестируйте поведение программы при отрицательных числах, близких к максимально возможным по модулю. Какое исключение возможно в этом случае?

2) Это действительно так. Генерируется команда VLDMIA с суффиксом IA (инкрементировать после доступа к данным) и дополнительным суффиксом «!» авто-обновления содержимого базового регистра r1 после завершения доступа. В списке только один регистр сопроцессора –{s1-s1}.

```
0x0000003C ECF10A01  VLDMIA        r1!, {s1-s1}
44:                            VLDR s1,[r1],#4
```

3) Если сумма не выходит за диапазон чисел с плавающей точкой однократной точности, то единственное исключение, которое может возникнуть – это IXC (Неточность), когда она не может быть представлена одним из репрезентабельных чисел и округляется до ближайшего представимого числа. Иная ситуация возникает, когда сумма превышает максимально допустимое число 3.4E38 в формате однократной точности. Формируется флаг исключения OFC (Переполнение вверх), а также IXC (Неточность). В качестве результата возвращается положительная бесконечность +∞ (кодируется как 1.#INF). Тестовый массив приведен ниже:





4) В этом случае возникают те же два типа исключений OFC и IXC. Дело в том, что переполнение вверх означает превышение допустимого числа по модулю. Переполнение вниз означает попадание результата в диапазон субнормальных чисел или обнуление. Отрицательная бесконечность обозначается как -1.#INF.

21.11 Команды сохранения/восстановления данных из регистров сопроцессора в стеке/из стека

Команды работы со стеком используются для временного сохранения данных из регистров сопроцессора в стеке, а также при сохранении/восстановлении контекста при обращении к подпрограммам. Так как число регистров сопроцессора достаточно велико, можно использовать в основной программе одни регистры, а в подпрограмме – другие, и обойтись, в ряде случаев, без сохранения/восстановления контекста. Тем не менее, при разработке системного программного обеспечения общего назначения и библиотек специальных функций эти операции могут понадобиться.

Таких операций две:

```
VPUSH{cond} list
VPOP{cond} list
```

где:

cond – опциональный код условного выполнения команды.

list – список (листинг) из нескольких последовательно загружаемых из стека/сохраняемых в стеке регистров сопроцессора, заключенный в фигурные скобки. Имена регистров в листинге могут разделяться запятой или представляться в виде диапазона. В листинге должен присутствовать, по крайней мере, один регистр. Имена регистров однократной точности Sd и двойной точности Dd не должны смешиваться.

Команда записи регистров сопроцессора в стек сохраняет их содержимое в *порядке возрастания номеров регистров*_(Стек) ← (Fi)…(Fk). Список регистров должен быть последовательным, без пропусков, от самого младшего регистра к старшему.

Команда восстановления содержимого регистров сопроцессора из стека загружает данные из стека в обратном порядке: сначала в регистр с максимальным номером, а затем – по мере убывания номеров регистров: (Fi)…(Fk) ← (Стек) .

По существу, это не команды процессора, а *псевдокоманды*. Транслятор автоматически заменяет их командами множественной загрузки/сохранения с нужными суффиксами IA (инкрементировать после доступа к памяти) или DB (декрементировать перед доступом). Так как стек в процессорах ARM Cortex-M4 по умолчанию автодекрементный при записи, то суффикс DB используется при сохранении содержимого регистров в стеке, а суффикс IA – при восстановлении данных из стека. В любом из этих случаев в качестве базового регистра используется указатель стека SP. Для того, чтобы выполнялось автообновление содержимого указателя стека после всех записей в стек/извлечений из стека, после имени указателя стека SP обязательно указывается суффикс обновления «!». Ниже (табл. 21.6) приведены эквивалентные операции:

Таблица 21.6 Замена псевдокоманд реальными командами процессора

Псевдокоманды работы со стеком	Какой командой фактически заменяется
VPUSH {list}	VSTMDB SP!, {list}
VPOP {list}	VLDMIA SP!, {list}

Таким образом, при просмотре программы в окне дизассемблера вместо команд записи данных в стек и восстановления из стека, Вы увидите эквивалентные им команды множественного сохранения в памяти/загрузки из памяти регистров сопроцессора.

Пример 1. Проверим, как работают команды записи в стек и восстановления из стека на примере сохранения контекста основной программы при вызове подпрограммы расчета контрольной суммы массива чисел с плавающей точкой (MyProg_3.s в том же проекте FPU_4). В подпрограмме в качестве регистра временного хранения данных использован регистр сопроцессора s1. Предположим, что в основной программе используются два регистра s1 и s2, значения которых не должны «портиться» подпрограммой. Имитируем использование регистров s1, s2 их загрузкой начальными данными 16.0 и 2.0. Передадим параметры массива в подпрограмму и вызовем ее.

```
13  ; Основная программа
14  ; Число чисел в массиве
15  N       EQU 8
16  ; Имитация использования в основной программе регистров
17  ; сопроцессора s1,s2 (загрузка константами)
18          VMOV.F32 s1,#16.0
19          VMOV.F32 s2,#2.0
20  ; Передать два параметра в подпрограмму расчета контрольной
21  ; суммы массива чисел в формате FS
22          MOV r0,#N           ; Число чисел в массиве
23          LDR r1,=Array_FS    ; Начальный адрес массива
24  ; Вызвать подпрограмму расчета контрольной суммы
25          BL Summa_FP
26  ; Результат возвращается в регистр s0
27  ; Значения в регистрах s1,s2 должны быть восстановлены
28  ; в подпрограмме
```

```
34  ;************************************************
35  ; Подпрограмма расчета контрольной суммы массива
36  ; чисел с плавающей точкой
37  ; с сохранением и восстановлением контекста в стеке
38  ; Входы:
39  ; (r0) - Число чисел в массиве
40  ; (r1) - Начальный адрес массива в памяти
41  ; Выходы:
42  ; (s0) - Контрольная сумма массива
43  ; Используемые регистры:
44  ; (s1) - Копия текущего элемента массива
45  ; (s2) - Имитация использования
46  ;************************************************
47  Summa_FP
48  ; Сохранить контекст основной программы
49  ; Содержимое регистров s1,s2 - в стек
50        VPUSH {s1,s2}
51
52  ; Имитация использования регистра s2
53        VMOV.F32 s2,#1.5
54  ; Обнулить регистр текущей контрольной суммы
55        VLDR.F32 s0,=0
```

```
56  Loop
57  ; Считать очередное число из массива в регистр s1
58  ; с пост-авто-смещением указателя (в регистре r1)
59        VLDR s1,[r1],#4
60  ; Накопить в регистре контрольной суммы
61        VADD.F32 s0, s1
62  ; Декрементировать счетчик числа циклов и если
63  ; не равен нулю, повторить
64        SUBS r0,#1
65        BNE Loop
66  ; Восстановление контекста основной программы
67        VPOP {s1,s2}
68  ; Возврат в основную программу
69        BX LR
70  ;************************************************
```

Первое, что мы должны сделать в подпрограмме – это сохранить контекст основной программы (содержимое регистров s1, s2) в стеке. Используем для этого команду VPUSH {s1,s2}.

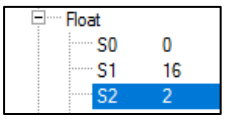
В подпрограмме будет использоваться регистр s1 – для временного хранения очередного элемента массива. Имитируем дополнительно и использование регистра s2. Загрузим его константой 1.5.

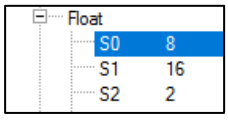
Тело исходной подпрограммы поиска суммы массива чисел с плавающей точкой оставим без изменений.

Очевидно, что перед возвратом из подпрограммы необходимо восстановить контекст основной программы, считав из стека значения регистров s1, s2. Эта операция выполняется командой VPOP {s1,s2}. Обратите внимание на то, что в листинге этой команды регистры указаны в том же порядке, что и в команде VPUSH.

Теперь можно выполнить возврат в основную программу.

Проинициализируем массив чисел с плавающей точкой единицами и проверим работу программы и подпрограммы. Содержимое регистров s0, s1, s2 перед вызовом подпрограммы и после возврата в основную программу свидетельствует о том, что контекст (в s1,s2) был правильно сохранен в стеке и правильно восстановлен из него. В регистре s0 – найденная сумма массива - 8.






```
40  ; Объявление секции даных в ОЗУ
41              ALIGN
42              AREA MyData, DATA, ReadWrite
43  ; Резервирование памяти под авто-инкрементный
44  ; и авто-декрементный стеки пользователя
45  MyStack_Inc SPACE N*4
46              SPACE N*4
47  MyStack_Dec
```

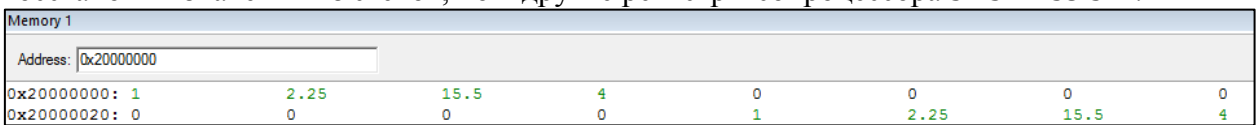
```
12  ; Основная программа
13  ; Число слов для каждого из типов стеков
14  N     EQU 8
15  ; Инициализировать указатели пользовательских стеков
16  ; r0 - авто-инкрементного
17  ; r1 - авто-декрементного
18        LDR r0,=MyStack_Inc
19        LDR r1,=MyStack_Dec
20  ; Загрузить регистры сопроцессора s0-s3 начальными данными
21        VMOV.F32 s0,#1.0
22        VMOV.F32 s1,#2.25
23        VMOV.F32 s2,#15.5
24        VMOV.F32 s3,#4.0
25  ; Сохранить эти значения в авто-инкрементном стеке
26        VSTMIA r0!,{s0-s3}
27  ; Сохранить те же значения в авто-декрементном стеке
28        VSTMDB r1!,{s0-s3}
29  ; Восстановить те же значения из авто-инкрементного стека
30  ; но в регистры s4-s7 сопроцессора
31        VLDMDB r0!,{s4-s7}
32  ; Восстановить те же значения из авто-декрементного стека
33  ; но в регистры s8-s11 сопроцессора
34        VLDMIA r1!,{s8-s11}
```

Пример 2. Создадим, кроме системного стека, свой собственный стек небольшого объема в оперативной памяти для временного хранения данных. Проверим, как выполняется запись данных и их извлечение из пользовательского стека (см. MyProg_4.s в проекте FPU_4). Системный стек резервируется в программе SturtUp_2.s. Для пользовательского автоинкрементного стека выделим первые N*4 ячеек памяти в секции данных MyData, следующие N*4 ячеек – выделим для пользовательского автодекрементного стека. Метки MyStack_Inc и MyStack_Dec будут адресами вершин соответствующих стеков. Для доступа к первому стеку

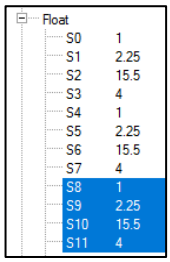
используем в качестве базового регистр r0, а ко второму – регистр r1.

Установим указатели на вершины стеков и проинициализируем первые 4 регистра сопроцессора s0-s3.

Выполним сохранение данных из этих регистров в стеках двух типов, а затем восстановим значения из стеков, но в другие регистры сопроцессора s4-s7 и s8-s11.



Результаты записи данных в стеки представлены в окне дампа памяти: с адреса 0x20000000 расположены 8 ячеек автоинкрементного стека, а начиная с адреса 0x20000040 в сторону уменьшения адресов – 8 ячеек автодекрементного стека. Обратите внимание на то, что в автоинкрементный стек сохранение данных выполняется, начиная с регистра с минимальным номером, и продолжается в сторону возрастающих номеров. Напротив, в автодекрементный стек сохранение данных выполняется, начиная с регистра с максимальным номером, – в сторону убывающих номеров. Другими словами, *данные из регистров сопроцессора со старшими номерами всегда сохраняются в памяти по старшим адресам, а младшие – по младшим*. Восстановление содержимого регистров выполняется для обоих типов стеков в обратном порядке (см. содержимое s4-s7 и s8-s11).



Для любого типа стека, достаточно в списке регистров просто перечислить имена регистров в порядке возрастания номеров, без пропусков – и запись в стек и извлечение данных из него будет корректной! Работайте в определенном месте программы (например, в подпрограмме) со стеком только одного типа. Переходите на другой тип стека только в случае необходимости. Это позволит избежать ненужных ошибок.

1) Какой тип имеет системный стек процессоров Cortex-M4F?
2) Какая команда множественного сохранения регистров сопроцессора фактически используется для записи данных в стек?
3) Почему в этой команде обязательно требуется указание суффикса «!» после имени указателя стека SP?

1) Используется стек типа Full Descending, автодекрементный, с заполнением данными (при записи) в направлении убывающих адресов.
2) Команда VSTMDB SP!, {list}.
3) Только в этом случае будет выполняться автообновление указателя стека после каждой записи.

21.12 Команды межрегистрового обмена данными

21.12.1 Пересылка данных между регистрами сопроцессора

При выполнении арифметических расчетов некоторые данные приходится сохранять в регистрах временного хранения. Выполняя вычисления с плавающей точкой, Вы имеете в своем распоряжении достаточно большой банк регистров s0-s31 (32 шт.), часть из которых

можно использовать в качестве регистров временного хранения данных, не обращаясь для этой цели ни к регистрам ЦПУ, ни к стеку:

```
VMOV{cond}.F32 Sd, Sm
VMOV{cond}.F64 Dd, Dm
```

Первая команда выполняет пересылку данных из регистра-источника числа с плавающей точкой однократной точности Sm в регистра-приемник Sd. Если нужно переслать сразу два числа однократной точности, расположенных последовательно (в регистре двойной точности Dm), то можно воспользоваться второй командой, которая копирует содержимое двойного слова из регистра-источника Dm в регистр-приемник двойной точности Dd.

21.12.2 Обмен данными между регистрами процессора и сопроцессора

Первая команда выполняет пересылку данных из регистра-источника в ЦПУ Rm в регистр сопроцессора однократной точности Sd. Вторая – в обратном направлении.

```
VMOV{cond} Sd, Rm
VMOV{cond} Rd, Sm
```

Как и в случае пересылки данных между регистрами сопроцессора, данные пересылаются «как есть», без проверки того, попадают они или нет в диапазон допустимых чисел в формате с плавающей точкой. Они пересылаются просто как 32-разрядные битовые последовательности. Следовательно, допустима пересылка любых данных, в том числе «Не чисел», «Бесконечностей», «Сигнальных не чисел», возможно несущих в себе дополнительную символьную информацию. Никакие флаги, в том числе флаги исключений при межрегистровых пересылках – не формируются.

Если Вы работаете исключительно с числами с фиксированной точкой, то можете использовать дополнительно 32 регистра сопроцессора как регистры временного хранения данных. Все, что нужно для этого – разрешить работу сопроцессора в стартовом файле проекта. После этого в Вашем распоряжении окажутся все 32 регистра сопроцессора.

Если Вы работаете с числами в формате с плавающей точкой, то регистры ЦПУ преимущественно используются:

- В качестве регистров-указателей для доступа к данным в памяти и периферийных устройствах;
- В качестве буферных регистров, принимающих информацию из периферийных устройств (в том числе, по каналам связи) с последующим преобразованием в формат числа с плавающей точкой (глава 22).
- В качестве буферных регистров, выдающих результаты расчета (управляющие воздействия) в периферийные устройства. Данные предварительно преобразовываются из формата числа с плавающей точкой в целочисленный формат (глава 22).
- В качестве регистров временного хранения, если их общего числа достаточно для решения более важных трех, перечисленных выше задач.

Регистры R13 (SP), R14 (LR), R15(PC), как специальные регистры, желательно не использовать.

21.12.3 Двойной обмен данными между регистрами процессора и сопроцессора

Имеются две команды обмена данными сразу между двумя регистрами ЦПУ и двумя регистрами сопроцессора, специфицированными в поле операндов команд:

```
VMOV{cond} Sd, Sd1, Rn, Rm    ; (Sd) ← (Rn); (Sd1) ← (Rm);
VMOV{cond} Rd1, Rd2, Sm, Sm1  ; (Rd1) ← (Sm); (Rd2) ← (Sm1);
```

Отличие этих команд от команды пересылки двойного слова в том, что регистры могут иметь любые номера, не обязательно последовательные. Применение таких команд сокращает код программы и повышает ее производительность.

Никаких сложностей в применении команд пересылки нет, поэтому примеры не приводятся. В следующей главе рассматриваются команды поддержки операций цифровой фильтрации и цифрового регулирования с переменными в формате с плавающей точкой, а также операции преобразования форматов.

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User’s Manual. Texas Instruments Inc. – 2011.
3) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
4) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
5) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

22 ПОДДЕРЖКА ЦИФРОВОГО РЕГУЛИРОВАНИЯ И ФИЛЬТРАЦИИ С ПЛАВАЮЩЕЙ ТОЧКОЙ

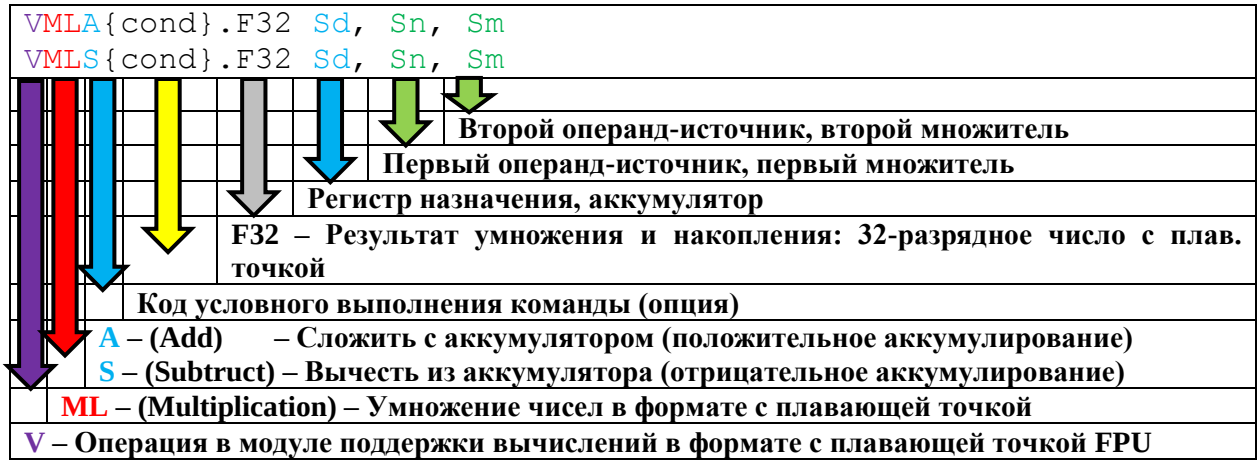
Оглавление

22.1 Операции умножения с накоплением чисел в формате с плавающей точкой

Цифровая обработка сигналов, значения которых преобразованы к формату чисел с плавающей точкой однократной точности, имеет большое практическое значение, так как существенно расширяет диапазон изменения переменных по сравнению с диапазоном представления переменных в формате с фиксированной точкой. По существу, снимаются ограничения, связанные с необходимостью представления переменных в относительных единицах – они могут быть представлены в физических единицах, так, как это удобно и разработчику, и конечному пользователю, что заметно упрощает отладку. В основе многих алгоритмов цифрового регулирования и фильтрации лежат операции умножения с накоплением, от эффективности которых зависит как производительность, так и точность реальной системы управления.

