Под общей редакцией А. Ю. Романова, Ю. В. Панчула

Цифровой синтез: практический курс

ДМК Пресс
издательство

Москва, 2020

УДК 004.01
ББК 32.972
Р69

Коллектив авторов
Антонов А. А., Барабанов А. В., Данчек Ч. Т., Жельнио С. Л.,
Иванец С. А., Кудрявцев И. А., Панчул Ю. В., Романов А. Ю.,
Романова И. И., Телятников А. А., Шуплецов М. С.

Р69 Цифровой синтез: практический курс / под общ. ред. А. Ю. Романова, Ю. В. Панчула. – М.: ДМК Пресс, 2020. – 556 с.

ISBN 978-5-97060-850-0

Книга представляет собой расширенный практический курс, ориентированный на язык Verilog и обеспечивающий возможность выполнения практических задач на дешевых отладочных платах. Этот практикум дополняет и объединяет теоретические курсы по цифровой логике, языкам описания аппаратуры, компьютерной архитектуре и микроархитектуре, а также подготавливает студентов к работе с промышленными процессорными ядрами, к созданию специализированных вычислителей (например, ускорителей нейросетей) и курсов VLSI по проектированию массовых микросхем ASIC.

Материал каждой главы можно изучать автономно. В конце глав приводятся вопросы и упражнения, позволяющие преподавателям встраивать данный материал в любой учебный курс, а читателям книги – закрепить новые знания, самостоятельно выполнив предлагаемые задания.

Издание предназначено для студентов технических вузов, разработчиков аппаратно-программных систем, а также специалистов в области прикладной математики, интересующихся алгоритмами САПР.

УДК 004.01
ББК 32.972

ISBN 978-5-97060-850-0

Оглавление

Цифровой синтез: практический курс

Введение

Verilog – не просто один из редких языков, а обязательный инструмент современного разработчика электроники

До сих пор встречаются люди, которые считают, что **Verilog** – просто один из редких языков программирования, а **ПЛИС** – устройство для очень специальных применений вроде обработки сигнала с радиотелескопа. В действительности же **Verilog** и **ПЛИС** – вход во всю современную цифровую электронику. Это так, поскольку подавляющее большинство цифровых микросхем, разработанных за последние 25 лет, использует технологию компиляции (синтеза) схем из языков описания аппаратуры, главный из которых – **Verilog**. Огромное число инженеров, которые сейчас разрабатывают микросхемы в Apple, Intel и других электронных компаниях, во время учебы в таких университетах, как Беркли и MIT, прошли через лабораторные работы с использованием учебных отладочных плат на **ПЛИС**. Такого рода практические занятия позволяют наработать опыт в технологии проектирования на уровне регистровых передач (**Register Transfer Level**, **RTL**), которая используется для создания массовых микросхем внутри популярных цифровых устройств вроде Apple iPhone.

Данный учебник – важный шаг на пути построения экосистемы разработки современной электроники в России и в других странах бывшего СССР. России предстоит пройти тот же путь, который прошли Япония, Южная Корея, Тайвань и который сейчас проходит Китайская Народная Республика. На этом пути необходимо создать большое количество групп разработчиков разной специализации, готовых слаженно работать вместе. Таких разработчиков необходимо выращивать из сегодняшних студентов:

- Некоторые из студентов после окончания университета будут специализироваться в разработке микроархитектуры процессоров, сетевых устройств и других логически сложных блоков. Они будут или сами использовать **Verilog**, или создавать модели устройств, основываясь на понимании того, как работает технология **RTL**.
- Другие студенты будут специализироваться на физическом уровне проектирования. Им придется решать проблемы физического уровня, возникающие при превращении логического графа схемы в план расположения дорожек и транзисторов на пластине кремния при ее фабричном производстве. Хотя эти инженеры не будут писать на Verilog сами, им нужно понимать основы того, как работает в пространстве и **времени** граф, который они раскладывают.
- Третья группа студентов будет специализироваться на создании программ автоматизированного проектирования (**САПР**), которые помогают работать разработчикам аппаратуры. Компании, разрабатывающие такие программы, образуют целую небольшую индустрию автоматизации проектирования (**Electronic Design Automation**, **EDA**). В этой индустрии востребованы математически мыслящие инженеры, умеющие решать алгоритмически сложные задачи, которые возникают в программах синтеза, размещения и трассировки схем, автоматического доказательства их свойств и проверки их эквивалентности высокоуровневым моделям. Этим инженерам также необходимо пони-

мать основы цифрового синтеза и программирования на языках проектирования аппаратуры (**HDL**).

- Знать основы логического проектирования электроники необходимо и создателям аппаратно-программных систем. В компьютерах и встраиваемых системах XX века аппаратура и программы были довольно сильно разделены. В XXI веке, когда повышение скорости процессоров за счет простого уменьшения размера транзисторов зашло в тупик, началось быстрое развитие специализированных вычислителей. Сначала появились графические процессоры для вычислений трехмерной графики, сейчас бурно развиваются ускорители нейронных сетей, чипы для машинного зрения и даже специализированные вычислители криптовалют. Создателям всех этих устройств необходимо понимать и программную, и аппаратную стороны вычислений.

Обоснования для создания этой книги

Данная книга – «Цифровой синтез: практический курс» – создана совместными усилиями преподавателей и инженеров из нескольких университетов и компаний не только из России, но и из Украины и США. Этот практикум – один из нескольких образовательных проектов, нацеленных на подъем электроники в странах постсоветского пространства, которые все вместе можно уже рассматривать как осознанную стратегию.

К таким проектам относятся, например:

- Симулятор **MIPT-MIPS**[1], созданный базовой кафедрой Intel в МФТИ.
- Совместный курс компьютерной архитектуры и **ПЛИС**, созданный ВМК МГУ в партнерстве с европейскими университетами.
- Курс по интернету вещей, созданный в российском отделении Samsung в партнерстве с российскими университетами.
- Учебное софт-процессорное **MIPS** ядро **schoolMIPS**[2], разработанное Станиславом Жельнио из IVA Technologies.

Предтечами создания практикума стали три проекта:

- Перевод вводного учебника Дэвида Харриса и Сары Харрис «Цифровая схемотехника и архитектура компьютера». Этот перевод сделала в 2015 году группа из сорока с лишним преподавателей российских и украинских университетов, русских сотрудников компаний в Silicon Valley (включая MIPS, AMD, Synopsys, Apple и NVidia) и российских компаний (включая НИИСИ, МЦСТ, Модуль). Начинание поддержали британская компания Imagination Technologies, образовательное отделение РОСНАНО, а также российское издательство «ДМК-Пресс». Эта книга закрыла брешь в теоретической части преподавания языков описания аппаратуры и микроархитектуры, связала их с основами цифровой логики и программированием. Данный практикум можно рассматривать как расши-

[1] https://mipt-ilab.github.io/mipt-mips/.
[2] https://github.com/MIPSfpga/schoolMIPS.

ренное продолжение и дополнение к этому курсу, ориентированное на практическое применение.

- Семинары **MIPSfpga**, организованные в 2015–2017 годах Imagination Technologies в партнерстве с российскими, украинскими и казахскими университетами (МГУ, НИУ ВШЭ (МИЭМ), МИФИ, МФТИ, МИЭТ, ИТМО, Самарский университет, Томский ТГУ, украинские КПИ и КНУ, казахский AlmaU). **MIPSfpga** – это базовая конфигурация разработанного на **Verilog** промышленного процессора **MIPS interAptiv UP**, различные варианты которого используют такие компании, как Microchip Technology, Broadcom и Байкал Электроникс. Проекты, где студенты соединяют с **MIPSfpga** свои собственные блоки и синтезируют для **ПЛИС** простые системы на кристалле, позволяют им работать с тем же кодом, с которым работают инженеры в промышленности. К сожалению, **MIPSfpga** слишком сложен для начальной демонстрации основных принципов микроархитектуры. Данный практикум содержит целую главу, посвященную проекту **schoolMIPS**, который значительно проще, чем **MIPSfpga**. При этом **schoolMIPS** позволяет студенту понять базовое устройство процессоров, работу процессорного конвейера и прерываний.
- Цикл популярных семинаров Nanometer ASIC, организованный РОСНАНО, МИСиС, КПИ и Imagination Technologies, состоялся в 2016 году. Автор этих материалов – Чарльз Данчек, преподаватель Университета Калифорнии Санта-Круз и бывший инженер Intel. Мистер Данчек написал приложение к данному практикуму. В нем рассказано, как студент, выполнивший упражнения на **ПЛИС**, сможет в будущем перейти к разработке заказных микросхем (**ASIC**, **Application Specific Integrated Circuits**). К **ASIC** относятся практически все микросхемы, которые применяются в массовых электронных устройствах.

Юрий Панчул,
разработчик процессорных ядер MIPS и ускорителя нейросетей Wave,
Саннивейл, Калифорния

История создания данной книги

История создания книги «Цифровой синтез: практический курс» традиционно для таких проектов не проста. После успеха переводной версии учебника «Цифровая схемотехника и архитектура компьютера» на русском языке стало ясно, что необходимо его продолжение в виде расширенного практического курса, ориентированного на **Verilog** и обеспечивающего возможность выполнения практических задач на дешевых отладочных платах. Ясно было также и то, что одному человеку (и даже коллективу университетской кафедры) создать такой курс – непосильная задача, при том что издание книги требовало финансирования. У истоков идеи написания данной книги стоял Юрий Панчул, сумевший объединить для ее воплощения множество преподавателей и инженеров из России, Украины и США. Второй фактор, который помог появиться книге, – это знакомство Юрия Панчула (в рамках семинаров Nanometer ASIC) со мной, Александром Романовым, руководителем Учебной лаборатории Систем автоматизированного проектирования Московского института электроники и математики Национального университета «Высшая школа экономики» (УЛ САПР НИУ ВШЭ). Несмотря на экономический уклон НИУ ВШЭ, МИЭМ в прошлом был отдельным техническим вузом, готовившим студентов в области электроники и математики. Сейчас же, опираясь на необходимые технические и людские ресурсы, МИЭМ является одним из ведущих центров компетенций, в том числе и по цифровому синтезу. Идея создания учебника получила поддержку на уровне руководства в лице Евгения Аврамовича Крука, после чего началась работа по написанию книги, сбору и редактированию глав от разных авторов, рецензированию и разработке дополнительных материалов.

Чем эта книга отличается от других?

Особенность данной книги в том, что во всех главах каждый пример кода сопровождается листингом и тестбенчем, которые находятся в дополнительных материалах к книге (https://github.com/RomeoMe5/DDLM). Это позволяет читателю легко использовать уже готовые исходные коды, проводить моделирование, прототипирование на учебной плате с ПЛИС и модифицировать работающие примеры программ.

Немного про отладочные платы: поскольку команда, разрабатывавшая книгу, находилась территориально в разных местах, одна из целей курса состояла в том, чтобы сделать его максимально доступным для людей разного достатка. Изначально книга ориентирована на плату **De10-Lite** от компании Terasic на основе **ПЛИС MAX 10K** производства **Intel FPGA** (в прошлом – компания Altera). Выбор платы обусловлен ее относительно небольшой стоимостью (около $55 по академической цене) и доступностью, а также тем, что она обладает достаточной периферией, функционалом, емкостью ресурсов и даже может быть интегрирована с платформой Arduino. Платы на основе **ПЛИС Intel FPGA** (**Altera**) популярны в России и ближнем зарубежье и имеются во многих академических организациях. При этом исходные коды примеров могут быть легко перенесены и на другие платы. Поскольку данный проект ориентирован на широкое сообщество, в ближайшее время планируется миграция примеров кодов на другие попу-

лярные отладочные платы на основе **ПЛИС**: **Марсоход**, **Terasic De1-SoC**, **Terasic De10-Standard**, **Digilent Nexys 3/4** и др. Для выполнения работ не требуется какого-либо платного программного обеспечения, так как **Quartus Prime Lite Edition** + **Modelsim** / **GTKWave** распространяются свободно. Таким образом, чтобы приступить к изучению цифрового синтеза, достаточно иметь только эту книгу; при желании увидеть результаты не только моделирования, но и прототипирования понадобится какая-либо отладочная плата на основе **ПЛИС**.

Содержание книги

Еще одна особенность книги – то, что авторы представили ее главы в виде отдельных независимых разделов. То есть можно изучать тот раздел, который необходим сейчас, и обращаться к другим главам по необходимости. Каждую главу в основном писали один-два автора, после чего редактировали и рецензировали другие люди. Вот почему каждая глава имеет свой неповторимый оригинальный стиль, приведенный, впрочем, к единому оформлению и терминологии. Таким образом, в случае если одна глава воспринимается читателем тяжело, то, возможно, другая будет читаться им намного легче.

Поскольку книга задумана как практикум, ее подразделы сопровождаются заданиями для самостоятельной проработки. В конце каждой главы приводятся вопросы и упражнения, позволяющие преподавателям встраивать данный материал в любой учебный курс, а читателям книги – закрепить новые знания, самостоятельно выполнив предлагаемые задания.

Рассмотрим кратко по главам содержание книги:

Глава 1. Основы комбинационной логики. Маршрут разработки цифровых схем
Глава знакомит читателя с типичным циклом разработки цифровой системы на примере проектирования простой комбинационной схемы, которая содержит всего несколько логических вентилей. Сначала демонстрируется, как описать цифровую схему с помощью графического редактора. Далее та же схема проектируется с использованием языка описания аппаратуры. После этого демонстрируются этапы моделирования и прототипирования.

Глава 2. Основы последовательностной логики. Управление энергопотреблением цифровой схемы
Данная глава посвящена разработке последовательностных устройств. Рассматриваются защелки и триггеры на примерах их различных реализаций на языке **Verilog**. Глава позволяет понять, чем комбинационная логика отличается от последовательностной и как можно управлять энергопотреблением цифровых устройств на этапе их проектирования.

Глава 3. Шифраторы и дешифраторы. Скорость работы комбинационных блоков
В этой главе приводятся примеры реализаций таких важных комбинационных блоков, как шифраторы и дешифраторы, а также дается методика оценки временных характеристик цифровых блоков и их оптимизации.

Глава 4. Мультиплексор, демультиплексор и селектор. Построение иерархических модулей
В главе на примере разработки различных вариантов реализаций таких комбинационных блоков, как мультиплексор, демультиплексор и селектор, демонстрируется иерархический подход к проектированию цифровых устройств. Также вводятся понятие параметризации модулей и конструкция **generate**. В конце главы демонстрируются некоторые приемы использования мультиплексоров на практических примерах.

Глава 5. Сумматор, компаратор, устройство сдвига и АЛУ. Повышение скорости арифметических операций
В главе рассматриваются примеры всевозможных реализаций комбинационной арифметики (сумматоров, компараторов, устройств сдвига и АЛУ на их основе). Отдельный раздел посвящен повышению скорости арифметических блоков на этапе их проектирования.

Глава 6. Последовательностная логика. Счетчики и сдвиговые регистры
В данной главе изложение возвращается к последовательностной логике и особенностям ее разработки на **Verilog** (блокирующие/неблокирующие присвоения, понятие защелок и т. д.) на примере разработки счетчиков и сдвиговых регистров. В конце главы даны примеры организации взаимодействия цифровых систем с простыми периферийными модулями.

Глава 7. Память: регистровый файл и стек
Эта глава посвящена различным вариантам реализации памяти: регистровая память, однопортовая/многопортовая память, стек, очередь и т. д. В конце главы приводится небольшой пример проектирования на **HDL** памяти с привязкой к библиотекам фабрик-производителей **ASIC**.

Глава 8. Конечные автоматы: основы
В главе приводятся основные понятия и приемы для проектирования конечных автоматов. Иллюстрируются особенности проектирования конечных автоматов Мили и Мура и рассматриваются наиболее оптимальные случаи их использования. Также демонстрируется использование специальных инструментов для проектирования и анализа конечных автоматов.

Глава 9. Использование конечных автоматов для связи с периферийными устройствами
Эта глава расширяет тему проектирования конечных автоматов путем формализации академического подхода к проектированию автоматов; также демонстрируются и другие автоматы, например на основе счетчика. Глава обосновывает применение конечных автоматов в проектировании цифровых устройств как ключевого блока управления ими.

Глава 10. Конвейерная обработка данных
Глава посвящена описанию конвейерного подхода к обработке данных и особенностям разработки на **Verilog**. Приводится сравнение комбинационного, многотактного и конвейерного подходов на примере разработки арифметического

блока; описываются дополнительные приемы повышения эффективности конвейерных схем.

Глава 11. Софт-процессор: основы микроархитектуры
Данная глава объединяет в себе необходимые знания из предыдущих разделов этой книги при проектировании **HDL**-реализации относительно простого, но при этом реально функционирующего, однотактного софт-процессорного ядра с архитектурой **MIPS**. В главе даются понятия микроархитектуры, тракта данных, устройства управления и т. д.; демонстрируется процесс добавления новых инструкций и расширения процессорного ядра и его блоков.

Приложение А. Путь вперед: от FPGA-устройств к массовому рынку ASIC для популярных гаджетов
Приложение дает общее представление об этапах проектирования чипов **ASIC**, начиная с идеи специализированного чипа и заканчивая его конечной реализацией на кристалле. Это первый шаг в подготовке инженеров, прошедших обучение проектированию на **ПЛИС**, к применению имеющихся у них навыков в разработке **ASIC** для конкретных приложений.

Приложение Б. История успеха победы российской команды на международном конкурсе InnovateFPGA от Intel
Это приложение – небольшое интервью с участником российской команды, занявшей второе место на одном из основных международных конкурсов по проектированию на **ПЛИС** – **InnovateFPGA** от Intel.

Таким образом, данный практикум по **Verilog** и **ПЛИС** дополняет и объединяет теоретические курсы по цифровой логике, языкам описания аппаратуры, компьютерной архитектуре и микроархитектуре. Практикум также подготавливает студентов к работе с промышленными процессорными ядрами, к созданию специализированных вычислителей (например, ускорителей нейросетей) и курсов **VLSI** по проектированию массовых микросхем **ASIC**. Он будет полезен разработчикам аппаратно-программных систем, а также прикладным математикам, интересующимся алгоритмами **САПР**.

Выражаем надежду на то, что практикум станет такой же надежной основой курсов по цифровой электронике для большого количества университетов России и стран СНГ, какой уже стал переводной учебник «Цифровая схемотехника и архитектура компьютера» авторов Дэвида и Сары Харрис. В результате этого в России появится новое поколение инженеров, способных объединять лучшие мировые практики с изобретательностью, присущей народам наших стран; так Россия вместе со своими соседями займет достойное место в мировой промышленной электронике.

Александр Юрьевич Романов,
к. т. н., доцент МИЭМ НИУ ВШЭ,
преподаватель курсов «Проектирование систем на кристалле»
и «Системное проектирование цифровых устройств»,
г. Москва, Россия

Данный учебник позволяет первично познакомиться с цифровым дизайном через изучение языка Verilog и правильных design-style и best practices

Прочтение этой книги вызвало у меня воспоминания об относительно далеком прошлом. В те времена, будучи студентом факультета технической информатики Мангеймского университета в Германии, я имел первый опыт программирования на языке **Verilog** и загрузки программного кода в **ПЛИС**. Самой сложной задачей была реализация программы для рисования, и самые продвинутые студенты рисовали круги. Помнится, что студенты одной из групп смогли разработать элементарный процессор и запрограммировать отрисовку кругов на ассемблере для этого процессора. Эти студенты стали теперь профессорами в университетах Германии.

Купив в 1990 году компанию Gateway Design Automation, в которой работал изобретатель языка **Verilog** Фил Мурби, компания Cadence Design Systems стала правообладателем **Verilog**. Этот язык изначально разрабатывался для описания микросхем для цифрового симулятора. Со временем **Verilog** также стал использоваться для синтеза, то есть превращения описания микросхемы на более абстрактном уровне **Register-Transfer-Logic RTL** в менее абстрактный **Gate-Level**, где описываются вентили, которые предоставляют конкретные изготовители микросхем (такие как: TSMC, UMC – Тайвань, GlobalFoundries – США). Этих изготовителей принято называть foundries; российские foundries – Ангстрем и Микрон в Зеленограде.

Verilog настолько успешен, что существуют несколько языков, которые имеют с **Verilog** общее название, но используются для разных целей. Например, **Verilog-A** – для моделирования аналоговых микросхем, **Verilog-AMS** – для моделирования цифроаналоговых микросхем, **SystemVerilog** – для верификации больших цифровых микросхем.

Конкретно этот учебник позволяет первично ознакомиться с цифровым дизайном, но охватывает при этом более широкие аспекты – обучает не только самому языку, но правильному **design-style** и **best practices**, что чрезвычайно важно для недопущения ошибок при подготовке к синтезу. Даже самое умное ПО не может синтезировать оптимальный **PPA** (**power**, **performance**, **area** – три самых важных показателя качества микросхемы), если исходный код (даже будучи грамматически правильным) не соблюдал определенные **design-styles**. Таким образом, данный учебник обладает двумя весомыми преимуществами: во-первых, он включает в себя базовую информацию, поиск которой требует, как правило, больших затрат времени и сил (проблемы выбора подходящего и, по возможности, бесплатного ПО и совместимой **ПЛИС**, проблемы их установки/настройки), и во-вторых, может использоваться для самообучения как студентами, так и научными сотрудниками, поскольку снабжен всеми необходимыми материалами от подготовленных исходных кодов программ для моделирования и синтеза до презентаций к каждой главе.

Компания Cadence (сотрудником которой я являюсь уже в течение 16 лет) выпускает программное обеспечение, применяемое на самых различных стадиях

проектирования и разработки комплексных микросхем, а также целых систем (приборы с печатными платами, **RF**-компонентами и несколькими системами на кристалле (**SoC**) в одной упаковке). Хотя ПО компании Cadence существенно облегчает процесс проектировки систем на кристалле с миллиардами элементов, именно талантливые и хорошо обученные инженеры являются главным капиталом компаний – разработчиков микросхем. Поэтому Cadence в высокой степени заинтересована в обучении следующего поколения инженеров-микроэлектронщиков, так как видит в них и будущих разработчиков, и будущих клиентов. В 2007 году была создана Cadence Academic Network (группа внутри компании Cadence), поддерживающая связи с ведущими университетами и предоставляющая им академические лицензии для доступа к ПО. Следует отметить, что данное программное обеспечение используется разработчиками самых передовых компаний мира, и Cadence Academic Network распространяет учебные материалы для обучения этим комплексным программам.

Компания Cadence приветствует появление такой нужной книги на российском рынке и будет рада контактам с читателями, усвоившими азы разработки цифровых микросхем и готовыми к дальнейшему обучению. Хотелось бы пожелать всем читателям этого учебника успехов и новых познаний.

Антон Клотц,
University Program Manager
Cadence Design Systems

Точно так же, как до этого в России стала массовой профессия программиста, данный курс поможет сделать массовой профессию электронного инженера – разработчика цифровой аппаратуры любой сложности

История развития мировой вычислительной техники насчитывает уже более 70 лет и является прямым отражением той жестокой конкурентной борьбы сверхдержав и их союзников, которую они ведут за контроль над мировой экономикой, финансовыми и товарными потоками с целью получения долгосрочных преимуществ перед остальными странами. Кроме того, разработка быстродействующих компьютеров является неотъемлемой частью технологического соперничества нынешних сверхдержав – США и Китая. Галопирующее развитие и широкое внедрение массовых коммуникаций и встроенных микрокомпьютеров в контексте интернета вещей приводит к тотальной цифровизации экономики.

Технологическая зависимость в области электронной компонентной базы может вызвать катастрофические последствия в случае разного рода санкций и ограничений на поставки со стороны зарубежных партнеров.

Несмотря на то что цифровая электронная инженерия была невостребованной в российской промышленности и устойчиво деградировала с 1990-х гг., в настоящее время наличие собственных инженерных компетенций в данной области на массовом уровне становится условием выживания любой страны, претендующей даже на частичный технологический суверенитет.

Учебное пособие «Цифровой синтез: практический курс» как комплекс практических работ является очередным этапом в создании массовой школы цифровой электронной инженерии на всем пространстве СНГ. Этому предшествовало издание профильных учебников и проведение серий семинаров, нацеленных на создание практических работ по различным разделам проектирования цифровых схем и микропроцессоров, о чем подробно написано Юрием Панчулом в предисловии.

Различные лабораторные работы по проектированию и синтезу цифровых схем на языках регистровых передач для описания аппаратуры стали необходимой частью подготовки электронных инженеров – проектировщиков микросхем для массовых электронных изделий. Через подобный практикум проходят будущие создатели смартфонов, разработчики автомобильной электроники и процессоров для различных встроенных применений, включая электронику для космических зондов. В США известным примером такого практического курса является 6.111 из MIT[1], а также различные варианты лабораторных заданий с **Verilog**/**VHDL** и **ПЛИС**/**FPGA** предусмотрены учебными программами практически для всех, кому преподают электронику, включая студентов местных университетов в небольших штатах.

Авторы надеются, что данный курс поможет сделать массовой профессию электронного инженера – разработчика цифровой аппаратуры любой сложности. Точно так же, как до этого в России стала массовой профессия программиста или

[1] http://web.mit.edu/6.111/volume2/www/f2018/index.html.

программного инженера – разработчика систем и приложений. Основами языков регистровых передач или **Register Transfer Level** (**RTL**) нужно владеть не только самим разработчикам, но и представителям смежных профессий – верификаторам, специалистам по физическому проектированию, а также алгоритмистам, которые пишут инструментальные программы для разработчиков электроники. Программисты встроенных систем и программ искусственного интеллекта тоже получат пользу от понимания того, как работают классические процессоры, GPU и нейроускорители.

Данное пособие создано усилиями международного авторского коллектива специалистов из ведущих университетов и ИТ-компаний. Помимо традиционных основ проектирования цифровой техники, авторы ввели в курс дополнительные проекты, которых не хватало в имеющихся курсах. Они включают развернутое объяснение принципов конвейерных вычислений, введение в микроархитектуру процессоров, а также детальные разъяснения о том, как студент может использовать свой опыт, полученный от лабораторных работ на ПЛИС, в своей дальнейшей карьере. Полученные практические навыки обеспечивают статус разработчика массовых изделий на специализированных полузаказных микросхемах **ASIC** (**Application Specific Integration Circuits**) и крупных систем на кристалле (СнК), которых так остро не хватает в отечественной индустрии.

Выражаю уверенность в том, что это пособие имеет очень хороший потенциал для его последующего перевода на английский язык и массового использования в кооперации с компаниями, разрабатывающими инструментарий электронной инженерии. Такими партнерами могут стать крупные компании Cadence, Synopsys и Mentor Graphics.

С пожеланиями успехов авторам и пользователям данного учебного пособия,

Тимур Турсунович Палташев,
доктор технических наук, профессор,
руководитель академических проектов,
Radeon Technology Group,
Корпорация Advanced Micro Devices.
17 марта 2020 года

Книга «Цифровой синтез: практический курс» является, по сути, продолжением классического учебника по проектированию микроэлектроники «Цифровая схемотехника и архитектура компьютера» Дэвида Харриса и Сары Харрис, новая редакция которого (на русском языке) вышла всего несколько лет назад

Рецензируемая книга представляет собой расширенный практический курс, ориентированный на **Verilog** и обеспечивающий возможность выполнения практических задач на широкодоступных отладочных платах **FPGA**. Именно такой подход позволяет эффективно организовать подготовку квалифицированных разработчиков для отечественной микроэлектроники. Следует отметить, что данное издание, подготовленное международным авторским коллективом русскоязычных специалистов из ведущих университетов и ИТ-компаний, обладает всем необходимым потенциалом для последующего перевода на английский язык и дальнейшего использования в университетской среде.

История развития систем автоматизированного проектирования берет свое начало в 70–80-х годах прошлого века. К тому времени сложность систем, возраставшая по мере увеличения количества транзисторов на микросхеме по закону Мура, стала такой, что ручное проектирование схемотехники микроэлектронных устройств становилось практически невозможным. Проектирование аппаратуры повторило историю развития языков программирования: если первые вычислительные системы программировались на уровне кодов, то уже в 50-е годы приходилось приступать к разработке систем автоматизации программирования и, соответственно, языков программирования высокого уровня. По тому же пути (несколько позднее) пошли и разработчики аппаратуры – для проектирования новых вычислительных систем использовалось программное обеспечение, разработанное для уже существующих компьютеров. Но если для проектирования печатных плат устройств уже в середине 80-х годов существовали специализированные САПР, то к промышленному проектированию схемотехники микроэлектронных изделий удалось приступить только в начале 90-х (при том что сам **Verilog** был разработан в середине 80-х). Однако первоначально он был ориентирован на описание и моделирование логических схем; применение его для синтеза на уровне логических элементов и гейтов было реализовано лишь с ростом популярности основанных на нем средств моделирования и отладки.

Знание **Verilog** и использование его обширного инструментария с целью проектирования цифровых систем сегодня является абсолютно необходимым навыком для любого инженера-электронщика (кроме разве что специалистов в аналоговой и силовой электронике, где количество элементов весьма ограничено, зато чрезвычайно важны их физические характеристики). В микроэлектронике ситуация прямо противоположная – все цифровые системы строятся из огромного (как правило, исчисляемого миллионами, а иногда десятками и сотнями миллионов) количества физически одинаковых элементов. Поэтому для инженера-схемотехника необходимы инструменты иерархической абстракции, позволяющие манипулировать логикой крупных модулей и строить из них системы, не опускаясь на уровень отдельных транзисторов и имея при этом возможность анализировать их поведение во времени. Именно такую возможность и предоставляет **Verilog**

с его инструментарием, зачастую поддерживающий и другие языки описания аппаратуры (в частности, **VHDL**).

В течение достаточно длительного времени одной из основных проблем отладки и тестирования микроэлектронной аппаратуры являлись технологические процессы производства микросхем. Ситуация коренным образом изменилась с появлением в середине 80-х и широким распространением в 90-е годы прошлого века технологии **FPGA** (**ПЛИС** в русскоязычной литературе) – программируемых логических матриц, состоящих из сотен тысяч и миллионов одинаковых индивидуально программируемых гейтов. В то же время именно **Verilog** и другие языки описания аппаратуры сделали эффективным проектирование и использование систем на **FPGA**. И если в 90-е годы они использовались в основном в телекоммуникационном и сетевом оборудовании, то сегодня трудно представить большую цифровую систему, не использующую **FPGA** в качестве акселераторов тех или иных процессов. Эта технология оказалась чрезвычайно удачным компромиссом между эффективностью, сравнимой с чисто аппаратными решениями (хотя и уступающей им), и гибкостью, простотой использования, характерной для программного обеспечения.

Таким образом, **FPGA** стали активно применяться в тестировании блоков схемотехники или целых цифровых систем, ориентированных на дальнейшую реализацию в микроэлектронном исполнении (**ASIC**). Хотя до сих пор не существует полностью автоматизированного процесса переноса **Verilog**-программы из **FPGA**-прототипа на технологический процесс конкретного изготовителя, этот процесс существенно повышает эффективность и снижает стоимость отладки аппаратных решений микроэлектроники. Для упрощения процесса прототипирования на **FPGA** все основные производители стали выпускать готовые отладочные платы, включающие (кроме самой микросхемы **FPGA**) все необходимое окружение, в том числе стандартные интерфейсы с компьютером; отпала нужда в специализированных программаторах, логических анализаторах и другом инженерном оборудовании стадии отладки.

В то же время с массовым распространением микросхем **FPGA**, отладочных плат на их основе и радикальным падением стоимости за счет эффекта масштаба стало возможным использовать их для учебных задач. Современные инженерные курсы в области микроэлектроники, как правило, используют **FPGA** в качестве основного инструмента проверки и тестирования программ на **Verilog** даже при ориентации на создание в дальнейшем **ASIC** и **SoC** (систем на кристалле).

Предлагаемый в книге «Цифровой синтез: практический курс» практикум по инструментам и технологиям цифрового синтеза схемотехники полностью охватывает все основные разделы цифровой схемотехники, а также используемый при проектировании бесплатный инструментарий. Покрывается практически весь материал, представленный в учебнике «Цифровая схемотехника и архитектура компьютера» Дэвида Харриса и Сары Харрис. Важной особенностью рецензируемой книги является то, что во всех главах каждый пример кода сопровождается исходным кодом и тестбенчем, которые находятся в дополнительных материалах к книге.

В заключение хотелось бы отметить, что необходимость в подготовке квалифицированных кадров в области проектирования микроэлектроники является вполне объективной – востребованность таких кадров будет только расти в связи с массовым распространением устройств интернета вещей и переводом все большего объема функционала на системы на кристалле. Если в последние десятилетия основным фокусом при обучении специалистов по цифровым системам была подготовка программистов и (позднее) аналитиков данных, то в ближайшие годы будут все более востребованы инженеры со специализацией на стыке программирования и аппаратуры, специалисты по киберфизическим системам и электронике; в нашей стране их готовят в ограниченном количестве учебных заведений, при этом ощущается существенный дефицит опыта и литературы. Представляется важным, чтобы МИЭМ НИУ ВШЭ стал одним из лидеров этого направления в российской академической среде; таким образом, издание рецензируемой книги в «ДМК Пресс» – ведущем издательстве, специализирующемся на выпуске компьютерной и радиотехнической литературы, – является важным шагом в этом направлении.

Игорь Рубенович Агамирзян,
вице-президент НИУ ВШЭ,
профессор факультета компьютерных наук НИУ ВШЭ,
канд. физ.-мат. наук

Сергей Иванец, Александр Романов

Цифровой синтез: практический курс

Глава 1. Основы комбинационной логики. Маршрут разработки цифровых схем

Содержание

*Глава посвящена основам цифрового дизайна и знакомит с логическими вентилями – основными элементами цифровых систем. Вначале описывается проектирование простой схемы, содержащей всего несколько логических вентилей, с помощью графического редактора. Далее спроектирована та же схема, но с использованием языка описания аппаратуры (**Hardware Description Language**, **HDL**). Спроектированная схема проверяется с помощью симулятора – специальной программы для тестирования цифровых схем. Для того чтобы увидеть, как работает схема «в железе», программируется микросхема **ПЛИС** (**Программируемая логическая интегральная схема, Field-Programmable Gate Array, FPGA**). Выполнив все описанные шаги, вы познакомитесь с типичным циклом разработки цифровой системы.*

Требования к аппаратным и программным средствам

Для выполнения практических работ понадобится следующее программное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime Lite Edition 17.0**[1];*
- *пакет **ModelSim Altera Edition**;*
- *программы **Icarus Verilog** и **GTKWave**[2].*

*Программы Quartus и ModelSim являются платными, но они имеют и студенческие бесплатные версии, которые могут быть свободно скачаны с сайта производителя **ПЛИС Altera** (**Intel FPGA**).*

*Также в данном практикуме используется отладочная плата компании **Terasic DE10Lite**[3]. Она содержит микросхему **ПЛИС** компании **Intel FPGA MAX10**[4] (**10M50DAF484C7G**). В папке doc дополнительных материалов к настоящей главе* (https://github.com/RomeoMe5/DDLM, lab_01/doc) *размещены инструкция к данной отладочной плате и ее электрическая схема (эти же документы могут быть бесплатно загружены с сайта компании **Terasic**).*

*Хотя в этом практикуме используется отладочная плата с микросхемой **MAX10** от компании **Intel FPGA**, концепции и методологии, которые вы узнаете при выполнении работ, могут быть использованы и при работе с **ПЛИС** от других производителей, например **Xilinx**. Однако следует учитывать то, что инструменты для проектирования и микросхемы быстро развиваются, и последние версии **САПР** компании **Xilinx** (**Vivado Design Suite**)[5] больше не поддерживают схематический редактор, а только разработку на основе языков описания аппаратуры.*

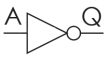
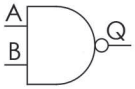
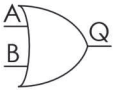
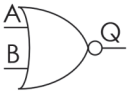
---

1 http://dl.altera.com/?edition=lite.
2 Инсталлятор можно найти в папке **pkg** материалов к данной главе.
3 http://de10-lite.terasic.com/.
4 https://www.altera.com/products/fpga/max-series/max-10/overview.html.
5 https://www.xilinx.com/support/answers/53764.htm.