22.1.1 Умножение с округлением произведения, накоплением, округлением накопленного

Представленные ниже две команды умножения с накоплением относятся к *классическим командам умножения с накоплением*, в которых: 1) вычисляется произведение двух чисел; 2) полученное произведение округляется; 3) результат добавляется к значению в аккумуляторе или вычитается из него; 4) полученная сумма/разность еще раз округляется и сохраняется в регистре назначения.



Графическая иллюстрация алгоритма выполнения этих команд показана на рис. 22.1.

Команды умножения с накоплением не вырабатывают флагов, но могут генерировать следующие исключения:

- Недопустимая операция (Invalid Operation);
- Положительное переполнение (Overflow);
- Отрицательное переполнение (Underflow);
- Неточность (Inexact);
- Входное денормализованное число (Input Denormal).

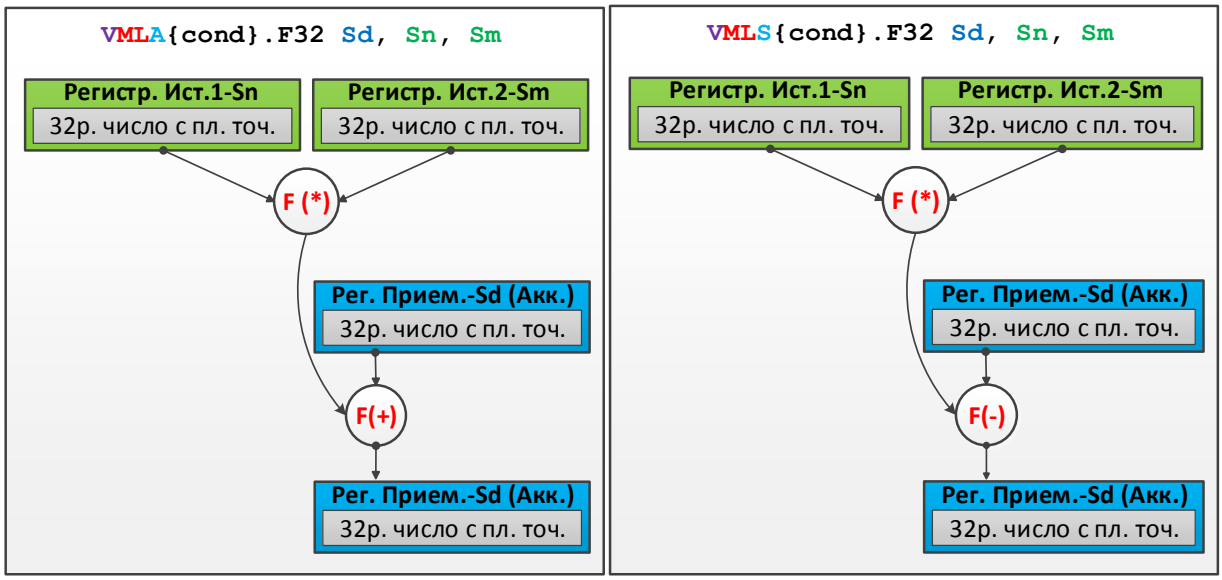


Рис. 22.1 Алгоритм выполнения команд умножения с округлением произведения и накоплением

22.1.2 Умножение, совмещенное с накоплением

Классическая операция умножения с накоплением чисел в формате с плавающей точкой выполняется в несколько этапов: умножение; округление результата умножения; добавление к содержимому аккумулятора (аккумулирование); округление результата сложения. Аппаратное устройство, которое выполняет эти действия, называется *умножителем-аккумулятором*. Так как это одна из важнейших операций в цифровой обработке сигналов, то ее качественной аппаратной реализации уделяется особое внимание при разработке сопроцессоров.

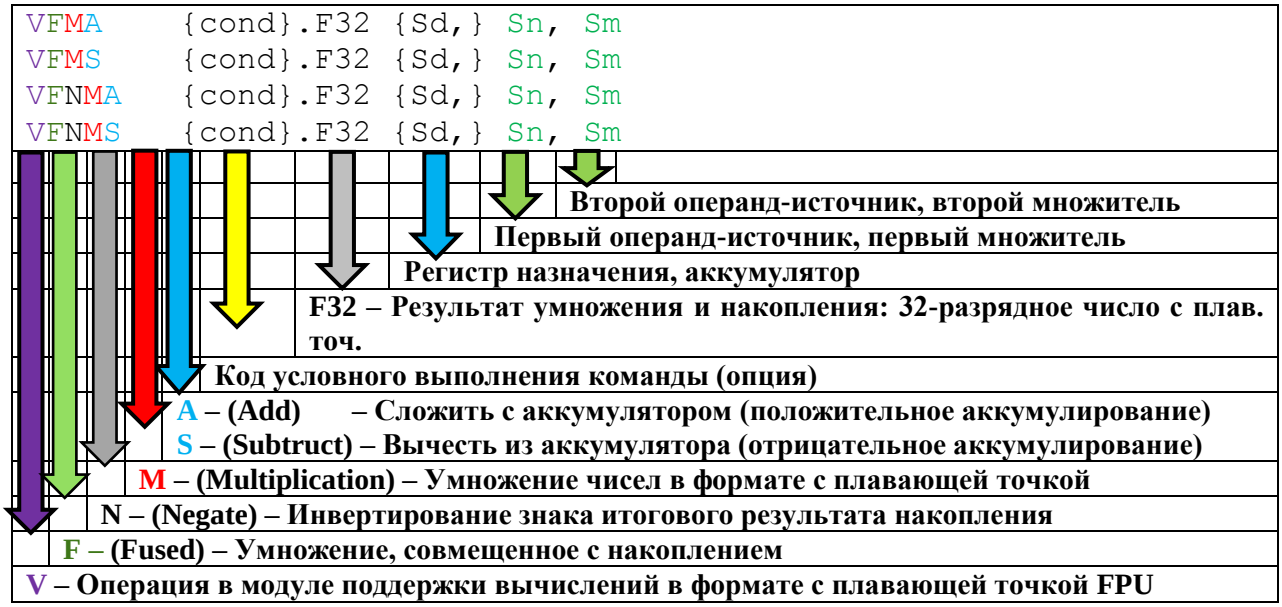
В последнее годы появились новые технологии реализации умножителя-аккумулятора, основанные на *одновременном выполнении сразу двух операций* – умножения и накопления. В этом случае вместо двух округлений требуется только одно (финальное), и процесс вычисления ускоряется и становится более точным. Такие операции умножения с накоплением получили название **Fused Multiplication**, что можно перевести с английского языка как «Совмещенные (сплавленные) операции умножения-накопления». В отличие от классического метода, когда сначала вычисляется произведение как сумма частичных произведений, а затем результат аккумулируется, новая технология предполагает аккумулирование каждой частичной суммы, полученной в процессе умножения. Округлению подвергается лишь конечный результат накопления всех частичных сумм.

Если раньше эта технология применялась только в очень мощных процессорах, то сейчас она стала доступной даже в микроконтроллерах, в том числе в ядрах Cortex-M4F. В международном стандарте IEEE 754 утверждается, что операция умножения с накоплением при одном округлении дает *более точный результат* и поэтому рекомендуется для всех операций цифровой обработки сигналов, где требуется накопление.

Специалисты по обработке чисел в формате с плавающей точкой предупреждают, что факт повышения точности зависит от конкретного приложения: в ряде случаев более точной оказывается, наоборот, классическая операция умножения с накоплением (с двумя округлениями). Поэтому, разработчики системы команд для Cortex-M4F оставили в системе команд обе возможности (классическую и «более прогрессивную»). Право выбора

конкретной команды остается за программистом. Все команды выполняются за три процессорных цикла, и итоговое быстродействие процессора оказывается одинаковым. А точность должна обязательно проверяться программистом на тестовых примерах во всем диапазоне изменения переменных.

Ниже представлена группа команд умножения с накоплением, основанная на новой технологии *параллельного умножения и накопления*, требующей только одного округления накопленного результата:



Представленная группа команд максимально адаптирована к задачам цифровой обработки сигналов, так как за один раз позволяет вычислить любое из четырех выражений:

$$A+(B\times C);\quad A-(B\times C);\quad -[A+(B\times C)];\quad -[A-(B\times C)]; \qquad (22.1)$$

Обратите внимание на то, что символ «N» в мнемонике команды означает не инвертирование произведения, а инвертирование результата положительного или отрицательного накопления. Возможность быстро вычислить любое из выражений (22.1) особенно важна при реализации цифровых регуляторов.

1) Разработайте программу, которая выполняет инициализацию трех регистров модуля FPU начальными значениями из оперативной памяти, выполняет вычисление четырех выражений (22.1) с использованием: а) классических операций умножения с накоплением; б) операций типа Fused. Сохраняет все результаты в оперативной памяти для сопоставления.
2) Задавая разные значения переменных в окне дампа памяти, убедитесь в правильности работы программы. Сравните результаты.

Возможный вариант решения представлен в проекте FPU_5 (программа MyProg.s). В конце программы в секции данных резервируется место под содержимое аккумулятора A, коэффициент K и очередную выборку цифрового регулятора/фильтра V. Предполагая, что каждая строка дампа памяти будет содержать по 8 ячеек, зарезервируем ячейки под результаты 4-х операций умножения с накоплением, выполненных классическим способом и с помощью операций типа Fused так, чтобы они располагались строго друг под другом для удобства сравнения.

```
      AREA MyData, DATA, ReadWrite
; Резервирование памяти под исходные числа с плавающей точкой
; A-аккумулятор; B - коэффициент; C - выборка
A     SPACE 4
K     SPACE 4
V     SPACE 4
REZ_1 SPACE 4   ; A+(K*V)
REZ_2 SPACE 4   ; A-(K*V)
REZ_3 SPACE 4   ; -[A+(K*V]
REZ_4 SPACE 4   ; -[A-(K*V]
      SPACE 4*4
FR_1  SPACE 4   ; A+(K*V)
FR_2  SPACE 4   ; A-(K*V)
FR_3  SPACE 4   ; -[A+(K*V]
FR_4  SPACE 4   ; -[A-(K*V]
```

Загрузим регистры сопроцессора s0-s3 исходными данными из памяти с использованием команды множественной загрузки VLDM. Установим значения указателей в регистрах r1 и r2 на адреса ячеек памяти, содержащих результаты расчетов.

```
; Демонстрация работы разных команд умножения с накоплением
MUL_ACC
; Считать исходное состояние аккумулятора, коэффициента и
; выборки в регистры сопроцессора s0,s1,s2 из памяти
      LDR r0,=A
      VLDM r0,{s0,s1,s2}
; Загрузить указатели адресами выходных ячеек памяти
      LDR r1,=REZ_1
      LDR r2,=FR_1
```

Вычислим выражения (22.1) с использованием классических операций умножения с накоплением с сохранением результатов в памяти с помощью команд VSTMIA с автоинкрементированием содержимого базового регистра после каждой записи, кроме последней.

```
; Последовательно вычислить результаты умножения с накоплением
; при использовании классических команд умножения/накопления
; и сохранить результаты в памяти
M1
      VMOV.F32 s5,s0
      VMLA.F32 s5,s1,s2
      VSTMIA r1!,{s5}
M2
      VMOV.F32 s5,s0
      VMLS.F32 s5,s1,s2
      VSTMIA r1!,{s5}
M3
      VMOV.F32 s5,s0
      VMLA.F32 s5,s1,s2
      VNEG.F32 s5,s5
      VSTMIA r1!,{s5}
M4
      VMOV.F32 s5,s0
      VMLS.F32 s5,s1,s2
      VNEG.F32 s5,s5
      VSTM r1,{s5}
```

Аналогичные вычисления произведем с использованием команд умножения с накоплением типа Fused. Как видите, эти команды отличаются тем, что, кроме операции умножения и положительного или отрицательного накопления, могут *дополнительно инвертировать итоговый результат*. Программа будет выполняться быстрее.

```
; Последовательно вычислить результаты умножения с накоплением
; с использованием "модифицированных" команд умножения/напкопления
; и сохранить результаты в памяти
L1
      VMOV.F32 s5,s0
      VFMA.F32 s5,s1,s2
      VSTMIA r2!,{s5}
L2
      VMOV.F32 s5,s0
      VFMS.F32 s5,s1,s2
      VSTMIA r2!,{s5}
L3
      VMOV.F32 s5,s0
      VFNMA.F32 s5,s1,s2
      VSTMIA r2!,{s5}
L4
      VMOV.F32 s5,s0
      VFNMS.F32 s5,s1,s2
      VSTM r2,{s5}
```

Один из вариантов входных переменных и результатов расчета выражений (22.1) представлен ниже. Он подтверждает правильность работы команд умножения с накоплением двух типов. Отличия могут проявиться за счет округления только на границах диапазона представления чисел в формате с плавающей точкой и то при большом числе последовательно выполненных операций.

Memory 1

Address: 0x20000000

```
0x20000000: 1.5      2.25     3.3378   9.01005  -6.01005 -9.01005 6.01005  0
0x20000020: 0        0        0        9.01005  -6.01005 -9.01005 6.01005  0
```

При использовании переменных с плавающей точкой задача выбора формата переменных и соответствующих им команд, как это было при работе с числами в формате с фиксированной точкой, *просто отпадает*. Все коэффициенты, выборки и результаты расчетов будут представлены в одном

и том же формате F32. Более того, полностью снимается проблема возможных переполнений при выходе результата за формат данных с фиксированной точкой и связанного с этим эффекта возможной неустойчивости систем управления, требующего специальных мер защиты: насыщения результата умножения с накоплением. Программа цифрового регулятора/фильтра становится, с одной стороны, более простой, а с другой – более надежной, *робастной* – нечувствительной к широкому диапазону изменения входных данных.

22.2 Команды преобразования форматов переменных

Выше отмечены очевидные преимущества цифровой обработки сигналов в формате с плавающей точкой. Однако остается нерешенной проблема интерфейса цифрового регулятора/фильтра, работающего в формате с плавающей точкой, с окружением процессора – портами ввода/вывода данных и периферийными устройствами, большинство которых поддерживают целочисленные переменные или переменные в формате с фиксированной точкой.

Для решения этих проблем и создания полного спектра команд, обеспечивающего комплексный подход к разработкам цифровых систем управления в формате с плавающей точкой, создатели сопроцессора снабдили его широкими встроенными возможностями по преобразованию форматов данных. Программисты получили удобный итнерфейс на уровне системы команд *между обработчиками данных с фиксированной и плавающей точками*. Дополнительные команды получили название *операций конвертирования форматов* переменных.

Сначала вводимые извне данные, являющиеся целыми числами или числами с фиксированной точкой, конвертируются в формат чисел с плавающей точкой, затем полностью обрабатываются в этом формате, а в конце вычислений, перед выдачей управляющих воздействий, преобразуются обратно в формат с фиксированной точкой.

Система команд ARM Cortex-M4F содержит все необходимые для этого операции, существенно облегчая труд программиста. При этом для выполнения конвертации данных потребуется только одна команда сопроцессора, в то время как для обычных микроконтроллеров для этой цели придется создавать или покупать специальную библиотеку подпрограмм конвертирования данных. Алгоритмы преобразования форматов данных известны, но достаточно объемны. Их реализация – трудоемкое дело, особенно для любого, произвольного формата числа с фиксированной точкой (I.Q). То, что разработчики сопроцессора включили команды конвертации в список «штатных» команда сопроцессора, то есть реализовали их на аппаратном уровне, делает им честь и существенно облегчает жизнь разработчиков.

Все команды конвертации выполняются исключительно в сопроцессоре и поэтому имеют перед мнемокодом команды префикс «V». Исходный операнд и результат операции конвертации находятся в регистрах сопроцессора. Поэтому, если нужные данные для конвертации содержатся в памяти или в регистрах ЦПУ, они должны быть предварительно загружены в регистры сопроцессора.

Сопроцессор поддерживает операции преобразования формата чисел с плавающей точкой однократной точности F32 в формат чисел с плавающей точкой половинной точности F16 и обратно. Они могут потребоваться только для хранения больших массивов данных в памяти с целью сжатия

(компрессии) данных. Обработка данных непосредственно в этом формате сопроцессором не поддерживается. Поэтому, далее такие команды преобразования форматов не рассматриваются.

22.2.1 Преобразование целых чисел в формат с плавающей точкой

Напомним, что целые числа в 32-разрядных микроконтроллерах могут быть двух типов: 1) S32 – целые со знаком в дополнительном коде; 2) U32 – целые без знака. Это могут быть входные переменные, полученные из портов ввода или периферийных устройств (в том числе интерфейсных).

Для обоих вариантов имеются команды преобразования данных в формат числа с плавающей точкой однократной точности F32:

```
VCVT{cond}.F32.S32 Sd, Sm
VCVT{cond}.F32.U32 Sd, Sm
```

К мнемокоду команды конвертирования данных «CVT» добавляется один из суффиксов типа преобразования «**.F32.S32**» или «**F32.U32**». На первом месте указывается тип выходного формата, а далее – тип входного формата. Направление преобразования данных всегда *справа – налево*, в строгом соответствии с направлением передачи данных от регистра-источника Sm к регистру-приемнику Sd. В процессе преобразования используется установленный в регистре статуса и управления модулем FPU (FPSCR) режим округления. Для большинства применений – к ближайшему возможному репрезентабельному числу в формате с плавающей точкой.

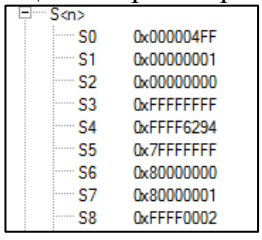
Исходные целочисленные данные загружаются в регистр-источник Sm. Имена регистров источника и приемника могут совпадать. При этом результат преобразования будет сохраняться там же – в регистре-источнике. Обычно так и делают, поскольку входные данные больше не понадобятся. Дальнейшая обработка – это уже обработка чисел в формате с плавающей точкой. При этом экономятся регистры сопроцессора.

Для демонстрации возможностей команд преобразования целых чисел в формат с плавающей точкой обратимся к проекту FPU_6. В основной программе сначала выполняется загрузка регистров общего назначения процессора целочисленными константами со знаком в дополнительном коде с охватом практически всего диапазона представления целых чисел. При этом используются псевдокоманды загрузки регистров ЦПУ константами, генерирующие в конце программы «литеральные пулы». Будем комментировать программу, одновременно представляя результаты выполнения ее фрагментов в отладчике.

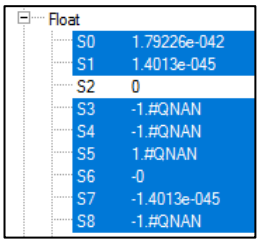
```
12  ; Загрузка регистров ЦПУ исходными данными
13      LDR r0,=+1279
14      LDR r1,= +1
15      LDR r2,=0
16      LDR r3,= -1
17      LDR r4,=-40300
18      LDR r5,=0x7FFFFFFF  ; +2147483647
19      LDR r6,=0x80000000  ; -2147483648
20      LDR r7,=0x80000001  ; -2147483647
21      LDR r8,=0xFFFF0002  ; -65534
```

```
22  ; Копирование исходных данных в регистры сопроцессора
23      VMOV s0,r0
24      VMOV s1,r1
25      VMOV s2,s3,r2,r3  ; Сдвоенная пересылка слов
26      VMOV s4,s5,r4,r5
27      VMOV s6,s7,r6,r7
28      VMOV s8,r8
```

Для преобразования констант в числа с плавающей точкой скопируем содержимое регистров ЦПУ в регистры сопроцессора.



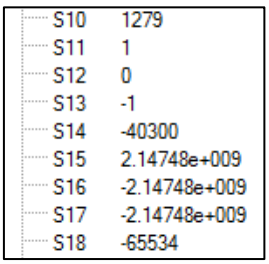
Используем как обычные команды однократной пересылки данных VMOV, так и команды двойной пересылки, когда при выполнении команды данные сразу из двух регистров-источников поступают в два регистра-приемника. Это повышает быстродействие программы.



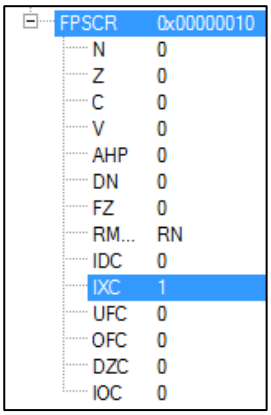
Как видите, загрузка регистров выполнена правильно, данные скопированы «как есть». При этом некоторые целые числа в интерпретации стандарта чисел с плавающей точкой оказались денормированными числами (в регистрах S0, S1, S7), некоторые – нормальными (S2, S6), а некоторые – положительными или отрицательными, так называемыми «Тихими Не Числами» qNaN.

После того, как исходные числа скопированы в регистры сопроцессора, можно выполнить их преобразование в формат с плавающей точкой. Для удобства оценки результатов преобразования выберем в качестве регистров-приемников вторую десятку регистров сопроцессора (S10-S18). Преобразование выполняется командами VCVT.F32.S32. В комментариях к ним показаны целые числа со знаком в дополнительном коде, которые должны быть преобразованы в формат F32. При выполнении этого фрагмента программы получим результат, соответствующий ожиданиям:

```
29  ; Преобразование целых чисел со знаком F32 <- S32
30      VCVT.F32.S32 s10,s0 ;+1279
31      VCVT.F32.S32 s11,s1 ;+1
32      VCVT.F32.S32 s12,s2 ;0
33      VCVT.F32.S32 s13,s3 ;-1
34      VCVT.F32.S32 s14,s4 ;-40300
35      VCVT.F32.S32 s15,s5 ;+2147483647
36      VCVT.F32.S32 s16,s6 ;-2147483648
37      VCVT.F32.S32 s17,s7 ;-2147483647
38      VCVT.F32.S32 s18,s8 ;-65534
```

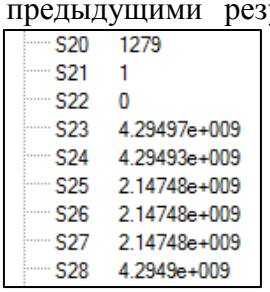


На первый взгляд, все операции преобразования целых чисел со знаком в формат с плавающей точкой (S32→F32) выполнены корректно. При преобразовании большинства чисел никаких исключительных ситуаций не возникает, о чем свидетельствует содержимое регистра статуса и управления модулем плавающей точки FPSCR. Так и должно быть, так как диапазон целых чисел является поддиапазоном чисел с плавающей точкой. Однако при преобразовании максимально возможного положительного числа со знаком +2147483647 в формат F32 формируется флаг исключения IXC (Inexact – Неточность). Он свидетельствует о том, что исходное число в новом формате F32 находится между двумя репрезентативными («представимыми») числами, и в процессе преобразования было выполнено округление результата. Следовательно, преобразование целых чисел со знаком S32 в формат с плавающей точкой F32 выполняется правильно, но, при некоторых значениях чисел, не абсолютно точно, возможно, с округлением результата.



Далее, не меняя исходного содержимого регистров S0-S8, будем рассматривать его в виде чисел без знака U32. Выполним преобразование этих чисел в формат с плавающей точкой F32. Для удобства сравнения с предыдущими результатами в качестве регистров-приемников выберем регистры второй двадцатки S20-S28.

```
40  ; Преобразование целых чисел без знака F32 <- U32
41      VCVT.F32.U32 s20,s0 ;+1279
42      VCVT.F32.U32 s21,s1 ;+1
43      VCVT.F32.U32 s22,s2 ;0
44  ; Максимально возможное целое число без знака
45      VCVT.F32.U32 s23,s3 ; -1
46  ; Близкое к максимально возможному
47      VCVT.F32.U32 s24,s4 ; -40300
48  ; Числа в середине диапазона целых чисел
49      VCVT.F32.U32 s25,s5 ; 0x7FFFFFFF  ;+2147483647
50      VCVT.F32.U32 s26,s6 ; 0x80000000  ;-2147483648
51      VCVT.F32.U32 s27,s7 ; 0x80000001  ;-2147483647
52  ; Близкое к максимальному
53      VCVT.F32.U32 s28,s8 ; 0xFFFF0000  ;-65534
```



Соответствующий фрагмент основной программы и результаты его выполнения говорят о корректности и этого типа преобразования.

Число (-1) в формате U32 является максимально возможным целым числом без знака 2^{32}-1= 4,294,967,296 (см. содержимое S23).

Для большинства чисел в конце диапазона целых чисел без знака точное преобразование в формат с плавающей точкой невозможно: результат округляется до ближайшего репрезентабельного числа и формируется флаг IXC («Неточность») в регистре FPSCR. Однако, погрешность при этом невелика и практически не влияет на результаты расчетов, поскольку речь идет об очень больших числах.

1) Какие числа с плавающей точкой считаются репрезентабельными?
2) Какой из режимов округления установлен в Cortex-M4F по умолчанию?
3) Можно ли не обращать внимания на флаг IXC («Неточность») в процессе преобразования целых чисел в числа с плавающей точкой?

1) Такие, значения которых отличаются только одним младшим разрядом мантиссы, те, которые могут быть представлены в формате числа с плавающей точкой однократной точности F32.
2) Это режим округления к ближайшему возможному числу в формате F32. Более точно – к ближайшему «четному» числу, младший бит мантиссы которого (23-й) равен нулю – RNE (см. 20.5.3). Этот режим рекомендуется для большинства приложений, как обеспечивающий максимально возможную точность.
3) Да. Этот флаг говорит только о том, что результат преобразования не точен в рамках 32-разрядного представления числа F32 (точное число может иметь в 23-м разряде мантиссы другой бит).

22.2.2 Преобразование чисел с плавающей точкой в формат целого числа

Обратное преобразование обычно используется при выводе рассчитанных управляющих воздействий в порты вывода и периферийные устройства. Оно выполняется командами:

```
VCVT{R}{cond}.S32.F32 Sd, Sm
VCVT{R}{cond}.U32.F32 Sd, Sm
```

Как и в предыдущем случае, в мнемокоде команды сначала специфицируется формат данных в регистре приемнике (S32 – целое со знаком или U32 – целое без знака), а затем – в регистре источнике (F32). В таком же порядке в поле операндов указываются имена регистров приемника и источника. Оба этих регистра должны быть регистрами сопроцессора. Если итоговые целочисленные данные должны находиться в регистрах ЦПУ, то потребуется дополнительная команда для их пересылки VMOV Rd, Sm. Можно воспользоваться и командой сопроцессора для вывода данных непосредственно в ячейку памяти (регистр периферийного устройства), например, VSTM Rn,{Sm}, в которой адрес ячейки памяти должен быть предварительно загружен в базовый регистр Rn.

Разумеется, точное преобразование числа с плавающей точкой в целое не всегда возможно:

1) Число с плавающей точкой может содержать дробную часть, например, 2.756;

2) Число с плавающей точкой может выходить из диапазона возможных целых чисел со знаком $[-2^{31} \div +(2^{31}-1)]$ или диапазона целых чисел без знака $[0 \div (2^{32}-1)]$.

В первом случае, когда преобразуемое число с плавающей точкой находится внутри диапазона целых чисел, программист может выбрать один из двух возможных вариантов округления результата преобразования:

1. Опция {R} опущена – режим округления до нуля (Zero rounding mode), когда дробная часть числа в формате F32 просто отбрасывается, – «режим усечения»;
2. Опция {R} указана – используется режим округления, установленный в регистре статуса и управления модуля плавающей точки FPSCR. Обычно это режим округления к ближайшему возможному числу RNE, представимому в формате целого числа (к ближайшему четному, если результат преобразования находится ровно посередине).

Естественно, что второй режим преобразования предпочтительнее с точки зрения обеспечения максимальной точности.

Графическая иллюстрация команды преобразования числа с плавающей точкой в целое число со знаком S32 представлена на рис. 22.2.

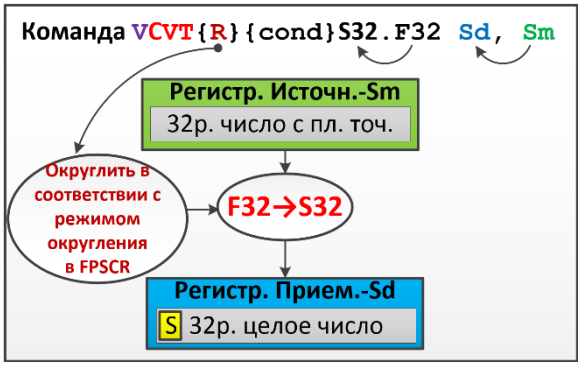


Рис. 22.2 Команда преобразования числа с плавающей точкой в целое число со знаком S32

При выходе входного операнда x(F32) за диапазон возможных целых чисел со знаком S32 вверх $[x > +(2^{31}-1)]$ или вниз $[x < -2^{31}]$ формируется флаг исключения «Недопустимая операция» (IOC), и в качестве результата возвращается максимальное положительное (+2,147,483,647) или максимальное отрицательное число (– 2,147,483,648).

Аналогичная ситуация возникает и при преобразовании числа с плавающей точкой в формат целого числа без знака U32. Если входной операнд x(F32) превышает максимально возможное число без знака $x > (2^{32}-1)$, или является отрицательным ($x < 0$), то также формируется флаг «Недопустимой операции» (IOC) и в качестве результата возвращается максимально возможное число 4,294,967,295, или ноль, соответственно.

Все входные операнды в субнормальном диапазоне преобразуются в нули.

Флаг IXC («Неточность») выставляется всегда, когда сопроцессору приходится выполнять операцию округления результата, что бывает довольно часто.

Исследуем особенности обратного преобразования чисел из формата с плавающей точкой в целочисленный формат, в том числе условия возникновения исключений (проект FPU_7). Как и в предыдущем случае, будем комментировать программу по отдельным фрагментам, предполагая, что читатель сам в состоянии в отладчике поставить точки останова после каждого фрагмента, выполнить фрагмент и проверить результаты его работы.

Выполним инициализацию регистров сопроцессора числами с плавающей точкой, используя для этой цели псевдокоманды загрузки регистров константами, которые генерируют в конце программы «литеральные пулы» с данными, а в тексте программы – операции относительного доступа к ним по содержимому счетчика команд PC и непосредственному смещению.

```
; Инициализация регистров сопроцесора s0-s11
    VLDR.F32 s0,=-100.0e12
    VLDR.F32 s1,=-12.2
    VLDR.F32 s2,=-12.5
    VLDR.F32 s3,=-12.77
    VLDR.F32 s4,=-3.0e-40
    VLDR.F32 s5,=-0.0
    VLDR.F32 s6,=+0.0
    VLDR.F32 s7,=+3.0e-40
    VLDR.F32 s8,=+12.2
    VLDR.F32 s9,=+12.5
    VLDR.F32 s10,=+12.77
    VLDR.F32 s11,=+100.0e12
```

Выберем константы так, чтобы проследить за механизмом округления, используемым сопроцессором.

```
    AREA MyData, DATA, ReadWrite
; Исходные числа в формате с плавающей точкой
VAR_F32
    SPACE 4*12
; Результаты в формате чисел со знаком S32
VAR_S32
    SPACE 4*12
; Результаты в формат чисел без знака U32
VAR_U32
    SPACE 4*12
```

Конвертируем числа в регистрах s0-s11 в формат целых чисел со знаком S32 с использованием команд VCVTR.S32.F32 с суффиксом «R», которые будут выполнять округление результата до ближайшего возможного целого числа. Можно было бы переслать результаты преобразования в регистры ЦПУ. Но значения в них отображаются только в Hex-формате, что неудобно для анализа. Используем возможности окна

```
; Сохранить исходные числа в памяти
    LDR r0,=VAR_F32
    VSTM r0,{s0-s11}
```

памяти среды µVision представлять данные в окне в любом требуемом формате. Зарезервируем в конце программы в секции данных по 12 ячеек памяти для представления исходных чисел в формате F32 и преобразованных данных в форматах S32 и U32. Сохраним исходные числа F32 в памяти, начиная с адреса VAR_F32, воспользовавшись командой множественного сохранения регистров сопроцессора в памяти VSTM.

```
; Конвертировать в формат целых
; чисел со знаком F32->S32
    VCVTR.S32.F32 s20,s0
    VCVTR.S32.F32 s21,s1
    VCVTR.S32.F32 s22,s2
    VCVTR.S32.F32 s23,s3
    VCVTR.S32.F32 s24,s4
    VCVTR.S32.F32 s25,s5
    VCVTR.S32.F32 s26,s6
    VCVTR.S32.F32 s27,s7
    VCVTR.S32.F32 s28,s8
    VCVTR.S32.F32 s29,s9
    VCVTR.S32.F32 s30,s10
    VCVTR.S32.F32 s31,s11
; Сохранить результаты в памяти
    LDR r0,=VAR_S32
    VSTM r0,{s20-s31}
```

Выполним преобразование чисел с плавающей точкой F32 в формат целых чисел со знаком S32 с помощью команд VCVTR с дополнительным суффиксом «R», что обеспечит округление результата преобразования до ближайшего целого числа со знаком.

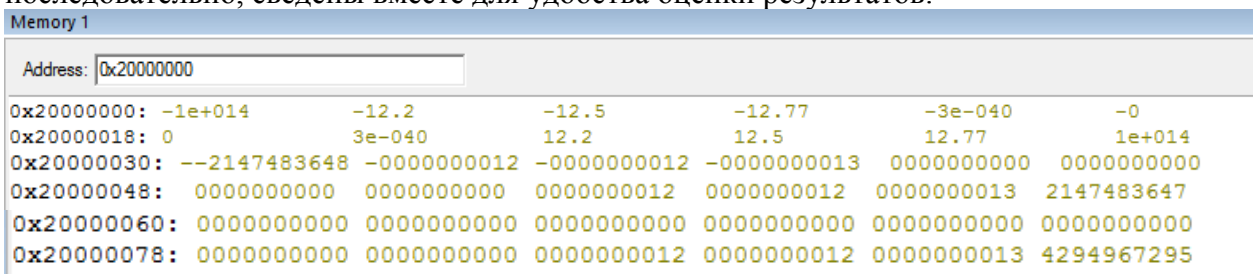
С использованием команд множественного сохранения содержимого регистров сопроцессора VSTM запишем результаты преобразования в память, начиная с адреса VAR_S32. Заметим, что данные будут пересылаться «как есть», то есть преобразованные к формату S32, что и требуется.

```
; Конвертировать в формат целых чисел
; без знака F32->U32
    VCVTR.U32.F32 s20,s0
    VCVTR.U32.F32 s21,s1
    VCVTR.U32.F32 s22,s2
    VCVTR.U32.F32 s23,s3
    VCVTR.U32.F32 s24,s4
    VCVTR.U32.F32 s25,s5
    VCVTR.U32.F32 s26,s6
    VCVTR.U32.F32 s27,s7
    VCVTR.U32.F32 s28,s8
    VCVTR.U32.F32 s29,s9
    VCVTR.U32.F32 s30,s10
    VCVTR.U32.F32 s31,s11
; Сохранить результаты в памяти
    LDR r0,=VAR_U32
    VSTM r0,{s20-s31}
```

Аналогично выполним преобразование исходных чисел в формате F32 к формату целых чисел без знака U32. Сохраним результаты преобразования в памяти, начиная с адреса VAR_U32.

Теперь можно приступить к отладке программы, открыв окно дампа памяти, начиная с адреса VAR_F32 (0x20000000). Выполните программу до метки Stop. Результаты преобразования будут сохранены в памяти. С помощью контекстного меню окна дампа памяти сначала установите режим

отображения данных в окне памяти типа *Float*, затем знаковых 32-разрядных чисел *Signed Int* в десятичном представлении *Decimal*, и, наконец, 32-разрядных чисел без знака *Unsigned Int* также в десятичном представлении. Среда µVision, выполняя функцию встроенного преобразователя форматов чисел, покажет Вам содержимое памяти в нужном формате. Три указанных выше варианта отображения данных в окне памяти, полученные последовательно, сведены вместе для удобства оценки результатов:

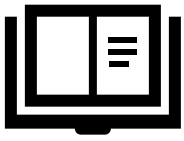


1. Преобразование чисел F32 в S32 ограничивается сверху и снизу максимальными 32-разрядными числами со знаком в дополнительном коде.
2. Преобразование чисел F32 в U32 ограничивается сверху максимальным 32-разрядным числом без знака, а снизу – нулем.
3. Все субнормальные числа автоматически заменяются нулями.
4. При наличии суффикса «R» результат округляется к ближайшему четному целому числу.
5. Знаком «--» в окнах памяти отображаются максимальные по модулю отрицательные числа -2147483648, не имеющие соответствия среди положительных чисел.

1) Удалите в командах преобразования форматов суффикс «R» – опцию округления. Получите результат преобразования. Что изменилось?
2) Определите экспериментально, по состоянию регистра статуса сопроцессора FPSCR, какие исключения формируются при выполнении команд преобразования форматов в целые числа со знаком?

1) Числа ±12.77 будут преобразованы в ±12. Дробная часть будет отброшена.
2) Для первого и последнего числа будет выставлен флаг IOC (Недопустимая операция), так как исходное число с плавающей точкой выходит за допустимый диапазон целых знаковых чисел. Для всех чисел с ненулевой дробной частью будет выставлен флаг IXC (Неточность), так как преобразование сопровождалось округлением. И только при преобразовании нулей никаких исключений не возникнет.

22.2.3 Преобразование чисел с фиксированной точкой в формат числа с плавающей точкой и обратно



Это две очень мощные команды преобразования данных, которые обеспечивают программный интерфейс между модулями проекта, работающими в формате чисел с фиксированной точкой и модулями проекта, работающими в формате чисел с плавающей точкой.

Первая команда предназначена для преобразования данных, вводимых из портов ввода, выходных регистров АЦП или полученных по каналам связи непосредственно в формат чисел с плавающей точкой, а вторая – для обратного преобразования, перед выдачей данных вовне (см. табл. 22.1).