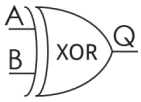
1.1. Краткие теоретические сведения

Рассмотрим цикл разработки **комбинационного устройства**, схема которого будет содержать логические вентили – **И**, **ИЛИ**, **НЕ**, а также **исключающее ИЛИ**. Особенностью комбинационных схем является то, что они выполняют только заданную логическую функцию над входными сигналами, но не сохраняют их значения. В следующей главе также рассмотрены **последовательностные** устройства, которые содержат элементы для хранения значений, и их состояние поэтому может зависеть не только от текущего набора входных сигналов, но и от предыстории. Логические вентили являются теми основными «кирпичиками», с помощью которых строятся все остальные элементы цифровых систем – от простых элементов, таких как дешифратор или триггер, до самых сложных – процессоров и систем на кристалле (**system-on-chip**, **SoC**).

Для построения устройства на основе логических элементов необходимо определить логические функции, которые описывают требуемые логические операции. В таблице, приводимой ниже, показаны основные логические элементы, их обозначения, уравнения и таблицы истинности.

Вентиль	Символ	Уравнение	Таблица истинности
НЕ (NOT)		$Q = \overline{A}$	A Q: 0 1; 1 0
И (AND)		$Q = A \cdot B = A \& B$	A B Q: 0 0 0; 0 1 0; 1 0 0; 1 1 1
И-НЕ (NAND)		$Q = \overline{A \cdot B} = \overline{A \& B}$	A B Q: 0 0 1; 0 1 1; 1 0 1; 1 1 0
ИЛИ (OR)		$Q = A + B$	A B Q: 0 0 0; 0 1 1; 1 0 1; 1 1 1
ИЛИ-НЕ (NOR)		$Q = \overline{A + B}$	A B Q: 0 0 1; 0 1 0; 1 0 0; 1 1 0

Исключающее ИЛИ (XOR)



$Q = A \oplus B$

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

На начальном этапе для разработки принципиальной схемы будет использован схемотехнический редактор. Этот метод создания цифровых систем использовался в начале 1980-х годов и позднее был вытеснен языками описания аппаратуры. Описание схемы на языке описания аппаратуры с помощью компилятора синтезируется в принципиальную схему, которая в дальнейшем реализуется либо на микросхеме **ПЛИС**, либо с помощью специализированной микросхемы (**Application-Specific Integrated Circuit**, **ASIC**). Такой подход позволяет значительно увеличить скорость создания цифровых устройств и обеспечивает существование наиболее передовых современных разработок в виде систем на кристалле, которые используются в мобильных телефонах, планшетах и других устройствах. Данный практический курс в первую очередь ориентирован на создание цифровых устройств с помощью языков описания аппаратуры.

Еще одним преимуществом использования языков описания аппаратуры является увеличение скорости отладки проектов. Это происходит потому, что исправлять схему гораздо дольше, чем код, ее описывающий. Кроме того, HDL содержат в своем составе конструкции, предназначенные для написания отладочных тестов (**test bench**), которые используются для симуляции цифровых устройств, что еще больше ускоряет отладку и верификацию проектов.

На рис. 1.1 приведены основные операции, необходимые для создания устройства с использованием **ПЛИС**. Рассмотрим их более подробно. Сначала необходимо создать спецификацию или техническое задание – документ, содержащий полный список требований к устройству. Затем выполняется разработка с помощью языков описания аппаратуры. На этом этапе можно использовать библиотеки, в которых могут быть описаны блоки различной сложности. Далее выполняется верификация проекта в симуляторе и синтез списка связей (**netlist**). Все описанные выше этапы относят к **front end** разработке микросхем или проектов на **ПЛИС**. **Back end** процесса разработки, часто называемый **place and route**, включает в себя размещение элементов на кристалле и разводку межсоединений. База данных разводки (например, **GDSII**-файл), полученная после этого этапа, может быть использована производителем чипов для изготовления специализированной микросхемы.

Поскольку изготовление микросхем является весьма дорогостоящей операцией, в данном практическом курсе предлагается более дешевый метод – с использованием **ПЛИС**. **ПЛИС** – это специализированная микросхема, содержащая матрицу ячеек с реконфигурируемой логической функцией. Ячейка может быть сконфигурирована для выполнения различных логических функций: одна для выполнения операции **И**, другая – операции **ИЛИ** и т. д. Функции ячеек и их соединение между собой могут быть изменены и записаны в специальной конфигурационной памяти **ПЛИС**.

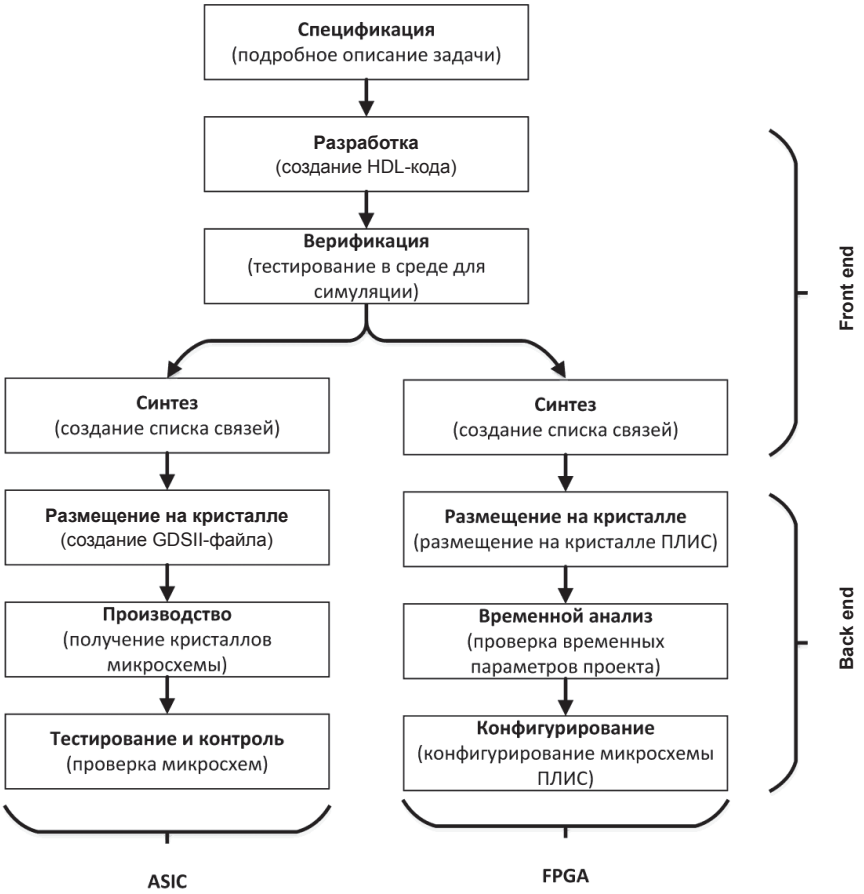


Рис. 1.1 Порядок разработки проектов для ASIC и FPGA

Ключевой концепцией данного практического курса является анализ и учет на этапе проектирования временных параметров устройства. Хотя сигнал через логический вентиль проходит очень быстро (данный параметр исчисляется единицами и десятыми долями наносекунды), задержка сигнала внутри вентиля не является нулевой. Таким образом, сигналу необходимо какое-то время для того, чтобы пройти со входа на выход. На рис. 1.2 показана задержка распространения сигнала через буфер, то есть повторитель сигнала. Данный параметр для одинаковых микросхем может варьироваться в зависимости от различных факторов: от максимального значения **tpd** (**propagation delay**) до минимального **tcd** (**contamination delay**). Для последовательно соединенных нескольких вентилей результирующая задержка представляет собой сумму задержек отдельных вентилей, и ее значение может превышать максимально допустимое. Далее при выполнении работ будет показано, что инструменты синтеза учитывают разброс значений задержки, суммируют задержку на нескольких вентилях и т. д. Задача разработчика – определить реалистичные временные параметры проекта. Если тактирование схемы не выполняется, то разработчик должен разделить проект

на несколько небольших частей, с которыми он будет работать во время анализа временных параметров. Далее в последующих главах будет введено понятие тактового, или синхронизирующего, сигнала, который используется для упрощения синхронизации сложных последовательностных схем путем размещения фрагментов логики между регистрами.

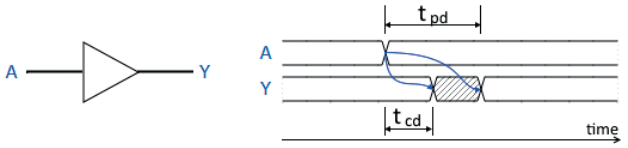


Рис. 1.2 Минимальная и максимальная задержки

1.2 Использование схемотехнического редактора

Quartus Prime – это интегрированная среда разработки, которая используется для проектирования на основе **ПЛИС** от компании **Intel FPGA**. Проекты могут отличаться по своей сложности – от простого управления светодиодом до сложной системы на кристалле, содержащей одно или несколько процессорных ядер. Quartus содержит инструменты для разработки проекта, его синтеза, размещения на кристалле и разводки межсоединений (данная операция носит название **place-and-route** или **fitting** для проектов на **ПЛИС**), генерации конфигурационного файла и загрузки этого файла в микросхему на плате.

Конфигурирование **ПЛИС** – это процесс записи определенной последовательности бит в конфигурационную память микросхемы, то есть программирования ее на выполнение определенной логической функции. Поэтому загрузку конфигурации в микросхему **ПЛИС** часто называют программированием. Важно отличать это от понятия программирования как процесса разработки программ.

1.2.1 Установка пакета Quartus Prime

Существует несколько версий пакета Quartus Prime:

- **Quartus Prime Lite Edition**[1] – полностью бесплатная версия пакета **Quartus Prime**;
- **Quartus Prime Pro Edition** или **Quartus Prime Standard Edition** – коммерческая версия пакета.

Quartus Prime Lite Edition поддерживает все функции, необходимые для разработки проекта на **ПЛИС**, и поэтому в данном практическом курсе будет использоваться именно эта версия программы. Необходимую версию пакета **Quartus Prime Lite Edition** можно скачать с сайта компании **Intel FPGA**. В настоящем практическом курсе будет использована версия 17.0 данного пакета. Различие между версиями пакета описано в документации на **Quartus Prime**[2]. При скачивании не-

[1] https://fpgasoftware.intel.com/?edition=lite.
[2] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/po/ss-quartus-comparison.pdf.

обходимо позаботиться о том, чтобы скачать не только сам пакет **Quartus Prime Lite Edition**, но и поддержку микросхемы **MAX 10**, а также пакет ModelSim – **Intel FPGA Edition**.

В процессе установки **Quartus Prime Lite Edition** необходимо обратить внимание на несколько параметров:

- папку, в которую будет устанавливаться пакет (желательно, чтобы ее имя было на латинице и без пробелов);
- поддерживаемые семейства **ПЛИС**. Поддержка различных семейств и определенных микросхем зависит от версии пакета, поэтому перед его установкой следует убедиться в том, что необходимая микросхема поддерживается выбранной версией пакета. Иногда необходимо устанавливать более старые версии Quartus для работы с семействами микросхем, выпускавшимися ранее;
- устанавливаемые компоненты. Настоятельно рекомендуется выбирать пункт **Modelsim – Intel FPGA Starter Edition** при инсталляции. Это позволит установить бесплатную версию пакета Modelsim – мощного симулятора цифровых систем, используемого в промышленности.

Основные этапы установки приведены на рис. 1.3, 1.4, 1.5, 1.6, 1.7.

Рис. 1.3 Начало установки Quartus Prime Lite Edition

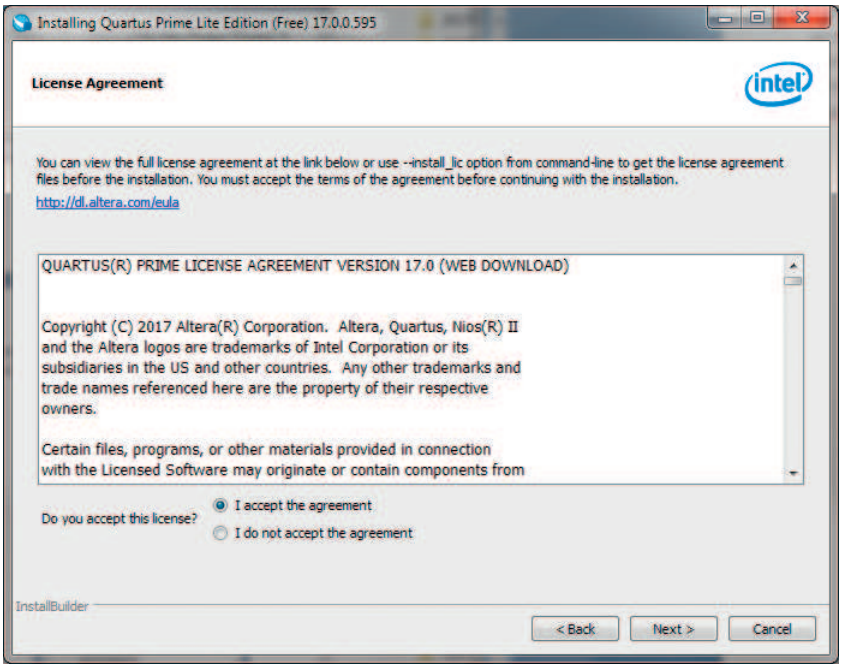


Рис. 1.4 Соглашение о лицензии в Quartus Prime Lite Edition

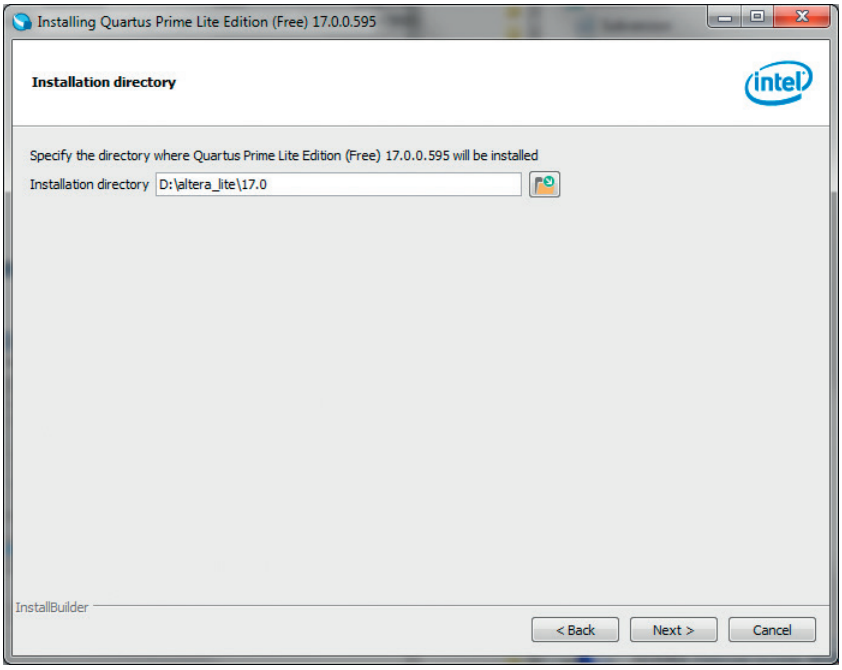


Рис. 1.5 Назначение папки, в которую будет установлен Quartus Prime Lite Edition

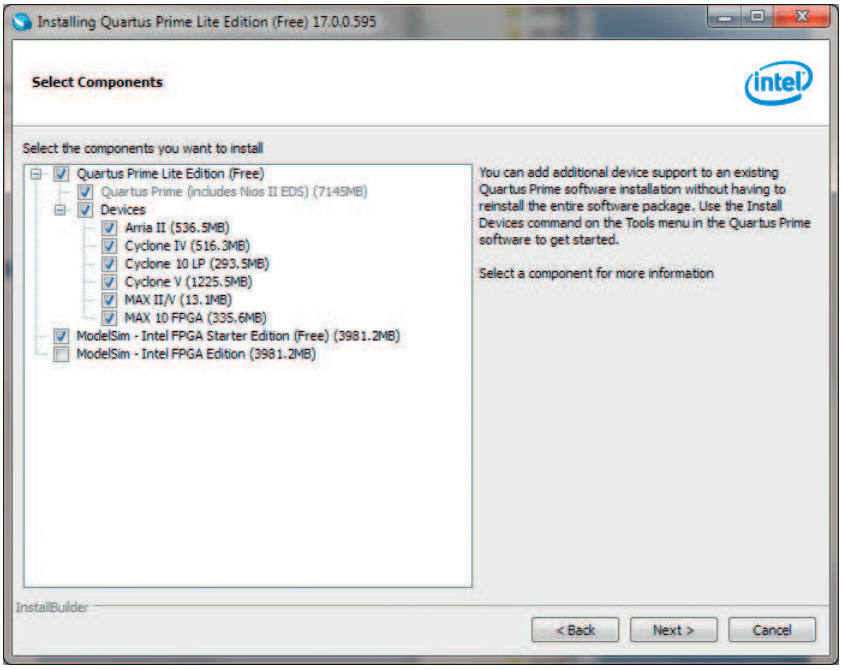


Рис. 1.6 Выбор компонентов Quartus Prime Lite Edition и поддерживаемых семейств микросхем

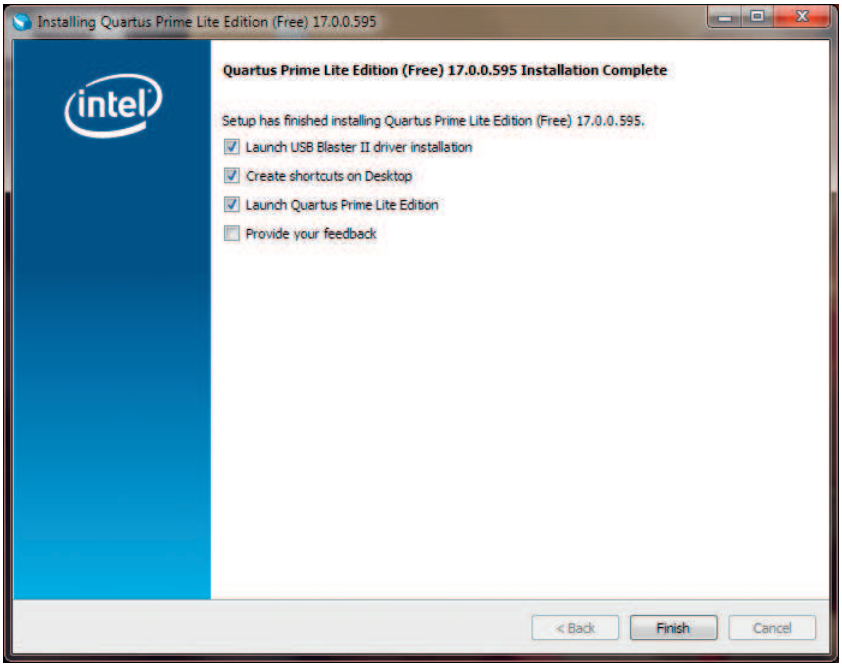


Рис. 1.7 Окончание установки Quartus Prime Lite Edition

1.2.2 Создание проекта в схемотехническом редакторе

Quartus Prime рассматривает любую схему независимо от ее размера (будь то несколько вентилей или система на кристалле) как проект (**project**). Каждый проект соответствует папке, в которой он находится. Все файлы проекта располагаются в папке проекта независимо от того, созданы ли они разработчиком или являются результатом работы компилятора, поэтому рекомендуется использовать разные папки для разных проектов. **Quartus Prime** может работать одновременно только с одним проектом.

Перед созданием нового проекта необходимо определить некоторые условия:

- папку, в которой будет располагаться проект на диске;
- имя проекта. Недопустимо использование русских букв в имени проекта или пути к нему, а также символов ^ & ? * < >;
- имя объекта верхнего уровня, то есть имя модуля, который вы разрабатываете либо в виде схемы, либо описываете на языке описания аппаратуры;
- целевое семейство микросхем (в нашем случае это **MAX10**);
- дополнительные файлы библиотек;
- другие необходимые файлы, например файлы тестбенчей.

Для создания нового проекта выберите пункт меню **File → New Project Wizard...** (рис. 1.8).

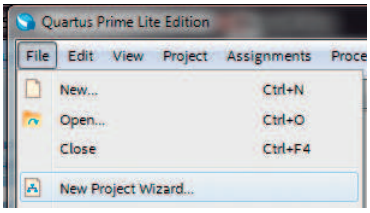


Рис. 1.8 Запуск мастера нового проекта New Project Wizard

Примечание: пункт меню **File → New...** (или кнопка 🗋) создает новый файл, а не новый проект.

Запускается помощник, который помогает создать проект и ввести всю необходимую информацию. Рассмотрим все этапы создания нового проекта по шагам.

Первый шаг при создании проекта – это определение путей для размещения проекта и его имени (рис. 1.9).

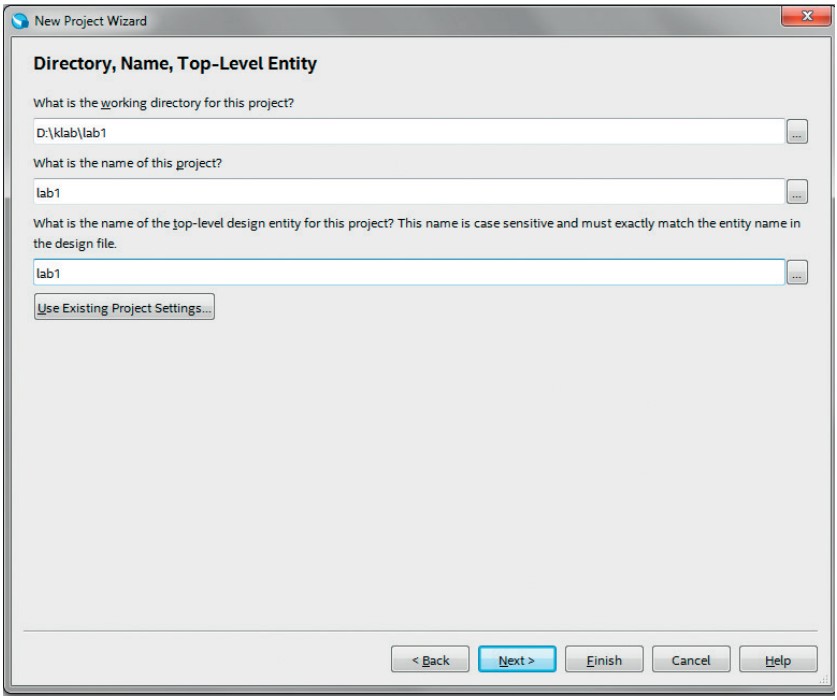


Рис. 1.9 New Project Wizard. Шаг 1

What is the working directory for this project? – Определите имя папки, в которой будет размещаться ваш проект. **Quartus Prime** создает большое количество файлов во время работы, поэтому строго рекомендуется хранить разные проекты в разных папках.

What is the name of this project? – Определите имя проекта. Часто разработчики устанавливают имя проекта таким же, как и имя файла верхнего уровня. Это является хорошим тоном.

What is the name of the top-level design entity for this project? – Определите имя файла верхнего уровня, который может быть как схемой, так и файлом на языке описания аппаратуры. Следует обращать внимание на то, что имя чувствительно к регистру.

Use Existing Project Settings... – Использовать параметры уже существующего проекта или нет.

На втором шаге (рис. 1.10) необходимо выбрать **Empty project**, если для создания проекта не используется шаблон.

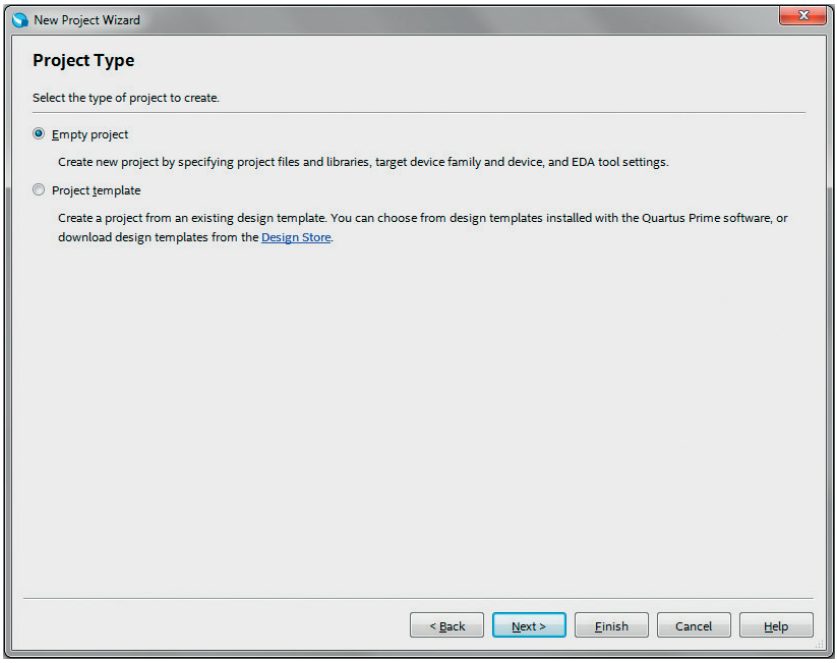


Рис. 1.10 New Project Wizard. Шаг 2

Диалог, приведенный на рис. 1.11, определяет дополнительные библиотеки или файлы пользователя, которые могут быть подключены к проекту. К проекту могут быть добавлены файлы таких типов: графические (**.bdf**, **.gdf**), файлы с описанием на языках описания аппаратуры (**VHDL**, **Verilog**, **SystemVerilog**), файлы **EDIF**-формата. При этом нет необходимости добавлять файлы, находящиеся в рабочей папке проекта. Также можно подключить к проекту библиотеки пользователя с помощью кнопки **User Libraries...**.

В этой практической работе не нужны дополнительные файлы, поэтому данный шаг можно пропустить.

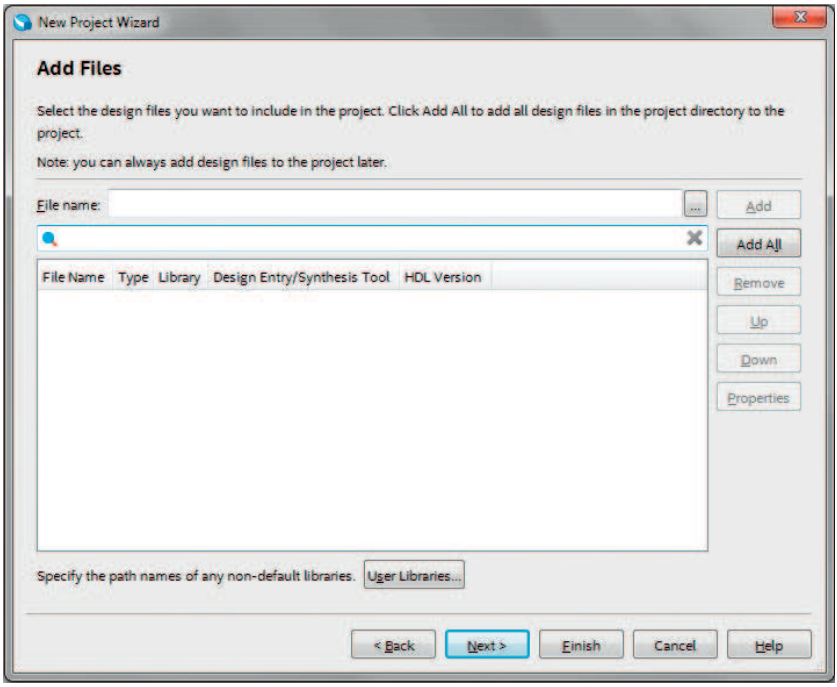


Рис. 1.11 New Project Wizard. Шаг 3

В следующем окне следует задать нужное семейство микросхем и конкретную микросхему из этого семейства (рис. 1.12). Так как работа ориентирована на плату **Terasic DE10-Lite FPGA**, нужно выбирать микросхему **10M50DAF484C7G**, как это показано на рисунке. Выбор микросхемы конкретно с такими параметрами очень важен, так как в обозначении микросхемы задаются количество вентилей, тип корпуса (**Package**), количество выводов (**Pin count**) и градация скорости (**Core speed grade**), поэтому другая микросхема из этого же семейства будет отличаться от нужной, что может привести к совершенно иным результатам компиляции. Кроме того, если выбрать не ту микросхему, это не позволит сконфигурировать микросхему на отладочной плате. Временные параметры микросхемы, используемые при компиляции проекта, определяются параметром **speed grade**. Этот параметр не описывает задержку сигнала в микросхеме, а служит для обозначения более быстрых или более медленных микросхем. Так, микросхема со **speed grade 4** быстрее микросхемы со **speed grade 5**.

Замечание: используйте фильтр при поиске (**Show in 'Availble devices' list**), чтобы упростить поиск микросхемы в списке.

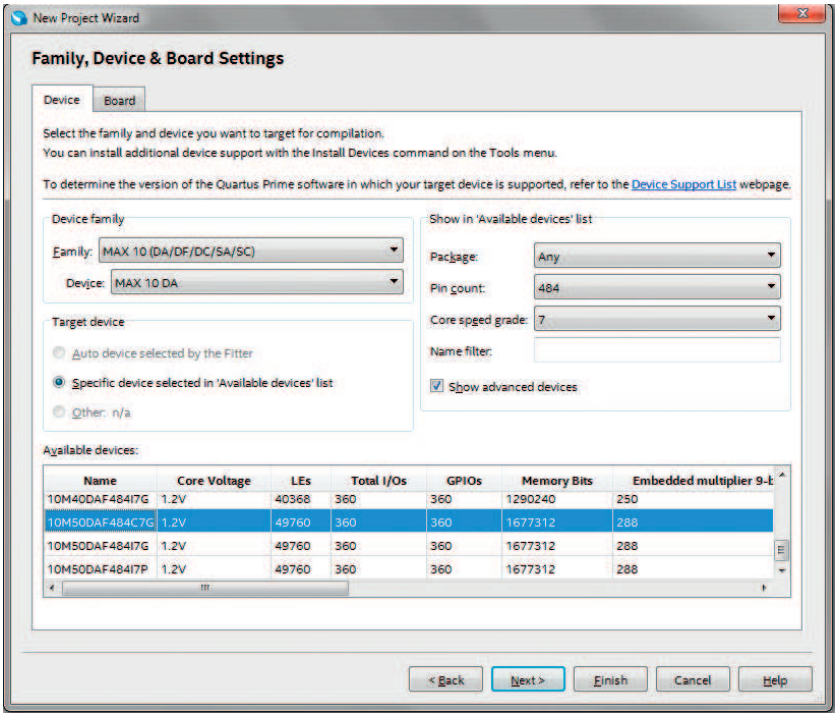


Рис. 1.12 New Project Wizard. Шаг 4

Следующий диалог определяет нестандартные средства для отладки, верификации и синтеза (рис. 1.13). Здесь ничего менять не нужно.

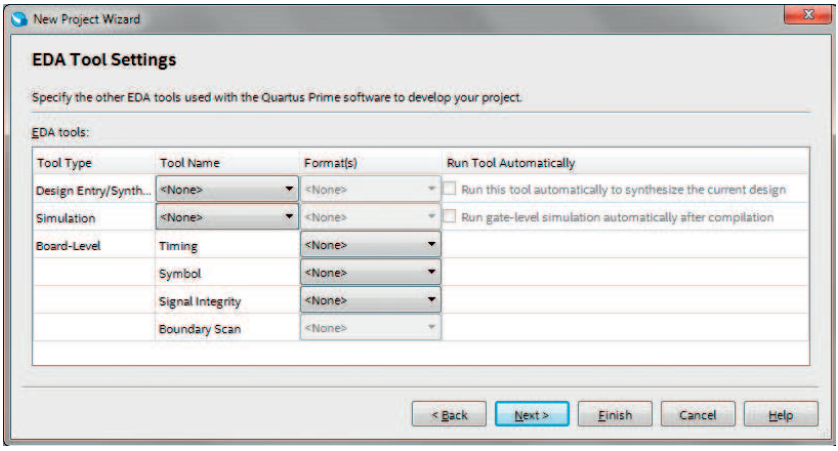


Рис. 1.13 New Project Wizard. Шаг 5

Последний диалог выводит суммарную информацию о выбранной конфигурации нового проекта (рис. 1.14). Это окно не требует никаких действий и служит для проверки корректности установок, заданных на предыдущих шагах. Если все сде-

лано правильно, то нажатие кнопки **Finish** закроет окно мастера создания нового проекта. Теперь проект готов для разработки нового устройства.

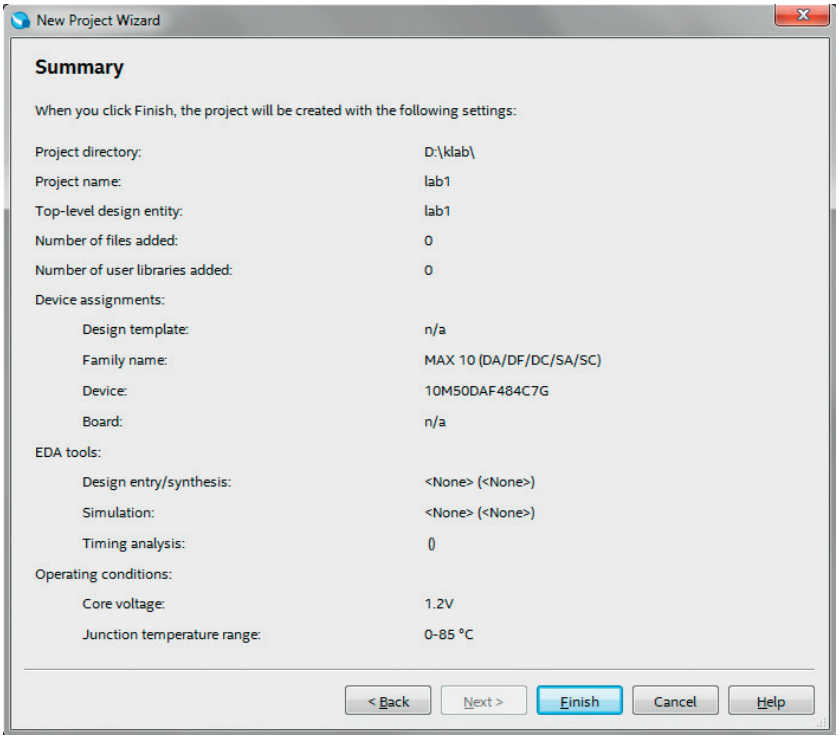


Рис. 1.14 New Project Wizard. Финальный шаг

1.2.3 Создание файла в схемотехническом редакторе

Для создания нового файла со схемой необходимо добавить файл с расширением **.bdf** к уже созданному проекту. Для этого нужно выбрать пункт меню **File → New...** или нажать на кнопку 🗋, в результате чего откроется окно, приведенное на рис. 1.15.