Таблица 22.1 Команда преобразования числа с плавающей точкой в целое число со знаком

VCVT{cond}.F32.Td Td = S32, U32, S16, U16	**Sd, Sd, #fbits**	**[F32 в (Sd)] ← [Td из (Sd)] #fbits = Q (для I.Q)**	**Исключ. в FPSCR**
Преобразование числа с плавающей точкой в число с фиксированной точкой			
VCVT{cond}.Td.F32 Td = S32, U32, S16, U16	**Sd, Sd, #fbits**	**[Td в (Sd)]← [F32 из (Sd)] #fbits = Q (для I.Q)**	**Исключ. в FPSCR**

Данные в формате с фиксированной точкой Td могут быть либо 32-разрядными числами со знаком или без знака (S32, U32), либо 16-разрядными числами со знаком или без знака (S16, U16). Для первой команды они должны быть предварительно загружены в регистр сопроцессора – источник данных Sd, который одновременно будет и приемником результата преобразования. Для второй команды регистр-источник должен содержать данные в формате с плавающей точкой. В него же и будет записан результат преобразования в число с фиксированной точкой.

Эта команда – *исключение из общих правил*. В ней регистр-источник и регистр-приемник должны быть одним и тем же регистром, причем оба одинаковых имени регистров обязательно должны быть указаны в команде.

16-разрядные данные (S16, U16) должны находиться в младшем полуслове 32-разрядного регистра Sd при конвертации в формат F32. В младшем полуслове регистра Sd будет получен и результат обратного преобразования числа из формата F32 в формат S16 или U16.

В поле операндов команд, кроме имени регистра приемника/источника данных Rd, указывается число разрядов дробной части числа с фиксированной точкой #fbits, соответствующее формату I.Q.

Команды не вырабатывают флагов, но могут генерировать флаги исключений в регистре статуса и управления сопроцессора FPSCR: IOC (Недопустимая операция) и IXC (Неточность).

Способ округления, используемый в команде, зависит от направления преобразования. Для преобразования из формата с фиксированной точкой в формат с плавающей точкой используется режим Round Ties To Even (до ближайшего репрезентабельного числа). Для обратного преобразования из формата с плавающей точкой в формат с фиксированной точкой – режим Round Toward Zero (в направлении нуля с отбрасыванием дробной части). При этом имеется ввиду дробная часть, выходящая за пределы 32-разрядного числа вправо (используемая в сопроцессоре для повышения точности расчетов).

В большинстве практических задач то, что регистр-источник и регистр-приемник являются одним тем же регистром Sd, не является ограничением. При вводе/выводе информации, данные, подлежащие преобразованию, в большинстве случаев больше не нужны.

22.2.3.1 Преобразование входного однополярного аналогового сигнала в формат числа с плавающей точкой

Предположим, что требуется оцифровка аналоговых переменных, таких как давление, температура, напряжение и т.д., которые являются *однополярными*. Сигналы с датчиков физических величин аппаратным способом приводятся к стандартному входному диапазону сигналов АЦП

(например, 0-3.3 В) и далее преобразуются в цифровой код (подробно см. в 16.9). Обычно разрядность АЦП не превышает 10 – 16 разрядов, и код представляет собой целое число без знака. На рис. 22.3 представлена передаточная характеристика 12-разрядного АЦП.

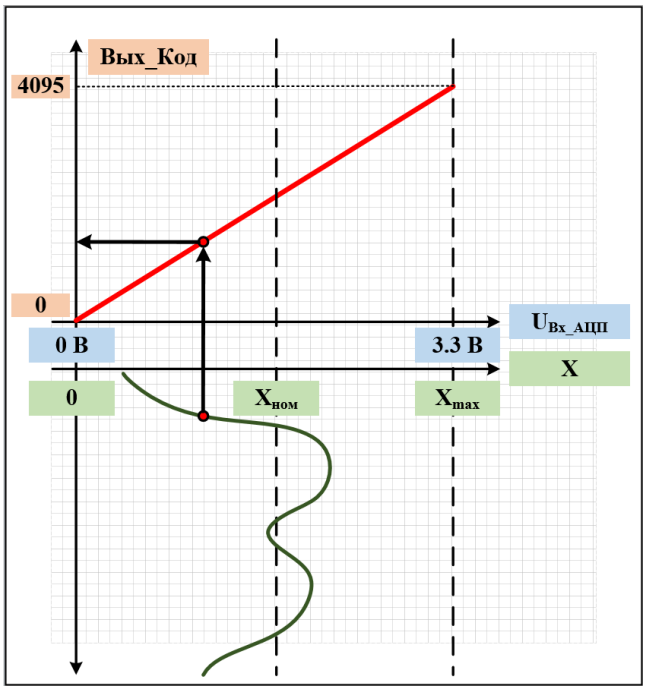


Рис. 22.3 Оцифровка однополярного аналогового сигнала

Выходной код обычно располагается в младшей или в старшей части 16-разрядного выходного регистра АЦП. Для 12-разрядного АЦП исходный формат переменной может быть U12.0 (код в младшей части полуслова) или U12.4 (код в старшей части полуслова).

Рассмотрим, для примера, второй случай. Первое, что нужно сделать, – это считать содержимое выходного регистра АЦП в один из регистров сопроцессора. Дальше нужно преобразовать полуслово U16, содержащее число в формате с фиксированной точкой U12.4, в формат с плавающей точкой X_F32[code]. Это можно сделать всего одной командой сопроцессора, например, **VCVT.F32.U16 s3,s3,#4**, где #4 – число дробных разрядов числа после десятичной точки.

Осталось только получить значение переменной, например, напряжения постоянного тока в физических величинах. Для этого нужно знать максимальное измеряемое значение переменной X_{max}[В] и максимальный код АЦП, который ему соответствует – в нашем случае 2^{12}-1=4095. Расчет выполняется по очевидной формуле:

$$X\_F32[В]=X\_F32[code]*Xmax[В]/4095. \qquad (22.2)$$

Пример программы, выполняющей это преобразование, – в проекте FPU_8. Алгоритм преобразования оформлен в виде подпрограммы Conv_U, которая вызывается из основной программы. В качестве параметра в подпрограмму передается адрес выходного порта АЦП. Порт имитируется ячейкой памяти в секции данных.

```
54      AREA MyData, DATA, ReadWrite
55    ; Имитация выходного регшистра АЦП
56    ADC_OUT SPACE 4
```

Если Вы поставите точку останова на строке 18, то можете модифицировать исходные данные в окне дампа памяти и повторить преобразование для нового значения переменной.

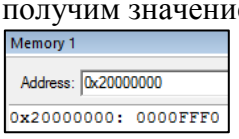
```
12  ; Установить указатель r0 на адрес выходного порта АЦП
13        LDR r0,=ADC_OUT
14  ; Вызвать подпрограмму считывания и преобразования кода
15  ; сигнала напряжения в формат F32
16  Get_U
17        BL Conv_U
18        B Get_U     ; Повторить ввод для другого значения
```

В подпрограмме устанавливаются значения необходимых коэффициентов: максимального измеряемого напряжения и максимального кода в АЦП, ему соответствующего.

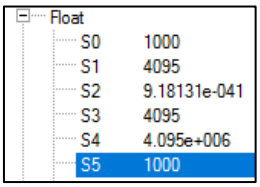
```
;*******************************************************
; Подпрограмма ввода аналогового сигнала напряжения
; с преобразованием в формат с плавающей точкой F32[В]
; Входы:
; (r0) – адрес выходного регистра АЦП
; Выходы:
; (s5) – напряжение в формате с плавающей точкой
; Используемые регистры:
; s0-s4
;*******************************************************
Conv_U
; Максимальное значение сигнала напряжения [В]
      VLDR.F32 s0,=1000.0
; Максимальный код, полученный с 12-разрядного АЦП
      VLDR.F32 s1,=4095.0
; Считать выходной код АЦП в регистр сопроцессора s2
      VLDM r0,{s2}
; Сохранить в s3
      VMOV.F32 s3,s2
; Преобразовать из формата U12.4 в формат F32
      VCVT.F32.U16 s3,s3,#4
; Преобразовать к переменной в физических единицах
; [Вольтах]в формате F32
      VMUL.F32 s4,s3,s0
      VDIV.F32 s5,s4,s1
; Возврат в основную программу
      BX LR
;*******************************************************
```

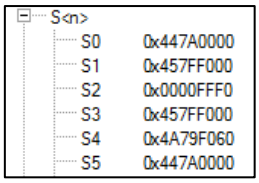
Код из выходного регистра АЦП считывается непосредственно в регистр сопроцессора s2 и преобразуется из формата с фиксированной точкой U12.4 в формат с плавающей точкой F32.

Две последние команды вычисляют выражение (22.2), после чего в регистре s5 получим значение напряжения в формате с плавающей точкой в физических единицах [В].



Пример последовательного преобразования исходного значения в АЦП 0x0000FFF0 (максимально возможного кода) показан ниже. Вы видите, что коэффициенты 1000 и 4095 загружены в регистры s0 и s1. В регистр s2 вводится исходное значение кода с АЦП (см. представление S2 в Нех-формате S<n>), которое из формата U12.4 преобразуется в число с плавающей точкой 4095 в регистре s3. Окончательный результат расчета 1000[В] в точности соответствует ожидаемому.





1) Проверьте правильность преобразования во всем диапазоне изменения входных данных, в частности, при нулевом значении сигнала и в середине диапазона.

2) Какое исключение может генерироваться при выполнении этой программы?

1) Для исходного кода 0x00000000 получим 0.0 В, а для кода 0x00008000 – значение 500.112 В.

2) Так как при выполнении операции деления возможно округление до репрезентабельного значения числа с плавающей точкой, то может формироваться флаг IXC (Неточность).

22.2.3.2 Преобразование входного разнополярного аналогового сигнала в формат числа с плавающей точкой

Большинство переменных в реальных системах управления являются знакопеременными. Следовательно, входные сигналы с датчиков будут разнополярными. Они аппаратно преобразуются на плате

микропроцессорного контроллера к стандартному промышленному диапазону (например, -5 В – +5 В) и далее – к однополярному входному диапазону АЦП (например, 0 – 3.3В) – подробно см. в 16.9. Таким образом, передаточная характеристика 12-разрядного АЦП будет выглядеть так, как показано на рис. 22.4.

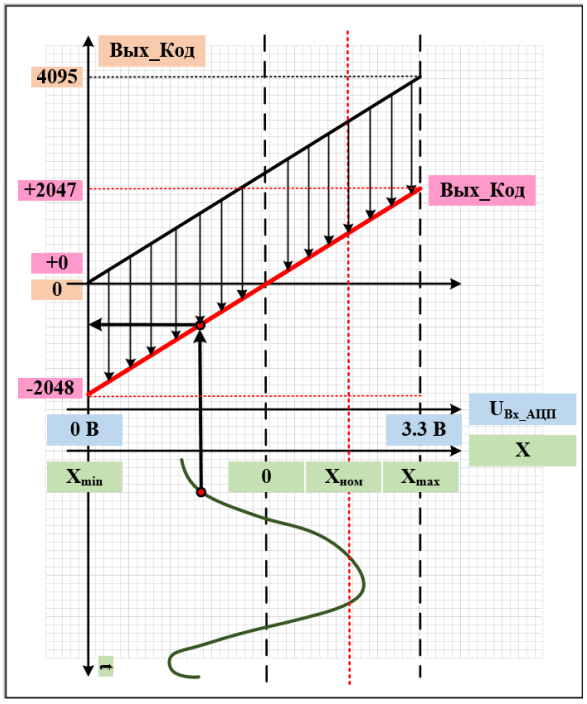


Рис. 22.4 Передаточная характеристика 12-разрядного АЦП

Выходной код АЦП может располагаться либо в младшей части 16-разрядного регистра, либо в старшей. Рассмотрим, для примера, первый вариант.

Первое, что требуется – это преобразовать исходный код числа без знака в число со знаком в дополнительном коде. Для этого содержимое выходного регистра АЦП считывается в один из регистров ЦПУ как полуслово без знака (с расширением нулями в сторону знакового разряда).

Если теперь из полученного значения вычесть целое число 2048 (половину максимального кода), то считанное из АЦП значение окажется представленным в формате S12.0 целого числа со знаком. Для его преобразования в формат числа с плавающей точкой F32[code] достаточно одной команды VCVT.F32.S16 s3,s3,#0.

Чтобы получить значение переменной, например, тока в физических величинах [A] нужно знать максимальное измеряемое значение переменной X_{max}[A] и максимальный код с АЦП, который ему соответствует – в нашем случае $2^{12}/2=2048$. Расчет выполняется по формуле:

$$X\_F32[A]=X\_F32[code]*Xmax[A]/2048. \quad (22.3)$$

```
23  ;*******************************************************
24  ; Подпрограмма ввода аналогового сигнала тока
25  ; с преобразованием в формат с плавающей точкой F32[В]
26  ; Входы:
27  ; (r0) - адрес выходного регистра АЦП
28  ; Выходы:
29  ; (s5) - ток в формате с плавающей точкой
30  ; Используемые регистры:
31  ; s0-s4
32  ;*******************************************************
33  Conv_I
34  ; Максимальное значение сигнала тока [A]
35        VLDR.F32 s0,=200.0
36  ; Значение, ему соответствующее, в единицах кода АЦП
37        VLDR.F32 s1,=2048.0
38  ; Считать выходной код АЦП в регистр ЦПУ r1 с расширением
39  ; до слова (как числа без знака)
40        LDRH r1,[r0]
41  ; Вычесть смещение 2048 для получения значения в
42  ; дополнительном коде S12.0 (или S16.0)
43        SUB r1,r1,#2048
44  ; Переслать в регистр сопроцессора s3
45        VMOV.F32 s3,r1
46  ; Преобразовать из формата S12.0 в формат F32
47        VCVT.F32.S16 s3,s3,#0
48  ; Преобразовать к переменной в физических единицах
49  ; [Амперах]в формате F32
50        VMUL.F32 s4,s3,s0
51        VDIV.F32 s5,s4,s1
52  ; Возврат в основную программу
53        BX LR
54  ;*******************************************************
```

Модифицируем основную программу проекта FPU_8. В новом файле MyProg_1.s также используем подпрограмму, которая будет выполнять всю необходимую последовательность действий для получения значения сигнала в формате с плавающей точкой в физических координатах. В отличие от предыдущей подпрограммы, прямое считывание из выходного регистра АЦП в регистр сопроцессора нерационально. Оно выполняется сначала в регистр r1 ЦПУ, где выполняется предварительное преобразование целого числа без знака в целое число со знаком S12.0 (S16.0 – эквивалентный формат в данном случае).

Только после этой операции выполняется загрузка регистра сопроцессора s3 содержимым r1 (S12.0).

Собственно преобразование в формат числа с плавающей точкой F32[code] выполняется одной командой VCVT.F32.S16 s3,s3,#0. Две последующие команды реализуют соотношение (22.3), в результате вычисления которого регистр s5 будет содержать значение тока в формате с плавающей точкой в физических единицах – Амперах.

1) Задайте в выходном регистре АЦП три значения кода, соответствующие конечной, средней и начальной точкам диапазона измерения аналогового сигнала: 0x0FFF; 0x0800; 0x0000. Получите результат преобразования.
2) Почему в данной подпрограмме пришлось вначале считывать показания АЦП в регистр ЦПУ, а не в регистр сопроцессора?
3) Какие флаги исключительных ситуаций может генерировать данная процедура преобразования форматов?

1) Получим следующие результаты: +199.902; 0.0; -200.0.
2) Нам пришлось сначала преобразовать значение числа в формате целого без знака в формат целого числа со знаком в дополнительном коде. Эта операция может быть выполнена только в ЦПУ.
3) Только флаг IXC (Неточность). Это будет всегда, когда потребуется округление до ближайшего репрезентативного числа в формате с плавающей точкой.

Операции конвертирования целочисленных форматов и форматов чисел с фиксированной точкой в формат чисел с плавающей точкой и обратно обеспечивают удобный и эффективный программный интерфейс между модулями программы, работающими в форматах с плавающей и фиксированной точкой, а также с устройствами ввода и вывода данных.

22.3 Цифровая фильтрация в формате с плавающей точкой

Ранее (глава 16, глава 17) были рассмотрены основы цифрового регулирования и цифровой фильтрации с использованием чисел в формате с фиксированной точкой. Эти технологии распространяются и на вычисления в формате с плавающей точкой. Более того, расчеты становятся проще, так как выполняются в одном и том же формате F32, не требуют тщательного подбора типов команд процессора и постоянного контроля переполнений. Наличие команд конвертирования форматов заметно облегчает алгоритмы ввода значений переменных и вывода управляющих воздействий, как это показано в предыдущем параграфе. Еще одно заметное преимущество – более высокая точность и робастность алгоритмов (нечувствительность к широкому диапазону изменения входных переменных), возможность работы с переменными непосредственно в физических единицах.

Дополнительное преимущество состоит в том, что отладка программ, написанных с использованием чисел в формате с плавающей точкой, становится более простой и наглядной. В этой главе мы покажем, как для этой цели можно использовать графические возможности логического анализатора среды µVision.

В проекте CPU_23 дан пример создания цифрового фильтра так называемого скользящего среднего с использованием переменных в формате с фиксированной точкой

(см. подробно в 17.5.3). Покажем, как та же технология с использованием кольцевого буфера выборок в памяти работает в формате чисел с плавающей точкой – проект FPU_9. В составе проекта три файла: стартовый StartUp_2.s, основная программа MyProg.s и библиотека процедур обслуживания кольцевого буфера данных Circ_Buffer.s, немного модифицированная для работы с числами в формате с плавающей точкой.