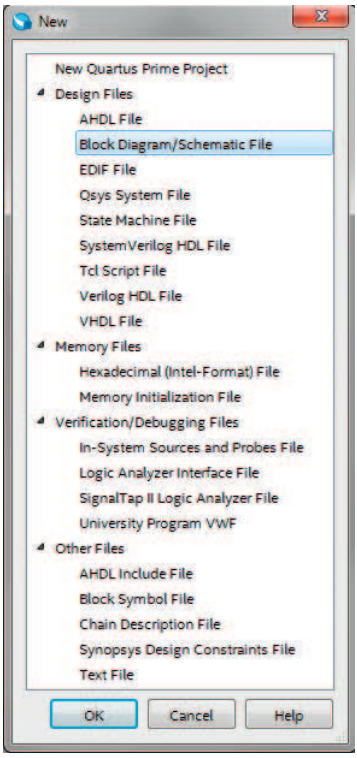


Рис. 1.15 Диалог создания нового файла

Выберите **Block Diagram/Schematic File**. Это действие откроет окно схемотехнического редактора (**Schematic editor**), показанное на рис. 1.16.

Описание кнопок панели инструментов:

- – разблокировать окно
- – инструмент выбора
- – вставить текст
- – вставить графический символ
- – вставить блок
- – добавить проводник
- – добавить линию передачи
- – растягивание соединений при перемещении связанных с ними элементов схемы
- – частичное выделение линии
- – масштаб
- – добавить шину

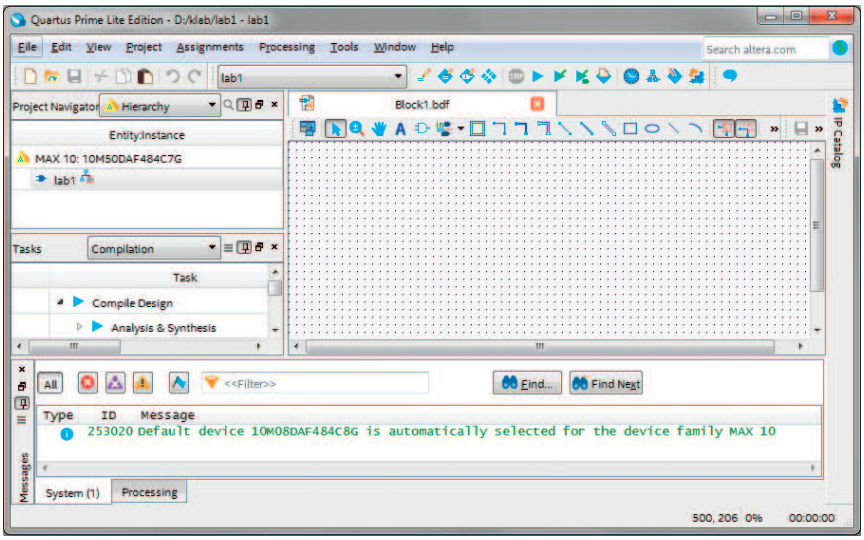


Рис. 1.16 Окно схемотехнического редактора

1.2.4 Использование схемотехнического редактора (Schematic editor)

Разрабатываемая схема будет содержать вентили **И**, **ИЛИ** и несколько инверторов. Для вставки символа необходимо нажать кнопку или сделать двойной щелчок на пустом месте в окне графического редактора для вызова соответствующего диалога. Окно выбора символа показано на рис. 1.17. Выберите символ **AND2**. Для этого нужно ввести имя символа в поле **Name**. Следует отметить, что список **Libraries** содержит не только стандартные или определяемые пользователем библиотеки, но и символы элементов, содержащихся в проекте.

По умолчанию пользователю доступно несколько библиотек[1]. Список доступных библиотек показан ниже. В данном проекте используются библиотеки **logic** и **pin**.

primitives – библиотека примитивов
buffer – буферы SOTF, WIRE, LCELL, GLOBAL и другие
logic – логические элементы И, ИЛИ, НЕ и др.
other – примитивы земли (GND), питания (VCC), константы
pin – выводы: INPUT, OUTPUT, BIDIR
storage – триггеры
other – другие примитивы
maxplus2 – примитивы 74-й серии микросхем (не рекомендуются к использованию)
megafunctions – мегафункции, настраиваемые с помощью инструмента MegaWizard

[1] Описание примитивов может быть найдено по следующей ссылке: http://quartushelp.altera.com/17.0/index.htm#hdl/prim/prim_list.htm.

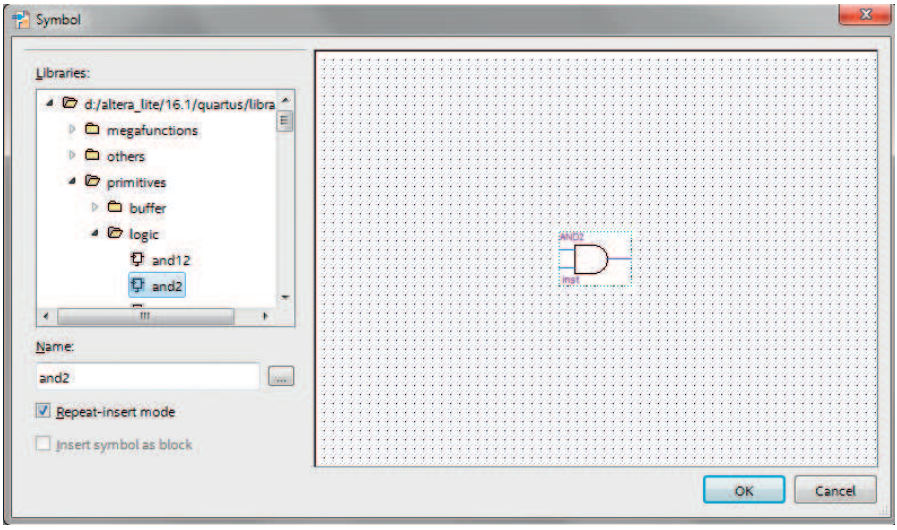


Рис. 1.17 Диалог вставки символа

Цель данной практической работы – создать такую же схему, как показано на рис. 1.18.

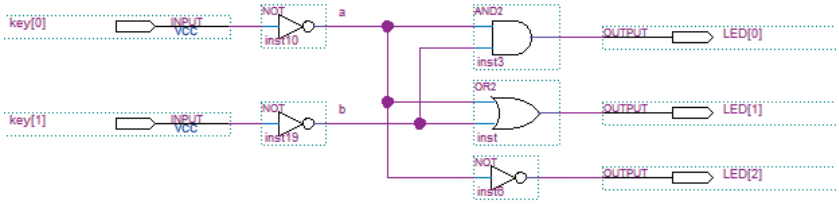


Рис. 1.18 Разрабатываемая схема, представленная в Block/Symbol Editor

Рассмотрим подробнее действия, которые необходимо выполнить для создания данной схемы.

Вставка символа

На этом шаге добавляются новые вентили. Вначале они не будут соединены – это действие будет выполнено позднее.

Для вентиля **AND2** выполняем следующие действия:

- откройте диалог **Symbol** (рис. 1.17). В поле **Name** введите **AND2** и нажмите **OK**;
- щелкните для вставки символа в нужном месте поля редактора.

Повторите описанные действия для вентилей **OR2**, **NOT**, для двух символов **Input** и трех символов **Output**.

В результате должна получиться схема, похожая на ту, которая изображена на рис. 1.19.

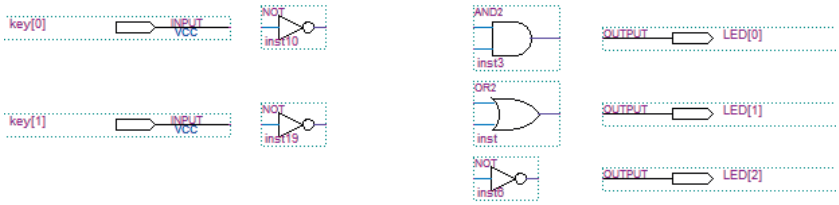


Рис. 1.19 Символы в **Block/Symbol Editor**

Соединение элементов

Соединим вентили между собой с помощью проводников. Вместо того чтобы переключаться в режим рисования проводников, достаточно просто подвести курсор к любому выводу вентиля, в результате чего его форма изменится на такую: ⌝. После этого необходимо нажать левую кнопку мыши и протянуть соединение к другому выводу. Соедините компоненты так, как показано на рис. 1.18. Для более сложных цепей можно использовать другой вариант: два проводника соединяются между собой, если они имеют одинаковые имена. Для того чтобы присвоить проводнику имя, выберите его. Затем нажмите правую кнопку мыши и в контекстном меню выберите пункт **Properties** (рис. 1.20). В диалоге **Node Properties** выберите закладку **General** и там введите имя проводника в поле **Name**.

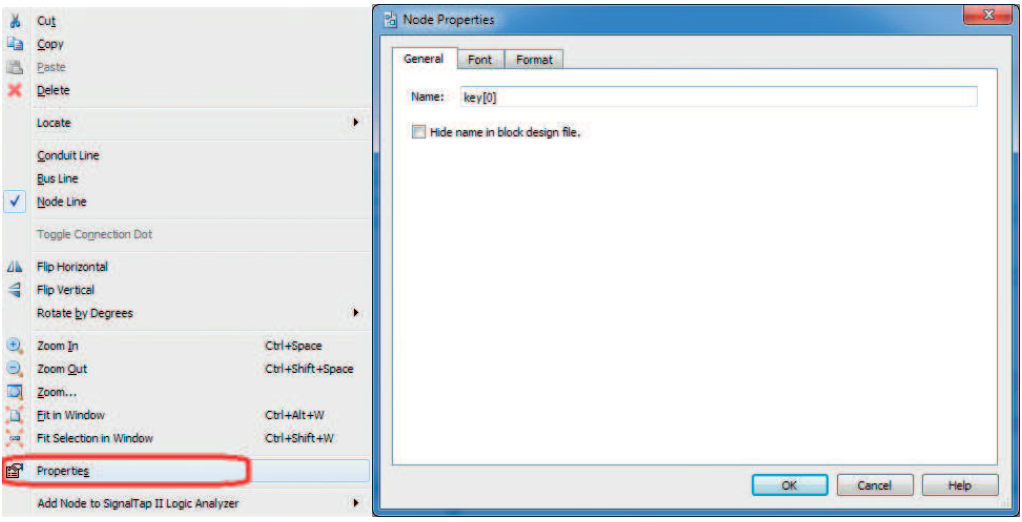


Рис. 1.20 Диалог свойств проводника **Node Properties**

Переименование входов, выходов и проводников

На плате **DE10-Lite** имеются две кнопки, которые подключены к двум входам микросхемы **MAX 10**. Также в разрабатываемом проекте будет три выхода, которые подключены к светодиодам. На рис. 1.21 и рис. 1.22 изображено, как выполнены эти подключения на отладочной плате. Отметим, что кнопки по умолчанию подтянуты к питанию, и при нажатии на входах **ПЛИС** формируется сигнал низкого уровня, поэтому эти сигналы необходимо проинвертировать внутри **ПЛИС**.

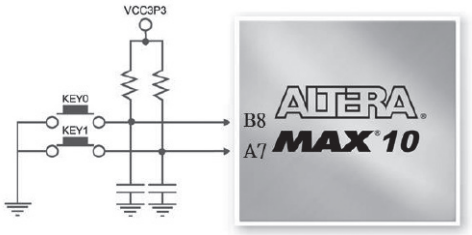


Рис. 1.21 Подключение кнопок к микросхеме **MAX10 (DE10-Lite User Manual)**

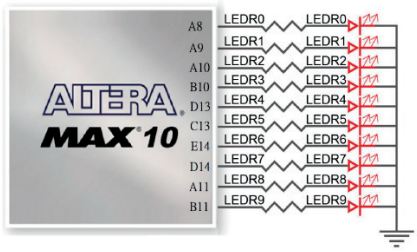


Рис. 1.22 Подключение светодиодов к микросхеме **MAX10 (DE10-Lite User Manual)**

Переименуйте входы в проекте **key[0]** и **key[1]**, как это показано на рисунке ниже.

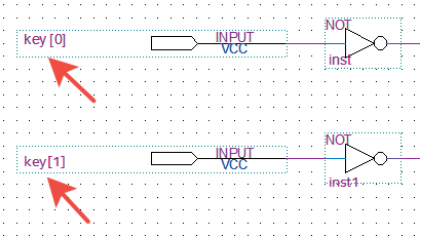


Рис. 1.23 Изменение имен входов

Для этого необходимо выполнить двойной щелчок по имени входа и ввести имя в соответствии с рис. 1.23.

Также необходимо назначить имена двум внутренним узлам схемы. Для этого щелкните на узле схемы, показанном стрелкой (рис. 1.24). Нажмите правую кнопку мыши и введите имя вывода «**a**». Также введите имя внутреннего узла «**b**», как это показано на рис. 1.18.

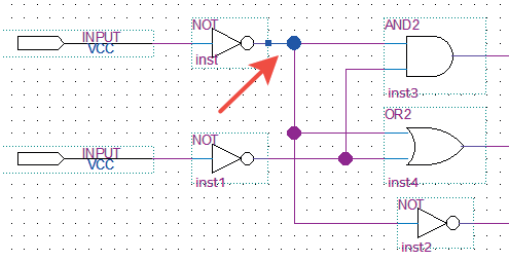


Рис. 1.24 Выбор внутреннего узла

Теперь таким же образом переименуйте выходы от **LED[0]** до **LED[2]**. Ваша первая схема в **Quartus Prime** готова, и вы научились основным действиям, необходимым для создания схемы проекта. Разработанная схема содержит всего несколько вентилей и займет доли процента от общего объема микросхемы **MAX10**. Микросхема же **10M50**, которая используется в данном примере, содержит 50 000 логических элементов, каждый из которых эквивалентен четырехвходовому вентилю. Далее будет показан процесс создания более сложной схемы, которая будет использовать все 10 светодиодов на плате.

1.2.5 Разработка более сложной схемы

Расширим разрабатываемую схему более сложными логическими функциями, как показано на рис. 1.25. В табл. 1.1 приведен полный список логических вентилей, используемых в проекте, и имена выводов, подключенных к светодиодам.

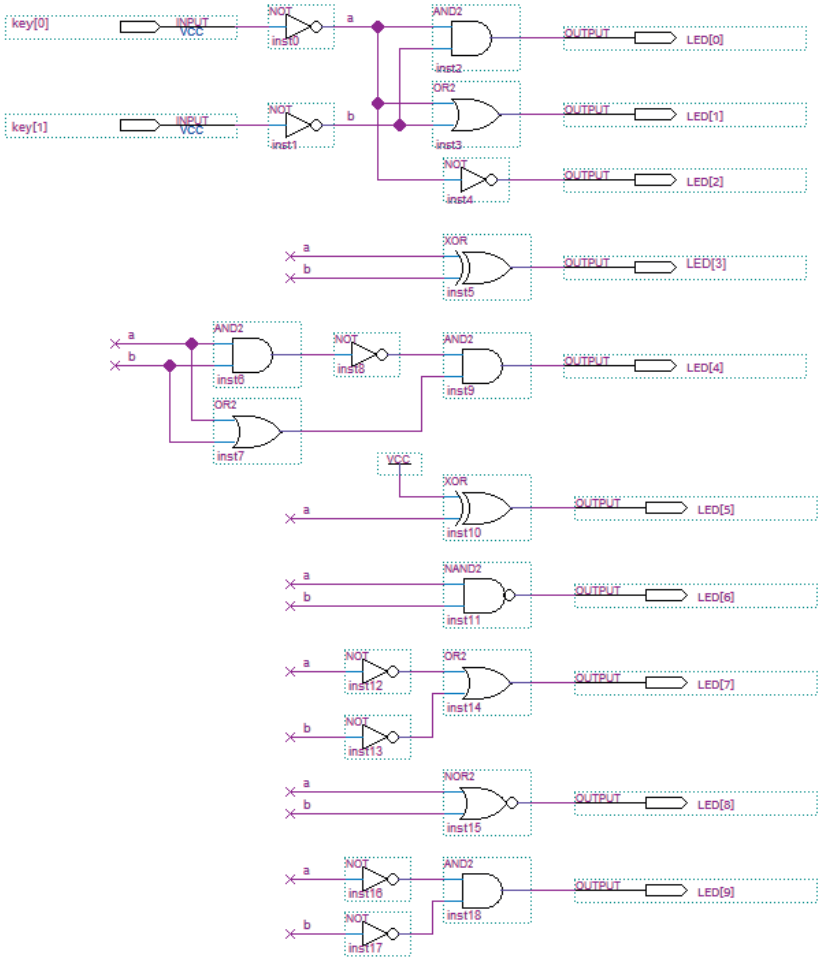
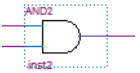
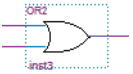
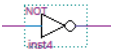
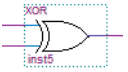
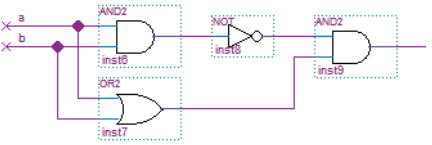
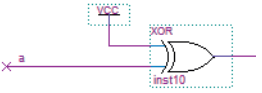
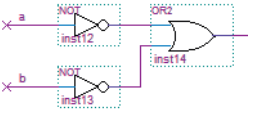
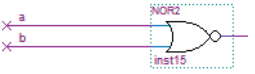
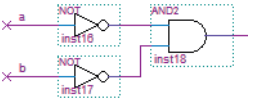


Рис. 1.25 Окончательный вариант схемы

Табл. 1.1 Логическая функция, названия выводов и символ для каждой цепи

Логическая функция	Выход	Символ или схема	Таблица истинности
И (AND)	LED[0]		
ИЛИ (OR)	LED[1]		
НЕ (NOT)	LED[2]		
Исключающее ИЛИ (XOR)	LED[3]		
Исключающее ИЛИ с использованием только И, ИЛИ, НЕ	LED[4]		
НЕ с использованием исключающего ИЛИ с 1	LED[5]		
Иллюстрация законов де Моргана	LED[6]		
	LED[7]		
	LED[8]		
	LED[9]		

Заполните последнюю колонку таблицы. Проверьте получившуюся схему на соответствие рис. 1.25. Если схема соответствует рисунку, сохраните окончательный вариант схемы в файле с именем **lab1.bdf**. Это файл верхнего уровня для разрабатываемого проекта. Данный проект содержит всего один файл. Отметим, что более сложные проекты лучше делать иерархическими, когда модули файла верхнего уровня раскрываются в файлах более низкого уровня. Соединение с выводами **ПЛИС** в таком случае выполняют в файле верхнего уровня. Файлы более низкого уровня более детально описывают логику работы устройства и могут быть откомпилированы и отлажены до того, как будут включены в большой проект.

Этап ввода проекта (**design entry**) закончен. Следующий шаг разработки проекта на **ПЛИС** – это компиляция проекта (**compile**), во время которого схема, понятная человеку, будет преобразована в формат, готовый для синтеза.

1.3 Компиляция проекта

Разрабатываемый проект должен быть откомпилирован и синтезирован. Для этого запустите компилятор (**Compiler**), нажав кнопку ▶ или выбрав пункт меню **Processing → Start Compilation**.

Если компиляция прошла успешно, появится окно, приведенное на рис. 1.26 (в дальнейшем, когда вы будете компилировать файлы на языке описания аппаратуры, могут возникать ошибки, которые будут показаны в окне сообщений **Messages**). Во время анализа и синтеза (**Analysis & Synthesis**) компилятор проверяет проект на наличие ошибок, синтезирует логические уравнения для всего проекта и затем оптимизирует их для выполнения заданных опций синтеза. Опции синтеза могут задавать уменьшение размера схемы или увеличение скорости работы схемы. Исходя из этих требований, будет изменяться размер схемы и ее быстродействие. В данном проекте не было задано никаких параметров для синтеза, поэтому в результате компиляции выводится сообщение, что проект содержит не полностью определенные параметры (**Design is not fully constrained**). На данном этапе это пока не важно, поэтому данное сообщение можно проигнорировать.

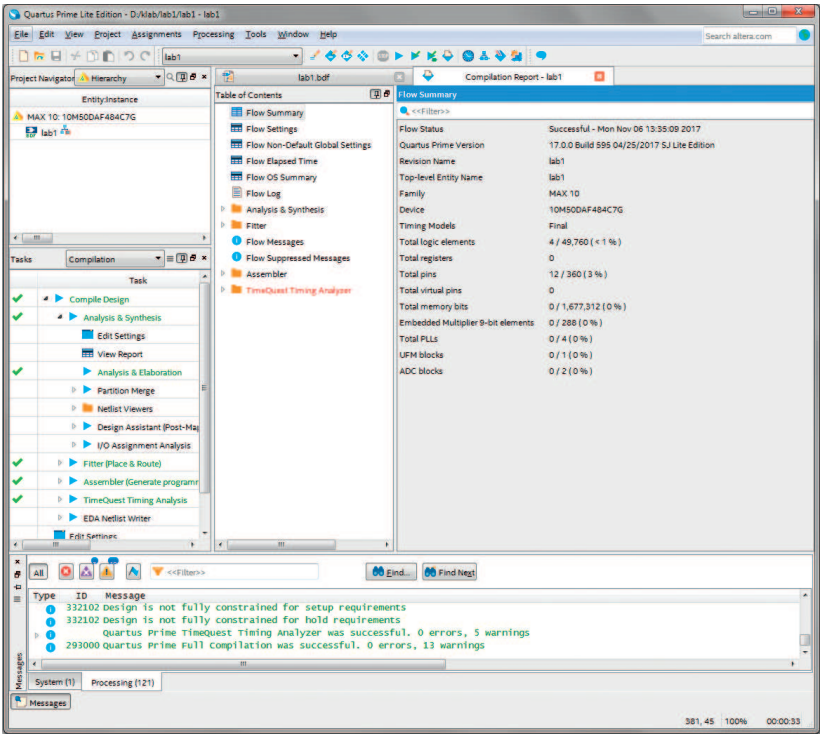


Рис. 1.26 Отчет компилятора (Compilation report)

1.3.1 Использование RTL Viewer

Графическое представление проекта на **ПЛИС** может быть получено с помощью модуля **RTL Viewer** (рис. 1.27). Для того чтобы открыть **RTL Viewer**, выберите пункт меню **Tools → Netlist Viewers → RTL Viewer** (обязательно должна быть предварительно выполнена компиляция путем вызова пунктов меню **Analysis & Elaboration** или **Analysis & Synthesis**).

В окне **RTL Viewer** отображается логическая реализация проекта в графическом виде. Схема представлена в немного оптимизированном виде: два инвертора, управляющих светодиодом **LED[2]**, объединены в один общий буфер с именем **inst4**. Остальные инверторы преобразованы в инвертирующие входы вентилей, например **inst11** (рис. 1.27).

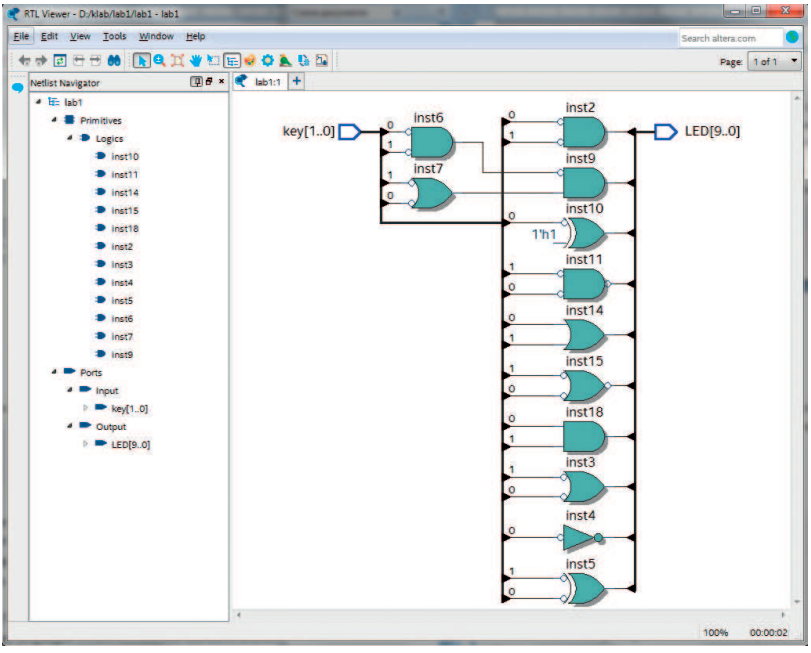


Рис. 1.27 RTL Viewer

Возможности **RTL Viewer** позволяют увидеть схему, порождаемую компилятором из HDL-описания. Эта возможность будет использоваться в данном практическом курсе очень часто. Кроме того, в **RTL Viewer** можно увидеть, как используются другие ресурсы **ПЛИС** – логические блоки, сумматоры, умножители, триггеры, элементы ввода-вывода и т. д. Еще раз отметим, что логика, показанная в **RTL Viewer**, – это функциональный эквивалент схемы, разработанной в графическом редакторе или описанной на языке описания аппаратуры.

1.4 Назначение выводов. Компиляция проекта

Pin Planner (рис. 1.28) – это интерактивное средство, которое используется для соединения портов проекта с выводами микросхемы ПЛИС. Мы можем его за-

пустить с помощью пункта меню **Assignments → Pin Planner** или нажатием кнопки .

Основная работа в **Pin Planner** происходит в таблице **All pins**, которая содержит следующие столбцы:

Node Name – имя вывода в схеме или в HDL-описании;

Direction – направление передачи данных: вход (**input**), выход (**output**) или двунаправленный (**bidirectional**);

Location – расположение вывода на корпусе **ПЛИС** (номер вывода);

I/O Bank – номер банка выводов. Все выводы в **ПЛИС** сгруппированы в несколько банков;

Vref Group – опорное напряжение группы выводов;

Fitter Location – номер вывода после предыдущего запуска компоновщика (**Fitter**);

I/O Standard – стандарт ввода-вывода (3,3 В ТТЛ, КМОП, наличие триггера Шмидта и т. д.);

Reserved – зарезервировано: показывает, что данный порт ввода-вывода не может быть использован в данном чипе;

Current Strength – предельное значение тока вывода, зависящее от стандарта ввода-вывода.

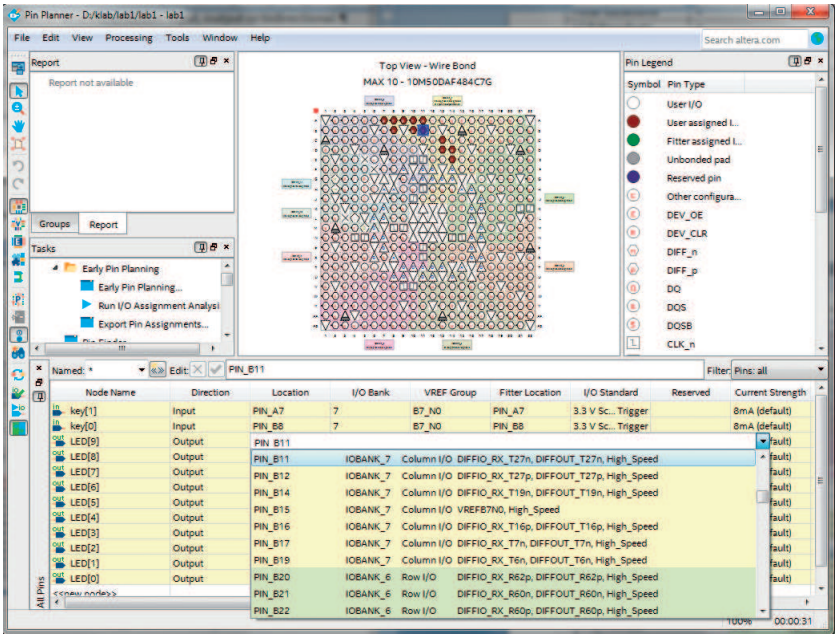


Рис. 1.28 Окно модуля Pin Planner

Порядок назначения выводов в **Pin Planner**:

- откомпилируйте проект, чтобы заполнить список портов модуля верхнего уровня;

- запустите **Pin Planner** с помощью пункта меню **Assignments → Pin Planner** или кнопки ;
- найдите нужный порт в окне **All Pins**;
- дважды щелкните в столбце **Location** в таблице параметров (рис. 1.28). Выберите нужный вывод из списка;
- дважды щелкните в столбце **I/O Standard** в таблице параметров (рис. 1.28). Выберите нужный стандарт из списка;
- перекомпилируйте проект после изменения настроек всех портов.

Расположение выводов и стандарты ввода-вывода зависят от подключений микросхемы на печатной плате. Информация по подключениям на отладочной плате **DE10-Lite** показана в табл. 1.2 и 1.3.

Signal Name	FPGA Pin No.	Description	I/O Standard
KEY0	PIN_B8	Push-button[0]	3.3 V SCHMITT TRIGGER"
KEY1	PIN_A7	Push-button[1]	3.3 V SCHMITT TRIGGER"

Табл. 1.2 Назначение выводов для кнопок (в соответствии с DE10-Lite User Manual)

Signal Name	FPGA Pin No.	Description	I/O Standard
LEDR0	PIN_A8	LED [0]	3.3-V LVTTL
LEDR1	PIN_A9	LED [1]	3.3-V LVTTL
LEDR2	PIN_A10	LED [2]	3.3-V LVTTL
LEDR3	PIN_B10	LED [3]	3.3-V LVTTL
LEDR4	PIN_D13	LED [4]	3.3-V LVTTL
LEDR5	PIN_C13	LED [5]	3.3-V LVTTL
LEDR6	PIN_E14	LED [6]	3.3-V LVTTL
LEDR7	PIN_D14	LED [7]	3.3-V LVTTL
LEDR8	PIN_A11	LED [8]	3.3-V LVTTL
LEDR9	PIN_B11	LED [9]	3.3-V LVTTL

Табл. 1.3 Назначение выводов для светодиодов (в соответствии с DE10-Lite User Manual)

1.5 Конфигурирование ПЛИС

Для загрузки конфигурационного файла, соответствующего нашему проекту, в микросхему **ПЛИС** необходимо выполнить следующие действия:

1. Присоединить отладочную плату к компьютеру с помощью USB-кабеля (рис. 1.29).
2. Запустить **Programmer** (рис. 1.30) с помощью пункта меню **Assignments → Programmer** или нажать кнопку .
3. Если вы в первый раз используете режим программирования, то необходимо будет выбрать загрузочный кабель. Для этого щелкните кнопку **Hard-**

ware Setup (рис. 1.29) и выберите загрузочный кабель **USB Blaster** из списка **Currently selected hardware** (рис. 1.31).

4. Выбрать файл конфигурации **ПЛИС** (**lab1.sof**), который находится в основной таблице данного окна. Если файла там нет – добавить его вручную с помощью кнопки **Add File**.
5. Запустить программирование микросхемы с помощью кнопки **Start**.
6. Если получился такой же результат, как на рис. 1.32, то конфигурирование **ПЛИС** прошло успешно.

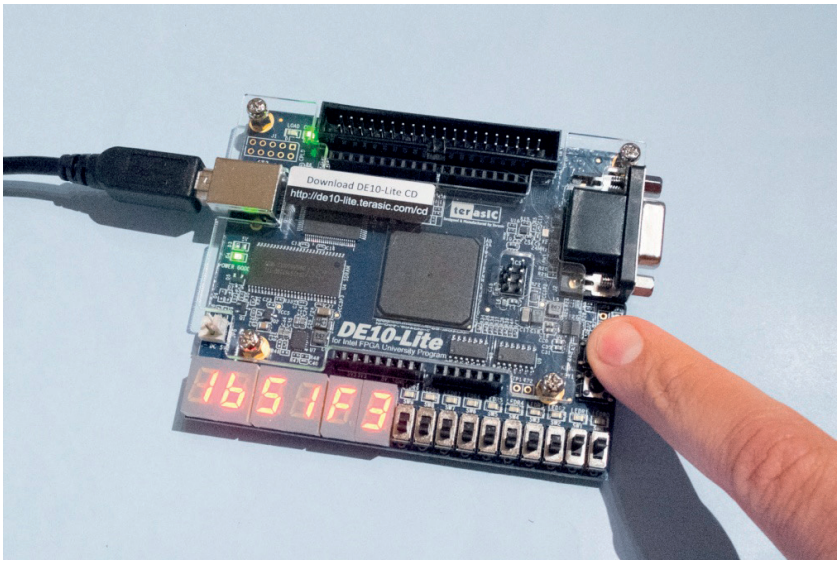


Рис. 1.29 Плата DE10-Lite, подключенная к ПК

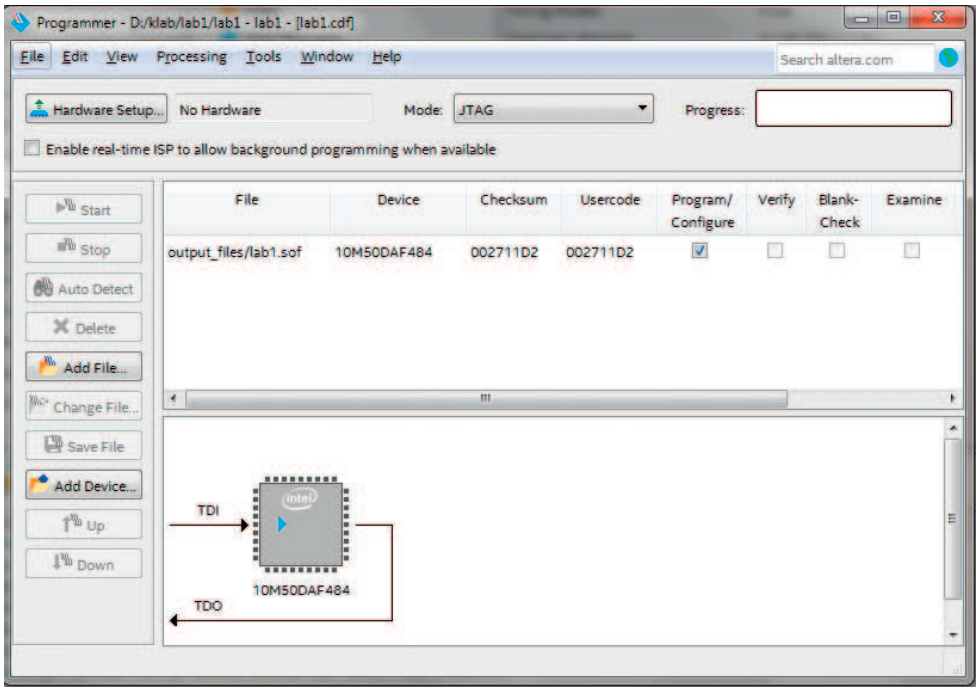


Рис. 1.30 Окно программатора

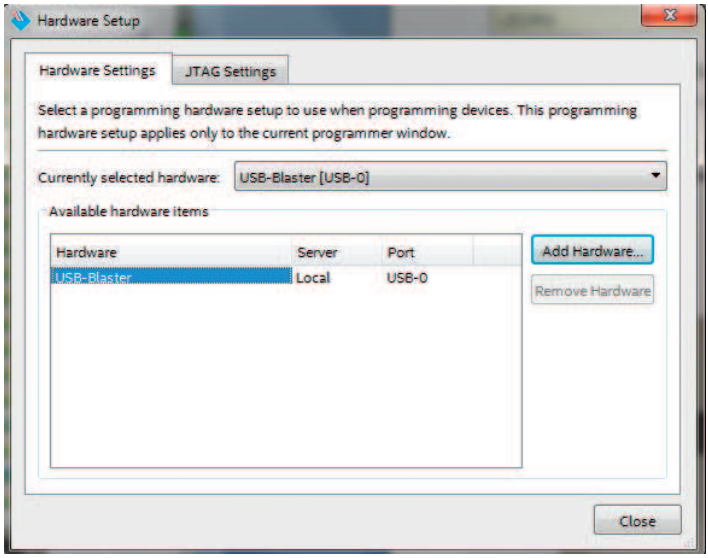


Рис. 1.31 Выбор загрузочного кабеля

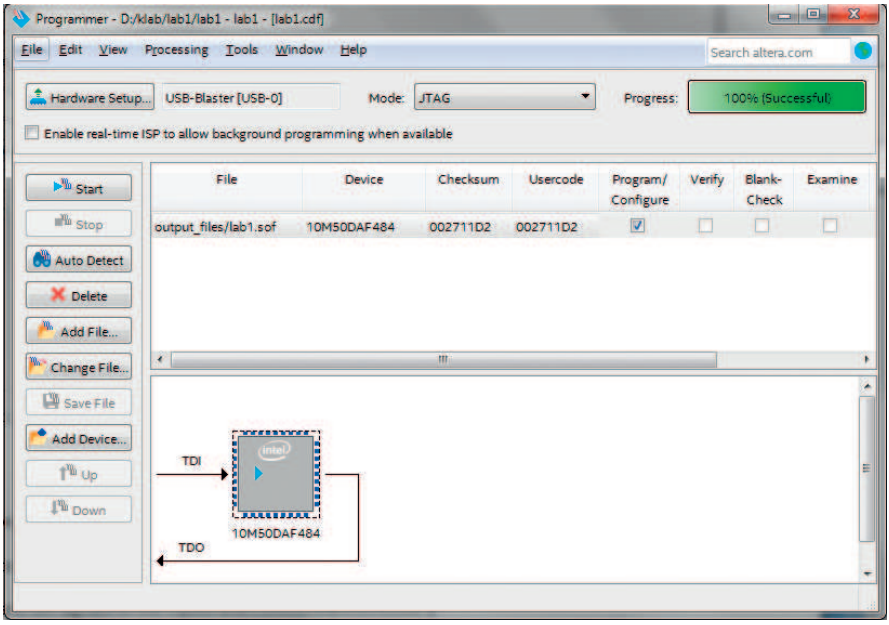


Рис. 1.32 Успешная конфигурация ПЛИС

Проверка работы проекта на отладочной плате

Теперь вы имеете отладочную плату с микросхемой **ПЛИС**, в которую загружен проект. Необходимо убедиться в том, что проект действительно работает правильно. Для этого требуется выполнить следующие действия.