В связи с подробным изложением особенностей работы такого фильтра с реализацией его в формате данных с фиксированной точкой (см. 17.5), дадим только краткую информацию, отражающую особенности работы с числами в формате с плавающей точкой. Как и ранее, в конце основной программы резервируется место в кодовой памяти, где размещаются коэффициенты фильтра. Так как кольцевой буфер выборок содержит 8 значений (выборок), все коэффициенты одинаковы и равны 1/8. Для их размещения используются директивы Ассемблера DCFS, позволяющие сохранять данные в кодовой памяти в формате с плавающей точкой F32.

```
66  ; Зарезервировать в кодовой памяти одинаковые коэффициенты
67  ; фильтра "скользящего среднего" в формате F32
68  ; соответствующие значению 1/N =1/8=0.125
69          EXPORT Koeff
70  Koeff   DCFS  0.125
71          DCFS  0.125
72          DCFS  0.125
73          DCFS  0.125
74          DCFS  0.125
75          DCFS  0.125
76          DCFS  0.125
77          DCFS  0.125
```

```
84              AREA MyData, DATA, ReadWrite
85  ADR_0
86              SPACE N*4
87  ADR_1
88              EXPORT ADR_0  ; Начальный адрес буфера
89              EXPORT ADR_1  ; Конечный адрес буфера
90  ; Входная и выходная переменные фильтра - глобальные
91  ; для наблюдения в логическом анализаторе
92  ; Зарезервировать одно слово в памяти для имитации
93  ; порта ввода данных (выходной регистр АЦП -
94  ; расположен в младшем полуслове)
95  Port_ADC_32
96              SPACE 4
97              EXPORT Port_ADC_32
98  ; Зарезервировать одно слово в памяти для имитации
99  ; порта вывода управляющего воздействия (выход фильтра)
100 ; в формате U32
101 Port_Out_32
102             SPACE 4
103             EXPORT Port_Out_32
```

Кольцевой буфер выборок резервируется в ОЗУ, он содержит N=8 полных слов, так как каждая переменная в формате F32 занимает слово (4 байта). Начальный и конечный адреса кольцевого буфера помечаются метками ADR_0 и ADR_1. Предполагается, что данные будут вводиться через ячейку памяти, имитирующую выходной регистр АЦП – Port_ADC_32. АЦП – 10 разрядный и используется для оцифровки сигнала напряжения постоянного тока (выходной код занимает младшие 10 разрядов в младшем полуслове регистра).

```
20  ; Инициализация параметров кольцевого буфера,
21  ; предварительная очистка кольцевого буфера
22          BL Inicy_Circ_Buffer_F32
23  ; Задать число циклов работы фильтра
24          MOV r10, #20
25  ; Тело цифрового фильтра "скользящего среднего"
26  Filter_F32
27  ; Получить очередную выборку данных из АЦП
28  ; Конвертироать выборку в формат F32
29          BL Convert_X_F32
30  ; Сохранить выборку в кольцевом буфере
31          BL  Write_Circ_Buffer_F32
32  ; Рассчитать выходное управляющее воздействие
33    ; Очистить регистр сопроцессора -аккумулятор s8
34          VLDR.F32 s8,=0.0
35    ; Инициализировать счетчик числа умножений/накоплений
36          MOV r8,#N
37    ; Установить указатель в r9 на начало таблицы коэффициентов
38          ADR r9,Koeff
```

```
39  Loop_Filter
40    ; Считать очередной коэффициент в регистр s6 с пост-авто
41    ; инкрементированием указателя
42          VLDR.F32 s6,[r9],#4
43    ; Считать очередную выборку в регистр s7 с авто-модификацией
44    ; указателя в кольцевом буфере выборок
45          BL Read_Circ_Buffer_F32
46    ; Умножение с накоплением коэффициента на выборку
47    ; [(s6)*(s7)+s8] -> s8
48          VFMA.F32 s8,s6,s7
49    ; Декрементировать счетчик числа циклов
50          SUBS r8,#1
51    ; Если не все операции выполнены, повторить
52          BNE Loop_Filter
```

Для вывода результата фильтрации используется порт Port_Out_32, воспринимающий цифровой код в формате целого без знака U32.

В начале программы выполняется инициализация регистров, обслуживающих кольцевой буфер данных (в подпрограмме Inicy_Circ_Buffer_F32), и задается число циклов работы фильтра. Каждый цикл начинается с получения очередной выборки данных из порта АЦП, конвертирования ее в формат числа с плавающей точкой в подпрограмме Convert_X_F32 и сохранения выборки в кольцевом буфере Write_Circ_Buffer_F32. Далее следует расчет текущего выходного сигнала на основе всех 8-и выборок, находящихся в данный момент в кольцевом буфере, и таблицы коэффициентов фильтра в ПЗУ. Расчет выполняется с использованием *команды умножения с накоплением* чисел с плавающей точкой *типа Fused* – VFMA. При этом в качестве регистра-аккумулятора

используется регистр s8, в качестве регистра текущего коэффициента фильтра – s6, а в качестве регистра выборки сигнала – s7.

Это внутренний цикл, позволяющий выполнить одно усреднение 8-и последовательных выборок.

```
; Выдать управляющее воздействие в порт
; Port_Out_Control_32 в формате целого числа без знака U32
        BL Output_Control_U32
; Декрементировать счетчик числа циклов работы фильтра
        SUBS r10,#1
; Если не все циклы фильтрации выполнены,
; повторить для новой выборки
        BNE Filter_F32
; фильтрация заданного числа выборок выполнена
```

После этого можно преобразовать текущее выходное значение из формата числа с плавающей точкой в формат U32 числа без знака и выдать в порт вывода – подпрограмма Out_Control_U32.

Мы заказали 20 проходов фильтра. Пока они все не будут выполнены – внешний цикл будет повторяться вновь с получением очередной выборки данных, преобразованием ее в формат F32 и сохранением в кольцевом буфере.

Все подпрограммы обслуживания кольцевого буфера находятся в отдельном файле и для основной программы являются внешними. Укажем на некоторые особенности этих подпрограмм.

```
;*************************************************************
; Подпрограмма преобразования входной переменной X_U,
; считанной из порта Port_ADC, в число с плавающей
; точкой F32 в физических единицах - Вольтах
; Максимальное значение измеряемого напряжения 1000 В,
; соответствующий ему код с АЦП - 1023
; Вход: Данные с датчика напряжения в порту Port_ADC_32
;       Исходный формат U10.0 (в младшем полуслове)
; Выход: Переменная X_U в формате числа с плавающей точкой F32
;        в регистре сопроцессора s5 в физических единицах, В
;*************************************************************
Convert_X_F32
; Инициализация регистров сопроцессора
        VLDR.F32 s0,=1000.0 ; Максимальное измеряемое напряжение
        VLDR.F32 s1,=1023   ; Соответствующий ему код АЦП
; Считать код из выходного регистра АЦП в регистр сопроцессора
        VLDM.F32 r3, {s2}
; Преобразовать в формат с плавающей точкой F32_U[code]
        VCVT.F32.U32 s2,s2
; Перейти к физическим единицам - Вольтам
        VMUL.F32 s3,s2,s0
        VDIV.F32 s5,s3,s1
; Результат - в регистре сопроцессора s5
; Выход из подпрограммы
        BX lr
;*************************************************************
;*************************************************************
; Подпрограмма записи новой выборки в кольцевой буфер
; Входы: новое значение входной переменной x[k] в формате
;        F32_U[B] в регистре сопроцессора s5
; Выход: модифицированный кольцевой буфер в памяти
;*************************************************************
Write_Circ_Buffer_F32
; Записать новую выборку по текущему указателю адреса
; "свободной" ячейки в кольцевом буфере с авто-смещением
; указателя кольцевого буфера в регистре r1
        VSTR.F32 s5,[r1],#4
; Проверить превышение максимально допустимого адреса
        CMP r1, r2
; Если да, пере-инициализировать указатель кольцевого
; буфера начальным адресом буфера
        MOVHI r1,r0
; Выход из подпрограммы
        BX lr
;*************************************************************
;*************************************************************
; Подпрограмма чтения очередной выборки из
; кольцевого буфера с предварительным декрементированием
; указателя и проверкой выхода за минимально допустимый
; адрес
; Выход: s7 - считанные из буфера данные в формате F32
;*************************************************************
Read_Circ_Buffer_F32
; Декрементировать на 4 текущее значение в указателе
; кольцевого буфера
        SUB r1,#4
; Сравнить с минимально допустимым адресом
        CMP r1, r0
; Если "строго ниже", заменить максимальным адресом
        MOVLO r1,r2
; Считать выборку в регистр s7 сопроцессора
        VLDR.F32 s7,[r1]
; Выход из подпрограммы
        BX lr
;*************************************************************
```

Подпрограмма инициализации кольцевого буфера рассматривает его как линейный буфер слов, обнуляя в формате фиксированной точки. Это можно делать, так как нули в двух разных форматах представляются одинаково.

Подпрограмма Convert_X_F32 использует возможность непосредственной загрузки данных из выходного регистра АЦП в регистр s2 сопроцессора с последующим преобразованием полученного кода как 32-разрядного числа без знака в формат F32 с помощью команды VCVT.F32.U32.

Дальше выполняется коррекция полученного кода для получения значения в физических величинах – вольтах.

Подпрограмма записи новой выборки в кольцевой буфер использует для этой цели команду VSTR.F32. Ее особенность – автосмещение указателя кольцевого буфера на 4 позиции в сторону больших адресов. Если указатель выходит за максимально допустимый адрес буфера, он переинициализируется начальным адресом кольцевого буфера.

Особенность процедуры считывания текущей выборки из кольцевого буфера в том, что данные считываются сразу в регистр сопроцессора s7 с использованием косвенной адресации через регистр-указатель r1. Считывание выполняется

после операции пре-декрементирования указателя и проверки его возможного выхода за нижний допустимый адрес буфера. Если это имеет место – указатель переинициализируется максимальным адресом кольцевого буфера.

```
137  ;************************************************************
138  ; Подпрограмма выдачи управляющего воздействия
139  ; Вход: Накопленное произведение в регистре-аккумуляторе s8
140  ;       в формате числа с плавающей точкой F32
141  ; Выход: 32-разрядное слово в формате U32 в порту
142  ;        Port_Out_Control_32
143  ; Используемый регистр: r4
144  ;************************************************************
145  Output_Control_U32
146  ; Преобразовать выходную переменную из формата с плавающей
147  ; точкой в формат целого числа без знака
148         VCVT.U32.F32 s9,s8
149  ; Выдать рассчитанное выходное воздействие
150         VSTR.F32 s9,[r4]
151  ; Завершить подпрограмму
152         BX lr
153  ;************************************************************
```

Подпрограмма выдачи отфильтрованного выходного значения сопровождается попутным преобразованием числа из формата с плавающей точкой F32 в формат целого числа без знака U32. Для этого требуется только одна команда VCVT.U32.F32.

В нашем проекте выходная переменная является числом без знака. Для наблюдения за такими переменными удобно использовать логический анализатор среды µVision. Откроем в отладчике симулятора окно логического анализатора и укажем в качестве наблюдаемой переменной Port_Out_32.

Разумеется, для отладки программы придется открыть окно дампа памяти, начиная с адреса 0x20000000 для наблюдения за содержимым кольцевого буфера, а также уточнить, по какому адресу компоновщик разместил ячейку памяти, имитирующую порт ввода Port_ADC_32 (это будет 0x20000020). Именно в нее нужно будет вводить исходные данные.

Получим реакцию цифрового фильтра скользящего среднего на скачок максимально возможного значения напряжения – код в выходном регистре АЦП -1023 (см. рис. 22.5).

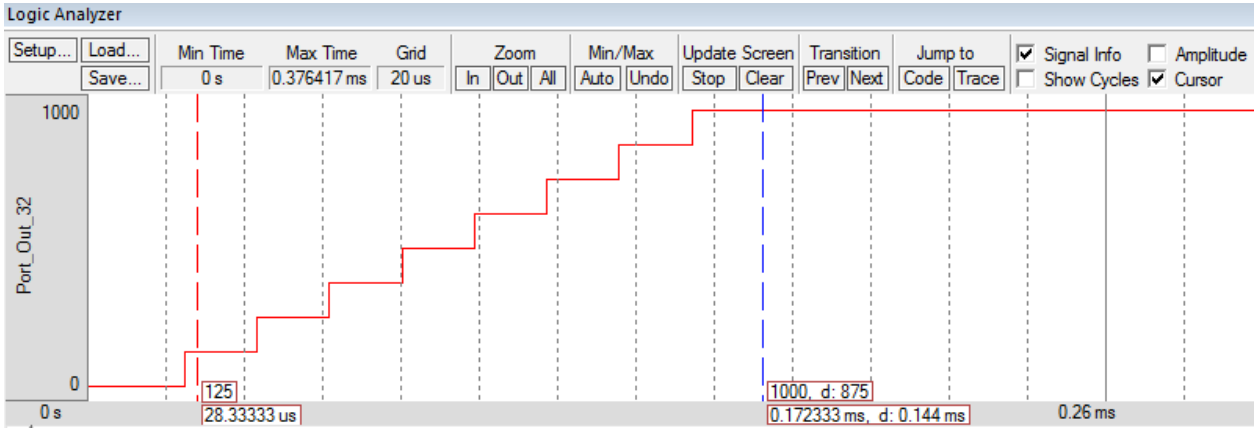


Рис. 22.5 Реакция цифрового фильтра на скачок максимального значения напряжения

Видно, что с каждым проходом фильтра выходное напряжение увеличивается с 0 на 125 В, что соответствует (1/8)*1000 В, а в установившемся режиме выход фильтра равен входному напряжению 1000 В, что и должно быть. Фильтр работает правильно.

1) Исследуйте поведение цифрового фильтра при другом значении входной переменной, например, соответствующем аналоговому сигналу 500 В. Получите график изменения выходной переменной фильтра.

2) Замените «продвинутую» команду умножения с накоплением VFMA на классическую команду VMLA с раздельным выполнением округления после операций умножения и накопления. Сравните полученные результаты.

1) Изменим код на выходе АЦП на значение 512 (ровно половина входного диапазона). После преобразования в число с плавающей точкой в физических единицах ему будет соответствовать 500.489 В (проверьте по содержимому кольцевого буфера). Это ближайшее репрезентабельное число в формате с плавающей точкой. После преобразования в целое число без знака U32 получим значение 500. График выходного сигнала представлен ниже. Работоспособность программы фильтра «скользящего среднего» подтверждается (см. рис. 22.6).

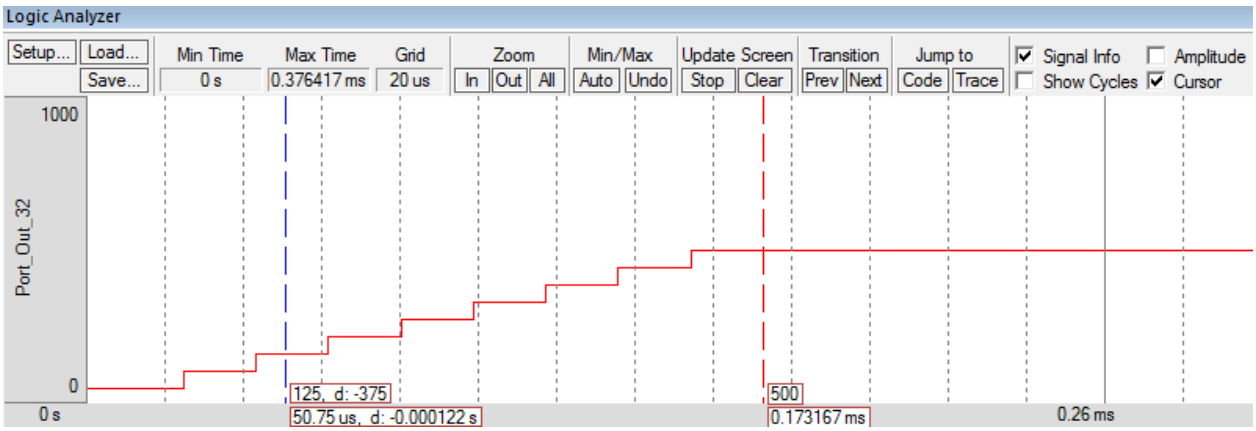
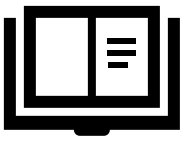


Рис. 22.6 График выходного сигнала

2) Этот вариант программы см. в файле MyProg_1.s. Разумеется, никаких отличий по точности представления результатов Вы не найдете. Для того, чтобы они проявились, нужно выполнить огромное число операций, причем с числами формата F32 на границе диапазона.

Система команд сопроцессора ядер Cortex-M4F обеспечивает эффективную реализацию алгоритмов цифровой фильтрации и цифрового регулирования в формате с плавающей точкой. По существу, Вы получаете возможность решения в реальном времени даже систем дифференциальных уравнений, моделирующих и объект управления, и управляющее устройство. Это уже новый уровень разработки и отладки систем цифрового управления. Причем, Вы можете это делать не только на языке высокого уровня, но и на Ассемблере, достигая максимальной эффективности как плотности кода, так и быстродействия.

22.4 Линейная интерполяция функций по двум опорным точкам



Линейная интерполяция функциональных зависимостей часто применяется на практике – рис. 22.7.

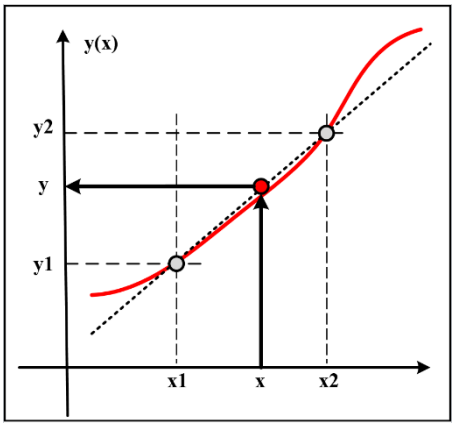


Рис. 22.7 Принцип линейной интерполяции

Речь идет об определении любого промежуточного значения функции по ее значениям в двух опорных точках x1 и x2. Вычисление y(x) выполняется по начальному значению функции y1, значению производной и приращению аргумента на интервале интерполяции:

$$y = y1 + (\frac{dy}{dx}) \cdot (x - x1) \qquad (22.4)$$

Таблица опорных значений функции может быть рассчитана аналитически или получена в результате эксперимента. Обычно она сохраняется в постоянной памяти микропроцессорной системы. По текущему значению аргумента определяется интервал, в котором справедливо соотношение (x1 ≤ x < x2), и извлекаются значения функции в опорных точках x1 и x2: y1(x1); y2(x2). Расчет требуемого значения y выполняется в соответствии с выражением (22.4).

```
;*********************************************
; Подпрограмма линейной интерполяции функции по
; двум опорным точкам
; Входы:
; s0 - x; s1 - x1; s2 - x2
; s3 - y1; s4 - y2
; Выход: s3 - y
;*********************************************
Lin_Int
; Приращения аргумента и функции на интервале
; интерполяции dx, dy
      VSUB.F32 s2,s1       ; dx -> s2
      VSUB.F32 s4,s3       ; dy -> s4
; Производная функции на интервале интерполяции
      VDIV.F32 s4,s2       ; dy/dx
; Приращение аргумента по отношению к первой
; опорной точке
      VSUB.F32 s0,s1       ; x-x1 -> s0
; Выходное значение функции
      VFMA.F32 s3,s4,s0    ; y=y1+(dy/dx)*(x-x1)
; Возврат в основную программу
      BX LR
;*********************************************
```

```
; Передать параметры в процедуру интерполяции
; Значение аргумента x (x1<=x<=x2)
      VLDR.F32 s0,=10.75
; Опорные точки и значения функций в них
; x1 -> s1, x2 -> s2
      VLDR.F32 s1,=10.2
      VLDR.F32 s2,=11.8
; Соответствующие аргументам значения функций
; y1 -> s3, y2 -> s4
      VLDR.F32 s3,=1000.0
      VLDR.F32 s4,=1200.5
; Вызвать подпрограмму линейной интерполяции
      BL Lin_Int
```

Рассмотрим подпрограмму Lin_Int, представленную в проекте FPU_10 (MyProg.s). В подпрограмму передаются пять параметров: аргумент x, значения аргумента в начале и в конце текущего интервала интерполяции x1, x2 и соответствующие им значения функций y1, y2.

В подпрограмме вычисляются приращения аргумента dx и функции dy на интервале интерполяции, определяется производная dy/dx.

Рассчитывается приращение аргумента по отношению к его значению в начале интервала интерполяции и значение функции в соответствии с (22.4).

Обратите внимание на использование команды умножения с накоплением типа Fused. Результат возвращается в основную программу содержимым одного из входных параметров в регистре s3. Для проверки

работы подпрограммы в основной программе выполняется загрузка значений в регистры s0-s4 и вызывается процедура Lin_Int. По содержимому регистрового окна Вы можете убедиться в правильности расчета значения функции.