1. Заполнить последнюю колонку табл. 1.1 соответствующими таблицами истинности.
2. Используя кнопки, перебрать комбинации значений входных сигналов и визуально проверить результаты работы схемы на светодиодах.
3. Проверить таблицы истинности из табл. 1.1.

1.6 Разработка схемы с использованием языка описания аппаратуры. Симуляция

Постоянное усложнение электронных устройств требует от разработчиков увеличения скорости разработки. Использование языков описания аппаратуры (**Hardware Description Language**, **HDL**) позволяет обеспечить описание сложных систем на высоком уровне абстракции, поддержку библиотек проектов, кросс-платформенность кода, увеличение скорости разработки. В настоящее время наиболее часто используемыми языками описания аппаратуры являются **VHDL**, **Verilog HDL** и **SystemVerilog**. Эти языки считаются стандартными и поддерживаются практически всеми средствами моделирования и синтеза устройств на **ПЛИС**. Кроме упомянутых выше языков, существуют и менее распространенные языки, такие как **AHDL**, **Abel** и **SystemC**.

Программное обеспечение для моделирования позволяет упростить тестирование HDL-кода. Эти пакеты позволяют разработчику анализировать прохождение сигналов внутри модулей HDL и помогают находить ошибки в проекте, не загружая конфигурацию в микросхему **ПЛИС**. В этой главе будет освещена работа со свободной версией мощного коммерческого симулятора **ModelSim** и с его аналогом с открытым исходным кодом – **Icarus Verilog** в связке с **GTKWave**. Также можно использовать бесплатную версию **Active-HDL Student Edition** от компании **ALDEC** для моделирования и верификации, однако в данной книге описание работы с этим программным пакетом не приводится.

1.6.1 Загрузка и установка ModelSim Starter Edition

Пакет **ModelSim** может быть загружен и установлен во время установки пакета **Quartus Prime Lite Edition**, как это было показано в разделе 1.2.1. Или же он может быть загружен с сайта компании Intel FPGA[1] и установлен независимо от Quartus Prime, как это показано ниже. Основные этапы установки приведены на рис. 1.33, 1.34 и 1.35.

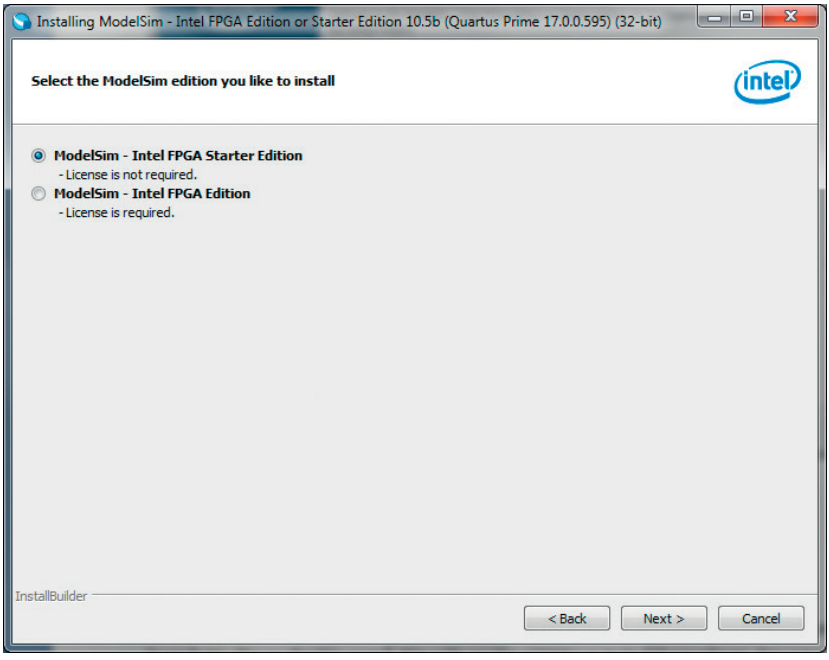


Рис. 1.33 Установка ModelSim. Шаг 1

[1] http://dl.altera.com/?edition=lite.

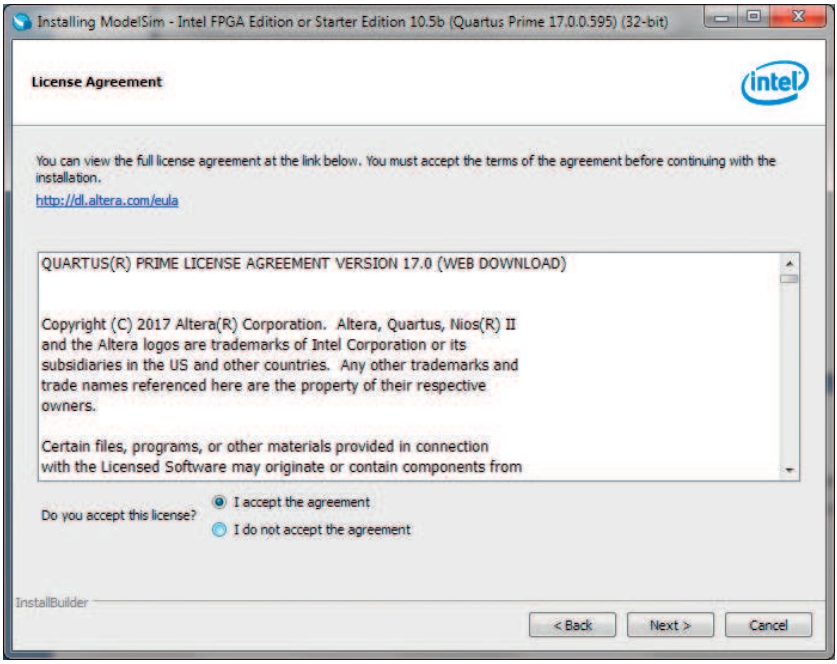


Рис. 1.34 Установка ModelSim. Шаг 2

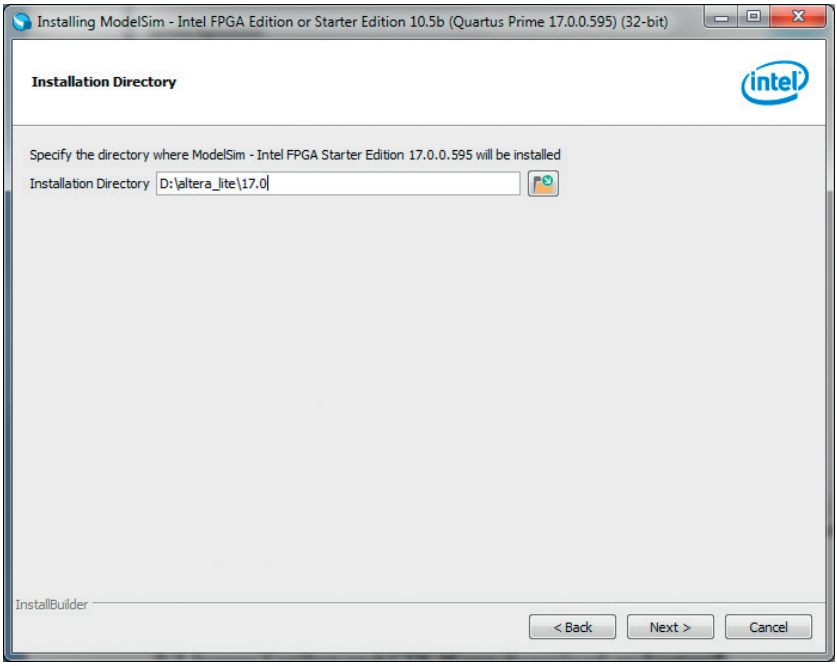


Рис. 1.35 Установка ModelSim. Шаг 3

Когда установка закончена, рекомендуется добавить папку с пакетом **ModelSim modelsim_ase\win32aloem** в системные пути (**PATH**), для того чтобы работали скрипты, приведенные в разделе 1.8.1.

1.6.2 Загрузка и установка пакетов Icarus Verilog и GTK Wave

Icarus Verilog – это свободно распространяемый симулятор и средство синтеза языка **Verilog HDL**. **Icarus Verilog** можно загрузить со страницы **Pablo Bleyer Kocik**[1]. Для построения временных диаграмм также будет использован пакет **GTKWave**, который можно загрузить с официальной страницы[2]. Еще можно использовать установщик для Windows из папки **pkg** для последовательной установки **Icarus Verilog** и **GTKWave**. Основные этапы установки показаны на рис. 1.36 и рис. 1.36.

После того как установка будет окончена, рекомендуется добавить папки **iVerilog\bin** и **iVerilog\GTKWave\bin** в системные пути **PATH**, чтобы запускать скрипты, приведенные в разделе 1.8.2.

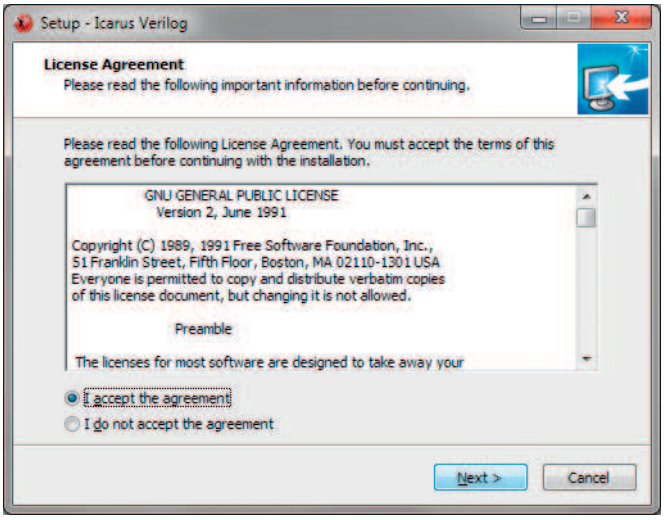


Рис. 1.36 Установка Icarus Verilog и GTKWave. Шаг 1

[1] http://bleyer.org/icarus/.
[2] http://gtkwave.sourceforge.net/.

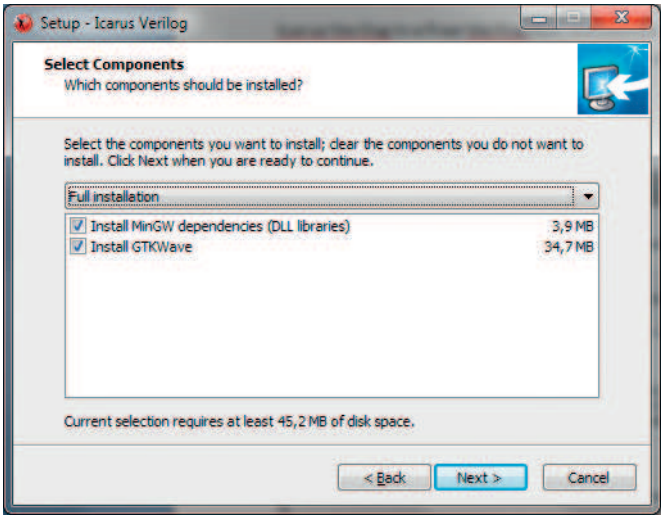


Рис. 1.37 Установка Icarus Verilog и GTKWave. Шаг 2

1.7 HDL-модуль и его описание

Основным элементом **Verilog HDL** является модуль (**module**). Каждый модуль описывает какое-то устройство или же его часть. Исходный код модуля, идентичный схеме, созданной в разделе 1.2.5, показан в листинге 1.1.

Название модуля записывается после слова «**module**». Имя модуля должно совпадать с именем файла. Любое цифровое устройство содержит внешние контакты. Аналогом этого в **Verilog HDL** являются входы (**inputs**) и выходы (**outputs**). Для нашего устройства это «**KEY**» и «**LED**».

```
module lab1
(
   input [1:0] KEY,   // KEYs
   output [9:0] LED   // LEDs
);
   wire a = ~ KEY [0];
   wire b = ~ KEY [1];

   // Basic gates AND, OR and NOT
   assign LED [0] = a & b;
   assign LED [1] = a | b;
   assign LED [2] = ~ a;

   // XOR gates (useful for adders, comparisons,
   // parity and control sum calculation)
   assign LED [3] = a ^ b;

   // Building XOR only using AND, OR and NOT
   assign LED [4] = (a | b) & ~ (a & b);
```

```
    // Building NOT by XORing with 1
    assign LED [5] = a ^ 1'b1;

    // De Morgan law illustration
    assign LED [6] = ~ ( a & b );
    assign LED [7] = ~ a | ~ b;
    assign LED [8] = ~ ( a |  b );
    assign LED [9] = ~ a & ~ b;

endmodule
```

Листинг 1.1 Схема с логическими элементами

Входы и выходы в устройстве могут быть как одноразрядными, так и многоразрядными. Пример одноразрядного входа:

```
input rst_n
```

Для многоразрядного входа или выхода запись будет выглядеть так:

```
input [1:0] KEY,     // KEYs
output [9:0] LED   // LEDs
```

В квадратных скобках указывают количество бит – два для входа **KEY** и десять для выхода **LED**. Это значит, что два сигнала от кнопок будут подключены к входам модуля, а десять выходных сигналов будут подключены к светодиодам.

Внутри модуля входной сигнал управляет проводником (wire). В нашем примере это два сигнала: «**a**» и «**b**». Эти сигналы поступают после инверсии входных сигналов с кнопок. Сигнал «**a**» – как инверсия **key [0]**, сигнал «**b**» – как инверсия **key [1]**:

```
wire a = ~ KEY [0];
wire b = ~ KEY [1];
```

Логические функции, используемые в языке **Verilog**, приведены ниже:

! – НЕ	| – ИЛИ	^ – исключающее ИЛИ
~ – инверсия	~& – И-НЕ	~^ или ^~ – исключающее ИЛИ-НЕ
& – И	~| – ИЛИ-НЕ	

Опишем схему с использованием логических уравнений. Например, описание логического **И** выглядит так:

```
assign LED [0] = a & b;
```

Это означает, что выход «**led [0]**» будет управляться логическим **И** от «a» и «b».

Оператор «**assign**» задает непрерывное присваивание значения сигналу. «Непрерывное» означает, что любое изменение одного из операндов в правой части выражения сразу вызывает изменение значения выходного сигнала. Этот вид присваивания всегда синтезируется в комбинационную схему.

Таким образом осуществляется описание отдельных элементов и всей схемы. Обратите внимание на то, что запись кода на языке **Verilog HDL** занимает гораздо меньше времени, чем построение схемы в редакторе схем, поэтому при проекти-

ровании современных устройств схемы не строят в схемотехническом редакторе, а описывают на HDL.

Замечание: особенностью языков описания аппаратуры является параллелизм операторов. Все операторы в программе выполняются параллельно – точно так же, как и работают различные части схемы. Если изменить положение строк внутри кода, алгоритм работы схемы не изменится. Это одно из ключевых отличий от обычных языков программирования, таких как C или Java, в которых порядок следования команд очень важен.

1.8 Тестбенч и его описание

Тестбенч (**Test bench**) – это тестирующее оборудование или отладочный модуль, используемый для проверки кода в HDL. Тестбенч генерирует набор сигналов для нашего устройства, а затем принимает сигналы от устройства для анализа. Рассмотрим код, описывающий тестбенч для созданного проекта (листинг 1.2).

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns / 1ps
module testbench;
   // input and output test signals
   reg [1:0] key;
   wire [9:0] led;
   // creating the instance of the module we want to test
   // lab1 - module name
   // dut - instance name (‘dut’ means ‘device under test’)
   lab1 dut ( key, led );
   // do at the beginning of the simulation
   initial
      begin
         key = 2’b00;       // set test signals value
         #10;          // pause
         key = 2’b01;       // set test signals value
         #10;          // pause
         key = 2’b10;       // set test signals value
         #10;          // pause
         key = 2’b11;       // set test signals value
         #10;          // pause
      end
   // do at the beginning of the simulation
   // print signal values on every change
   initial
      $monitor("key=%b led=%b", key, led);
   // do at the beginning of the simulation
   initial
```

```
      $dumpvars;   //iVerilog dump init
endmodule
```

Листинг 1.2 Тестбенч для модуля с простейшей логикой

Тестбенч начинается с задания масштаба времени и точности вычисления интервалов:

```
`timescale 1ns / 1ps
```

Далее следует декларация модуля. Если бы это было описанием какого-либо устройства, то далее было бы описание портов модуля, но тестбенч не содержит внешних портов, а только внутренние сигналы:

```
reg [1:0] key;
wire [9:0] led;
```

Описание «проводника» «wire» уже ранее встречалось. Проводник не сохраняет информацию, в отличие от «**reg**» – элемента хранения данных, который сохраняет значение сигнала до тех пор, пока ему не будет присвоено новое значение.

Затем объявляется экземпляр модуля, который будет тестироваться. Тестируемая схема называется тестируемым устройством (**DUT**, **device under test**) или тестируемым модулем (**UUT**, **unit under test**). В нашем тестовом окружении DUT описывается следующей строкой:

```
lab1 dut ( key, led );
```

Следующие строки запускают initial-блок:

```
initial
   begin
   ...
end
```

Каждый **initial**-блок запускается только один раз, в начале моделирования. Если же описано несколько начальных блоков, то они будут работать параллельно.

В данной главе напишем простейший тестбенч. Этот тестбенч будет подавать сигналы на входы тестируемого модуля через определенные промежутки времени. В **DUT** есть два входа, на которые будут подаваться последовательные сигналы с интервалом в 10 единиц модельного времени:

```
key = 2'b00;  // set test signals value
#10;          // pause
```

Формат чисел, подаваемых на входы, следующий: **2'b00**. Это означает, что 2-битный вектор со всеми битами, равными **0. 2'b11**, соответствует 2-битному вектору со значением **11** в бинарном виде. Синтаксис «#» используется для указания задержки. Это означает, что сначала сигнал «**key**» будет установлен в значение «**00**». Затем (после задержки в **10** единиц модельного времени) будет выполнена следующая по порядку команда.

Следующая строка содержит команду:

```
initial $monitor("key=%b led=%b", key, led);
```

Она дает команду симулятору выводить все изменения сигналов «**led**» и «**key**» на консоль.

В конце листинга содержится следующая строка:

```
$dumpvars;      //iVerilog dump init
```

Эта команда используется для записи выбранных выше переменных в файл дампа во время моделирования. Если функция не имеет аргументов, то записываются значения всех переменных.

1.8.1 Симуляция с использованием ModelSim

После выполнения предыдущих шагов имеются две программы на языке **Verilog HDL – lab1.v** и **testbench.v**. Проведем симуляцию проекта. Для этого необходимо выполнить следующие действия:

- открыть командную строку Windows:
  `cmd.exe`
- сменить текущую папку на папку «simulation»:
  `cd D:\klab\lab1_hdl\simulation`
- запустить скрипт симуляции в ModelSim (листинг 1.3):

  `>01_simulate_with_modelsim.bat`

После этого выведется окно ModelSim с результатами моделирования (рис. 1.38).

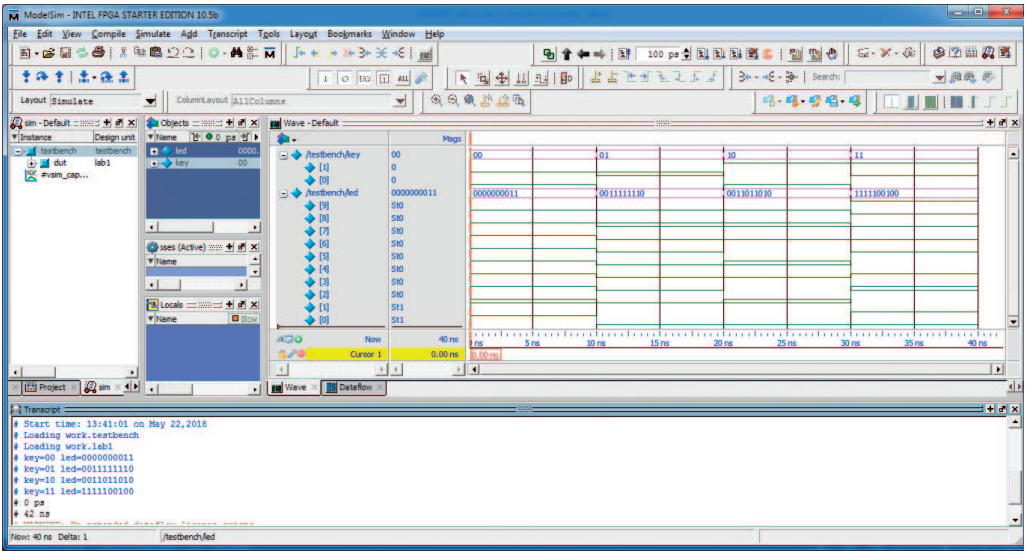


Рис. 1.38 Результаты симуляции в Modelsim

Скрипт листинга 1.3 написан на языке **TCL** – скриптовом языке, используемом в ModelSim в качестве внутреннего языка для управления работой программы. Все команды данного скрипта соответствуют действиям в графическом интерфейсе ModelSim. Использование скриптов позволяет значительно ускорить работу при симуляции, избавиться от выполнения однотипных действий и таким образом сэкономить значительное количество времени.

```
rem recreate a temp folder for all the simulation files
rd /s /q sim
md sim
cd sim

rem start the simulation
vsim -do ../modelsim_script.tcl
rem return to the parent folder
cd ..
```

Листинг 1.3 Скрипт для запуска моделирования в ModelSim

```
# create modelsim working library
vlib work

# compile all the Verilog sources
vlog ../testbench.v ../../lab1.v

# open the testbench module for simulation
vsim work.testbench

# add all testbench signals to time diagram
add wave sim:/testbench/*

# run the simulation
run -all

# expand the signals time diagram
wave zoom full
```

Листинг 1.4 Скрипт для управления моделированием в ModelSim

1.8.2 Симуляция с использованием Icarus Verilog и GTK Wave

Для получения результатов симуляции с использованием **Icarus Verilog** (пакет для симуляции) и **GTKWave** (среда визуализации временных диаграмм) необходимо выполнить следующие действия:

- открыть командную строку Windows:

  ```
  cmd.exe
  ```
- сменить текущую папку на папку «**simulation**»:

  ```
  cd D:\klab\lab1_hdl\simulation
  ```
- запустить скрипт симуляции в **Icarus Verilog** (листинг 1.5):

  ```
  >02_simulate_with_icarus.bat
  ```

После этих действий будет запущен **GTKWave**, в котором будут отображены результаты моделирования (рис. 1.39). Для просмотра сигналов в виде временных диаграмм необходимо выбрать их в левом нижнем углу окна и нажать кнопку **Append**.

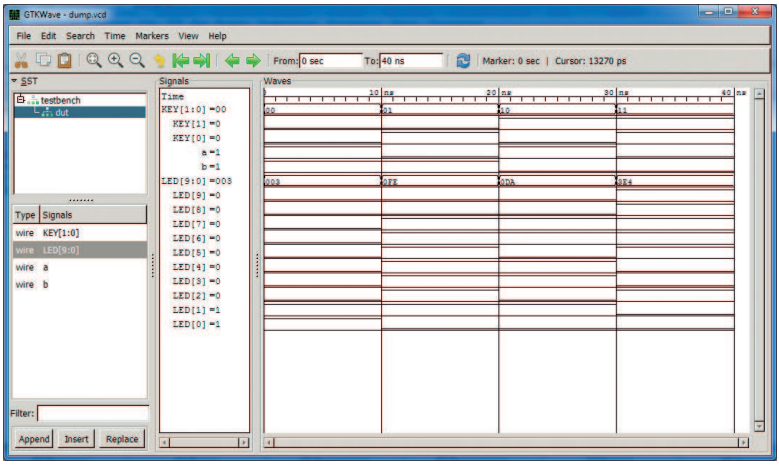


Рис. 1.39 Результаты симуляции в Icarus Verilog, открытые в GTKWave

Пакет **Icarus Verilog** – это консольная программа. Для ее запуска с нужными параметрами необходимо использовать скрипт, приведенный в листинге 1.5, который служит для получения результатов симуляции и просмотра их в **GTKWave**:

```
rem recreate a temp folder for all the simulation files
rd /s /q sim
md sim
cd sim
rem compile Verilog files for simulation
iVerilog -s testbench ..\testbench.v ..\..\lab1.v
rem run the simulation
vvp -la.lst -n a.out -vcd
rem show the simulation results in GTKWave
GTKWave dump.vcd

rem return to the parent folder
cd ..
```

Листинг 1.5 Скрипт для запуска моделирования в Icarus Verilog

1.9 Создание описания схемы на языке Verilog HDL. Синтез схемы

В этом разделе приведена последовательность действий, необходимых для синтеза конфигурации **ПЛИС** из файлов HDL и загрузки ее на плату **DE10-Lite**. Для этого необходимо:

- открыть командную строку Windows:
  `cmd.exe`

- сменить текущую папку на папку «**synthesis**»:

```
cd D:\klab\lab1_hdl\synthesis
```

Данная папка содержит следующие файлы:

```
lab1.qpf
lab1.qsf
make_project.bat
```

Пакетный файл Windows **make_project.bat** (листинг 1.6) создает копию файлов проекта во временной папке с именем «**project**». Создавать отдельную папку необходимо потому, что **Quartus Prime** создает большое количество временных файлов в рабочей папке. Создание временной папки позволяет отделить файлы с описанием проекта от временных файлов проекта.

```
rd /s /q project
mkdir project
copy *.qpf project
copy *.qsf project
```

Листинг 1.6 Пакетный файл для создания проекта в Quartus

- Запустить **make_project.bat** (листинг 1.6):

```
> make_project.bat
```

- запустить **Quartus Prime** (рис. 1.40):

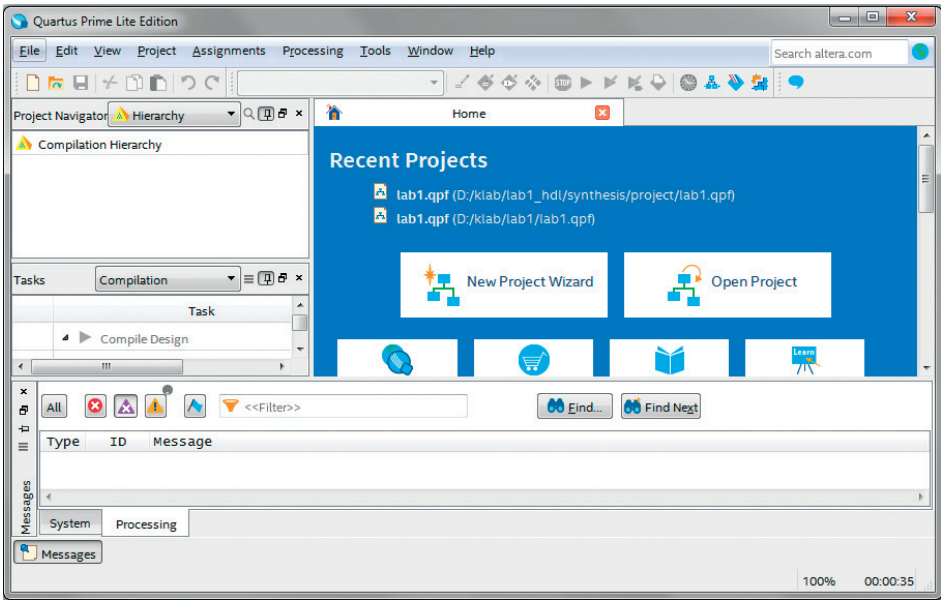


Рис. 1.40 Главное окно Quartus Prime Lite Edition

- нажать пункт меню **File → Open Project** и выбрать файл проекта **.qpf**: synthesis\project\lab1.qpf (рис. 1.41);
- откомпилировать проект так, как это было описано ранее в разделе 1.3;

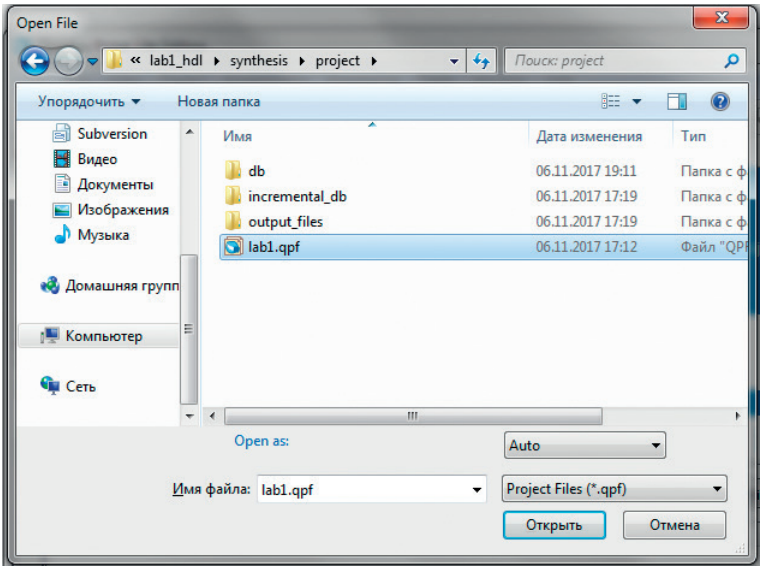


Рис. 1.41 Диалог открытия проекта в Quartus Prime

- нет необходимости использовать **Pin Planner** для назначения выводов в **ПЛИС**, так как вся нужная информация находится в **.qsf-**файле (листинг 1.7). Сравните этот файл с тем, который был создан в результате действий в разделе 1.2.5:

```
QUARTUS_VERSION = "17.0"
DATE = «12:11:55 November 06, 2017»
PROJECT_REVISION = "lab1"
```

Листинг 1.7 Файл проекта в Quartus

- Загрузить конфигурацию в микросхему **ПЛИС** так, как это описано в разделе 1.5;
- проверить работоспособность проекта на отладочной плате.

В файле назначений (листинг 1.8) обратите внимание на строки, содержащие назначение стандартов ввода-вывода:

```
set_instance_assignment -name IO_STANDARD «3.3-V LVTTL» -to LED*
```

Вместо перечисления всех выходов в таком виде:

```
set_instance_assignment -name IO_STANDARD «3.3-V LVTTL» -to LED[0]
```

используется символ «*» для обозначения всех выводов с одинаковыми частями в названии (листинг 1.8).