22.5 Табличная интерполяция функциональных зависимостей

22.5.1 Общие положения

В системах управления реального времени большое значение имеет не только точность вычисления функций, но и быстродействие. Компромисс между точностью и производительностью может быть достигнут при использовании *табличного метода вычислений*, когда на заданном интервале возможного изменения аргумента ($x_{min} \leq x < x_{max}$) значения функции задаются таблицей точных значений в опорных точках, расположенных по оси абсцисс с фиксированным шагом $h=\Delta x=(x_{max} - x_{min})/k$; $k =N-1$, где N – число опорных точек (точных значений функции), а любое промежуточное значение – вычисляется одним из методов интерполяции. Точные значения функции рассчитываются на компьютере, например, с использованием электронных таблиц, и сохраняются в памяти микропроцессорной системы. При расчете промежуточных значений функции в простейшем случае используется линейная интерполяция (см. 22.4), в более сложном – интерполяция второго, третьего порядка или сплайн-интерполяция.

Так, в системах цифрового векторного управления двигателями и силовыми преобразователями, работающими во вращающихся системах координат, требуется быстрая и точная реализация тригонометрических функций sin(x), cos(x), tg(x), ctg(x), а также различных функций преобразования координат, основанных на использовании тригонометрических функций.

Идея табличной интерполяции представлена на рис. 22.8. Весь возможный диапазон изменения аргумента функции разбит на интервалы, длиной h. Все опорные точки пронумерованы, начиная с 0 до N (максимально возможного номера опорной точки).

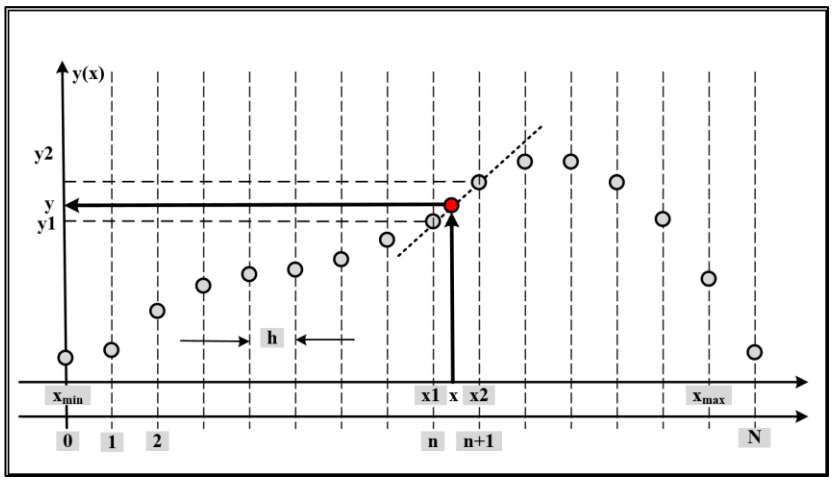


Рис. 22.8 Табличная интерполяция функции

На примере вычисления функции sin(x) покажем, как эта задача может быть быстро и достаточно точно решена с использованием команд процессорного ядра ARM-Cortex-M4F.

Значения функции для каждой опорной точки должны быть рассчитаны и в виде числа с плавающей точкой в формате F32 последовательно занесены в таблицу Tab_Sin. Для этого используется уже известная нам директива Ассемблера DCFS – зарезервировать ячейку памяти и разместить в ней константу в формате с плавающей точкой однократной точности.

Предположим, что аргумент функции sin(x) измеряется в градусах – x(F32), а значение функции должно быть определено на интервале изменения аргумента $0 \le$ x[град.] ≤ 90, включая минимальное и максимальное значения. При использовании линейной интерполяции по двум ближайшим точкам n и n+1, отличающимся ровно на 1 градус, последняя опорная точка будут иметь номер на единицу больше, чем номер, соответствующий максимальному значению аргумента x_{max}, то есть номер N=91, а всего в таблице синусов будет 92 записи.

Предположим, в качестве аргумента в процедуру расчета функции sin(x) содержимым регистра сопроцессора s0 передается значение x[град] в формате с плавающей точкой x(F32). Нужно извлечь из таблицы опорных точек Tab_Sin два последовательных числа, соответствующих рабочему интервалу интерполяции y1(F32) и y2(F32) и рассчитать выходное значение y(F32). Предлагается такой алгоритм:

1) Определить номер n нужной нам записи в таблице синусов. Для этого преобразуем значение аргумента с плавающей точкой x(F32) в формат целого числа без знака x(U32), используя режим усечения (без суффикса «R» округления до ближайшего целого) – VCVT.U32.F32. Команда эквивалентна операции получения целой части числа в формате с плавающей точкой.
2) Умножим полученный номер n на 4 (размер одной записи в таблице в байтах) для получения смещения от начального адреса таблицы до эффективного адреса расположения в ней числа y1(F32).
3) Добавим смещение к начальному адресу таблицы для получения адреса размещения y1(F32) в таблице.
4) С помощью команды множественной загрузки регистров сопроцессора данными из памяти VLDM r0, {s1,s2} считаем сразу два значения функции в начале и конце рабочего интервала интерполяции: y1(F32), y2(F32).
5) Определим относительное значение смещения аргумента Δx^* на текущем интервале интерполяции. Для этого выполним обратное преобразование целого без знака n(U32) в число с плавающей точкой n(F32) VCVT.F32.U32 и найдем разность Δx^*(F32) = x(F32)-n(F32). Эта операция соответствует получению дробной части аргумента.
6) Загрузим в один из регистров сопроцессора приращение функции на текущем интервале интерполяции y2(F32)-y1(F2) и умножим его на относительное значение аргумента на этом интервале Δx^*(F32). Добавим полученное значение к начальному значению функции y1(F32). Эта операция может быть выполнена всего одной командой умножения с накоплением VMLA или VFMA.

Обратите внимание, что представленный выше алгоритм работоспособен при любом значении h: как целом, так и вещественном. Целочисленное значение h предпочтительнее.

22.5.2 Подпрограмма расчета функции sin(x) в первом квадранте аргумента

В проекте FPU_11 представлена подпрограмма расчета функции sin(x) в первом квадранте возможного изменения аргумента (от 0 до 90 град.), даны примеры ее тестовых вызовов из основной программы MyProg.s для нескольких, заданных непосредственными константами, значений аргумента.

```
; Передать аргумент в подпрограмму содержимым регистра s0
; Вызвать подпрограмму расчета синуса угла Calc_Sin_90
; в первом квадранте аргумента
      VLDR.F32 s0,=0.0  ; 0 [Град.]
      BL Calc_Sin_90
M1
      VLDR.F32 s0,=0.578  ; 0,578 [Град.]
      BL Calc_Sin_90
M2
      VLDR.F32 s0,=30.00  ; 30.00 [Град.]
      BL Calc_Sin_90
M3
      VLDR.F32 s0,=30.222 ; 30.222 [Град.]
      BL Calc_Sin_90
M4
      VLDR.F32 s0,=30.999 ; 30.999 [Град.]
      BL Calc_Sin_90
M5
      VLDR.F32 s0,=45.00  ; 45.00 [Град.]
      BL Calc_Sin_90
M6
      VLDR.F32 s0,=45.07  ; 45.07 [Град.]
      BL Calc_Sin_90
M7
      VLDR.F32 s0,=90.00  ; 90.00 [Град.]
      BL Calc_Sin_90
```

В конце подпрограммы с помощью директив DCFS инициализируется таблица опорных значений функции с шагом 1 градус (показано начало таблицы). Эти значения Вы можете без труда получить на компьютере с использованием электронных таблиц и скопировать в ассемблерную программу. Вам придется только заменить запятую, используемую в электронных таблицах в качестве десятичной точки, на точку, которая является разделителем целой и дробной части числа в языке Ассемблер.

```
; Таблица значений функции Sin(x) с шагом 1 [Град.]
; Выравнять по границе полного слова (4-м байтам)
      ALIGN
Tab_Sin
      DCFS  0.0000000000  ; x[Град.]  0
      DCFS  0.0174524064  ; x[Град.]  1
      DCFS  0.0348994967  ; x[Град.]  2
      DCFS  0.0523359562  ; x[Град.]  3
```

Перед началом таблицы для ускорения доступа к 32-разрядным данным применяется директива ALIGN – выравнивания по границе полного слова.

Подпрограмма реализована в соответствии с описанным выше алгоритмом (см. 22.5.1). Отметим только некоторые принципиальные особенности.

Система команд сопроцессора не содержит специальных команд получения целой и дробной части числа в формате с плавающей точкой. Целая часть числа (в данном случае – значение n) может быть получена путем преобразования аргумента из формата x(F32) в формат x(S32) в режиме усечения – отбрасывания дробной части. Если полученный результат подвергнуть обратному преобразованию из формата n(S32) в формат n(F32) и вычесть из исходного значения аргумента x(F32), то можно получить дробную часть исходного числа. В подпрограмме, таким образом, рассчитывается относительное значение аргумента на интервале интерполяции.

```
;*****************************************************************
; Процедура расчета функции Sin(x) для значения аргумента
; в формате числа с плавающей точкой x(F32) в Градусах
; в первом квадранте 0.0 <= x(F32) <= 90.00
; Вход: s0 - значение аргумента x(F32)[Град.]
; Выход: s5 - значение функции Sin(x) - в формате F32
; Используемые регистры ЦПУ:
; r0 - базовый регистр для косвенного доступа к таблице Sin(x)
; r1 - число байт в слове с плавающей точкой F32
; r2 - смещение в байтах для доступа к значению y1(F32)
; r3 - номер опорной точки n - начала текущего интервала
;      интерполяции
; Используемые регистры сопроцессора:
;       s1,s2 - значения функции в начале и конце
;               интервала интерполяции y1(F32) и y2(F32)
;       s3 - x1(U32) - соответствует номеру опорной точки n
;       s4 - относительное приращение аргумента dx*(F32)
;*****************************************************************
```

Как мы уже не раз отмечали, команда множественной загрузки регистров сопроцессора VLDM очень эффективна. В данном случае она позволяет по косвенному адресу первого числа в таблице загрузить одновременно пару значений функции, участвующих в процедуре интерполяции.

```
59  Calc_Sin_90
60  ; Загрузить базовый регистр r0 начальным адресом таблицы Sin(x)
61  ; на интервале от 0 до 90 Град.
62        ADR r1, Tab_Sin
63  ; Загрузить регистр r1 числом байт в слове с плавающей точкой
64        MOV r2, #4
65  ; Преобразовать входное значение аргумента x(F32) в целое
66  ; без знака x1(U32) - номер n опорной точки в начале рабочего
67  ; интервала интерполяции функции
68        VCVT.U32.F32 s3,s0
69        VMOV.F32 r3,s3
70  ; Определить величину смещения адреса по таблице синусов
71  ; для доступа на начальному значению y1(F32)
72        MUL r2,r3          ; (r2)=4*n
73  ; Добавить смещение: получить эффективный адрес доступа
74        ADD r1,r2          ; (r1)= Tab_Sin+4*n
75  ; Получить значения функции sin(x) в начале и в конце
76  ; рабочего интервала интерполяции y1(F32), y2(F32)
77        VLDM r1,{s1,s2}
78  ; Рассчитать относительное приращение аргумента на интервале
79  ; интерполяции dx[F32]= x(F32)-x1(F32)
80    ; Обратное преобразование числа n(U32) в n(F32)
81        VCVT.F32.U32 s3,s3
82    ; Относительное значение преращения аргумента
83        VSUB.F32 s4,s0,s3
84  ; Инициализация регистра-аккумулятора s5 начальным значением
85  ; на интервале интерполяции
86        VMOV.F32 s5,s1       ; (s5)=y1[F32]
87  ; Приращение функции на интервале интерполяции
88        VSUB.F32 s2,s1       ; (s2)=dy[F32]=y2[F32]-y1[F32]
89  ; Интерполяция на текущем рабочем интервале
90        VFMA.F32 s5,s2,s4    ; (s5) =y1[F32]+dy[F32]*dx[F32]
91  ; Возврат из процедуры
92        BX LR
93  ;*****************************************************************
```

Обратите внимание и на команду умножения с накоплением, которая одна выполняет весь комплекс действий, реализующих алгоритм интерполяции.

Выполните трансляцию файлов проекта, сборку проекта, перейдите в режим отладки. Поставьте точки останова по всем меткам M1 – M7. В режиме прогона следите за результатами расчета функции на тестовых значениях аргумента.

Для ряда значений аргументов (0, 30, 90 град.) вы получите абсолютно точные значения функций. В остальных случаях возможны округления при вычислениях, сопровождаемые выработкой флагов IXC (Неточность).

1) Линейная интерполяция будет давать тем большую точность расчета, чем меньше шаг h по таблице значений функции. Определите экспериментально, с какой точностью работает наша программа при шаге по таблице в 1 град.

2) Что делать, если точность недостаточна для Вашего приложения?

1) Точность достаточна для большинства инженерных расчетов – ошибка проявляется только в 5-м, 6-м знаках числа после десятичной точки.

2) Первое – уменьшить шаг по таблице опорных значений функции, увеличив, соответственно, объем таблицы в памяти. Второе – использовать интерполяцию более высокого порядка, так как линейная интерполяция, по определению, дает ошибки (реальная кривая аппроксимируется всего лишь ломаной линией).

22.5.3 Подпрограмма расчета функции sin(x) на всем периоде изменения

Усложним задачу. Требуется рассчитать значение функции Sin(x) для любого положительного значения аргумента, выраженного в градусах [град] и являющегося числом с плавающей точкой в формате F32.

Функция Sin(x) периодическая, следовательно, ее значения будут повторяться через каждые 360 [град]. Первое, что нужно сделать, – привести значение аргумента x(F32) к диапазону значений, измеренных относительно начала текущего периода [0.0 ≤ x(F32) ≤ 360.0]. Назовем это значение аргумента эффективным значением внутри одного периода α(F32). Возможный алгоритм расчета α(F32):

1) Определим номер периода n_per, которому соответствует значение аргумента x(F32). Для этого разделим аргумент x(F32) на длину периода в градусах 360.0. Получим число с целой и дробной частью, например, n_per(F32) =17.345. Нас интересует только дробная часть – относительное приращение аргумента на текущем периоде изменения

функции Δx*(F32). Технологию поиска дробной части числа в формате F32 мы освоили ранее (см. 22.5.2):

a. Преобразуем n_per(F32) в формат целого числа n_per(U32) с помощью команды VCVT.U32.F32.
b. Выполним обратное преобразование n_per(U32) в формат числа с плавающей точкой с помощью команды VCVT.F32.U32 и вычтем полученное значение из исходного n_per(F32). Результат и будет представлять собой относительное приращение аргумента Δx*(F32) на текущем периоде изменения функции.

2) Остается только умножить это значение на 360.0, чтобы получить эффективное значение аргумента α(F32) на одном периоде изменения функции.

Используем уже созданную ранее подпрограмму расчета функции sin(x) в диапазоне изменения аргумента в первом квадранте функции sin(x) - от 0 до 90 град. На рис. 22.9 показано, как можно получить расчетное значение аргумента $\alpha_{рас}$ по его эффективному значению на периоде α, если функция ищется во II-м, III-м и IV-м квадрантах.

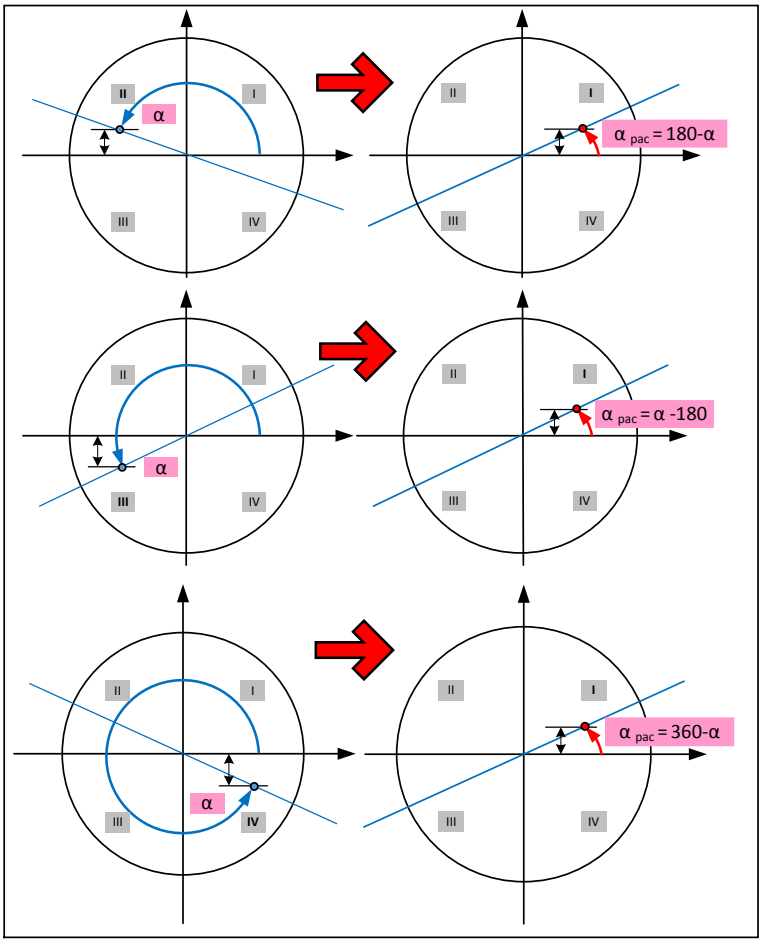


Рис. 22.9 Расчет функции sin(x) на полном периоде по значениям в первом квадранте

Алгоритма расчета функции sin(x), соответствующий рис. 22.9, представлен ниже (см. табл. 22.2).

Таблица 22.2 Алгоритма расчета функции sin(x)

Квадрант	Условие	Расчетное значение $\alpha_{рас}$	Значение функции
I	$\alpha \le 90$	$\alpha_{рас} = \alpha$	$\sin(\alpha_{рас})$
II	$\alpha \le 180$	$\alpha_{рас} = 180 - \alpha$	$\sin(\alpha_{рас})$
III	$\alpha \le 270$	$\alpha_{рас} = \alpha - 180$	$-\sin(\alpha_{рас})$
IV	$\alpha < 360$	$\alpha_{рас} = 360 - \alpha$	$-\sin(\alpha_{рас})$

```
10  ; Считать значение аргумента из памяти данных
11  ; в регистр сопроцессора s10
12        LDR r0,=X_grad
13        VLDM r0,{s10}
14  ; Вызвать подпрограмму расчета функции Sin(x)
15        BL Calc_Sin
16  ; Сохранить результат расчета в памяти
17        LDR r0,=Sin_X
18        VSTM r0,{s15}
19  ; Повторить расчет для нового значения аргумента
20        B MyProg
```

Обратимся к проекту FPU_12. В основной программе имеются две подпрограммы: верхнего уровня Calc_Sin, работающая с любым значением аргумента в формате плавающей точки x(F32)[град.], и нижнего уровня Calc_Sin_90, работающая с аргументом только в диапазоне от 0 до 90 град (см. предыдущий раздел 22.5.2). При этом подпрограмма верхнего уровня сначала получает расчетное значение аргумента для подпрограммы нижнего уровня, а затем – вызывает ее.

```
231       AREA MyData, DATA, ReadWrite
232  ; Аргумент
233  X_grad SPACE  4
234  ; Значение функции Sin(x)
235  Sin_X  SPACE  4
```

Для удобства отладки в конце основной программы в ОЗУ зарезервировано место под входную переменную и результат расчета.