```
# project and device settings
set_global_assignment -name FAMILY «MAX 10»
set_global_assignment -name DEVICE 10M50DAF484C7G
set_global_assignment -name TOP_LEVEL_ENTITY lab1
set_global_assignment -name PROJECT_OUTPUT_DIRECTORY output_files

# I/O ports settings
set_location_assignment PIN_B8 -to KEY[0]
set_location_assignment PIN_A7 -to KEY[1]
set_location_assignment PIN_A8 -to LED[0]
set_location_assignment PIN_A9 -to LED[1]
set_location_assignment PIN_A10 -to LED[2]
set_location_assignment PIN_B10 -to LED[3]
set_location_assignment PIN_D13 -to LED[4]
set_location_assignment PIN_C13 -to LED[5]
set_location_assignment PIN_E14 -to LED[6]
set_location_assignment PIN_D14 -to LED[7]
set_location_assignment PIN_A11 -to LED[8]
set_location_assignment PIN_B11 -to LED[9]
set_instance_assignment -name IO_STANDARD «3.3 V SCHMITT TRIGGER» -to
KEY*
set_instance_assignment -name IO_STANDARD «3.3-V LVTTL» -to LED*
# project files
set_global_assignment -name VERILOG_FILE ../../lab1.v
```

Листинг 1.8 Файл назначений

1.10 Упражнения

1.10.1 Основное задание

Составьте таблицу истинности для приведенного ниже уравнения. Используя **схемотехнический редактор** и язык **Verilog**, реализуйте указанное ниже уравнение. Напишите **тестбенч** для проверки работоспособности устройства. Используя переключатели и светодиоды, проверьте работоспособность проектов на отладочной плате.

1. $y = \overline{x_1 x_2} + x_3 \overline{x_4} + x_1 x_3 x_4 + x_3$;
2. $y = x_1 \overline{x_2} + \overline{x_3 x_4} + \overline{\overline{x_1}\,\overline{x_2}} + \overline{\overline{x_2}\,\overline{x_3}}$;
3. $y = \overline{x_1 \overline{x_2} x_3} + x_3 x_4 + (x_1 \overline{x_2} x_3 \overline{x_4} + \overline{x_1 x_3} + \overline{x_1 x_3 x_4})$;
4. $y = \overline{x_1 x_2 + x_3} + \overline{x_1 \overline{x_2}} + \overline{x_2 x_3}$;
5. $y = \overline{\overline{x_1 + x_2} + x_1 x_3} + \overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{x_4}$;
6. $y = \overline{x_1 x_2 x_3 + x_3} + x_1 \overline{x_2} x_3 \overline{x_3 x_4} + x_1 \overline{x_3}$;
7. $y = \overline{x_1 x_2 x_3 + x_4} + x_1 + \overline{x_1 x_2 x_3 x_4} + \overline{x_3 + x_4}$;

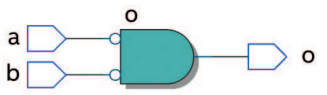
8. $y = \overline{x_1 + x_2 x_4 + \overline{x_3 x_4}} + \overline{x_2 x_3 \overline{x_4}\,\overline{x_1 x_2}} + x_1 \overline{x_2} x_3 + \overline{x_2 x_3 \overline{x_3 x_4}}$;
9. $y = \left(\overline{x_3 x_4} + x_1 x_4 \overline{\overline{x_2}\,\overline{x_3} x_4} + x_2\right)\left(\overline{x_1} x_3 \overline{x_4}\,\overline{x_2 x_3} + \overline{x_2}\,\overline{x_1 x_3} + x_4 + \overline{x_2} x_3\right)$;
10. $y = \overline{\overline{x_4} + x_1 x_2 \overline{x_3 x_4}} + x_2 \cdot \overline{\overline{x_2} x_4} + x_1 \cdot \overline{x_2 + \overline{x_3 \overline{x_4}}}$;
11. $y = \overline{x_1 \overline{x_3 x_4}} + \overline{\overline{x_2} x_4 \left(x_2 + \overline{x_3} + x_4\right)} + \overline{x_3}\,\overline{x_4} + \left(x_2 + x_1 \overline{x_2 x_3}\,\overline{x_4} + x_3\right)\overline{x_3}$;
12. $y = \overline{x_2 \overline{x_3}} + \overline{x_1}\,\overline{x_2 x_3 \overline{x_4}} + \overline{x_2 x_4 x_1 \overline{x_3}} + \overline{x_3 \overline{x_4}\,\overline{x_1 x_2 x_3}} + x_2 + \overline{x_3 + x_4}$;
13. $y = \overline{\overline{x_1} x_2 \overline{x_3 x_4}} + x_2 + x_3 + x_4 + \overline{x_1 \overline{x_2} x_3 \overline{x_2 x_3}} + \overline{x_2}\,\overline{x_3}$;
14. $y = \overline{x_1 + x_2 + x_3 + x_4} + x_2 + \overline{\overline{\overline{x_1} x_2} + x_2 \overline{x_3}} + x_3 + \overline{x_1} x_3 \overline{x_4 x_2 x_3}$;
15. $y = \overline{\overline{x_1} x_2 + x_2 + x_3} + \overline{\overline{x_2} x_4 + \overline{x_1}} + \overline{x_1 \overline{x_2} x_3 \overline{x_4} + \overline{x_1}\,\overline{x_2} x_3 x_4}$;

1.10.2 Контрольные вопросы

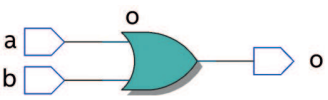
1. Какая схема является результатом компиляции приведенного модуля?

```
module logic
(
   input a,
   input b,
   output o
);
   assign o = a & b;
endmodule
```

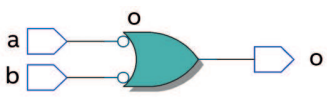
а)



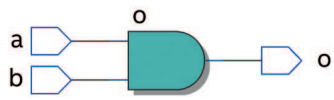
б)



в)



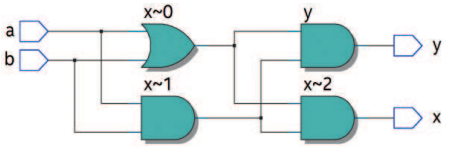
г)



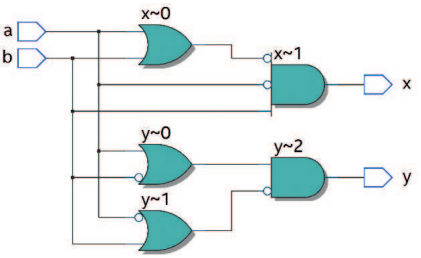
2. Какая схема является результатом компиляции приведенного модуля?

```
module logic
(
   input a,
   input b,
   output x,
   output y
);
   assign x = (a | b) & ~ (a & b);
   assign y = (a & ~b) & ~ (~a & b);
endmodule
```

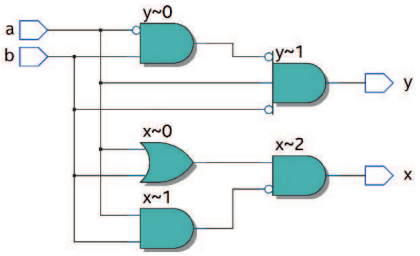
а)



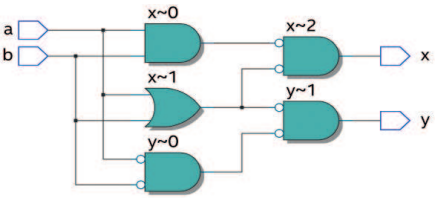
б)



в)



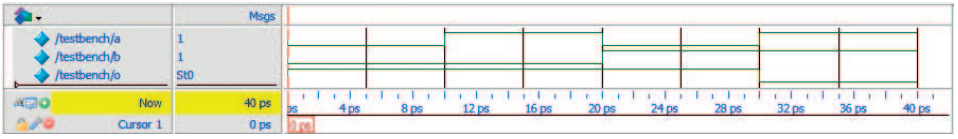
г)



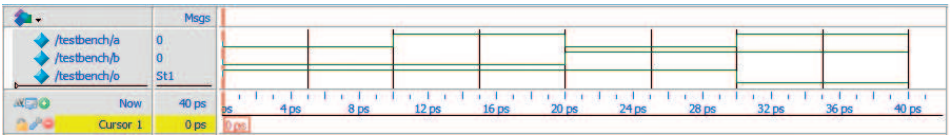
3. Какие диаграммы соответствуют описанию на языке Verilog HDL?

```
module design (input a, input b, output o);
   assign o = ~ a | ~ b;
endmodule

module tb;

   reg a;
   reg b;
   wire o;
   design design_inst (a, b, o);

   initial
   begin
      $dumpvars;
      $monitor ("a %b b %b o %b", a, b, o);
      a = 0; b = 0; #10;
      a = 1; b = 0; #10;
      a = 0; b = 1; #10;
      a = 1; b = 1; #10;
   end
```

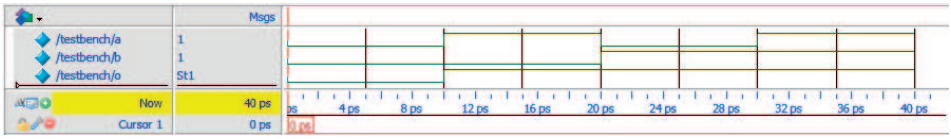
а)



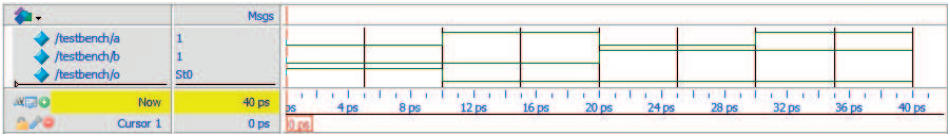
б)


Msgs
/testbench/a
/testbench/b
/testbench/o
0
0
St1
Now
40 ps
Cursor 1
0 ps
4 ps
8 ps
12 ps
16 ps
20 ps
24 ps
28 ps
32 ps
36 ps
40 ps

в)


Msgs
/testbench/a
/testbench/b
/testbench/o
1
1
St1
Now
40 ps
Cursor 1
0 ps
4 ps
8 ps
12 ps
16 ps
20 ps
24 ps
28 ps
32 ps
36 ps
40 ps

г)


Msgs
/testbench/a
/testbench/b
/testbench/o
1
1
St0
Now
40 ps
Cursor 1
0 ps
4 ps
8 ps
12 ps
16 ps
20 ps
24 ps
28 ps
32 ps
36 ps
40 ps

Илья Кудрявцев, Ирина Романова,
Александр Романов, Юрий Панчул

Цифровой синтез: практический курс

Глава 2. Основы последовательностной логики. Управление энергопотреблением цифровой схемы

Содержание

*Глава охватывает основные принципы разработки последовательностных устройств: рассмотрены защелки и триггеры, приведены примеры различных реализаций триггеров на языке **Verilog** и их использование при работе с платой **ПЛИС**. **Триггеры** – широко используемые блоки при сложном цифровом проектировании. Поскольку триггеры могут хранить состояние системы, на них можно строить сложные устройства в виде конечных автоматов. В свою очередь, применение конечных автоматов позволяет проектировать сложные цифровые системы с возможностью формальной верификации. Другое применение триггеров – это конвейерное проектирование, которое может увеличить быстродействие вычислительных систем.*

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime** (есть студенческая версия);*
- *пакет ModelSim Altera Edition или программы Icarus Verilog и GTKWave;*
- *отладочная плата компании **Terasic DE10Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах[1] к данной книге).*

2.1 Последовательностные элементы

Существует два вида цифровых логических схем – комбинационные и последовательностные. Комбинационные логические схемы можно описать при помощи таблиц истинности или логических выражений. В них все выходные значения зависят только от входных значений в текущий момент времени. Последовательностные логические схемы обладают памятью предыдущих состояний, то есть значение выхода такой схемы зависит не только от состояний на входах в текущий момент, но и от предыдущих состояний.

Рассмотрим основные принципы разработки последовательностных устройств. Особенность данной главы состоит в том, что исходный код на языке **Verilog**, который содержится в дополнительных материалах[2], приведен только в целях демонстрации. Код работоспособен, но такой стиль написания кода не используется разработчиками аппаратного обеспечения, поскольку в языке **Verilog** содержатся специальные средства для реализации последовательностных схем.

2.1.1 Асинхронный RS-триггер

Самой простой последовательностной схемой является асинхронный **RS-триггер** (**ResetSet**). Обычно такой триггер используется как строительный элемент при создании более сложных схем с памятью. Асинхронный **RS-триггер** состоит

[1] https://github.com/RomeoMe5/DDLM.
[2] Там же.

из двух элементов ИЛИ-НЕ с перекрестными обратными связями, как показано на рис. 2.1. Важным является наличие обратных связей, проходящих с выхода **N1** на вход **N2** и с выхода **N2** на вход **N1**. Данная конфигурация позволяет схеме хранить один бит данных. Обозначение асинхронного RS-триггера на схемах приведено на рис. 2.2.

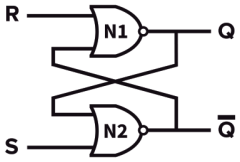


Рис. 2.1 Схема асинхронного RS-триггера

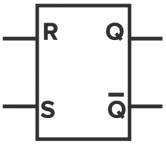


Рис. 2.2 Обозначение асинхронного RS-триггера

Работа асинхронного **RS-триггера** зависит от двух входов: **R** (**Reset**) – сброс и **S** (**Set**) – установка. Если вход **R** установлен в **1** и вход **S** установлен в **0** (режим сброса), то на выходе **N1** будет **0**, а на выходе **N2** – **1**. В результате на выходе **Q** будет **0** (в этом случае говорят, что триггер сброшен), а на инверсном выходе **Q'** (не **Q**) будет **1**. Таким образом, триггер хранит нулевое значение бита. Даже если вход **R** перейдет в **0** (**S = R = 0** – режим хранения), выход **Q'**, связанный обратной связью с **N1**, гарантирует, что на выходе **Q** останется **0**. Схема запоминает свое состояние.

В противоположном случае триггер работает аналогично. Если вход **R** установлен в **0** и вход **S** установлен в **1** (режим установки), то на выходе **N1** будет **1**, на выходе **N2** – **0**, а на выходе **Q** будет **1** (на выходе **Q'** будет **0**). В этом случае говорят, что триггер установлен, т. е. хранит единичное значение бита. Даже если выход **S** перейдет в **0**, выход **Q** сохранит состояние.

Можно сделать вывод, что пара битов на входах **S-R {0, 1}** работает как команда сброса, т. е. сбрасывают выход **Q** в **0**. Аналогично входные биты **{1, 0}** работают как команда установки и устанавливают выход **Q** в **1**. Еще одна команда – это **{0, 0}**. В этом случае не происходит никаких изменений. Триггер сохраняет предыдущее значение бита.

У этой простой схемы есть одно ограничение: когда оба входа **R** и **S** установлены в **1**, происходит логический конфликт (выходы **Q** и **Q'** равны **0** одновременно). Такое состояние называется неопределенным, а входная комбинация – запрещенной. При переходе из подобного состояния в режим хранения одновременным переключением входов в **0** состояние триггера может оказаться любым. Поэтому рекомендуется избегать поступления на вход триггера команды **{1, 1}**.

Асинхронный RS-триггер (моделирование)

Модуль **sr_latch** (листинг 2.1) демонстрирует реализацию асинхронного **RS-триггера**, приведенного на рис. 2.1. Имеются две входные переменные (**s** – установка и **r** – сброс) и две взаимосвязанные выходные переменные (**q** и **q_n**).

```
module sr_latch
(
   input s,
```

```
   input r,
   output q,
   output q_n
);
   assign q = ~ ( r | q_n );
   assign q_n = ~ ( s | q );
endmodule
```

Листинг 2.1 RS-триггер

Тестовый модуль, используемый для демонстрации работы модуля **sr_latch**, приведен в листинге предыдущей главы.

```
`timescale 1 ns / 100 ps
module testbench;
   reg s, r;
   wire q, q_n;
   sr_latch sr_latch (s, r, q, q_n);

   initial $dumpvars;
   initial
   begin
      $monitor ("%0d s %b r %b q %b q_n %b", $time, s, r, q, q_n);
      # 10; s = 0; r = 0;
      # 10; s = 1; r = 0;
      # 10; s = 0; r = 0;
      # 10; s = 0; r = 1;
      # 10; s = 0; r = 0;
      # 10; s = 1; r = 1;
      # 10; s = 0; r = 0;
      # 10; s = 0; r = 0;
      # 10;
      $finish;
   end
endmodule
```

Листинг 2.2 Тестовый модуль для RS-триггера

Данный код похож на тестовый модуль, описание которого дано в предыдущей главе:

- код начинается с команды **timescale**, которая нужна для того, чтобы задать временную шкалу, используемую симулятором. В данном примере заданный единичный интервал времени равен **1 нс**, а разрешение по времени – **100 пс**. Это означает, что **1 нс** – это один элемент задержки (**#1**) для тестового модуля, а минимальное время моделирования по временной шкале – **100 пс** (одна десятая от заданного элемента задержки):

  ```
  `timescale 1 ns / 100 ps
  ```

- следующая команда – это объявление тестового модуля:

```
module testbench;
```

- следующие две строки задают входные и выходные сигналы тестируемого устройства. Тип **reg** используется для входных сигналов тестируемого устройства, потому что эти сигналы хранят значения и устанавливаются непосредственно самим тестовым модулем. Для выходных сигналов, которые генерирует тестируемое устройство, используется тип wire:

```
reg s, r;
wire q, q_n;
```

- следующая строка – объявление тестируемого устройства. Здесь первая запись **sr_latch** – это имя модуля, а вторая – название кода. В скобках перечислены входные и выходные переменные тестируемого устройства.

```
sr_latch sr_latch (s, r, q, q_n);
```

- блоки **initial** запускаются только один раз в начале моделирования. Первый блок содержит инструкцию симулятору хранить все моделируемые значения сигналов в файле. Эта команда полезна при использовании **Icarus Verilog**, поскольку в этом случае для просмотра временных диаграмм моделируемых сигналов используется другая среда (**GTKWave**):

```
initial $dumpvars;
```

- второй блок **initial** начинается с команды **monitor**. Данная команда отображает значения параметров каждый раз, когда значения параметров изменяются. Блок заканчивается командой $finish, которая сообщает об окончании моделирования:

```
initial
begin
   $monitor ("%0d s %b r %b q %b q_n %b", $time, s, r, q, q_n);
   ...
   $finish;
end
```

- блок **initial**, описанный выше, также содержит команды для организации задержки: **#10** – это означает, что необходимо сделать задержку на **10** стандартных интервалов задержки (длительность элемента задержки была установлена командой **timescale**). За командами задержки следуют команды установки значений входных сигналов тестируемого устройства. Например, ниже приведены три команды: ожидать **10** временных интервалов, а затем установить сигналы **s** и **r** в **0**:

```
# 10; s = 0; r = 0;
```

Чтобы начать моделирование с помощью данного тестового модуля, необходимо запустить файл **01_simulate_with_modelsim.bat** в папке **lab_02\src\00_sr_latch_theory\simulation** дополнительных материалов[1] к данной главе.

[1] https://github.com/RomeoMe5/DDLM.

Временные диаграммы моделируемых сигналов должны быть такими же, как показано на рис. 2.3.

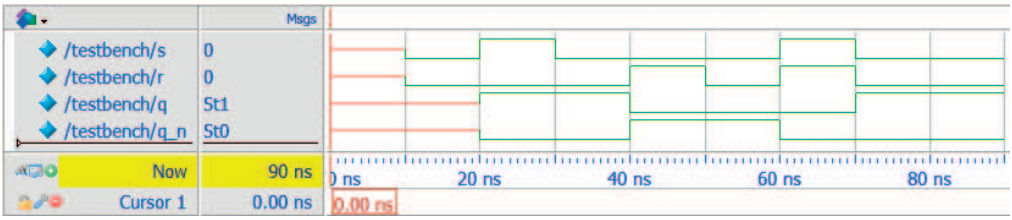


Рис. 2.3 Временные диаграммы моделируемых сигналов для RS-триггера

Эти временные диаграммы полностью соответствуют описанной ранее теории.

Асинхронный RS-триггер (аппаратная реализация)

Рассмотрим пример аппаратной реализации **RS-триггера**. Используемый аппаратный модуль верхнего уровня приведен в листинге 2.3.

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
   assign LEDR[9:2] = 7'b0;
   sr_latch sr_latch
   (
      .s ( ~KEY [0] ),
      .r ( ~KEY [1] ),
      .q ( LEDR[0] ),
      .q_n ( LEDR[1] )
   );
endmodule
```

Листинг 2.3 Реализация RS-триггера (модуль верхнего уровня иерархии)

Входной сигнал **s RS-триггера** соединяется с кнопкой **KEY[0]**, а сигнал **r** – с кнопкой **KEY[1]**. Сигналы инвертированы из-за схемы включения кнопок (кнопка возвращает **1**, когда нет нажатия, и **0** – когда есть нажатие). Выходы **RS-триггера** соединены с расположенными на плате светодиодами с помощью сигналов **LEDR[0]** и **LEDR[1]**.

Чтобы осуществить прототипирование проекта на базе **ПЛИС**, необходимо выполнить следующие шаги:

- запустить файл **make_project.bat**, чтобы создать папку для синтеза и файл проекта:

  `lab_02\src\00_sr_latch_theory\synthesis\de10_lite\make_project.bat;`

- открыть файл проекта в среде проектирования **Quartus Prime**:

  `lab_02\src\00_sr_latch_theory\synthesis\de10_lite\project\de10_lite.qpf;`

- скомпилировать проект (**Processing → Start Compilation**);
- после завершения компиляции открыть окно **RTL Viewer** в проекте (**Tools → Netlist Viewers → RTL Viewer)** и сравнить его с рис. 2.4;
- подключить плату и выполнить ее конфигурирование;
- нажимая кнопки **0** и **1** на плате в различных комбинациях, сравнить состояние светодиодов с результатами моделирования, полученными в предыдущем разделе.

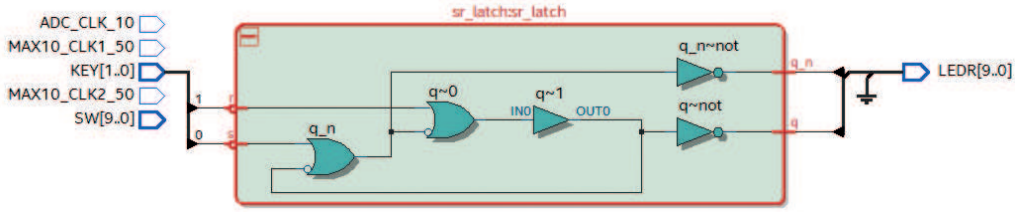


Рис. 2.4 Реализация асинхронного RS-триггера (RTL Viewer)

2.1.2 D-защелка[1]

Приведенный ранее асинхронный **RS-триггер** имеет ограничение: его выходы переходят в неопределенное состояние, когда оба входа **r** и **s** установлены в **1**. Чтобы преодолеть это ограничение, добавим еще один логический вход в схему асинхронного **RS-триггера**. Внутренняя структура получившейся **D-защелки** приведена на рис. 2.5, а схематичное изображение – на рис. 2.6. Наличие входа **CLK**, в отличие от ранее рассмотренного асинхронного **RS-триггера**, автоматически переводит устройство в разряд синхронных. Синхронные устройства обладают более высокой помехоустойчивостью по сравнению с асинхронными, так как при отсутствии единичного уровня тактового импульса (**CLK = 0**) помехи на информационных входах (в данном случае **D**) не влияют на состояние устройства.

[1] В отечественной литературе термин **защелка**, как правило, относится к последовательностным устройствам, синхронизируемым уровнем тактового сигнала. Чаще же используется термин **триггер** независимо от типа синхронизации, тогда как в зарубежной литературе термин **триггер** обычно обозначает последовательностное устройство, синхронизируемое передним или задним фронтом тактового сигнала. Здесь и ниже будет использоваться термин **защелка** по отношению к устройствам, синхронизируемым уровнем, как это сделано в книге: *Дэвид М. Харрис*, *Сара Л. Харрис*. Цифровая схемотехника и архитектура компьютера. М.: ДМК Пресс, 2017.

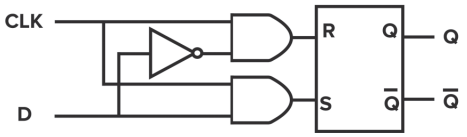


Рис. 2.5 Схема D-защелки

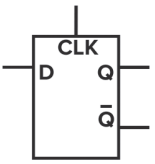


Рис. 2.6 Обозначение D-защелки

Инвертор на рис. 2.5 гарантирует, что входы **R** и **S** никогда не будут равны **{1, 1}**. Таким образом, **D-защелка** хранит один бит данных, поступающий на вход **D**. А тактовый сигнал **CLK** определяет момент, когда значение на входе **D** запоминается защелкой. На рис. 2.7 приведена таблица переходов для **D-защелки**.

CLK	D	$\overline{D}$	S	R	Q	$\overline{Q}$
0	X	$\overline{X}$	0	0	Qprev	$\overline{Qprev}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

Рис. 2.7 Таблица переходов для D-защелки

В соответствии с таблицей переходов, когда вход **CLK** равен **0**, **D-защелка** закрыта (непрозрачна) и текущее значение на входе **D** игнорируется. На выход **D-защелки** подается предыдущее сохраненное значение. Когда вход **CLK** установлен в **1**, **D-защелка** открывается (становится прозрачной) и передает на выход текущее значение входа **D**.

D-защелка характеризуется недостатком, присущим всем защелкам. Если сигнал **D** изменяется в то время, когда **D-защелка** открыта, изменения сразу передаются на выходы **Q** и **Q'**. Устройство хранения, преодолевающее этот недостаток и обновляющее сохраненное значение только в заданный момент времени, называется **D-триггером** и рассматривается в разделе 2.1.3.

D-защелка (моделирование)

Модуль **d_latch** (листинг 2.4) демонстрирует реализацию **D-защелки**, приведенной на рис. 2.5.

```
module d_latch
(
   input clk,
   input d,
   output q,
   output q_n
);
   wire r = ~d & clk;
   wire s = d & clk;
   sr_latch sr_latch (s, r, q, q_n);
endmodule
```

Листинг 2.4 D-защелка

Тестовый модуль, используемый для демонстрации работы модуля **d_latch**, аналогичен рассмотренному ранее. Чтобы начать моделирование, необходимо запустить файл **01_simulate_with_modelsim.bat** в папке **lab_02\src\01_d_latch_theory\simulation**.

Временные диаграммы моделируемых сигналов должны соответствовать рис. 2.8 в соответствии с приведенной ранее теорией.

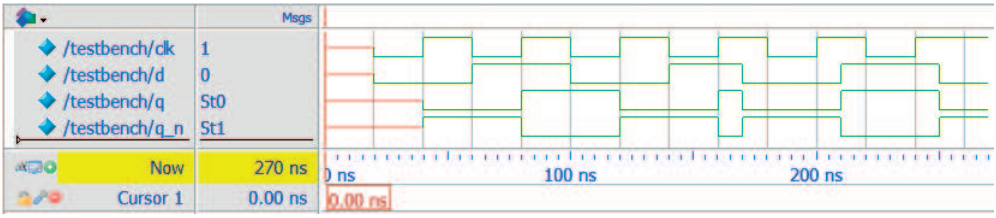


Рис. 2.8 Временные диаграммы моделируемых сигналов для D-защелки

D-защелка (аппаратная реализация)

Аппаратный модуль верхнего уровня иерархии приведен в листинге 2.5. Единственное отличие от аналогичного модуля для асинхронного **RS-триггера** (за исключением названия) – это то, что вместо входных сигналов **R** и **S** имеются входные сигналы **CLK** и **D**. Значение этих сигналов:

- **CLK** (**clock**) – тактовый сигнал (подключен к кнопке **KEY[0]**);
- **D** (**data**) – данные (подключен к кнопке **KEY[1]**).

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,

   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);

   assign LEDR[9:2] = 7'b0;

   d_latch d_latch
   (
      .clk ( ~KEY [0] ),
      .d ( ~KEY [1] ),
      .q ( LEDR[0] ),
      .q_n ( LEDR[1] )
   );
endmodule
```

Листинг 2.5 Реализация D-защелки (модуль верхнего уровня иерархии)

Последовательность действий для прототипирования **D-защелки** аналогична тем, что выполняются в разделе 2.1.2 для асинхронного **RS-триггера**. **RTL**-представление **D-защелки** дано на рис. 2.9.

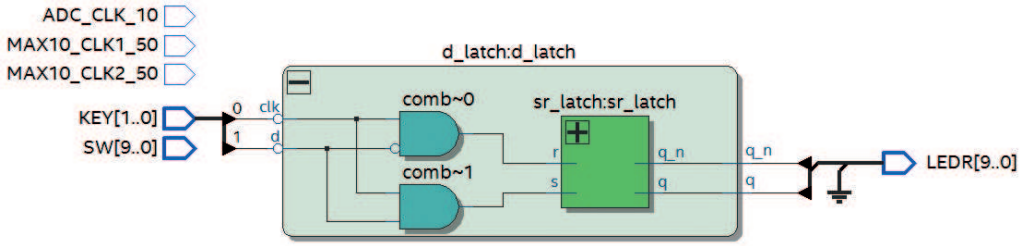


Рис. 2.9 Реализация D-защелки (RTL Viewer)

2.1.3 D-триггер

В отличие от **D-защелки**, **D-триггер** обновляет сохраненное значение только в заданный момент времени. **D-триггер** может быть построен из двух включенных последовательно **D-защелок**, называемых **ведущей** и **ведомой** (**master** и **slave**). В отечественной литературе такие схемы принято называть структурами мастер-помощник. Обе защелки управляются одним сигналом **CLK** (приведен на рис. 2.10). Защелка **L1** называется ведущей, а защелка **L2** – ведомой. Когда **CLK = 0**, ведущая защелка открыта и значение **N1** передается на вход **D**. В этот момент ведомая защелка закрыта и хранит предыдущее состояние. Когда **CLK** переходит в **1**, ведущая защелка закрывается, и значение **N1** передается в ведомую защелку. Таким образом, значение входа **D**, перед тем как сигнал **CLK** становится **1**, появляется на **N1**, а затем передается в защелку **L2**, и выход **Q** изменяет свое значение. **D-триггер** фиксирует значение входа **D** перед тем, как **CLK** устанавливается в **1**.

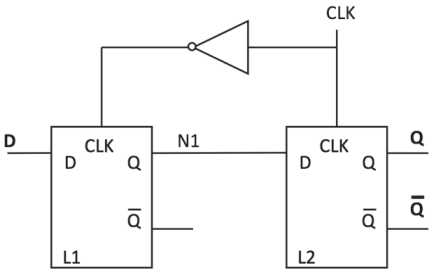


Рис. 2.10 Схема D-триггера

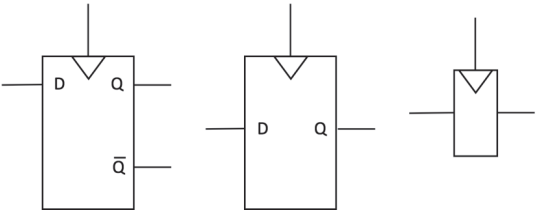


Рис. 2.11 Обозначение D-триггера

На рис. 2.11 приведено обозначение **D-триггера** на схемах. Треугольник около входа **CLK** означает, что вход синхронизируется по фронту. По умолчанию подразумевается передний фронт тактирующего сигнала (существуют также **D-триггеры** с синхронизацией по заднему фронту сигнала, при этом в обозначении используется кружок в углу треугольника). В отечественной литературе применяется обозначение, приведенное на рис. 2.12, и такие триггеры часто называют триггерами с динамическим управлением.

Рис. 2.12 D-триггеры: а) синхронизируемый передним фронтом; б) синхронизируемый задним фронтом

D-триггер, синхронизируемый передним или задним фронтом тактового сигнала, остается открытым очень небольшой промежуток времени (в течение переднего или заднего фронта сигнала **CLK**), в отличие от **D-защелки**, которая остается открытой в течение всего времени, пока **CLK = 1**. Преимуществом триггеров является более высокая помехоустойчивость по сравнению с защелками. Но **D-триггеры** имеют и свои недостатки. Сигнал на входе **D** должен быть стабилен непосредственно перед и сразу после фронта, описание чего дано в следующем разделе.

Кроме двухступенчатой структуры, управление по фронту или заднему фронту может быть реализовано более сложной схемой на базе бистабильных ячеек (рис. 2.13).

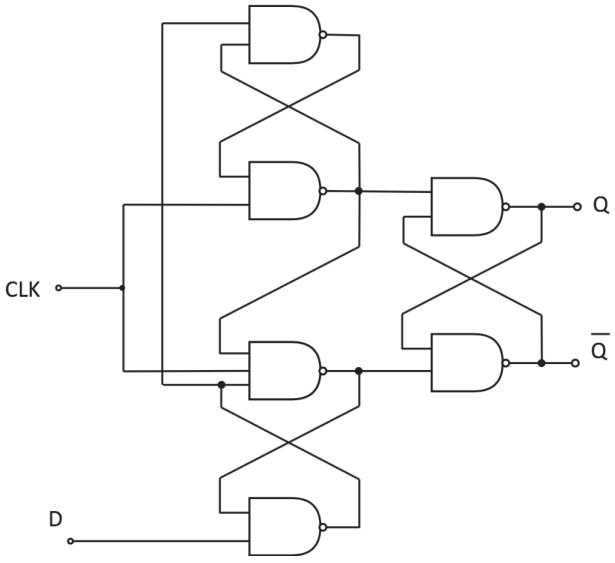


Рис. 2.13 D-триггер на базе бистабильных ячеек

Временные ограничения D-триггера с динамическим управлением

В соответствии с рис. 2.1 **RS-триггер** хранит один бит данных в петле обратной связи. Требуется время, равное двум задержкам в логических элементах, чтобы бит данных полностью прошел по петле. Если добавить еще один элемент, как показано на рис. 2.5, логично ожидать, что вход **D** будет стабилен на протяжении нескольких интервалов задержки.

В **D-триггере** сигнал с входа **D** передается на выход **Q** по переднему (или заднему) фронту тактирующего сигнала **CLK**. Этот процесс называется тактированием сигнала **D** по фронту **CLK**. Если сигнал **D** стабилен при тактировании, то выходной сигнал четко определен. Если сигнал **D** изменяется во время перепада тактирующего сигнала, могут возникнуть проблемы – в триггер запишется случайное или даже неверное (метастабильное) состояние.

Последовательностный элемент имеет временное окно в окрестностях фронта тактового сигнала, в течение которого входное значение должно быть стабильно, а на выходе триггера будет четко определенное значение. Чтобы определить данное временное окно, следует задать **время установки** (до перепада сигнала **CLK**) и **время удержания** (после перепада сигнала **CLK**). Время установки – это время перед приходом перепада сигнала **CLK**, в течение которого сигнал **D** должен быть стабилен. Время удержания – это время, в течение которого сигнал **D** должен быть стабилен, после прихода перепада сигнала **CLK**. Поскольку двухступенчатый триггер состоит из ведущей и ведомой защелок, которые соединены внутренней связью, следует определить время задержки между сигналами **CLK** и **Q** – время, в течение которого сигнал распространяется внутри элемента до выхода **Q** (это время называется **задержкой распространения**).

Таким образом, чтобы корректно использовать **D-триггер**, необходимо знать три временных параметра: время установки, время удержания и задержку распространения. В проектах на специализированных интегральных схемах (**ASIC**) защелки и триггеры – это библиотечные элементы, и их временные параметры известны и зависят от параметров технологического процесса (**32 нм**, **28 нм** и т. д.). Временные параметры элементов **ПЛИС** также зависят от быстродействия схемы (спецификация **Speed Grade**). В обоих случаях временные параметры известны средствам разработки и используются на этапе анализа временных параметров проекта. Один из параметров, который также вычисляется на данном этапе, – это максимальная частота, на которой может работать проект. Эта частота вычисляется на основе задержек из трех источников: задержек комбинационных элементов, задержек внутри **D-триггеров**, а также задержек в соединениях. Частота тактового сигнала должна позволять комбинационной логике вычислить новое состояние и записать его в **D-триггеры** схемы за период тактового сигнала. В среде проектирования **Quartus Prime** данный этап называется **TimeQuest Timing Analysis**.

D-триггер (моделирование)

Реализация **D-триггера**, изображенного на рис. 2.10, приведена в листинге 2.6.

```
module d_flip_flop
(
   input clk,
   input d,
   output q,
   output q_n
);
```

```
   wire n1;
   d_latch master
   (
      .clk ( ~clk ),
      .d ( d ),
      .q ( n1 )
   );
   d_latch slave
   (
      .clk ( clk ),
      .d ( n1 ),
      .q ( q ),
      .q_n ( q_n )
   );
endmodule
```

Листинг 2.6 Реализация D-триггера

Моделирование осуществляется точно так же, как и в предыдущих примерах. Итоговые временные диаграммы должны соответствовать рис. 2.14.

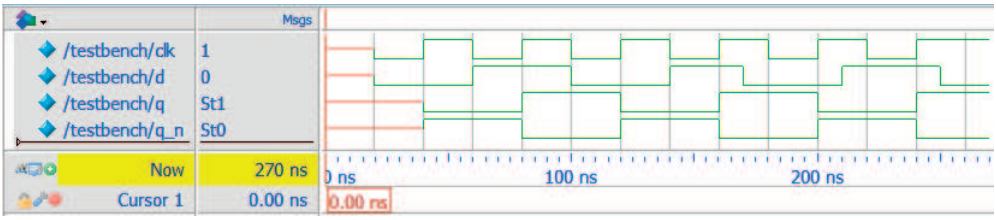


Рис. 2.14 Временные диаграммы моделируемых сигналов для D-триггера

D-триггер (аппаратная реализация)

Модуль верхнего уровня иерархии для аппаратной реализации **D-триггера** приведен в листинге 2.9. Он практически полностью аналогичен модулю из предыдущего примера. Процесс синтеза и последующей верификации **D-триггера** на отладочной плате также ничем не отличается. Результат **RTL**-синтеза **D-триггера** должен соответствовать рис. 2.15. Рисунок отлично иллюстрирует подход, когда две **D-защелки** располагаются последовательно для реализации **D-триггера**.

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);

   assign LEDR[9:2] = 7'b0;
```

```
   d_flip_flop d_flip_flop
   (
      .clk ( ~KEY [0] ),
      .d ( ~KEY [1] ),
      .q ( LEDR[0] ),
      .q_n ( LEDR[1] )
   );
endmodule
```

Листинг 2.7 Реализация D-триггера (модуль верхнего уровня иерархии)

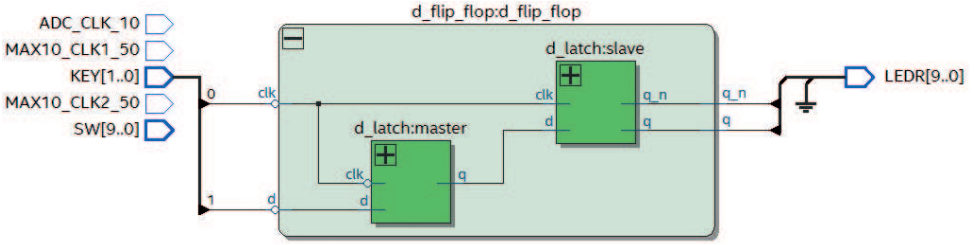


Рис. 2.15 Реализация D-триггера (RTL Viewer)

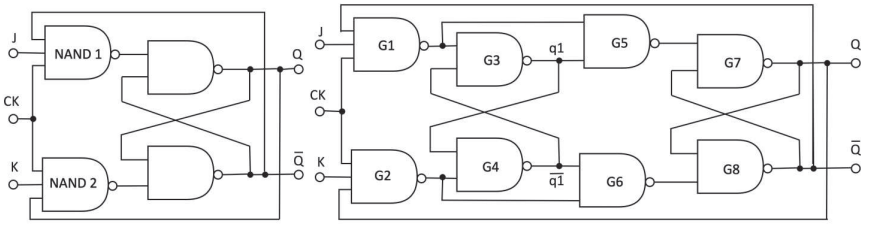
2.1.4 JK-триггер

JK-триггер принято называть универсальным триггером, так как на его основе можно построить триггеры других типов. Кроме того, он не имеет неопределенного состояния, как **RS-триггер**. Таблица переходов **JK-триггера** приведена на рис. 2.16.

CLK	J	K	Q
↑	0	0	Qprev
↑	1	0	1
↑	0	1	0
↑	1	1	$\overline{Qprev}$

Рис. 2.16 Таблица переходов JK-триггера

JK-триггер является синхронным и может быть построен в виде защелки или по принципу мастер-помощник (рис. 2.17).



а) защелка б) двухступенчатый JK-триггер

Рис. 2.17 Структура JK-триггера

Дополнительное задание для самостоятельной работы

Используя опыт, полученный при создании двухступенчатого **RS-триггера**, самостоятельно разработайте на языке **Verilog** модуль, описывающий структуру **JK-триггера**, приведенную на рис. 2.17.

2.1.5 T-триггер

T-триггер используется как делитель частоты на два и называется счетным, так как обеспечивает счет входящих импульсов (до двух). **T-триггеры** могут быть использованы для построения асинхронных счетчиков. Частота сигнала на его выходе в два раза меньше частоты сигнала на тактовом входе. **T-триггер** можно построить на базе **D-триггера** или **JK-триггера**, как показано на рис. 2.18.

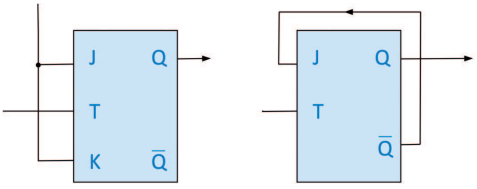


Рис. 2.18 Структура T-триггера

Дополнительное задание для самостоятельной работы

Используя созданные ранее **D-триггер** и **JK-триггер**, опишите **Verilog**-модули **T-триггера** в соответствии с рис. 2.18 и самостоятельно постройте таблицу переходов данного триггера.

2.2 Реализация последовательностной логики на языке Verilog

В языке **Verilog** существует три типа операторов присваивания. Один из них, оператор непрерывного присваивания (**assign**), уже использовался в примерах ранее. Этот оператор присваивания позволяет соединить сигналы внутри проекта и получить комбинационную схему:

```
assign o = a & b;
wire c = a | b;
```

Листинг 2.8 Оператор непрерывного присваивания

Следующий тип выражений, который следует рассмотреть, – это блокирующие и неблокирующие присваивания. Эти типы присваиваний могут использоваться только в специализированном блоке **always**. Данный блок начинается с оператора **always** и содержит список чувствительности (листинг 2.9). В представленном примере список чувствительности содержит сигналы a и b. Содержимое списка чувствительности зависит от используемого типа присваивания. В списке перечисляются все сигналы, изменение которых влияет на выходные сигналы, формируемые внутри блока.

```
always @ (a or b)
   ...
```

Листинг 2.9 Оператор always

Блокирующее присваивание (=) работает как оператор присваивания в языках программирования высокого уровня. Все выражения выполняются одно за другим. Результат предыдущего выражения может быть использован следующим выражением (листинге 2.10).

```
reg d;
wire a, b, c, w;
always @ (*)
begin
   d = c;
   d = d & a;
   w = d | b;
   q = w;
end
```

Листинг 2.10 Блокирующее присваивание

Результатом синтеза представленного кода является следующая комбинационная схема:

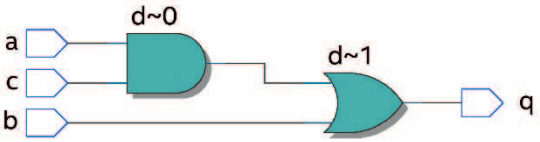


Рис. 2.19 Пример блокирующего присваивания: сгенерированная RTL-схема

В этом случае список чувствительности должен содержать все сигналы, используемые с правой стороны блокирующего присваивания **(a, b, c)**, кроме сигналов, используемых только внутри блока **always (w)**. Запись **(a or b or c)** можно заменить на **(*)**, как показано в листинге 2.10. Такая замена указывает компилятору, что в списке чувствительности будут использованы все сигналы, необходимые для выполнения операторов внутри блока. Компилятор может оптимизировать их набор по своему усмотрению, сохраняя логику, заданную разработчиком.

Операторы непрерывного присваивания можно трансформировать в операторы блокирующего присваивания и получить ту же схему в результате синтеза. Например, выражения из листинга 2.8 могут быть представлены в виде блокирующего присваивания (листинги 2.11 и 2.12). В первом случае (листинг 2.11) операторные скобки **begin… end** используются для того, чтобы разместить два блокирующих присваивания в одном блоке **always**.

```
always @ (*)
begin
   o = a & b;
   c = a | b;
end
```

Листинг 2.11 Блокирующее присваивание (один блок always)

Во втором случае (листинг 2.12) используются отдельные блоки **always** для каждого из блокирующих присваиваний, поэтому использование операторных скобок **begin… end** не обязательно. Поскольку обе строки присваиваний независимы, порядок их следования не важен, и поэтому они могут быть разнесены в разные блоки **always**.

```
always @ (*)
   o = a & b;

always @ (*) // допускается опустить скобки в данном выражении
   c = a | b;
```

Листинг 2.12 Блокирующее присваивание (отдельные блоки always)

Еще один тип присваиваний – это неблокирующее присваивание (<=), представляющее собой специальную операцию языка **Verilog** для последовательностных схем. В этом случае список чувствительности содержит сигналы, от значения которых зависит, когда произойдет сохранение значения в переменную. Этот список не одинаков для разных случаев (например, для **D-защелки**, **D-триггера** и т. д.). Все операции неблокирующего присваивания внутри блока **always** выполняются одновременно, причем откладываются до следующего цикла дельта-моделирования. Это отличается от стандартного подхода, используемого в других языках программирования, и может стать причиной непонимания у начинающего разработчика.

Правильное использование неблокирующего присваивания приведено в листинге 2.13. Это программный код простого **D-триггера**. Данная реализация подробно рассмотрена в последующих разделах.

```
wire d, clk;
reg q;
   always @ (posedge clk)
      q <= d;
```

Листинг 2.13 Пример неблокирующего присваивания

В следующем примере (листинг 2.14) демонстрируется неверное использование неблокирующих присваиваний. Поскольку неблокирующие присваивания выполняются одновременно, и при этом обе операции модифицируют регистр **d**, его значение будет неопределенным. В этом случае результаты моделирования и синтеза схемы могут различаться.

```
wire clk, c, d, d;
reg d;
always @ (posedge clk)
begin
   d <= c;
   d <= d & a;
end
```

Листинг 2.14 Неверное использование неблокирующего присваивания

В языке **Verilog** выражения присваивания имеют больше возможностей, чем инструменты синтеза. Существуют четыре правила, которые помогут избежать проблем и получить предсказуемые результаты при моделировании и синтезе схем.

1. Конструкцию **always @ (posedge clk)** и неблокирующие присваивания использовать для моделирования схем последовательностной логики:

```
always @ (posedge clk)
   q <= d;
```

2. Непрерывные присваивания применять для моделирования простых схем комбинационной логики:

```
assign o = a & b;
wire c = a | b;
```

3. Конструкцию **always @ (*)** и блокирующие присваивания использовать для моделирования сложных схем комбинационной логики:

```
always @(*)
begin
   d = c;
   d = d & a;
   d = d | b;
   q = d;
end
```

4. Не присваивать значение одному и тому же сигналу в различных блоках **always**. При необходимости можно осуществлять такие присваивания в пределах одного блока, но исключать возможность двойного присваивания.

Синтез проектов на **ПЛИС** и на специализированных интегральных схемах различается. В результате синтеза проекта на **ПЛИС** получается конфигурация **ПЛИС**, которая включает в себя конфигурацию и внутренние соединения логических элементов (**ЛЭ**) **ПЛИС** (одна микросхема **ПЛИС** содержит тысячи логических элементов).

Замечание: не путайте базовые логические элементы **И**, **ИЛИ**, **НЕ** с логическими элементами **ПЛИС**, описываемыми в данном разделе.

Структура логического элемента в **ПЛИС MAX 10 FPGA** производства **Intel FPGA** (**Altera**) приведена на рис. 2.20. Программируемый регистр каждого **ЛЭ** может быть сконфигурирован как любой из описанных в этой главе триггеров. В некоторых случаях программируемых регистров недостаточно, чтобы реализовать последовательностную логику, и тогда нужна дополнительная комбинационная логика. Такой пример рассматривается в разделе 2.3. Одной из важных задач разработчика схем на **ПЛИС** является эффективное использование ресурсов **ПЛИС**, поэтому следует избегать ситуаций, при которых генерируются дополнительные схемы, не предусмотренные логикой проекта.

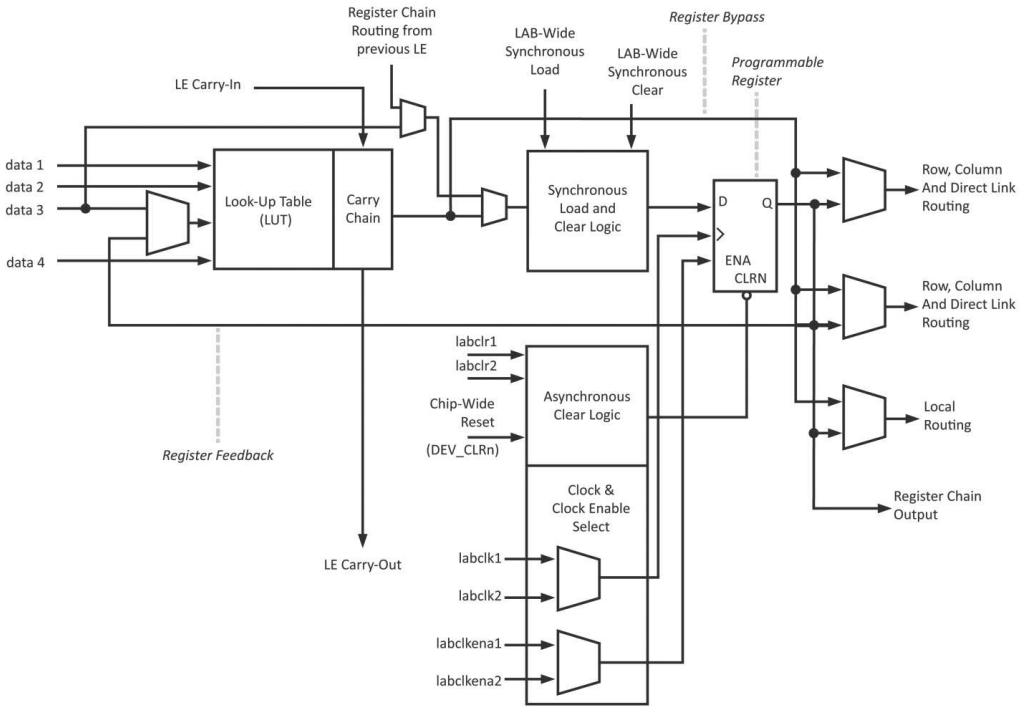


Рис. 2.20 Логический элемент в ПЛИС MAX 10 FPGA компании Intel FPGA (из руководства пользователя MAX10 FPGA[1])

Итак, следует использовать такие конструкции языка **Verilog**, в результате применения которых будет эффективно синтезирована аппаратная реализация схемы. Хорошим источником информации об этом служит документ **Recommended HDL Coding Style**[2]. Подобные документы предоставляются компаниями-разработчиками интегральных микросхем. Для среды проектирования **Quartus Prime** краткие подсказки по программированию можно найти в диалоговом окне **Edit → Insert Template** (рис. 2.21).

[1] MAX 10 FPGA Device Architecture. (n.d.). Retrieved January 7, 2019, from https://www.intel.com/content/www/us/ en/programmable/documentation/sss1397439908414.html.

[2] Intel Quartus Prime Pro Edition User Guide: Design Recommendations. (n.d.). Retrieved January 7, 2019, from https://www.intel.com/content/altera-www/global/en_us/index/documentation/sbc1513987577203.html# mwh1409959570946.

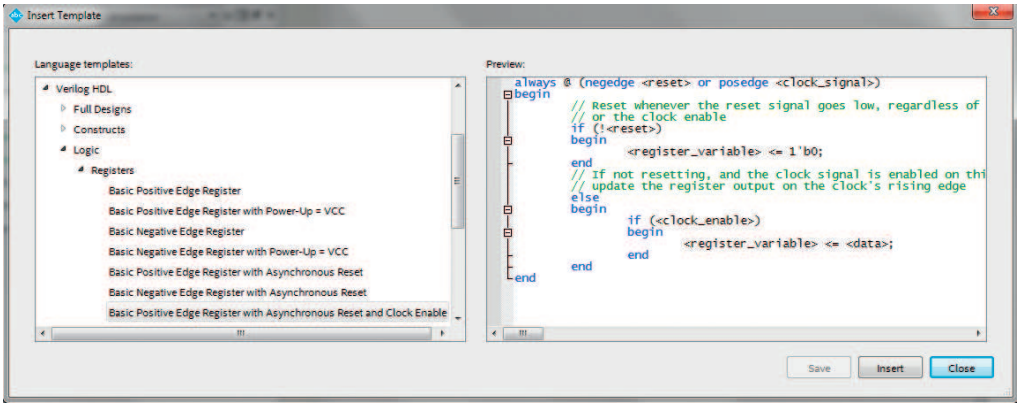


Рис. 2.21 Диалоговое окно Insert Template в среде проектирования Quartus Prime

Дополнительное задание для самостоятельной работы

Найти в сети интернет документ с рекомендациями по программированию на языке HDL для ПЛИС производства компании Intel FPGA.

2.2.1 Реализация D-защелки с использованием языка Verilog

В листинге 2.15 приведен пример того, как **D-защелка** может быть реализована с использованием операторов **Verilog**. В частности, для этого используется блок **always**, который активируется при изменении тактового сигнала **clk** или сигнала данных **d** (представлен в списке чувствительности блока), и также применяется неблокирующее присваивание (**<=**), которое работает только при высоком уровне тактового сигнала (из-за оператора **if**).

```
module d_latch
(
   input clk,
   input d,
   output reg q
);
   always @ (clk or d)
   if(clk)
      q <= d;
endmodule
```

Листинг 2.15 Реализация D-защелки

D-защелка (моделирование)

Результаты моделирования данной реализации **D-защелки** представлены на рис. 2.22. Сравните эти временные диаграммы с описанной ранее теорией.

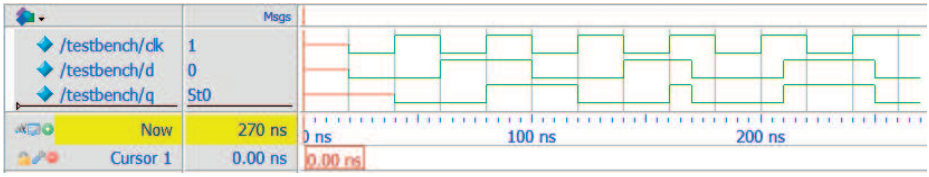


Рис. 2.22 Временные диаграммы моделируемых сигналов для D-защелки

D-защелка (аппаратная реализация)

Модуль верхнего уровня иерархии для прототипирования **D-защелки** представлен в листинг 2.16. **KEY[0]** используется как тактовый сигнал **clk**, **KEY[1]** – как информационный вход **d**, а **LEDR** для визуализации выходного сигнала **q**.

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
   assign LEDR[9:1] = 8'b0;
   d_latch d_latch
   (
      .clk ( ~KEY [0] ),
      .d ( ~KEY [1] ),
      .q ( LEDR[0] )
   );
endmodule
```

Листинг 2.16 Реализация D-защелки (модуль верхнего уровня иерархии)

Результаты **RTL**-синтеза представлены на рис. 2.23. Следует отметить, что в данном примере **D-защелка** реализована с использованием программируемого регистра **ПЛИС** (в конфигурации **защелка**). Вход программируемого регистра **DATAIN** используется для входа **d D-защелки**, а **LATCH_ENABLE** – для тактового сигнала. Выход **q D-триггера**-защелки подключен к выходу программируемого регистра **OUT0**. Здесь нет дополнительной комбинационной логики, т. е. ресурсы **ПЛИС** используются эффективно.

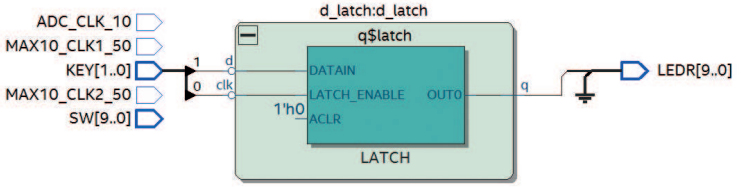


Рис. 2.23 Реализация D-защелки (окно RTL Viewer)

Дополнительное задание для самостоятельной работы

Отредактируйте код **D-защелки** (листинг 2.16) так, чтобы входное значение сохранялось при низком уровне тактового сигнала. Скомпилируйте код и сравните его **RTL**-представление с рис. 2.23.

2.2.2 Реализация D-триггера с использованием языка Verilog

D-триггер – это последовательностное устройство, которое передает значение входа **d** на выход **q** по каждому нарастающему фронту тактового сигнала **clk**. Также возможно определить **D-триггер**, который будет работать по заднему фронту тактового сигнала, но его аппаратная реализация зависит от возможностей используемой микросхемы **ПЛИС**. В некоторых случаях это может вызвать излишний расход аппаратных ресурсов микросхемы.

Пример реализации **D-триггера** на языке **Verilog** приведен в листинге 2.17. В отличие от предыдущего примера, поведение блока **always** зависит только от тактового сигнала. Ключевое слово **posedge** используется для уточнения того, что выражение внутри блока **always** должно работать только по положительному фронту тактового сигнала. Также используется тип сигнала **reg**, который в комбинации с неблокирующим присваиванием **<=** хранит значение **d**.

```
module d_flip_flop
(
   input clk,
   input d,
   output reg q
);
   always @ (posedge clk)
      q <= d;
endmodule
```

Листинг 2.17 Реализация D-триггера

D-триггер (моделирование)

Временные диаграммы моделируемых сигналов должны совпадать с рис. 2.24. Сравните вейвформы (временные диаграммы) с теорией, описанной ранее.

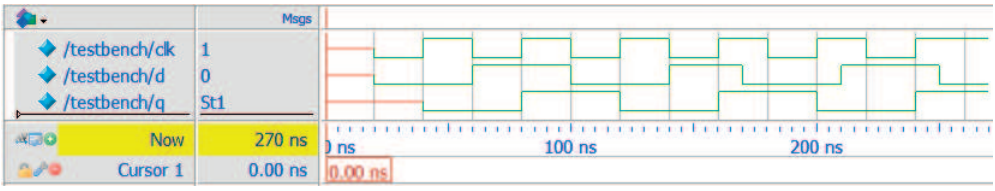


Рис. 2.24 Временные диаграммы моделируемых сигналов D-триггера

D-триггер (аппаратная реализация)

В листинга 2.18 приведен модуль верхнего уровня иерархии, нужный для прототипирования работы **D-триггера** на плате **ПЛИС**. Он похож на модуль, представленный в предыдущем примере. Здесь тактовый сигнал **clk** и входной информа-

ционный сигнал **b** подключены к кнопкам (**KEY[0]** и **KEY[1]**), а выходной сигнал **q D-триггера** управляет светодиодом.

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
   assign LEDR[9:1] = 8'b0;
   d_flip_flop d_flip_flop
   (
      .clk ( ~KEY [0] ),
      .d ( ~KEY [1] ),
      .q ( LEDR[0] )
   );
endmodule
```

Листинг 2.18 Реализация D-триггера (модуль верхнего уровня иерархии)

На рис. 2.25 приведены результаты **RTL**-синтеза данной реализации **D-триггера**. Следует отметить, что триггер синтезирован на программируемом регистре **ПЛИС**. При этом не сгенерировано какой-либо дополнительной комбинационной логики. Выводы **D-триггера** (вход и выход) подключаются к таким же выводам программируемого регистра (**d** к **D**, **clk** к **CLK** и **q** к **Q**).

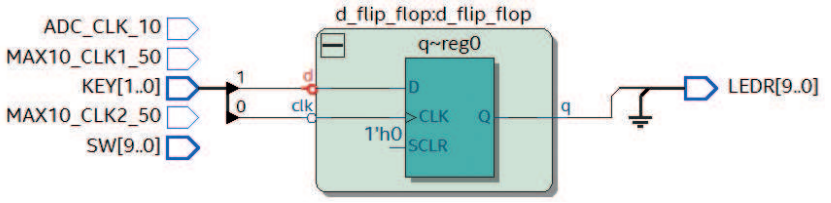


Рис. 2.25 Реализация D-триггера (окно RTL Viewer)

Дополнительное задание для самостоятельной работы

Измените **D-триггер**, код которого приведен в листинге 2.17, так, чтобы он работал под управлением заднего фронта тактового сигнала. Для этого необходимо заменить ключевое слово **posedge** на **negedge**. Скомпилируйте проект и сравните его **RTL**-представление с предыдущей реализацией на рис. 2.25.

2.2.3 Реализация JK-триггера с помощью языка Verilog

Код, иллюстрирующий поведение **JK-триггера** на языке **Verilog**, представлен в листинге 2.19.

```
module JKK(
   input J,
   input K,
   input CLK,
   output reg Q
   );

   always @(posedge CLK)
   begin
      if (J && K) Q <= ~ Q;
      else if (J && ~K) Q<=1;
      else if (~J && K) Q<=0;
   end
endmodule
```

Листинг 2.19 Реализация JK-триггера

Дополнительное задание для самостоятельной работы

Используя опыт, полученный при реализации **D-триггера**, самостоятельно отредактируйте код листинга 2.19. Проведите его моделирование, имитируя все возможные комбинации входных сигналов, приведенные в таблице переходов на рис. 2.16. Сравните получаемые результаты с этой таблицей.

Аналогичным образом отредактируйте код листинга 2.18, добавив еще одну кнопку. Проверьте работоспособность **JK-триггера**, наблюдая за состоянием светодиода, подключенного к выходу. Сравните затраты ресурсов, необходимых для создания **D-триггера** и **JK-триггера** в **RTL Viewer**.

2.3 Синхронный и асинхронный сбросы

В некоторых случаях полезно установить **D-триггер** в состояние по умолчанию. Этот процесс называется сбросом (**RESET**). **Сбрасываемые D-триггеры** обычно имеют инверсный дополнительный вход сброса (высокий уровень на нем означает, что сброс не требуется). При низком уровне на выводе сброса **D-триггер** игнорирует вход данных и устанавливает выход в **0**. В приведенных примерах сигнал сброса называется **n_rst**, где приставка **n_** означает, что вход инверсный.

Существует два варианта сброса: **синхронный** и **асинхронный**. Синхронный сброс означает, что триггер устанавливает в **0** значение только по нарастающему (спадающему) перепаду сигнала **CLK**. Асинхронный сброс означает, что триггер устанавливается в **0**, как только уровень сигнала сброса становится низким (по уровню), при этом значение на **CLK** игнорируется.

2.3.1 D-триггер с синхронным сбросом

Пример реализации синхронного сброса в **D-триггере** приведен на рис. 2.26. За счет логического элемента **И** сигнал инвертирующего сброса становится частью входных данных.

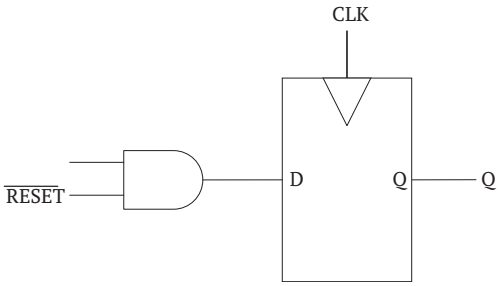


Рис. 2.26 Схема D-триггера с синхронным сбросом

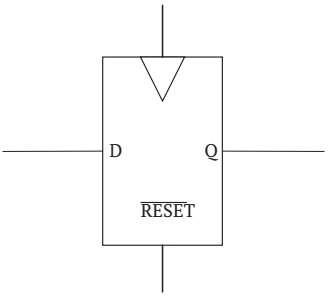


Рис. 2.27 D-триггер с символом сброса

Код, приведенный в листинге 2.20, практически идентичен базовой реализации **D-триггера** (рис. 2.31). Разница заключается только в том, что сначала проверяется значение сброса и только после этого новое значение присваивается выходу **Q**. Значение сброса не включено в список чувствительности блока **always**, потому что его значение проверяется только по переднему фронту сигнала **CLK**.

```
module dff_sync_rst_n
(
   input clk,
   input rst_n,
   input d,
   output reg q
);

   always @ (posedge clk)
      if (!rst_n)
         q <= 0;
      else
         q <= d;
endmodule
```

Листинг 2.20 Реализация D-триггера с синхронным сбросом

D-триггер с синхронным сбросом (моделирование)

Временные диаграммы моделируемых сигналов должны быть такими же, как показано на рис. 2.28. Сравните эти сигналы с теорией, рассмотренной ранее.

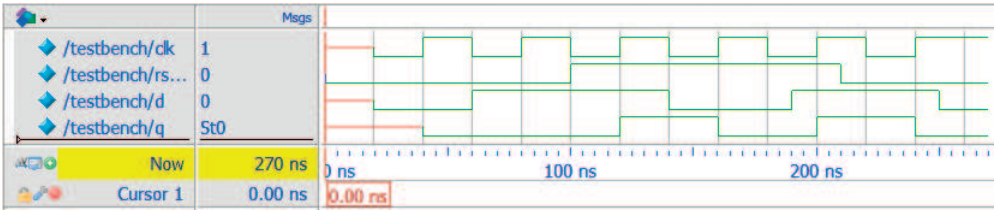


Рис. 2.28 Временные диаграммы моделируемых сигналов D-триггера с синхронным сбросом

D-триггер с синхронным сбросом (аппаратная реализация)

Ниже приведен модуль верхнего уровня иерархии, нужный для прототипирования работы **D-триггера** с синхронным сбросом на плате **ПЛИС**. Нужно использовать переключатель **SW[0]** в качестве источника сигнала сброса, поскольку на плате **DE10-Lite** имеется три входных сигнала (сброс, тактовый и информационный) и только две кнопки (**KEY[0]** и **KEY[1]**).

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
   assign LEDR[9:1] = 8'b0;
   dff_sync_rst_n dff_sync_rst_n
   (
      .clk ( ~KEY [0] ),
      .rst_n ( SW [0] ),
      .d ( ~KEY [1] ),
      .q ( LEDR[0] )
   );
endmodule
```

Листинг 2.21 Реализация D-триггера с синхронным сбросом (модуль верхнего уровня иерархии)

RTL-представление синтезированного триггера приведено на рис. 2.29. Следует обратить внимание на то, что рассмотренная конфигурация **D-триггера** с синхронным сбросом (листинг 2.20) не может быть реализована без дополнительной комбинационной логики, и в рассматриваемом случае для реализации схемы потребовался дополнительный мультиплексор на входе **D**.

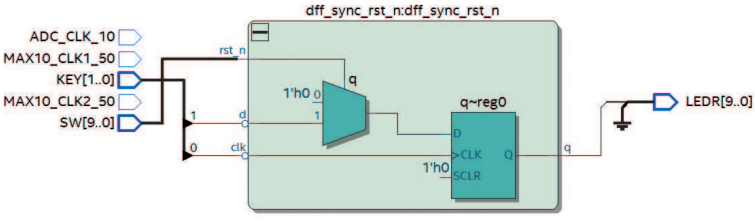


Рис. 2.29 Реализация D-триггера с синхронным сбросом (окно RTL Viewer)

Дополнительное задание для самостоятельной работы

Неинвертированный синхронный сигнал сброса часто называют сигналом очистки. Измените пример в листинге 2.20 для реализации **D-триггера** с сигналом очистки. Сравните его **RTL**-представление с рис. 2.29.

2.3.2 D-триггер с асинхронным сбросом

Изменим предыдущий пример так, чтобы сигнал сброса стал асинхронным. Все, что необходимо, – это добавить сигнал **rst_n** в список чувствительности блока **always**. Благодаря этому изменение значения на входе сброса с **1** на **0** приведет к обновлению значения *Q* (листинг 2.22).

```
module dff_async_rst_n
(
   input clk,
   input rst_n,
   input d,
   output reg q
);
   always @ (posedge clk or negedge rst_n)
      if (!rst_n)
         q <= 0;
      else
         q <= d;
endmodule
```

Листинг 2.22 Реализация D-триггера с асинхронным сбросом

D-триггер с асинхронным сбросом (моделирование)

Результаты моделирования **D-триггера** с асинхронным сбросом приведены ниже. Сравните эти сигналы с теорией, рассмотренной ранее.

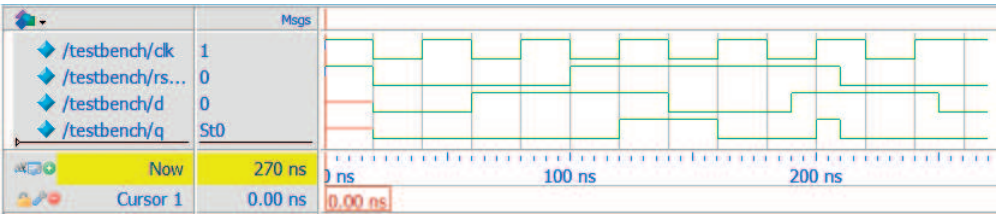


Рис. 2.30 Временные диаграммы моделируемых сигналов D-триггера с асинхронным сбросом

D-триггер с асинхронным сбросом (аппаратная реализация)

Ниже приведен модуль верхнего уровня иерархии, нужный для прототипирования работы **D-триггера** с асинхронным сбросом на плате **ПЛИС**.

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,

   input [ 1:0] KEY,
   input [ 9:0] SW,
```

```
   output [ 9:0] LEDR
);
   assign LEDR[9:1] = 8'b0;
   dff_async_rst_n dff_async_rst_n
   (
      .clk ( ~KEY [0] ),
      .rst_n ( SW [0] ),
   .d ( ~KEY [1] ),
   .q ( LEDR[0] )
   );
endmodule
```

Листинг 2.23 Реализация D-триггера с асинхронным сбросом (модуль верхнего уровня иерархии)

В отличие от предыдущей конфигурации с синхронным сбросом, рассмотренный **D-триггер** с асинхронным сбросом может быть реализован без дополнительных комбинационных логических блоков. Вход программируемого регистра **CLRN** используется для обеспечения асинхронного сброса.

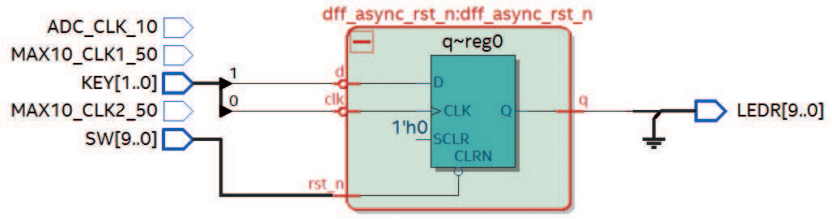


Рис. 2.31 Реализация D-триггера с асинхронным сбросом (окно RTL Viewer)

Дополнительное задание для самостоятельной работы

Добавьте в **D-триггер** асинхронный сброс с сигналом очистки, реализованным на предыдущем шаге. Модуль должен иметь положительный синхронный сигнал очистки и в то же время асинхронный вход **n_rst**. Сравните его аппаратную реализацию (**RTL**-представление) с рис. 2.31.

2.3.3 Преимущества и недостатки синхронного и асинхронного сбросов

Таким образом, суммируя изложенный материал, преимущества и недостатки синхронного и асинхронного сбросов при реализации триггеров можно свести в следующую таблицу:

Таблица 2.1 Преимущества и недостатки синхронного и асинхронного сбросов

	Асинхронный сброс	Синхронный сброс

Реализация на Verilog	`always @(posedge clk or negedge rst)`
	`  If(!rst) Q<=# 1'b0;`
	`  else`
	`  Q<=#1 D`
		`always @(posedge clk)`
		`  If(!rst) Q<=# 1'b0;`
		`  else`
		`  Q<=#1 D`

Продолжение табл. 2.1

	Асинхронный сброс	Синхронный сброс
Достоинства	Не требуется тактовый сигнал	Не надо беспокоиться о нарушении временных соотношений
Недостатки	Любая помеха в цепи сброса сбрасывает триггер (если имеются междоменные тактовые сигналы, необходимо синхронизировать сброс в каждом домене). Нужно контролировать снятие сигнала сброса	Необходимо контролировать установку и снятие сигнала сброса

Использование того или иного подхода зачастую диктуется аппаратурой и базой макроячеек, на которой будет реализовываться триггер. Наибольшее распространение получили реализации триггеров с синхронным сбросом.

2.3.4 Активируемый триггер

Аппаратная реализация **D-триггера** представляет собой набор транзисторов. Каждому из них необходима энергия для переключения, так как для каждого переключения полевого транзистора необходим ток, изменяющий заряд его затвора. Существует прием, с помощью которого можно снизить энергопотребление схемы за счет деактивации (заморозки) неиспользуемых частей. Для этого в схему **D-триггера** добавляют активирующий сигнал (**EN**).