Открыв окно дампа памяти, Вы можете вводить любое значение аргумента в формате F32, запускать программу на выполнение до точки останова и оценивать результат ее работы. В представленном фрагменте программы аргумент передается в подпрограмму верхнего уровня (содержимым регистра s10), и возвращенный из нее результат расчета (в регистре s15) сохраняется в памяти.

```
24  ;*************************************************************
25  ; Процедура расчета фукнции Sin(x) для любого значения
26  ; аргумента x(F32)[Град.]
27  ; Вход: s10 - значение аргумента
28  ; Выход: s15 - значение функции Sin(x)
29  ; Используемые регистры сопроцессора: s11-s14
30  ;*************************************************************
31  Calc_Sin
32  ; Так как из подпрограммы будут следовать вызовы подпрограммы
33  ; более низкого уровня - сохранить адрес в регистре связи в стеке
34        PUSH {lr}
35  ; Рассчитать номер текущего периода n_per(F32)
36        VLDR.F32 s11,=360.0   ; s11 <- 360.0
37        VDIV.F32 s12,s10,s11  ; s12 <- n_per(F32)
38  ; Преобразовать номер периода в целое число
39        VCVT.U32.F32 s13,s12  ; n_per(F32)-> n_per(U32)
40  ; Рассчитать относительное значение аргумента на текущем периоде
41        VCVT.F32.U32 s13,s13
42        VSUB.F32 s12,s13      ; s12 <- dx(F32)
43  ; Относительное значение аргумента в градусах
44        VMUL.F32 s12,s11          ; s12 <- x(F32)[Град]
```

Начальная часть подпрограммы верхнего уровня Calc_Sin особая. Из подпрограммы будет вызываться подпрограмма низкого уровня Calc_Sin_90. Следовательно, при сохранении контекста (если это необходимо) нужно обязательно сохранить в стеке содержимое регистра связи LR. При этом (подробно см. 14.6) возврат из подпрограммы верхнего уровня будет выполняться уже не командой BX lr, а командой прямого восстановления содержимого счетчика команд PC непосредственно из стека POP {pc}. Вызов подпрограммы более низкого уровня и возврат из нее при этом выполняются традиционным способом.

Итак, основная программа использует регистры сопроцессора только для передачи параметров в подпрограмму и приема из нее результата вычисления. Поэтому, в стеке сохраняется лишь содержимое регистра связи lr.

Остальные вычисления выполняются в соответствии с описанным выше алгоритмом получения номера текущего периода и расчета по нему сначала относительного значения аргумента на текущем периоде, а затем его значения в градусах x(F32)[град] в диапазоне от 0 до 360.

Следующая часть подпрограммы предназначена для определения квадранта функции sin(x), в котором располагается аргумент, и определения расчетного значения

$\alpha_{рас}$ в соответствии с рис. 22.9, которое будет передаваться в подпрограмму низкого уровня Calc_Sin_90 содержимым регистра s0.

```
45  ; Аргумент находится в первом квадранте?
46        VLDR.F32 s14,=90.0
47        VCMP.F32 s12,s14
48        VMRS APSR_nzcv, FPSCR
49  ; Если да, выполнить
50        VMOVLT.F32 s0,s12
51        BLLT Calc_Sin_90
52        VMOVLT.F32 s15,s5
53        POPLT {pc}
54  ; Нет. Аргумент находится во втором квадранте?
55        VLDR.F32 s14,=180.0
56        VCMP.F32 s12,s14
57        VMRS APSR_nzcv, FPSCR
58  ; Если да, выполнить
59        VSUBLT.F32 s0,s14,s12   ; s0 <- (180.0-alfa)
60        BLLT Calc_Sin_90
61        VMOVLT.F32 s15,s5
62        POPLT {pc}
63  ; Нет. Аргумент находится в третьем квадране?
64        VLDR.F32 s14,=270.0
65        VCMP.F32 s12,s14
66        VMRS APSR_nzcv, FPSCR
67  ; Если да, выполнить
68        VSUBLT.F32 s0,s14,s12
69        BLLT Calc_Sin_90
70        VNEGLT.F32 s15,s5
71        POPLT {pc}
72  ; Нет. Аргумент находится в четвертом квадранте
73        VSUB.F32 s0,s11,s12
74        BL Calc_Sin_90
75        VNEG.F32 s15,s5
76        POP {pc}
77  ; Конец процедуры
```

Эта часть подпрограммы иллюстрирует не только операции сравнения чисел с плавающей точкой, которые необходимы для определения текущего квадранта функции, но и показывает, как можно эффективно использовать результаты этого сравнения для создания удобного, хорошо читаемого кода с применением оригинальной технологии процессоров ARM – условного выполнения любых команд, в том числе, команд с плавающей точкой и, даже, вызова подпрограмм и возврата из них.

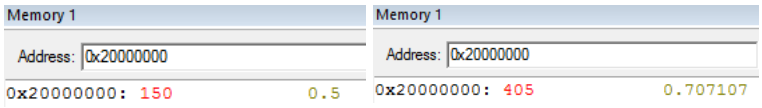
После любой операции сравнения содержимое регистра статуса и управления сопроцессора FPSCR считывается в регистр статуса приложения пользователя APSR, в результате чего активизируются коды условного выполнения. Для каждого квадранта эффективным является код LT (строго меньше). Блок условного выполнения в каждом случае состоит из 4-х команд: вычисления и передачи в подпрограмму расчетного значения аргумента (от 0 до 90 град.); вызова подпрограммы нижнего уровня; инверсии знака результата при необходимости (для квадрантов III и IV); возврата в основную программу.

Подпрограмма нижнего уровня рассмотрена ранее и не комментируется.

Использование технологии условного выполнения команд в процессорах ARM позволяет создать полностью линейный код, удобный для разработки, отладки и последующего сопровождения программы. Он не содержит ветвлений, сокращает объем программы и, как правило, повышает ее быстродействие.

1) Создайте исполнительный файл проекта и запустите его на отладку. В окне дампа памяти, начиная с адреса 0x20000000 (адрес X_grad), вводите разные исходные данные в формате float, получайте и анализируйте результаты вычисления функции sin_X.
2) Объясните, что такое блок условного выполнения команд? Проследите в окне дизассемблера за тем, какие дополнительные команды генерирует транслятор в начале каждого блока.
3) В чем заключается технология работы с вложенными подпрограммами в процессорах ARM?
4) Почему в качестве команды возврата из подпрограммы верхнего уровня используется команда POP {pc}?
5) Как будут выполняться команды блока условного выполнения при истинности альтернативного условия по отношению к условию LT (то есть при истинности GE)?

1) Результаты двух тестовых прогонов программы:



2) Блок условного выполнения начинается со специальной команды IT (Если То), которая автоматически генерируется транслятором для нескольких (от одной до 4-х) команд с одинаковыми или оппозитными кодами условного выполнения.
3) При вызове вложенной подпрограммы (низкого уровня) адрес возврата, который был записан в регистр связи LR при вызове подпрограммы верхнего уровня из основной программы, будет «перезаписан» новым адресом возврата уже в подпрограмму верхнего уровня. Чтобы не потерять этот адрес, его нужно предварительно сохранить в обычном стеке.
4) В конце подпрограммы верхнего уровня адрес возврата в основную программу восстанавливается из стека сразу в счетчик команд.
5) Они будут выполняться как команды NOP (пустые операции).

22.6 Генератор синусоидального сигнала

Генераторы периодических сигналов широко применяются для тестирования качества цифровых систем управления, в том числе, для экспериментального определения полосы пропускания и амплитудно-частотных характеристик системы. Генераторы обычно разрабатываются с использованием встроенных таймеров и системы прерывания процессора. С их помощью задается временной интервал, на котором выполняется дискретное приращение фазы периодического сигнала, рассчитывается и выдается очередное значение сигнала.

Разработаем программу, *имитирующую работу* генератора синусоидального сигнала (система прерываний и таймер не используются). Предположим, что задано начальное значение фазы сигнала, то есть аргумент функции sin(x) и приращение фазы (аргумента функции), а также число вызовов основного программного модуля, в котором выполняется приращение фазы и рассчитывается новое значение выходного сигнала. Возьмем за основу предыдущий проект FPU_12, содержащий подпрограмму расчета функции sin(x) для любого значения аргумента, и модернизируем только основную программу (см. MyProg_1.s), не меняя ни обеих подпрограмм, ни таблицы опорных значений функции sin(x).

```
 8  ; Генератор периодического синусоидального сигнала
 9  ; Считать из памяти начальное значение аргумента X_grad
10  ; в регистр сопроцессора s10 с авто-инкрементированием
11  ; указателя
12        LDR r0,=X_Grad
13        VLDMIA r0!,{s10}
14  ; Считать в регистр s16 приращение аргумента dx_Grad
15  ; на очередном такте работы генератора
16        VLDM r0,{s16}
17  ; Задать число вызовов генератора синусоидального сигнала
18        MOV r5,#1000
```

```
257             AREA MyData, DATA, ReadWrite
258  ; Начальное значение аргумента
259  X_Grad     SPACE 4
260  ; Приращение аргумента dx_Grad
261  dx_Grad    SPACE 4
262  ; Значение функции Sin(x) для текущего аргумента
263  Sin_X      SPACE 4
264  ; Значение целочисленой переменной 1000+1000*sin(x)
265  Sin_U32    SPACE 4
266  ; Объявить глобальной переменной для наблюдения
267  ; в логическом анализаторе
268             GLOBAL Sin_U32
```

Начальное значение аргумента в формате F32 считывается из памяти по адресу X_Grad, а приращение аргумента на очередном такте работы генератора - из ячейки памяти по следующему адресу dx_Grad.

В ОЗУ, кроме этих переменных, резервируются еще две ячейки памяти – одна для вывода текущего выходного сигнала в формате F32, а вторая – для целочисленной только положительной переменной Sin_U32, связанной непосредственно с выходом генератора,

которая может быть обычным способом графически отображена в окне логического анализатора среды µVision.

```
19  ; Сгенерировать нужное число точек синусоиды
20  Gen_Sin
21  ; Вызвать подпрограмму расчета функции Sin(x) для
22  ; очередного значения аргумента в регистре s10
23        BL Calc_Sin
24  ; Сохранить результат расчета из регистра s15 в памяти
25        LDR r0,=Sin_X
26        VSTM r0,{s15}
27  ; Вычислить выражение 1000 + 1000*Sin_X
28        VLDR.F32 s17,=1000.0
29        VFMA.F32 s17, s17,s15
30  ; Преобразовать в целое без знака для наблюдения
31  ; в логическом анализаторе
32        VCVT.U32.F32 s17,s17
33  ; Вывести это значение в порт вывода (в память)
34        LDR r0,=Sin_U32
35        VSTM r0,{s17}
36  ; Увеличить значение аргумента на величину
37  ; заданного приращения
38        VADD.F32 s10,s16
39  ; Декрементировать счетчик числа сгенерированных точек
40  ; синусоиды. Если не все точки сгенерированы, повторить
41        SUBS r5,#1
42        BNE Gen_Sin
43  ; Повторить расчет для нового значения аргумента
44        B MyProg
```

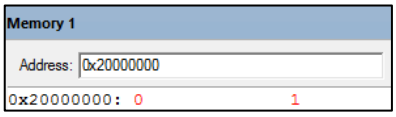
Эта переменная объявлена в качестве глобальной, так как только такие переменные могут быть наблюдаемыми. Она рассчитывается в соответствии с выражением: 1000 + 1000*sin(x), которое далее преобразуется в целое число без знака Sin_U32.

Такой подход позволяет нам просматривать в логическом анализаторе графики *разнополярных сигналов*, преобразуя их предварительно в *однополярные*. Как мы уже отмечали, к сожалению, для отображения в среде µVision знакопеременных сигналов в формате с фиксированной точкой и сигналов в формате с плавающей точкой необходимы дополнительные меры по преобразованию типов переменных, которые требуют знания языка высокого уровня C/C++. Мы покажем в следующем параграфе, как решается эта проблема.

Пока же поступим так: выполним двойное преобразование: из разнополярного сигнала в формате F32 – в однополярный в формате F32; из однополярного сигнала в формате F32 – в однополярный в формате U32. Коэффициент 1000 выбран условно. Вы можете использовать любое достаточно большое число без знака, которое позволит получить приемлемое разрешение в окне логического анализатора.

Основной блок генератора вычисляет значение функции синуса для текущего значения аргумента, преобразует его для вывода на экран анализатора и добавляет приращение к текущему значению аргумента. Основной блок повторяется в цикле заданное число раз.

В строке 44 можно поставить точку останова. Это позволит сгенерировать нужное число точек синусоиды при заданных параметрах (начальном значении аргумента и величине его приращения) и остановиться. Вводите новые исходные данные в окне памяти и запускайте программу на выполнение вновь. Пример графического вывода синусоидального сигнала в окно логического анализатора для начальной фазы 0 град. и приращения фазы 1 град. представлен ниже (см. рис. 22.10). Он убедительно иллюстрирует правильность работы двух описанных ранее подпрограмм Calc_Sin и Calc_Sin_90.



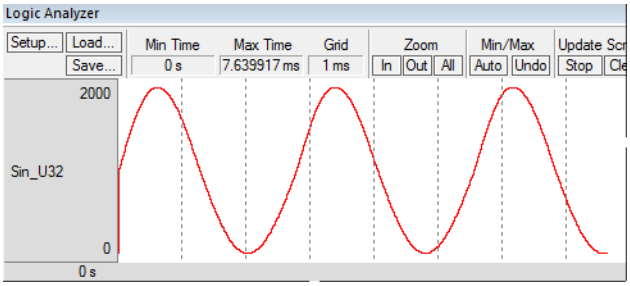


Рис. 22.10 Вывод синусоидального сигнала в окно логического анализатора

22.7 Дополнительные возможности логического анализатора среды µVision

При отладке сложных проектов весьма желательно графическое отображение всех переменных в динамике. По умолчанию логический анализатор поддерживает вывод на экран только целочисленных переменных, да и то беззнаковых, – в формате U32. Вместе с тем, при отладке программ, написанных на языке высокого уровня C/C++, имеется возможность наблюдения в окне логического анализатора и целочисленных переменных со знаком и переменных в формате с плавающей точкой.

Почему эти возможности недоступны в Ассемблере? Дело в том, что, объявляя любую переменную в языке C/C++, Вы обязательно должны указать ее тип (signed int, unsigned int, float и т.д.). Следовательно, компилятор связывает с любой переменной не только адрес ее фактического расположения в памяти системы, но и тип хранящихся в ней данных. Поэтому, при задании имени переменной в качестве наблюдаемой переменной, логический анализатор автоматически идентифицирует ее тип, чтобы в процессе трассировки данных правильно интерпретировать полученную битовую последовательность (значение переменной) и корректно вывести полученное значение на график. В языке Ассемблер только программист знает, что хранится в зарезервированной им ячейке памяти: это может быть целое число в формате U32 или S32, число в формате с плавающей точкой F32 или просто последовательность из 4-х байт, 2-х полуслов или 32-х бит.

Следовательно, мы сможем наблюдать любые переменные, в любом формате, если «сообщим» логическому анализатору *тип наблюдаемой переменной*. Для этого придется использовать более сложную технологию ввода имени переменной в окно Setup Logic Analyzer (инициализации логического анализатора). Эта технология основана на знании особенностей языка высокого уровня C/C++, поэтому пока примите описанный ниже порядок действий просто как шаблон (даны только минимальные комментарии):

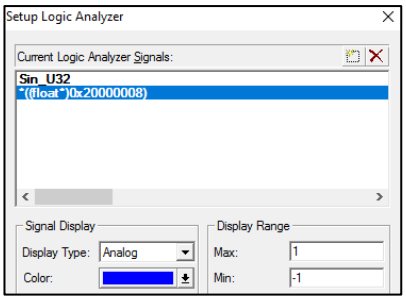
1) В программе на Ассемблере обязательно объявите имена нужных наблюдаемых переменных в качестве глобальных. Только они могут отображаться на графиках в логическом анализаторе.
2) Просмотрите файл с картой памяти проекта «.map» и запишите фактические адреса размещения наблюдаемых переменных в памяти системы.
3) Мы должны сообщить логическому анализатору тип наблюдаемой переменной. Рассмотрим, как это сделать на примере переменной Sin_X проекта FPU_12 – это переменная в формате с плавающей точкой F32 (тип float на языке C/C++), расположенная в памяти по адресу 0x20000008.
    a. Преобразуем адрес наблюдаемой переменной в *адрес переменной типа float*. Для этого используем выражение:

        (float*)&Sin_X или (float*)0x20000008.

        Оператор «&» используется в языке C/C++ для определения адреса переменной по ее символическому имени. Записи &Sin_X и 0x20000008 полностью эквивалентны. Префикс (float*) можно рассматривать как оператор преобразования формата данных по указанному адресу в формат числа с плавающей точкой float.
    b. В окно задания наблюдаемой переменной логического анализатора нужно ввести не имя переменной, а указатель на нее *(адрес переменной, преобразованной к нужному типу). В нашем случае возможны два варианта:

        ((float)&Sin_X) или *((float*)0x20000008)

4) Итак, укажем в качестве наблюдаемой переменной, переменную, расположенную по адресу 0x20000008, считая ее переменной типа float. Зададим максимальное (+1) и минимальное (-1) значения этой переменной. Тип сигнала сохраним – аналоговый. Дополнительно выберем цвет вывода сигнала в окно логического анализатора.



Все остальные операции по настройке логического анализатора мы уже выполняли. Зададим начальные значение параметров (аргумента и его приращения). Запустим процесс выполнения программы до точки останова.

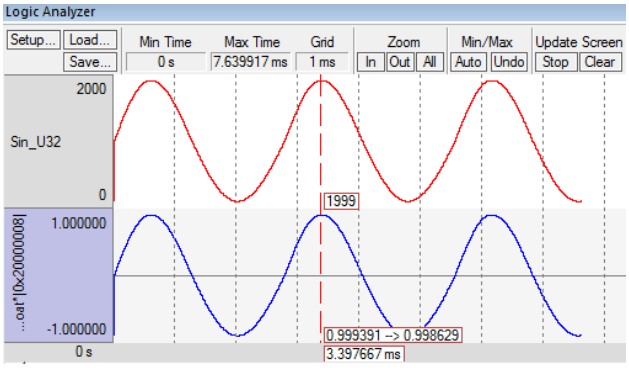


Рис. 22.11 Вывод синусоидального сигнала в окно логического анализатора

Вы видите (рис. 22.11), что на экране появилась «нормальная» синусоида, причем с дополнительной нулевой линией, что очень удобно при анализе разнополярных сигналов. Все остальные возможности логического анализатора, в частности, оцифровка значений на графиках, тоже работают.