При высоком уровне сигнала **EN** триггер работает в штатном режиме. При низком уровне **EN** активируемый **D-триггер** игнорирует значение **CLK** и сохраняет свое состояние. Варианты реализации активируемого **D-триггера** приведены на рис. 2.32. Первый из них (с мультиплексором) практически не влияет на энергопотребление схемы. Вторая же схема (с логическим элементом И в тактирующей схеме) позволяет уменьшить значение этой характеристики, поскольку активность всего триггера прекращается без тактирующего сигнала. Обозначение активируемого триггера представлено на рис. 2.33.

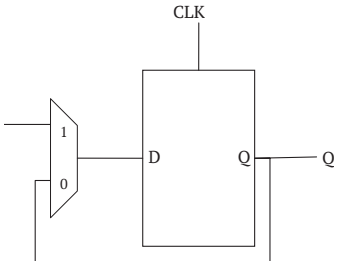


Рис. 2.32 Активируемый триггер: с мультиплексором MUX и вентилем И

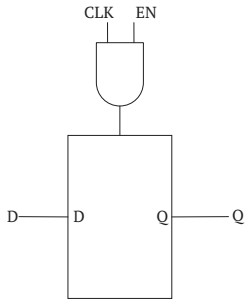


Рис. 2.33 Обозначение активируемого триггера

Следует отметить, что при разработке проектов на **ПЛИС** влияние на сигнал **CLK** считается не очень хорошей практикой, поскольку включение дополнительных элементов в цепь тактовых сигналов увеличивает задержку и может приводить

к возникновению ошибок. Хороший способ создания активируемого триггера в **ПЛИС** – разработка корректного **Verilog**-кода, результаты синтеза которого будут соответствующей конфигурацией программируемого регистра схемы **ПЛИС**. Пример такого кода приведен в листинге 2.24. Этот код содержит проверку разрешающего сигнала **en**. При его низком уровне сохраняемое значение не меняется.

```
module dff_with_en
(
   input clk,
   input en,
   input d,
   output reg q
);
   always @ (posedge clk)
      if (en)
         q <= d;
endmodule
```

Листинг 2.24 Реализация активируемого триггера

Активируемый триггер (моделирование)

Результаты моделирования **D-триггера** с асинхронным сбросом приведены на рис. 2.34. Сравните эти сигналы с теорией, рассмотренной ранее.

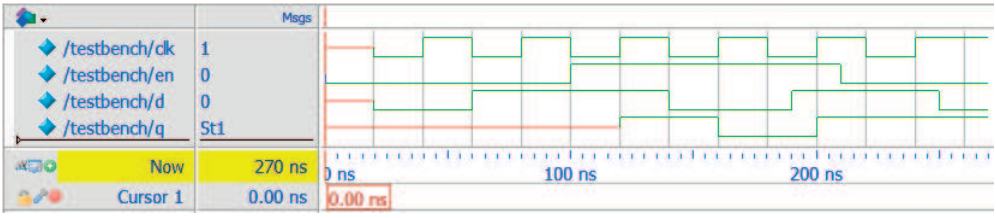


Рис. 2.34 Временные диаграммы моделируемых сигналов активируемого триггера

Активируемый триггер (аппаратная реализация)

Ниже приведен модуль верхнего уровня иерархии, нужный для прототипирования работы **D-триггера** с входом разрешения работы на плате **ПЛИС**. Главное отличие данного примера от предыдущего в том, что сигнал сброса заменен на сигнал разрешения (присваивание происходит при активном уровне сигнала **enable**).

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,

   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
```

```
   assign LEDR[9:1] = 8'b0;
   dff_with_en dff_with_en
   (
      .clk ( ~KEY [0] ),
      .en ( SW [0] ),
      .d ( ~KEY [1] ),
      .q ( LEDR[0] )
   );
endmodule
```

Листинг 2.25 Реализация активируемого триггера (модуль верхнего уровня иерархии)

Рисунок 2.35 демонстрирует программируемый регистр в конфигурации **D-триггера**. Вход программируемого регистра **ENA** используется как вход **EN** активируемого триггера. Здесь также не потребовалась дополнительная комбинационная логика.

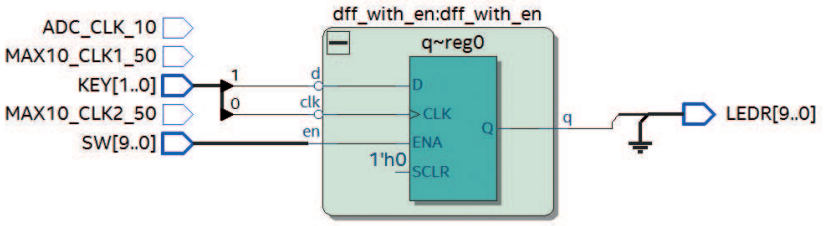


Рис. 2.35 Реализация активируемого триггера (окно RTL Viewer)

Дополнительное задание для самостоятельной работы

Измените активируемый триггер в листинге 2.24. Необходимо добавить синхронный положительный сигнал очистки и асинхронный сигнал **n_rst**. Сравните аппаратную реализацию (**RTL View**) с рис. 2.35.

2.3.5 Параметризированный триггер

Все примеры, которые обсуждались ранее, основаны на схеме хранения одного бита. Для упрощения конструкции более сложных схем (таких как умножители, сумматоры и др.) триггеры объединяются в многобитные конструкции – регистры. Сложная схема может быть построена из регистров с разной разрядностью (обычно из **8**, **16**, **32** и т. д. бит, но также бывает и из **3**, **5**, **9** бит и др.). Еще следует отметить, что эти регистры могут инициализироваться при сбросе не только нулевыми значениями.

В **Verilog** есть специальное ключевое слово **parameter**. Оно позволяет повторно использовать исходный код при разработке модулей разной разрядности. Пример такого модуля приведен в листинге 2.26. Это многобитный **D-триггер** (регистр) с асинхронным сбросом. Он имеет два параметра: разрядность (**WIDTH**) и сброс (**RESET**). Первый из них задает размер (разрядность) хранимой информа-

ции (в битах). Второй используется для установки начального значения регистра при низком уровне сигнала **rst_n**.

```
module dff_async_rst_n_param
#(
   parameter  WIDTH = 8,
              RESET = 8'b0
)
(
   input clk,
   input rst_n,
   input [WIDTH - 1 : 0] d,
   output reg [WIDTH - 1 : 0] q
);
   always @ (posedge clk or negedge rst_n)
      if (!rst_n)
         q <= RESET;
      else
         q <= d;
endmodule
```

Листинг 2.26 Параметризированный триггер

Типичное применение параметризованного модуля **Verilog** показано в листинге 2.27. Здесь разрядность регистра задана как **4** бита (т. е. он хранит значения от **0x0** до 0xF), а его начальное значение – **0xA** (или **10** в десятичной системе).

```
dff_async_rst_n_param #(.WIDTH(4), .RESET(4'ha))
 dff_async_rst_n_param (clk, rst_n, d, q);
```

Листинг 2.27 Пример использования параметризированного триггера

Параметризированный триггер (моделирование)

Проведем моделирование приведенного примера триггера. Временные диаграммы моделируемых сигналов должны быть такими же, как показано на рис. 2.36. Сравните эти сигналы с теорией, рассмотренной ранее.

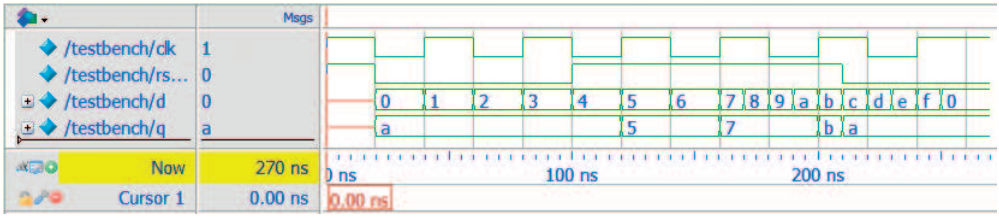


Рис. 2.36 Моделирование форм сигналов параметризованного D-триггера

Параметризированный триггер (аппаратная реализация)

Модуль верхнего уровня иерархии приведен в листинге 2.28. Регистр имеет разрядность **8** бит (устанавливается с помощью параметра **WIDTH**), а его начальное значение равно **0xF0**.

```
module de10_lite
(
   input ADC_CLK_10,
   input MAX10_CLK1_50,
   input MAX10_CLK2_50,

   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
   assign LEDR[0] = 1'b0;

   dff_async_rst_n_param
   #(
      .WIDTH ( 8 ),
      .RESET ( 8'hf0 )
   )

   dff_async_rst_n_param
   (
      .clk ( ~KEY [0] ),
      .rst_n ( SW [0] ),
      .d ( SW [9:1] ),
      .q ( LEDR [9:1] )
    );
endmodule
```

Листинг 2.28 Модуль верхнего уровня иерархии для параметризованного триггера

На рис. 2.35 приведена **RTL**-реализация программируемого регистра на **ПЛИС** в конфигурации **D-триггера**. В такой реализации модуль состоит из двух наборов **D-триггеров**. Для первого **D-триггера** программируемый вход регистра **CLRN** подключен к сигналу **rst_n** для очищения значения этих битов при сбросе, а для второго – сброс подключен ко входу **PRE**, что обеспечивает предустановку этих битов при сбросе.

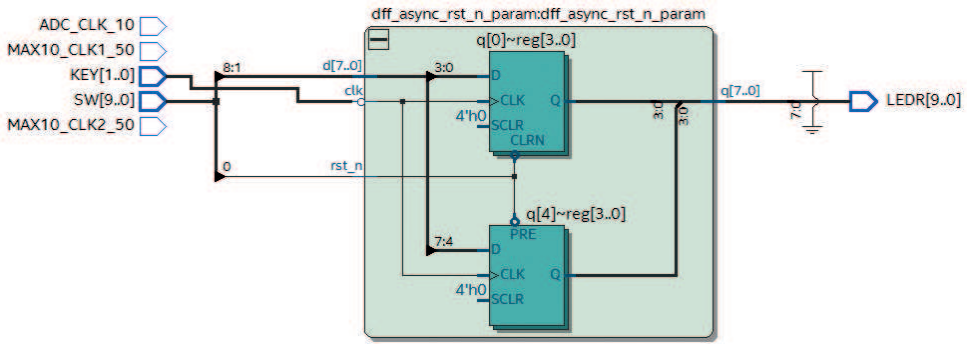


Рис. 2.37 Реализация параметризованного триггера (окно RTL Viewer)

Дополнительное задание для самостоятельной работы

Измените реализацию триггера, приведенную в листинге 2.28. Добавьте положительный сигнал синхронной очистки, асинхронный сброс **n_rst** и сигнал разрешения работы **en**. Сравните аппаратную реализацию (**RTL View**) с рис. 2.37.

2.4 Энергопотребление цифровых устройств

2.4.1 Общие сведения

Компании, производящие электронные компоненты, и университеты вплотную занялись исследованием управления энергопотреблением в середине 1990-х годов, когда начался бум сотовых телефонов. Физики начали проектировать более экономичные транзисторы, а разработчики библиотек стандартных ячеек для микросхем **ASIC** стали делать специальные версии библиотек для повышенного энергосбережения. В программы синтеза были внедрены специальные алгоритмы оптимизации, поддерживающие контроль энергопотребления. Появились также программные решения, позволяющие оценить энергопотребление схемы до ее производства. Даже на уровне программ, выполняемых на процессоре в рамках систем на кристалле (**СнК**), появились способы отключения неиспользуемых аппаратных блоков **СнК** непосредственно программистом.

Энергопотребление цифровой схемы состоит из двух составляющих: **статического** и **динамического** энергопотребления. Статическое энергопотребление связано в основном с выделением мощности на открытом транзисторе и примерно пропорционально количеству транзисторов в схеме. Динамическое энергопотребление связано с количеством переключений транзисторов, во время которых тратится дополнительная энергия.

Подавляющее большинство современных цифровых схем выполнено на базе технологии **КМДП** (**КМОП**, со структурой металл–диэлектрик–полупроводник). Особенностью данной технологии является достаточно низкое энергопотребление в статическом режиме, когда сигналы в схеме не меняют своего состояния. Но при этом происходит существенный рост потребляемой мощности в динамическом режиме (при переключении логических элементов).

Мощность статического энергопотребления определяется формулой:

$$P_{Static} = I_{DD} \cdot V_{DD},$$

где V_{DD} – это напряжение питания, а I_{DD} – суммарная величина тока, который протекает через схему в ее статическом состоянии, когда не происходит изменений никаких сигналов. Этот ток называется **током утечки** (**leakage current**) или **током покоя** (**quiescent supply current**).

Динамическое энергопотребление состоит из затрат энергии на переключение отдельных элементов в схеме и в основном определяется перезарядом емкостей, присутствующих в схеме. Динамическая мощность, потребляемая отдельным элементом, может быть определена по формуле:

$$P = \alpha \cdot C \cdot U^2 \cdot f, \quad (1)$$

где α – коэффициент в диапазоне от **0** до **1**, иллюстрирующий среднее число переключений в течение тактового периода, C – емкость нагрузки, U – напряжение питания, f – частота тактового сигнала. Емкость нагрузки складывается из емкости на транзисторах и емкости на соединениях в системе на кристалле. С конца 1990-х годов в субмикронных схемах влияние емкости соединений существенно возросло, что потребовало создания новых алгоритмов в средствах физического проектирования **ASIC**.

Так как энергопотребление пропорционально квадрату напряжения, самая большая экономия происходит при снижении напряжения, питающего схему. За последние полвека напряжения питания цифровых микросхем непрерывно снижались за счет уменьшения размеров и применения более продвинутых технологий изготовления транзисторов. В 1970–1980-х годах были популярны микросхемы, которые использовали напряжение питания 5 В, потом это напряжение стали снижать: **3.3 В**, **2.8 В**, **1.8 В**, **1.5 В**, **1.2 В**. Самые современные микросхемы, которые применяют технологии **14 нм** и **7 нм**, используют напряжение питания менее 1 В.

Значительной экономии энергопотребления можно достичь, если снизить тактовую частоту, на которой работает схема. Производители микроконтроллеров, как правило, используют процессорные ядра с тактовой частотой существенно ниже максимальной. Например, процессорное ядро **MIPS M5150** может работать на частоте **372 МГц** на технологии **65 LP** и на частоте **576 МГц** на технологии **28 HPM**. При этом компания **Microchip** использует это ядро внутри своего микроконтроллера **PIC32MZ** на максимальной частоте **252 МГц**, а также с целью экономии энергопотребления предоставляет возможность запускать его на гораздо более низких частотах (например, **8 МГц**).

Снижение потребляемой мощности стало популярно не только для микроконтроллеров и сотовых телефонов, но и для высокопроизводительных процессоров и суперкомпьютеров, рост частоты и размера кристалла в которых ограничен проблемой отвода тепла. Именно из-за этого частоты работы процессоров домашних компьютеров сейчас меньше, чем они были в 2000-х годах. При этом снижение частоты компенсируется более эффективной микроархитектурой.

Примерно с 2010 года стали популярны технологии подстройки напряжения и частоты непосредственно во время работы системы, в зависимости от ее условий работы и требуемой производительности. Примером такой технологии является технология **DVFS** (**Dynamic Voltage and Frequency Scaling**), применяемая в смартфонах. Когда смартфон не используется, его процессор может работать на низких частотах при низком потребляемом напряжении. Во время звонка, когда необходимо обрабатывать поток звукового сигнала, происходит переключение на более высокие частоты, а во время просмотра видео на экране частота работы процессора еще выше.

Развитием данной идеи стало использование внутри одной **СнК** нескольких высокопроизводительных ядер, работающих на высокой частоте, и небольших экономичных ядер, работающих на низкой частоте. Примером такой технологии является **ARM big.LITTLE**[1], появившаяся в 2011 году, в рамках которой в экономичном режиме работает небольшой процессор, а высокопроизводительный процессор при этом отключается и «спит» в режиме пониженного энергопотребления (**sleep mode**).

Полное отключение процессорного ядра может контролироваться программой, работающей на **СнК** вплоть до отключения подачи питания к нему. Для реализации такой схемы разработчики **СнК** используют специальные приемы, одним из которых является создание изолированных ячеек (**isolation cells**), представляющих собой специальные устройства, которые позволяют держать часть микросхемы выключенной при сохранении работоспособности остальной части **СнК**.

2.4.2 Разработка структуры устройства с учетом энергопотребления

С точки зрения разработки схемы основной возможностью, помимо описанных выше глобальных приемов, является манипулирование коэффициентом α из формулы (1). Это возможно, например, с помощью перехода к асинхронным автоматам, что, впрочем, требует полной переработки устройства и достаточно аккуратного подхода к анализу функциональности и временных соотношений, так как асинхронные устройства имеют ряд существенных недостатков (тем не менее работы, в основном научные, по данному направлению ведутся, но массового внедрения они не приобрели[2]). Кроме величины потребляемой мощности, проблемой может стать и пиковая мощность, особенно в том случае, когда устройство включает в себя большое количество синхронных блоков. В таких ситуациях практикуется разбиение устройства на отдельные домены с тактовыми сигналами, сдвинутыми по фазе друг относительно друга (**clock domain**). Подобная стратегия позволяет уменьшить пиковое энергопотребление устройства за счет разнесения пиков отдельных блоков во времени. Также в разных тактовых доменах может быть различная тактовая частота.

---

[1] Technologies | big.LITTLE – Arm Developer. (n.d.). Retrieved January 7, 2019, from https://developer.arm.com/ technologies/big-little.

[2] *Ковалев А. В.* Методы оптимизации энергопотребления в микроэлектронных системах: автореф. дис. … док. тех. наук: спец. 05.27.01 «Твердотельная электроника, радиоэлектронные компоненты, микро- и наноэлектроника на квантовых эффектах», 05.13.12 «Системы автоматизации проектирования (по отраслям)». Таганрог, 2009.

При разработке последовательностных устройств снижения энергопотребления можно добиться путем блокирования переключения неиспользуемых в данный момент элементов, как показано на рис. 2.38[1]. Такой прием носит название **gated clock**. Здесь тактовый сигнал подается на триггер только тогда, когда это разрешает специальный разрешающий сигнал **en** (**enable**).

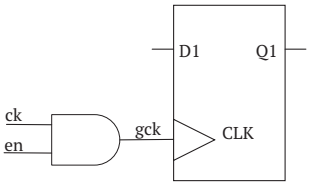


Рис. 2.38 Блокировка тактового сигнала дополнительным сигналом en

Ниже приведена диаграмма, иллюстрирующая такое управление тактовым сигналом.

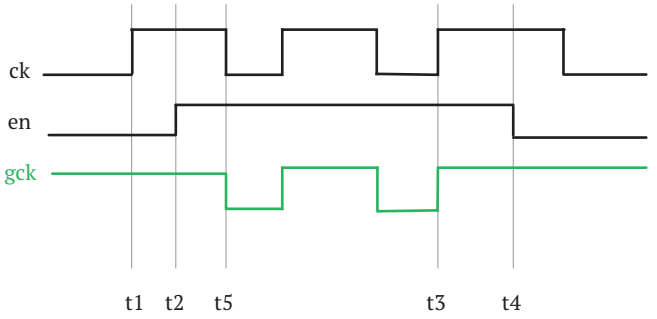


Рис. 2.39 Эпюры сигналов в схеме на рис. 2.38

Недостатком такой схемы является возможное влияние помех на сигнал **en**, как показано на рис. 2.40.

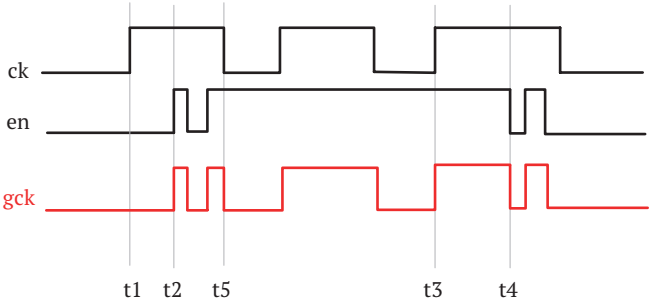


Рис. 2.40 Влияние помех в сигнале en

Кроме помех, возможны также проблемы в случаях, продемонстрированных на рис. 2.41.

[1] *Arar S.* How to Reduce Power Consumption with Clock Gating. Technical article. URL: https://www.allaboutcircuits.com/technical-articles/use-of-clock-gating-to-reduce-power-consumption/.

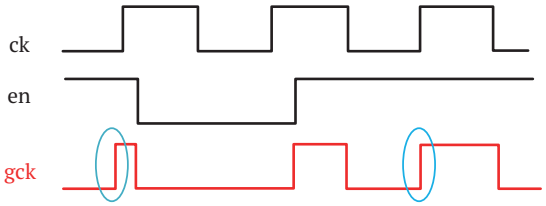
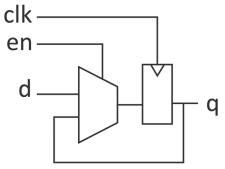


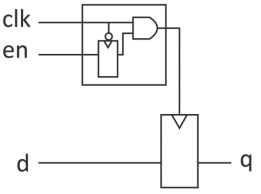
Рис. 41 Ошибочное формирование тактовых импульсов

Важным недостатком такой схемы управления является наличие в цепи тактового сигнала логических элемента, приводящих к появлению дополнительных задержек и разбросу в моментах прихода тактовых сигналов в различных триггерах схемы, что отрицательно влияет на функционирование последовательностных устройств, особенно при высоких частотах тактирования.

При реализации блокировки тактового сигнала, как правило, используют один из двух подходов, представленных на рис. 2.42[1].



а) схема рециркуляции



б) ячейка блокировки

Рис. 2.42 Основные схемы блокировки тактового сигнала

Схемы демонстрируют одинаковые функциональные возможности, при этом схема с рециркуляцией отличается более высоким потреблением ресурсов, а в схеме с ячейкой блокировки присутствует задержка в цепи тактового сигнала[2].

Следует отметить, что схема на рис. 2.42б, как правило, устанавливается в цепи тактового сигнала, тактирующего несколько триггеров, при этом при неактивном сигнале **en** тактовый сигнал воздействует только на входы ячейки блокировки и не распространяется дальше, что и приводит к снижению энергопотребления. При разработке цифровых интегральных микросхем инженеры применяют специальные программные инструменты, которые анализируют схему устройства и автоматически вставляют в них такие ячейки, когда встречают конструкцию на языке **Verilog** следующего типа:

```
always @(posedge clk)
   if (EN)
      D_out <= D_in
```

Листинг 2.29 Конструкция на языке Verilog, синтезируемая в ячейку блокировки

[1] *Narayana Koduri, Kiran Vittal.* (n.d.). Power analysis of clock gating at RTL. Retrieved January 6, 2019, from https://www.design-reuse.com/articles/23701/power-analysis-clock-gating-rtl.html.

[2] *Kathuria J., Ayoubkhan M., Noor A.* A review of clock gating techniques // MIT International Journal of Electronics and Communication Engineering, 2011, 1 (2), 106–114.

Для оценки итогового энергопотребления оптимизированной схемы существуют специальные программы (например, **Synopsys PrimeTime PX**), оценивающие динамическое энергопотребление с помощью комбинации данных о паразитных емкостях, полученных после синтеза, с данными о переключениях, полученных при симуляции синтезированного кода вместе с тестовым окружением.

Эти же соображения принимаются во внимание разработчиками цифровых устройств на базе **ПЛИС**, при этом основное внимание уделяется, как описано выше, блокировке воздействия тактового сигнала на не используемые в данный момент триггеры. Стоит отметить, что если на входе устройства действуют быстроменяющиеся информационные сигналы, то некоторые вентили заблокированного устройства могут по-прежнему переключаться и вносить свой вклад в общее энергопотребление. Снижения энергопотребления можно добиться переходом на защелки (вместо триггеров, управляемых фронтом или спадом сигнала), применением блокировки тактового сигнала с помощью специальных элементов **ПЛИС**, предназначенных для работы с тактовым сигналом, и др. Следует отметить, что применение блокировки тактового сигнала часто дает лучший результат по сравнению с блокировкой триггеров с помощью сигналов типа **clock enable**, предусмотренных схемами логических блоков **ПЛИС**. Наилучшей стратегией является – обращать внимание на энергопотребление еще на этапе проектирования и использовать специальные инструменты, предусмотренные в составе САПР для разработки цифровой техники на базе **ПЛИС**. Энергопотребление цифровых устройств на базе **ПЛИС** может также зависеть от дополнительных специфических факторов, например от топологии устройства (длины соединительных трасс на кристалле), что также требует внимания разработчика.

Для решения проблем с теплоотводом рекомендуется, с одной стороны, выбирать **ПЛИС** с как можно низким напряжением питания и наименьшим количеством элементов, а с другой – с наибольшим размером корпуса, снижающим тепловое сопротивление.

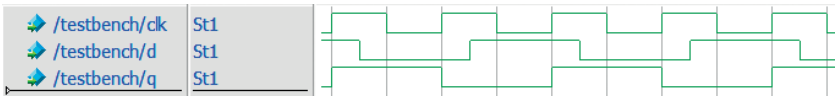
2.5 Упражнения

2.5.1 Основное задание

Выполните все дополнительные задания данной главы.

2.5.2 Контрольные вопросы

1. Форма сигналов какого типа последовательностных схем представлена ниже:



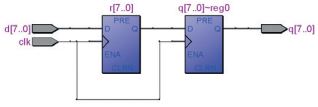
   а) асинхронный **RS-триггер**;
   б) **D-триггер**;
   в) **D-триггер** типа защелка;
   г) активируемый **D-триггер**.

2. Тестовый модуль, использованный для создания испытательных сигналов, выглядит следующим образом:

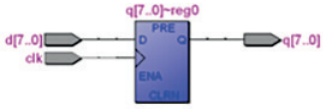
```
module testbench;
   logic clk;
   logic [7:0] d, q;
   dut dut (clk, d, q);
   initial
   begin
      clk = 0;

      forever
         #10 clk = ! clk;
      end
      initial
         begin
            $dumpvars ();
            for (int i = 0; i < 10; i++)
            begin
               @(posedge clk);
               #15;
               d = i;
            end
            $finish;
         end
endmodule
```

Поставьте в соответствие каждой схеме

A)



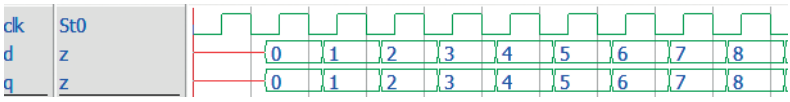
B)



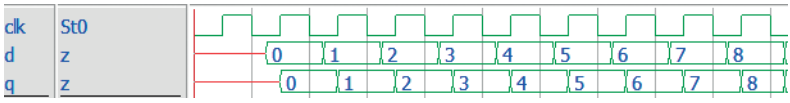
C)

полученную форму сигналов:

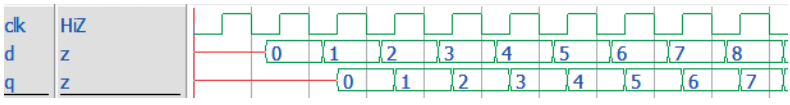
X)



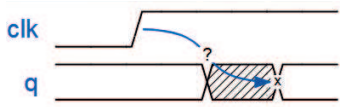
Y)



Z)



3. Какой тип задержки изображен на рисунке ниже:



   а) задержка распространения: $\mathbf{t}_{pd}$ = максимальная задержка от входа к выходу;
   б) задержка реакции: $\mathbf{t}_{cd}$ = минимальная задержка от входа к выходу;
   в) время удержания;
   г) время установки.

4. Опишите основные отличия синхронных триггеров от асинхронных. Охарактеризуйте их основные преимущества и недостатки.

5. Что такое запрещенная комбинация сигналов? Почему состояние **RS-триггера** при этой комбинации входных сигналов называется неопределенным?

6. В чем состоит преимущество **D-триггера** перед **D-защелкой**?

7. В чем отличие схемы мастер-помощник от схемы с использованием бистабильных ячеек при построении **D-триггера**?

8. Что означает параметр время установки для последовательностных устройств?

9. Что означает параметр время удержания для последовательностных устройств?

10. Почему **JK-триггер** называют универсальным триггером?

11. Разработайте схему **T-триггера** на основе **D-триггера**.

12. Опишите особенности применения оператора непрерывного присваивания в языке **Verilog**.

13. В чем отличие блокирующего присваивания от неблокирующего в языке **Verilog**?

14. В чем отличие синхронного сброса от асинхронного? Приведите пример реализации синхронного сброса на языке **Verilog**.

15. Опишите назначение и примеры применения ключевого слова **parameter** в языке **Verilog**.

16. От чего зависит величина мощности, потребляемой логическим элементом в динамическом режиме?

17. Что такое **ячейка блокировки**? Для чего она применяется?

18. В чем состоит проблема использования блокировки тактового сигнала (**gated clock**)?

Александр Барабанов, Александр Романов

Цифровой синтез: практический курс

Глава 3. Шифраторы и дешифраторы. Скорость работы комбинационных блоков

Содержание

В этой главе рассмотрены назначение, описание, синтез и моделирование работы одних из самых распространенных комбинационных цифровых устройств – шифраторов и дешифраторов. Также рассмотрены временны́е характеристики таких блоков и методы их анализа с использованием средств ***Quartus Prime****.*

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет* ***Quartus Prime*** *(есть студенческая версия);*
- *пакет* ***ModelSim Altera Edition*** *или программы* ***Icarus Verilog и GTKWave****;*
- *отладочная плата компании* ***Terasic DE10Lite*** *или другая отладочная плата на основе* ***ПЛИС Intel FPGA*** *или* ***Xilinx*** *(может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах*[1] *к данной книге).*

3.1 Шифраторы

Шифраторы и **дешифраторы** являются кодирующими устройствами, т. е. устройствами, преобразующими входной многоразрядный код в выходной, который построен по иному правилу.

В табл. 3.1 приведена таблица истинности классического **неприоритетного шифратора** (или **кодера**, **encoder**), который принимает на вход 16-разрядный унарный код (логическая единица может быть только в одном разряде или не быть вообще, если все разряды равны нулю), а на выходе формирует 4-разрядный двоичный код, который соответствует положению логической единицы на входе. Подача нескольких логических единиц на вход неприоритетного шифратора запрещена.

Кроме неприоритетного шифратора используются **приоритетный шифратор**, на вход которого можно подавать произвольный код, а шифратор формирует двоичный код, который соответствует положению старшей (или младшей) логической единицы на входе. Например, если на вход приоритетного шифратора с приоритетом младшего бита подать **0110**, то на его выходе будет сформирован сигнал **0010**, а если этот же код подать на вход приоритетного шифратора с приоритетом старшего бита – то **0100**. Такие шифраторы будут рассмотрены ниже.

Часто шифраторы имеют вход разрешения (**enable**): если он активен, то шифратор выполняет свои функции, если нет – выдает на выход нулевое значение.

Шифраторы используются для кодирования состояния переключателей, нажатых клавиш, датчиков (уровня, угла) и т. д. Приоритетные шифраторы используются в контроллерах прерываний микропроцессорных систем, на которые одновре-

[1] https://github.com/RomeoMe5/DDLM.

менно может подаваться несколько запросов прерываний, но в каждый момент времени обрабатываться может только прерывание с наибольшим приоритетом.

Таблица 3.1 Таблица истинности неприоритетного шифратора

Код на входе шифратора (унитарный)	Двоичный код на выходе
0000000000000000	0000
0000000000000001	0000
0000000000000010	0001
0000000000000100	0010
0000000000001000	0011
0000000000010000	0100
0000000000100000	0101
0000000001000000	0110
0000000010000000	0111
0000000100000000	1000
0000001000000000	1001
0000010000000000	1010
0000100000000000	1011
0001000000000000	1100
0010000000000000	1101
0100000000000000	1110
1000000000000000	1111

Дополнительное задание для самостоятельной работы

Разработайте принципиальную схему неприоритетного шифратора, таблица истинности которого приведена в табл. 3.1. Используйте логические элементы **И-НЕ**, **ИЛИ-НЕ**.

3.2 Описание шифраторов на языке Verilog

Наиболее общие рекомендации при описании **шифраторов** (как и других устройств комбинационной логики) можно сформулировать следующим образом.

1. Описание шифратора должно легко поддаваться отладке.
2. Каждый шифратор должен быть описан в отдельном модуле. Это дает возможность легко читать код и упрощает синтез.
3. Шифратор является комбинационным цифровым устройством, его следует описывать с использованием операторов непрерывного присваивания **assign** и **always**-блоков с блокирующим присваиванием =.

3.2.1 Неприоритетный шифратор на основе оператора непрерывного присваивания assign

Классический неприоритетный шифратор может быть построен на базе таблицы истинности (например, табл. 3.1). В листинге 3.1 приведено описание такого шифратора с использованием оператора непрерывного присваивания **assign**. Рассматриваемый шифратор имеет вход разрешения **enable**. Для каждого разряда выходного кода в явном виде записана соответствующая логическая функция.

```
module b1_enc_assign(
   input [15:0] in,
   output [3:0] binary_out,
   input enable
);
   assign binary_out[0] = (in[1] | in[3] | in[5] | in[7] | in[9] |
      in[11] | in[13] | in[15]) & enable;

   assign binary_out[1] = (in[2] | in[3] | in[6] | in[7] | in[10] |
      in[11] |   in[14] | in[15]) & enable;

   assign binary_out[2] = (in[4] | in[5] | in[6] | in[7] | in[12] |
      in[13] | in[14] | in[15]) & enable;

   assign binary_out[3] = (in[8] | in[9] | in[10] | in[11] | in[12]
      | in[13] | in[14] | in[15]) & enable;
endmodule
```

Листинг 3.1 Неприоритетный шифратор на основе оператора непрерывного присваивания assign

Из листинга 3.1 следует, что младший бит **binary_out[0]** равен логической единице, если она присутствует в любом четном бите (**2**, **4**, **6**, **8**, **10**, **12**, **14**, **16**) на входе (индекс элемента входного вектора нечетный, т. к. нумерация начинается с 0). Следующий бит **binary_out[1]** равен логической единице, если она присутствует в любой из четырех пар входных битов (**3**, **4**; **7**, **8**; **11**, **12**; **15**, **16**), **binary_out[2]** – если единица присутствует в одном из двух блоков по 4 бита (**5**, **6**, **7**, **8**; **13**, **14**, **15**, **16**), а **binary_out[3]** – в блоке из 8 битов (**9**, **10**, **11**, **12**, **13**, **14**, **15**, **16**). Можно сделать вывод, что такой шифратор корректно работает только при подаче на вход унарного кода.

Рассматриваемый шифратор является самым простым с точки зрения построения схемы, он состоит из небольшого количества элементарных логических элементов (**И**, **ИЛИ**) и имеет малые задержки прохождения сигналов (рис. 3.1).

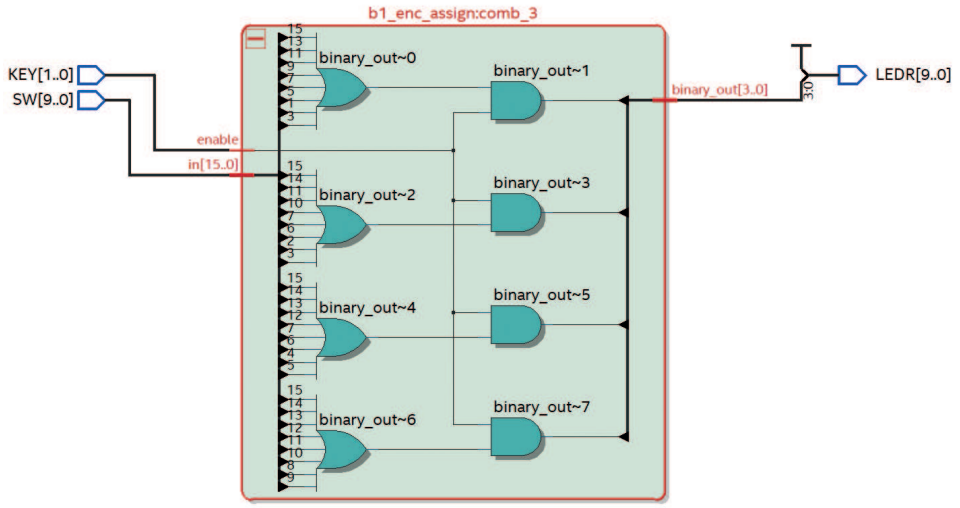


Рис. 3.1 Структура неприоритетного шифратора на основе оператора непрерывного присваивания assign в RTL Viewer

Результаты моделирования полученного модуля средствами **Quartus Prime** приведены на рис. 3.3.

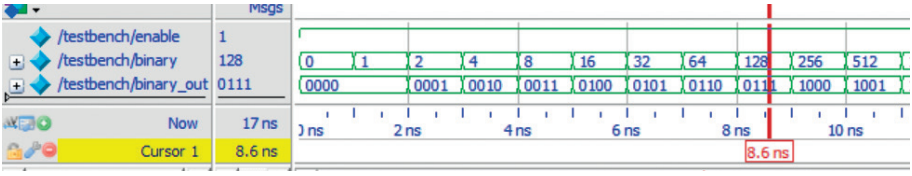


Рис. 3.2 Результаты моделирования работы неприоритетного шифратора на основе оператора непрерывного присваивания assign

3.2.2 Неприоритетный шифратор, реализованный с помощью операторов if

Неприоритетный шифратор можно реализовать в виде **always**-блока с несколькими операторами **if** и блокирующими присваиваниями (=) в них.

Код, приведенный в листинге 3.2, реализует функцию шифратора «в лоб» и поэтому прост для понимания.