Если переменная, которая должна быть представлена графически, является целым числом со знаком в дополнительном коде, то необходимо в качестве преобразователя формата указать (signed int *). Предположим, что в программе имеется переменная Sin_S32, значение которой получено после преобразования числа с плавающей точкой 1000*sin(x) в целое со знаком в формате S32. Для включения этой переменной в список наблюдаемых переменных введите: *((signed int*)&Sin_S32).

Возможности интегрированной среды µVision по графическому анализу динамических процессов в системах, работающих с переменными в любых форматах, в том числе, в формате с плавающей точкой, могут быть существенно расширены. Все, что нужно для этого – изменить технологию включения переменных типа signed int и float для их наблюдения в логическом анализаторе.

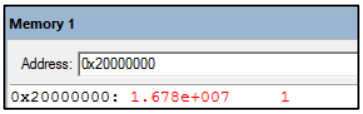
1) Исследуйте генератор периодических сигналов (MyPorg_1.s) при различных значениях входных параметров с использованием логического анализатора.

2) Добавьте в программу переменную Sin_S32. Убедитесь, что в окне логического анализатора можно графически представлять переменные в формате чисел со знаком в дополнительном коде.

2) Возможный вариант решения Вы найдете в файле MyProg_2.s в том же проекте. Подключите его и проверьте технологию задания в качестве наблюдаемой переменной целого числа со знаком S32.

22.8 «Подводные камни» при вычислениях в формате чисел с плавающей точкой

Надеемся, что Вы протестировали программу генератора периодических сигналов, представленную выше, на различных наборах исходных данных – начального значения аргумента и приращения аргумента. Попробуйте, что произойдет, если задать очень большое начальное значение аргумента и относительно маленькое значение его приращения?



Представленный ниже график (см. рис. 22.12) свидетельствует о том, что программа генерации синусоидального сигнала *не работает*. Вместе с тем, она работала абсолютно правильно для множества других, ранее введенных исходных данных. Более того, в режиме пошаговой отладки программы Вы можете убедиться в том, что первое значение функции синуса определено правильно. Далее оно почему-то не меняется вообще. Чем же этот набор переменных отличается от предыдущих?

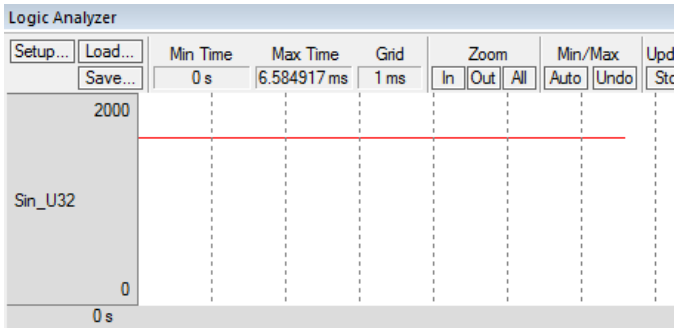
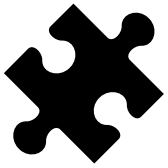


Рис. 22.12 Вывод сигнала в окно логического анализатора

Оказывается, мы задали такое большое начальное значение аргумента, что добавленная к нему единица дает число, которое не может быть представлено в формате с плавающей точкой однократной точности – не относится к числу репрезентабельных. Оно округляется при выполнении операции сложения чисел с плавающей точкой до исходного числа, то есть – не меняется. Это следствие неравномерности шкалы чисел в формате с плавающей точкой однократной точности: чем больше значение числа, тем больше дискретность его представления. Может вполне случиться так, что к очень большому числу добавляется очень маленькое, и результат остается тем же самым!

Программист должен хорошо представлять, как кодируются числа в формате с плавающей точкой, чтобы исключить подобные ситуации: либо правильным выбором диапазонов изменения переменных и коэффициентов, либо выбором других форматов переменных (например, целочисленных). Кроме того, разработанная программа должна обязательно тестироваться во всех возможных диапазонах изменения переменных и коэффициентов.

22.8.1 Выполняются ли математические законы в арифметике с плавающей точкой?



Существует несколько общеизвестных законов математики, истинность которых не подвергается сомнению. Как обстоит дело с этими законами, если операции выполнятся над числами в формате с плавающей точкой?

22.8.1.1 Коммутативный закон

Коммутативный закон гласит, что от перемены мест слагаемых или множителей, сумма или произведение не меняются. В арифметике с плавающей точкой этот закон выполняется только *применительно к двум операндам*. Так, следующие пары команд всегда будут генерировать один и тот же результат, независимо от значений исходных операндов:

```
VADD.F32 s5, s3, s4
VADD.F32 s6, s4, s3

VMUL.F32 s7, s3, s4
VMUL.F32 s8, s4, s3
```

Ситуация может измениться, если слагаемых или множителей больше двух. Все дело в округлении результата после каждой операции сложения или умножения. Поэтому, окончательный результат может незначительно отличаться (в младших битах мантиссы) при разной последовательности операций.

22.8.1.2 Ассоциативный закон

Ассоциативный закон утверждает, что при сложении или умножении любого числа операндов результат не меняется, если операнды объединяются в любые группы для вычисления промежуточных результатов, например, так:

$$(A + B) + C = A + (B + C).$$

И в этом случае, *за счет округления*, результаты расчетов могут чуть-чуть отличаться. Еще серьезнее обстоит дело с умножениями. При одной последовательности операций, в результате умножения больших чисел, *возможен выход за диапазон нормальных чисел с получением в результате умножения бесконечности*, а при другой последовательности этого можно избежать. Следовательно, при множественных умножениях и умножениях с накоплением нужно обязательно контролировать «Переполнения вверх», и, если они возникают, не удивляться, а пробовать изменить порядок операндов или диапазоны изменения переменных (что правильнее и надежнее).

22.8.1.3 Дистрибутивный закон

Дистрибутивный или распределительный закон предполагает выполнение соотношения:

$$A * (B + C) = (A * B) + (A * C).$$

Этот закон также может нарушаться из-за дискретности представления чисел в формате с плавающей точкой и необходимости округления результата до ближайшего репрезентабельного числа.

22.8.2 Основные рекомендации по организации вычислений с плавающей точкой

1. Возможные нарушения математических законов связаны исключительно с выходом за допустимый диапазон чисел с плавающей точкой, либо с округлениями. Как правило, они возникают при работе с очень большими по значению числами, дискретность представления которых велика (в конце диапазона нормальных чисел). Если Вы обнаружите такую ситуацию, обдумайте возможность реализации алгоритма (или его части) с использованием арифметики с фиксированной точкой.
2. Сопроцессор выполняет автоматическую замену очень больших чисел на бесконечность или на максимально возможное нормальное число, которая аналогична операции «насыщения». Работа с числами типа «бесконечность» допускается, а иногда и специально рекомендуется, при создании так называемых робастных систем управления (нечувствительных к широкому диапазону изменения переменных и коэффициентов). Это исключает резкие смены управляющих воздействий и связанную с этим потерю устойчивости систем управления.
3. Для постоянного контроля нестандартных ситуаций в процессоре реализован механизм исключений. После выполнения сомнительной математической операции обязательно проверяйте флаги исключений, и при их возникновении корректируйте алгоритм или диапазоны используемых переменных и коэффициентов.
4. Самым «критичным» является исключение IOC - «Недопустимая операция». Оно возникает тогда, когда результат арифметической операции не может быть определен сопроцессором. Выбором правильных алгоритмов и диапазонов переменных избегайте ситуаций, приведенных ниже (см. табл. 22.3).

Таблица 22.3 Условие возникновения «Недопустимой операции»

Команда сопроцессора	**Условие возникновения «Недопустимой операции»**
VADD	**(+∞) + (-∞); (-∞) + (+∞)**
VSUB	**(+∞) - (+∞); (-∞) - (+∞)**
VCMPE	**Один из операндов «Не Число» (NaN)**
VMUL, VNMUL	**±0 × (±∞);**
VDIV	**±0/±0; ±∞/±∞**
VMLA, VNMLA	**[±0 × (±∞)] – при умножении или [(+∞) + (-∞)] при сложении**
VMLS, VNMLS	**[±0 × (±∞)] – при умножении или [(+∞) - (+∞); (-∞) - (+∞)] при вычитании**
VSQRT	**Операнд отрицательный**
VCVT.S32.F32	**Операнд «Не Число» (NaN), ±∞ или превышает максимальное значение целого числа**

5. Если один из операндов арифметической команды или операнд команды преобразования формата является «Не числом», то такая операция считается недопустимой – формируется флаг IOC и возвращается результат в виде так называемого «Не числа по умолчанию» – Default NaN. В процессорах Cortex-M4F в качестве «Не числа по умолчанию» используется, так называемое, «Тихое Не Число»

(Quiet NaN – qNaN). Выполняйте арифметические операции только с нормальными и субнормальными числами в формате с плавающей точкой.

6. В процессорах Cortex-M4F гарантируется корректная работа и с очень маленькими числами – субнормальными. Практически, числовая шкала от -∞ до +∞ становится непрерывной. Однако, не следует забывать, что операции сложения/вычитания чисел в околонулевом диапазоне (субнормальных чисел) и в конце диапазона (больших нормальных чисел) из-за округления могут дать неточный результат. При получении неточного результата процессор всегда выставляет флаг IXC (Неточность), который Вы сможете проконтролировать.
7. Помните, что диапазон чисел с плавающей точкой однократной точности обеспечивает получение результата вычислений с точностью 5 – 6 знаков после десятичной точки, что вполне достаточно для большинства инженерных расчетов. Не используйте, по возможности, диапазон нормальных чисел, близких к бесконечностям, чтобы избежать нарушения основных математических законов.

Вы дочитали книгу до самого конца. Поздравляем! Надеемся, что процессорные ядра на базе ARM Cortex-M3/M4/M4F по своим потенциальным возможностям не обманули Ваших ожиданий!

Успехов в практическом решении ваших задач!

Список рекомендуемой литературы

1) Джозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство/ Ю. Джозеф; пер. с англ. А.В. Евстафеева. – М.: Додэка-XXI, 2012. – 552 с. ил. (Мировая электроника).
2) Айфичер Э., Джервис Б. Цифровая обработка сигналов: практический подход, 2-е издание.: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. – 992с. ил.
3) Texas Instruments. SPMU 159a. Cortex-M3/M4F Instruction Set. Technical User’s Manual. Texas Instruments Inc. – 2011.
4) Mahout V. Assembly language programming: ARM Cortex-M3. – Wiley-ISTE. – 2012.
5) Fisher M. ARM Cortex M4 Cookbook. – Packt Publishing Ltd. – 2016.
6) Hohl W., Hinds C. ARM Assembly language. Fundamentals and Techniques. Second Edition. – N.-Y.: CRC Press.– 2015. – 453 p.

ПРИЛОЖЕНИЕ 1. СПРАВОЧНИК ПО СИСТЕМЕ КОМАНД ПРОЦЕССОРНЫХ ЯДЕР CORTEX-M3/M4/M4F

Настоящее приложение (файл «Приложение_1_Справ_по_сист_ком») содержит подробный справочник по системе команд с указанием наличия команды в соответствующем процессорном ядре. Команды объединены в разделы по их основному функциональному назначению. Приводятся сведения по синтаксису команд и выполняемым ими действиям, по выработке флагов результатов операций, ограничениям в использовании регистров процессора. Даются примеры корректной записи команд на языке Ассемблер.

Алгоритмы выполнения некоторых, достаточно сложных команд, иллюстрируются графически.

Стиль оформления справочника унифицирован для всех команд и по возможности соответствует исходному описанию системы команд фирмы ARM и ведущих мировых производителей микроконтроллерной техники с ядрами ARM (в частности, Texas Instruments). Некоторые важные комментарии и замечания рассчитаны на повышение ясности излагаемого материала.

Все команды процессора разделены на 10 групп, которые полностью соответствуют последовательности и методике изучения системы команд, используемой в книге:

1. Команды пересылки данных
2. Арифметические команды
3. Логические команды
4. Команды умножения, умножения с накоплением, деления
5. Команды насыщения
6. Команды упаковки и распаковки данных
7. Команды работы с битовыми полями
8. Команды сопроцессора (обработки чисел в формате с плавающей точкой)
9. Команды передачи управления
10. Команды специального назначения

Внутри каждого из разделов имеется внутреннее оглавление, которое позволит читателю быстро найти описание нужной ему команды в соответствии с ее основным назначением и преимущественной областью применения.

Ввиду ограниченного объема книги некоторые команды только упомянуты в ней, но для полноты информации включены в справочник, что позволяет, при необходимости, самостоятельно изучить специфику их применения.

!!! Данное приложение распространяется бесплатно и доступно для свободного скачивания по ссылке.

ПРИЛОЖЕНИЕ 2. ПРОЕКТЫ

Настоящее приложение (каталог «Приложение_2_Проекты») содержит электронную версию всех проектов, включенных в книгу, которые читатель может без каких-либо ограничений загрузить в интегрированную среду разработки Keil µVision, установленную на своем компьютере, и выполнить.

Проекты можно использовать как при самостоятельной работе над материалом книги, так и при выполнении лабораторных и практических работ под руководством преподавателя соответствующего учебного курса или курса повышения квалификации.

Допускается и приветствуется любая модификация исходных файлов проектов в целях более детального изучения особенностей и возможностей системы команд процессорных ядер Cortex-M3/M4/M4F.

Проекты, иллюстрирующие систему команд центрального процессора, расположены в каталоге CPU, а проекты, иллюстрирующие систему команд сопроцессора для обработки чисел в формате с плавающей точкой однократной точности – в каталоге FPU. В каждом из проектов имеется стартовый файл, а также один или несколько исходных файлов пользователя на языке Ассемблер. Основная рабочая программа находится в файле MyProgN.s, где N – номер проекта. Вместо нее по мере изучения материала книги к текущему проекту могут подключаться модифицированные исходные файлы типа MyProgN_1.s, MyProgN_2.s и т.д., в том числе файлы с примерами возможных решений индивидуальных заданий. Для работы с модифицированными файлами отключите основной исходный файл из проекта и подключите нужный дополнительный файл.

Все проекты описаны в книге, за исключением проекта CPU12, который иллюстрирует работу с подпрограммами, включенными в пользовательские библиотеки. Он предназначен для самостоятельного изучения.

!!! Данное приложение распространяется бесплатно и доступно для свободного скачивания по ссылке.

ПРИЛОЖЕНИЕ 3. ОТЛАДОЧНЫЙ КОМПЛЕКТ VECTORCARD

Отладочный комплект на базе платы VectorCARD K1921BK01T является средством для начальной разработки программного обеспечения и оценки возможностей нового отечественного микроконтроллера **K1921BK01T производства АО «НИИЭТ»**.

Изделие изготовлено в виде 100-контактной втычной платы для установки в разъем стандарта DIMM. На плате реализована полная «обвязка» микроконтроллера (узлы тактирования и питания), поставлены защиты на аналоговые входы, светодиодная индикация и кнопки, имеется стандартный разъем JTAG для программирования и отладки, а также разъем, на который выведены все оставшиеся свободными выводы микроконтроллера. Плата содержит гальванически развязанный интерфейс USB с преобразователем интерфейсов UART-USB. Для работы с платой подходит любой стандартный JTAG-программатор для ядра ARM Cortex M4F (например, J-Link, ST-Link или аналоги).

Отличительной особенностью предлагаемой отладочной платы является совместимость с базовыми платами отладочных комплектов фирмы Texas Intruments: TMS320F28335 Experimenter Kit, DRV8301-HC-EVM, DRV8312-C2-KIT, High Voltage Motor Control Kit и другими.

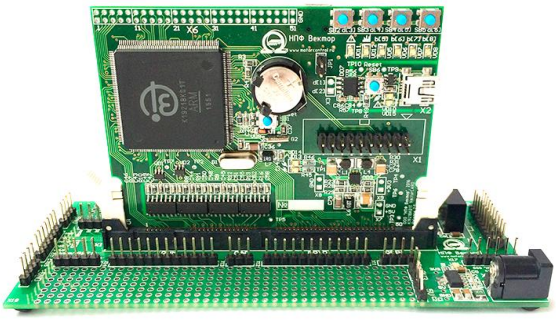
На основе комплекта Texas Intruments DRV8301-HC-EVM предлагается решение **для управления электродвигателями**. Базовая плата этого комплекта содержит шестиключевой инвертор с датчиками токов, интерфейсы с датчиками положения, CAN интерфейс. Вместе с платой VectorCARD K1921BK01T образуется полноценный стенд для создания и исследования систем управления электродвигателями на K1921BK01T.

ООО «НПФ Вектор» поставляет готовый комплект, включающий всё необходимое, чтобы **векторное управление** синхронной машиной с постоянными магнитами заработало на K1921BK01T «из коробки»: плата VectorCARD K1921BK01T, базовая плата с инвертором, USB-JTAG программатор, синхронный двигатель с постоянными магнитами и встроенными датчиками положения (инкрементальный и на эффекте Холла), среда разработки для микроконтроллера на основе Eclipse и свободных компиляторов GCC, программное обеспечение в исходных кодах на Си для K1921BK01T (обеспечивающее векторное датчиковое управление, позволяя управлять моментом, скоростью или положением), программное обеспечение персонального компьютера для настройки и наблюдения осциллограмм, документация, руководство пользователя, техническая поддержка по запуску комплекта в работу.

Также доступны другие виды отладочных комплектов на основе VectorCARD K1921BK01T для различных задач. Дополнительную информацию можно узнать на сайте ООО НПФ «Вектор» (в разделе Продукция-Отладочные платы).

Отладочный комплект VectorCARD K1921BK01T

на отечественном микроконтроллере семейства motorcontrol с ядром ARM Cortex M4F от ОАО «НИИЭТ»



Технические характеристики

- Совместимость с отладочными комплектами фирмы Texas Instruments
- Доступность на разъемах практически **всех** выходов микроконтроллера
- Гальванически развязанный интерфейс UART-USB

Комплектация для управления электродвигателем (motorcontrol)

- Инвертор на 60В U_{DC} с шунтовыми датчиками токов
- Гальванически развязанные интерфейсы связи CAN, RS-232 и SPI
- Синхронная машина с постоянными магнитами на роторе
- Датчики положения (инкрементальный и на эффекте Холла)
- USB-JTAG
- Среда разработки на базе Eclipse
- ПО для K1921BK01T в исходных кодах, реализующее векторное управление
- ПО компьютера для параметрирования, управления приводом и осциллографирования процессов

http://motorcontrol.ru
info@motorcontrol.ru
+7 (495) 303-37-54
Москва, ул. Фрязевская, д.4 с.2

НПФ Вектор
Электропривод, автоматика и энергосбережение

Учебное издание

Козаченко Владимир Филиппович
Анучин Алексей Сергеевич,
Алямкин Дмитрий Иванович
Жарков Александр Александрович
Лашкевич Максим Михайлович
Савкин Дмитрий Игоревич
Шпак Дмитрий Михайлович

ПРАКТИЧЕСКИЙ КУРС МИКРОПРОЦЕССОРНОЙ ТЕХНИКИ НА БАЗЕ ПРОЦЕССОРНЫХ ЯДЕР ARM-CORTEX-M3/M4/M4F

Учебное пособие
Текстовое электронное издание

Изд. № 19эи-023 Тираж 20 экз.