```
module b2_enc_if(
   input [15:0] encoder_in,
   output reg [3:0] binary_out,
   input enable
);
   always @ (*)
      begin
         binary_out = 0;
```

```
        if (enable) begin
           if (encoder_in == 16'h1) begin
              binary_out = 0;
           end if (encoder_in == 16'h2) begin
              binary_out = 1;
           end if (encoder_in == 16'h4) begin
              binary_out = 2;
           end if (encoder_in == 16'h8) begin
              binary_out = 3;
           end if (encoder_in == 16'h10) begin
              binary_out = 4;
           end if (encoder_in == 16'h20) begin
              binary_out = 5;
           end if (encoder_in == 16'h40) begin
              binary_out = 6;
           end if (encoder_in == 16'h80) begin
              binary_out = 7;
           end if (encoder_in == 16'h100) begin
              binary_out = 8;
           end if (encoder_in == 16'h200) begin
              binary_out = 9;
           end if (encoder_in == 16'h400) begin
              binary_out = 10;
           end if (encoder_in == 16'h800) begin
              binary_out = 11;
           end if (encoder_in == 16'h1000) begin
              binary_out = 12;
           end if (encoder_in == 16'h2000) begin
              binary_out = 13;
           end if (encoder_in == 16'h4000) begin
              binary_out = 14;
           end if (encoder_in == 16'h8000) begin
              binary_out = 15;
           end
        end
     end
endmodule
```

Листинг 3.2 Неприоритетный шифратор, реализованный с помощью операторов if

Следует обратить внимание на то, что выходной сигнал **binary_out** определен как регистр (**reg**). Это позволяет изменять значение этого сигнала явно, т. е. использовать процедурное назначение (**procedural assignment**). В современном языке **SystemVerilog** в аналогичных описаниях используется универсальный тип **logic**.

Достоинство данного шифратора раскрывается тогда, когда на вход подается не унарный код (т. е. на входе будет несколько логических единиц). В этом случае

ни одно из условий **if** не будет выполнено и на выходе будет выставлено значение **0000**.

Результаты синтеза такого описания приведены на рис. 3.3. Следует отметить, что количество логических элементов у такого шифратора намного больше, чем у предыдущего, а сама структура его более громоздка. Кроме того, мультиплексоры, реализующие функции **if**, соединены последовательно, что приводит к большой задержке прохождения сигнала.

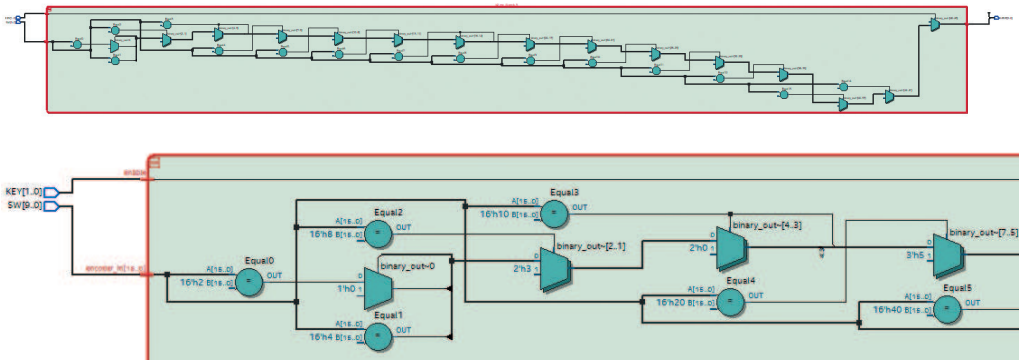


Рис. 3.3 RTL-представление неприоритетного шифратора, реализованного с помощью операторов if

3.2.3 Неприоритетный шифратор, реализованный с помощью оператора case

Вместо вложенных операторов **if** в **неприоритетном шифраторе** рекомендуется использовать оператор **case**. Такой подход более нагляден, снижает вероятность возникновения ошибок в коде и обеспечивает лучшую его читаемость. Код (листинг 3.3) и результаты синтеза шифратора **16в4** (рис. 3.4) приведены ниже. Схемы сравнения в правой части рисунка соответствуют отдельным ветвям оператора **case**. Следует отметить, что это самый распространенный подход к описанию шифраторов.

```
module b3_enc_case(
   input [15:0] encoder_in,
   output reg [3:0] binary_out,
   input enable
);
always @ (*)
   begin
      binary_out = 0;
      if (enable) begin
         case (encoder_in)
            16'h1 : binary_out = 0;
            16'h2 : binary_out = 1;
            16'h4 : binary_out = 2;
```

```
            16'h8 : binary_out = 3;
            16'h10 : binary_out = 4;
            16'h20 : binary_out = 5;
            16'h40 : binary_out = 6;
            16'h80 : binary_out = 7;
            16'h100 : binary_out = 8;
            16'h200 : binary_out = 9;
            16'h400 : binary_out = 10;
            16'h800 : binary_out = 11;
            16'h1000 : binary_out = 12;
            16'h2000 : binary_out = 13;
            16'h4000 : binary_out = 14;
            16'h8000 : binary_out = 15;
        endcase
    end
  end
endmodule
```

Листинг 3.3 Неприоритетный шифратор, реализованный с помощью оператора case

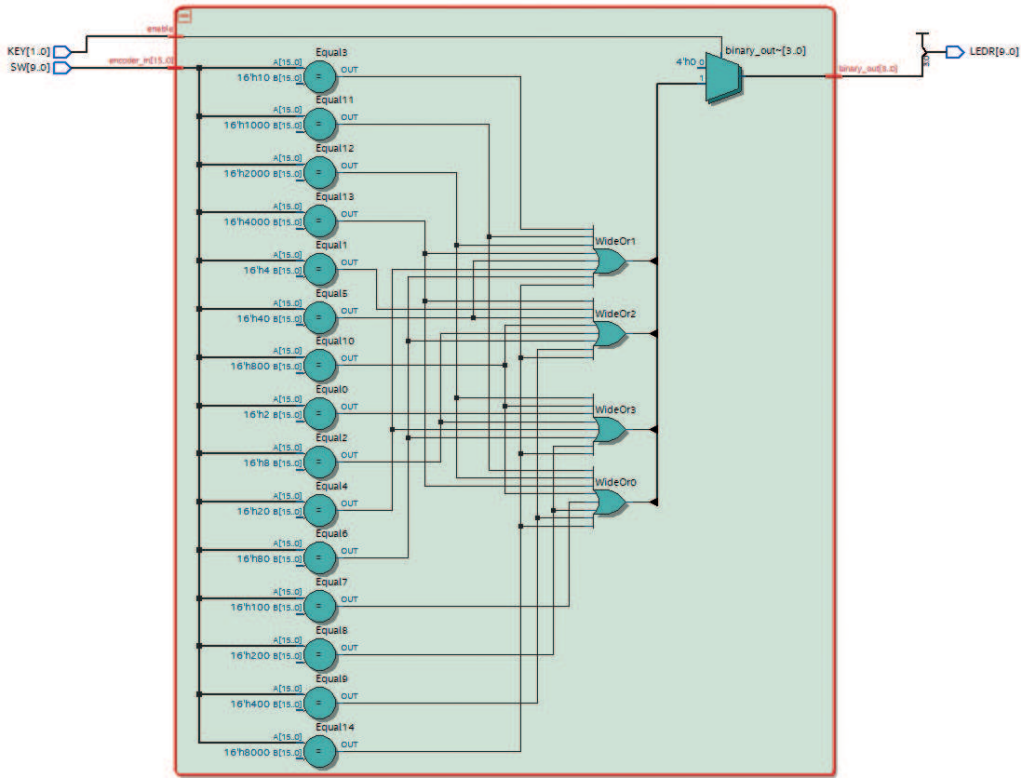


Рис. 3.4 Структура неприоритетного шифратора, реализованного с помощью оператора case в RTL Viewer

3.2.4 Приоритетный шифратор, реализованный с помощью операторов if

На вход **приоритетного шифратора**, в отличие от неприоритетного, можно одновременно подавать несколько логических единиц. В зависимости от приоритетности он вернет номер старшей единицы (или младшей, как в данном примере). Для разработки такого шифратора можно использовать подход, рассмотренный в разделе 3.2.1, при этом основное отличие состоит в наличии блока **else** у операторов **if** и в использовании вложенных операторов **if**. Таким образом, реализуется ветвление, благодаря чему будет выбран самый первый вариант, для которого логическое условие будет истинным. Порядок проверки разрядов входного сигнала в операторах **if** (от младшего к старшему или от старшего к младшему) определяет приоритет младшей или старшей единицы. Код **приоритетного шифратора** приведен в листинге 3.4. Результатом синтеза (рис. 3.5) является схема, состоящая из последовательно соединенных мультиплексоров, каждый из которых соответствует отдельному оператору **if**.

```
module b4_pri_enc_if(
   input [15:0] encoder_in,
   output reg [3:0] binary_out,
   input enable
);
always @ (*)
   begin
      binary_out = 0;
      if (enable) begin
         if (encoder_in[0] == 1'b1)
            binary_out = 0;
         else if (encoder_in[1] == 1'b1)
            binary_out = 1;
         else if (encoder_in[2] == 1'b1)
            binary_out = 2;
         else if (encoder_in[3] == 1'b1)
            binary_out = 3;
         else if (encoder_in[4] == 1'b1)
            binary_out = 4;
         else if (encoder_in[5] == 1'b1)
            binary_out = 5;
         else if (encoder_in[6] == 1'b1)
            binary_out = 6;
         else if (encoder_in[7] == 1'b1)
            binary_out = 7;
         else if (encoder_in[8] == 1'b1)
            binary_out = 8;
         else if (encoder_in[9] == 1'b1)
            binary_out = 9;
         else if (encoder_in[10] == 1'b1)
```

```
            binary_out = 10;
         else if (encoder_in[11] == 1'b1)
            binary_out = 11;
         else if (encoder_in[12] == 1'b1)
            binary_out = 12;
         else if (encoder_in[13] == 1'b1)
            binary_out = 13;
         else if (encoder_in[14] == 1'b1)
            binary_out = 14;
         else if (encoder_in[15] == 1'b1)
            binary_out = 15;
      end
   end
endmodule
```

Листинг 3.4 Приоритетный шифратор, реализованный с помощью операторов if

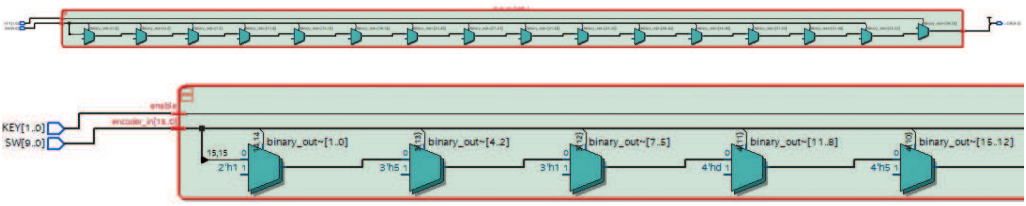


Рис. 3.5 Структура приоритетного шифратора, реализованного с помощью операторов if в RTL Viewer

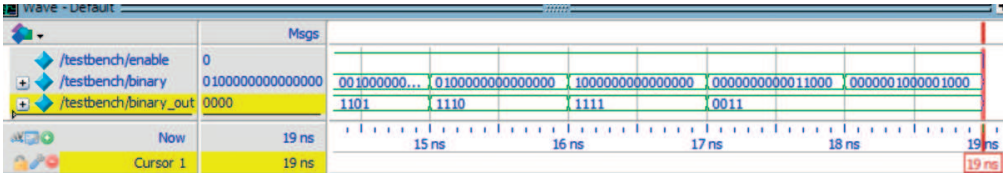


Рис. 3.6 Моделирование работы приоритетного шифратора, реализованного с помощью операторов if

Временна́я диаграмма, приведенная на рис. 3.6, иллюстрирует работу **приоритетного шифратора** с приоритетом младшей единицы (следует обратить внимание на интервал времени 17–19 нс).

3.2.5 Приоритетный шифратор с использованием оператора assign

В данном варианте шифратора (листинг 3.4) используется один оператор **assign** и несколько вложенных тернарных операторов (**?:**), которые являются аналогами оператора **if**.

```
module b5_pri_enc_assign(
   input [15:0] encoder_in,
   output [3:0] binary_out,
```

```
   input enable
);
assign binary_out = (enable) ? (
   (encoder_in[15] == 1'b1) ? 15 :
   (encoder_in[14] == 1'b1) ? 14 :
   (encoder_in[13] == 1'b1) ? 13 :
   (encoder_in[12] == 1'b1) ? 12 :
   (encoder_in[11] == 1'b1) ? 11 :
   (encoder_in[10] == 1'b1) ? 10 :
   (encoder_in[9] == 1'b1) ? 9 :
   (encoder_in[8] == 1'b1) ? 8 :
   (encoder_in[7] == 1'b1) ? 7 :
   (encoder_in[6] == 1'b1) ? 6 :
   (encoder_in[5] == 1'b1) ? 5 :
   (encoder_in[4] == 1'b1) ? 4 :
   (encoder_in[3] == 1'b1) ? 3 :
   (encoder_in[2] == 1'b1) ? 2 :
   (encoder_in[1] == 1'b1) ? 1 :
   (encoder_in[0] == 1'b1) ? 0 : 4'bxxxx):0;
endmodule
```

Листинг 3.5 Приоритетный шифратор с использованием оператора assign

Структура такого шифратора и результаты его моделирования точно такие же, как и у приоритетного шифратора на основе операторов **if**, который был рассмотрен в предыдущем разделе (рис. 3.5, 3.6).

3.2.6 Параметрический шифратор

Удобство рассматриваемого в данном разделе подхода состоит в том, что для изменения количества разрядов шифратора нет необходимости переписывать код, а достаточно изменить единственный параметр. Для этого используется оператор **while**. В листинге 3.6 приведен код такого приоритетного шифратора. Переменная **i** инкрементируется от **0** до обнаружения первой логической единицы. Соответствующее значение **i** поступает на выход шифратора. Данный код приводит к синтезу эффективной схемы (рис. 3.7), параметры которой приведены в табл. 3.3.

```
module b12_anybit_enc (in, enc_out, enable);
   parameter OUT_SIZE = 4;
   // configure this parameter only to get amount of output bits
   parameter IN_SIZE = 1<<OUT_SIZE;
   // input bits are calculated like 2^(output bits)

   input wire [IN_SIZE-1:0] in;
   output wire [OUT_SIZE-1:0] enc_out;

   reg [OUT_SIZE-1:0] out;
   assign enc_out = out;
   input enable;
```

```
   integer i;
   always @(in) begin
      if(enable) begin
         i=0;
         while (i<IN_SIZE-1 && !in[i]) i=i+1;
            out <= i;
      end else out<=0;
   end
endmodule
```

Листинг 3.6 Параметрический шифратор

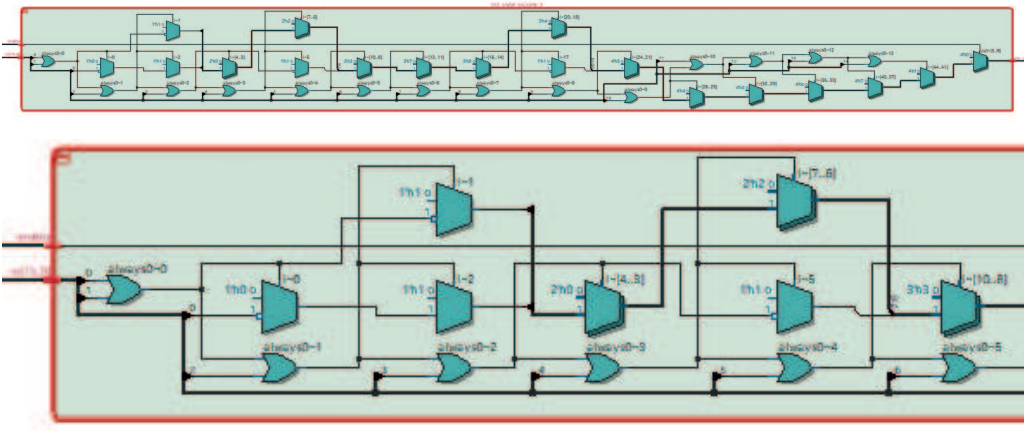


Рис. 3.7 Параметрический шифратор

3.2.7 Изучение временны́х характеристик цифровых устройств

Все цифровые устройства состоят из транзисторов, сопротивлений, емкостей и индуктивностей (возможно, паразитных). Изменение напряжения и тока в таких электронных схемах происходит с конечной скоростью, определяемой постоянными времени **RC** и **LR** цепей. Таким образом, изменение выходного сигнала схемы происходит с некоторой задержкой после изменения входного сигнала, а схема имеет конечное быстродействие.

Цифровые схемы характеризуются **задержкой распространения** (**propagation delay**, $\mathbf{t_{pd}}$) и **задержкой реакции** или отклика (**contamination delay**, $\mathbf{t_{cd}}$). Задержка распространения tpd – это максимальное время от начала изменения входного сигнала схемы до момента, когда все ее выходы достигнут своих стационарных состояний. **Задержка реакции** $\mathbf{t_{cd}}$ – это минимальное время от момента, когда входной сигнал изменился, до момента, когда любой из выходов начнет менять свое значение. Таким образом, выходной сигнал схемы может начать изменяться через время $\mathbf{t_{cd}}$ после изменения входного сигнала (но не раньше), и он точно примет новое стационарное значение не позднее, чем через время $\mathbf{t_{pd}}$ после изменения входного сигнала.

Когда разработчики говорят о задержке схемы, они в большинстве случаев имеют в виду наибольшее возможное значение задержки распространения. Конкретные числовые значения задержек распространения и реакции цифрового устройства определяются внутренней структурой схемы и технологией ее реализации. Задержка схемы обусловлена двумя физическими процессами, такими как:

- переключение логических элементов, из которых состоит схема;
- распространение сигналов через линии связи, соединяющие эти элементы.

При использовании **FPGA** часто второй процесс более медленный. Поэтому именно он обычно ограничивает быстродействие всего устройства.

Как отдельный логический элемент, так и сложная комбинационная логическая схема характеризуются задержками распространения и реакции. Конкретные значения этих величин определяются не только временны́ми характеристиками отдельных элементов, но и путем, который проходит сигнал от входа схемы до ее выхода. **Критический путь** комбинационной логической схемы (**critical path**) соответствует участку от входа до выхода с наибольшей задержкой и является самым медленным. Именно он ограничивает скорость, с которой работает схема. **Задержка распространения комбинационной схемы** является суммой задержек распространения всех элементов **критического пути**.

В современной цифровой электронике, как правило, используются синхронные последовательностные схемы, в которых цифровые комбинационные блоки размещены между регистрами, тактируемыми сигналами синхронизации (рис. 3.8). По фронту импульса синхронизации в первый регистр записывается новое значение. После этого оно появляется на его выходе, и в комбинационном блоке начинается переходный процесс, заканчивающийся установлением новых стационарных значений на его выходе. Для корректной работы последовательностной схемы критически важно, чтобы этот переходный процесс успел завершиться до поступления следующего импульса синхронизации, по которому значения выходных сигналов комбинационного блока будут записаны во второй регистр. Только при выполнении этого условия в регистр будут записаны корректные стационарные значения выходных сигналов комбинационного блока. Именно наибольшая задержка (обусловленная критическим путем) комбинационного блока ограничивает период сигнала синхронизации (снизу) и его частоту (сверху).

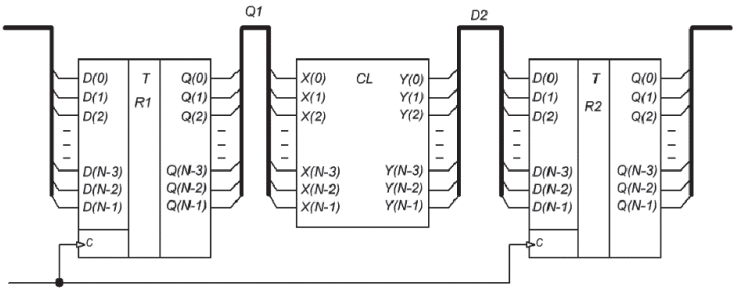


Рис. 3.8 Типичная структура цифровой последовательностной схемы

Для анализа **временны́х характеристик** (поиска критического пути, определения максимальной задержки) в **Quartus Prime** применяется средство **TimeQuest**. В данной работе будут исследованы **временны́е характеристики неприоритетного шифратора**.

Для использования **TimeQuest** для определения временны́х характеристик комбинационного блока необходимо задать:

- тактовый сигнал;
- входной регистр;
- выходной регистр.

Рисунок 3.9 и листинг 3.7 демонстрируют реализацию данного подхода. Изучаемый комбинационный блок размещен между двумя регистрами, тактируемыми сигналом синхронизации **clock**.

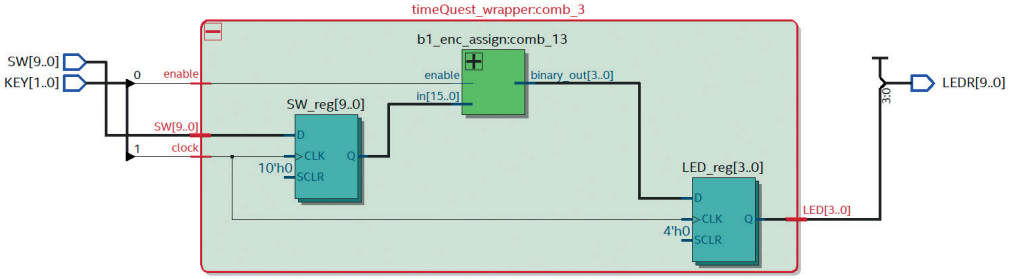


Рис. 3.9 Модуль верхнего уровня иерархии, используемый для определения задержек

```
module timeQuest_wrapper(clock,SW,LED,enable);
   input clock,enable;
   input [9:0] SW;
   output [3:0] LED;

   reg [9:0] SW_reg;
   reg [3:0] LED_reg; // registers for ‘catching’ time

   wire [3:0] LED_wire;
   wire [9:0] SW_wire;
   assign SW_wire = SW_reg;

   // creating our test instance
   b1_enc_assign (SW_reg,LED_wire,enable);

   // clock needed to determine at which step register was filled with data
   always @(posedge clock)
      begin
         SW_reg <= SW;
         // avoiding race and latch by setting ‘<=’ instead of ‘=’
         LED_reg <= LED_wire;
```

```
      end
   assign LED = LED_reg;
endmodule
```

Листинг 3.7 Модуль верхнего уровня иерархии для выполнения временного анализа

Откройте файл **1_lab3_hdl_4bit_enc_assign\lab3.v**, раскомментируйте модуль **timeQuest_wrapper** и закомментируйте модуль **b1_enc_assign** в конце файла (как показано в листинге 3.8), выполните синтез проекта (**Ctrl**+**L**). Далее выберите в меню **Tools → TimeQuest Timing Analyser** (рис. 3.10). Перейдите на вкладку **Tasks** (рис. 3.11) и сделайте двойной клик на **Report Clocks** (рис. 3.12). На выбранном тактовом сигнале нажмите **Report Timing** (рис. 3.13).

```
module lab3
(
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
   // b1_enc_assign (SW,LEDR,KEY[0]);
   // Please comment the line above and uncomment line below
   // to use timeQUest_wrapper
   timeQuest_wrapper(KEY[1], SW, LEDR, KEY[0]);
endmodule
```

Листинг 3.8 Необходимые изменения в коде для выполнения временно́го анализа

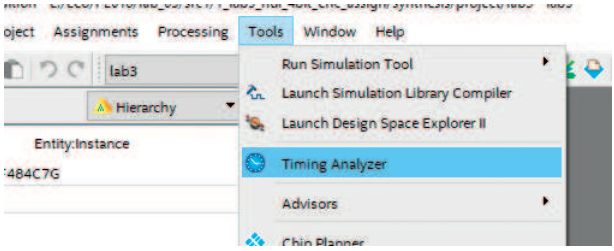


Рис. 3.10 Вызов модуля временно́го анализа (Timing Analyzer)

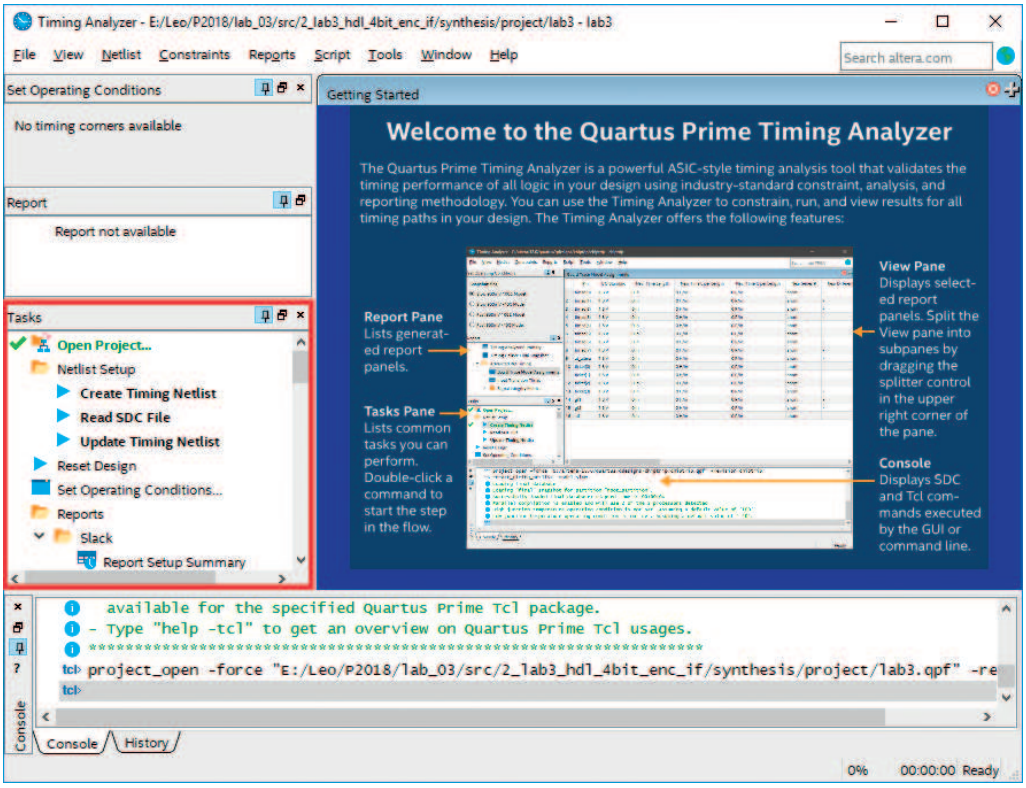


Рис. 3.11 Задачи (Tasks) в Timing Analyzer

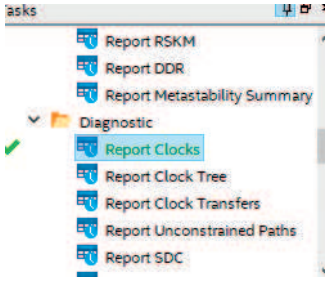


Рис. 3.12 Запуск анализа тактовых сигналов (Report Clocks)

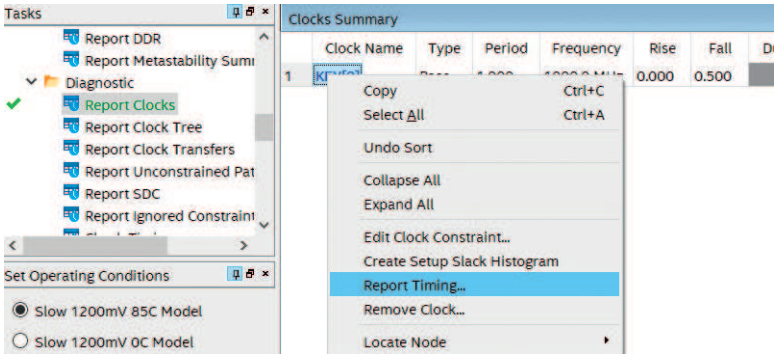


Рис. 3.13 Запуск генерации отчета в TimeQuest

Средство **TimeQuest** предлагает опционально выбрать различные тактирующие импульсы. Для путей прохождения сигналов следует оставить значения по умолчанию и нажать **Report Timing**. Будет проанализировано **10** самых медленных путей. 1-й путь является критическим, и именно он ограничивает быстродействие всего блока.

В окне результатов временного анализа (рис. 3.14) приведены следующие временны́е параметры изучаемой схемы:

- **Clock Delay** – задержка прохождения тактового сигнала от вывода **ПЛИС** (в нашем случае это вход **KEY [1]**) до тактового входа регистра **SW_reg**. Эта задержка не зависит от тактовой частоты;
- **Data Delay** – время прохождения данных от входного регистра модуля **time Quest_wrapper** до выходного;
- **Data Arrival** – время поступления данных на выходной регистр модуля time Quest_wrapper, Data Arrival = Clock Delay + Data Delay;
- **Data Required** – наибольшее допустимое время прибытия данных. При превышении этого параметра корректная работа схемы не гарантирована, временной анализ выдает ошибку;
- **Slack** – разница между Data Required и Data Arrival.

Из отчета, приведенного на рис. 3.14, следует, что наибольшая задержка на пути **SW_reg [3] -> LED_reg [1]** равна **1,603 нс**. Также можно отметить, что данные на выходной регистр приходят с опозданием, т. е. переходные процессы в шифраторе не успевают завершиться до поступления следующего тактового импульса. Запас по времени (интервал между завершением переходного процесса в комбинационном блоке и моментом записи значений его выходных сигналов с регистра) отрицательный и составляет **–0,631 нс** (выделен красным цветом), т. е. он отсутствует. Из отчета **Path Summary** (рис. 3.15) следует, что необходимое время поступления данных на регистр равно **3,774 нс**, а реальное время поступления данных составляет **4,405 нс**. Следовательно, с такой тактовой частотой (по умолчанию она равна **1 ГГц** с периодом следования сигнала, равным **1 нс**; рис. 3.16) устройство не будет работать корректно, и требуется уменьшить тактовую частоту.

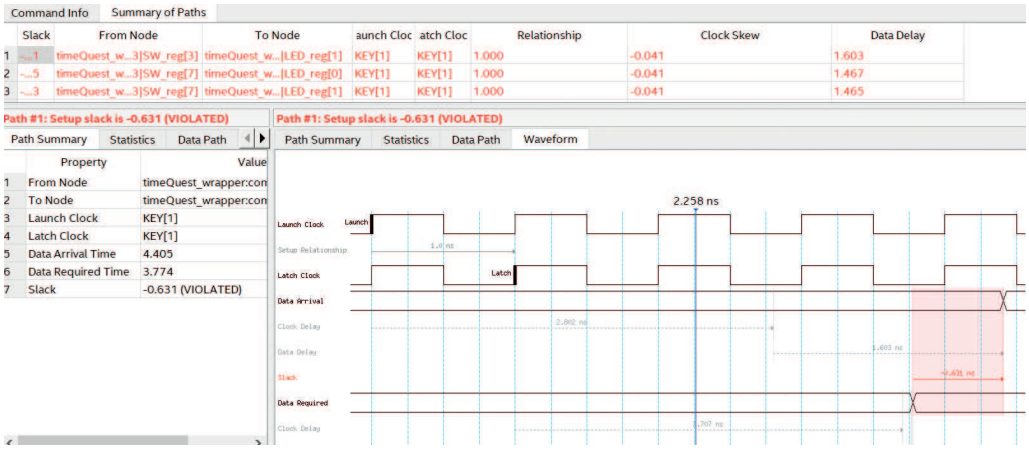
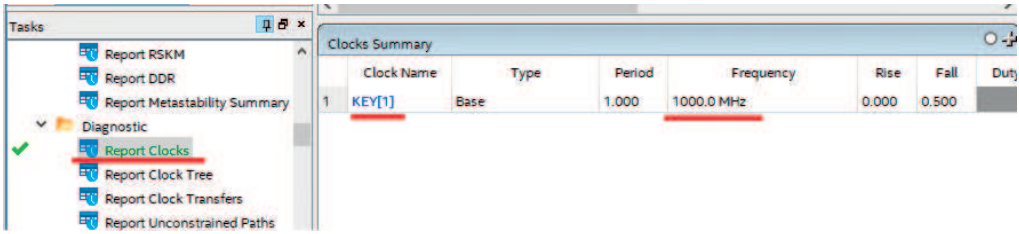


Рис. 3.14 Результаты временно́го анализа

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	timeQuest_wrapper:comb_3|SW_reg[3]
2	To Node	timeQuest_wrapper:comb_3|LED_reg[1]
3	Launch Clock	KEY[1]
4	Latch Clock	KEY[1]
5	Data Arrival Time	4.405
6	Data Required Time	3.774
7	Slack	-0.631 (VIOLATED)

Рис. 3.15 Обзор критического пути Path Summary



3.16 Обзор Clock Constraint

Выберите пункт меню **Edit Clock Constraint** (рис. 3.17). По умолчанию установлен период сигнала синхронизации **1 нс**, что соответствует частоте **1 ГГц**. Период можно увеличить до **2 нс**, таким образом установив частоту в **500 МГц**. После этого надо повторить временно́й анализ. В случае, приведенном на рис. 3.18, временно́й анализ успешен, а запас по времени – положительный и равен **0,369 нс**.

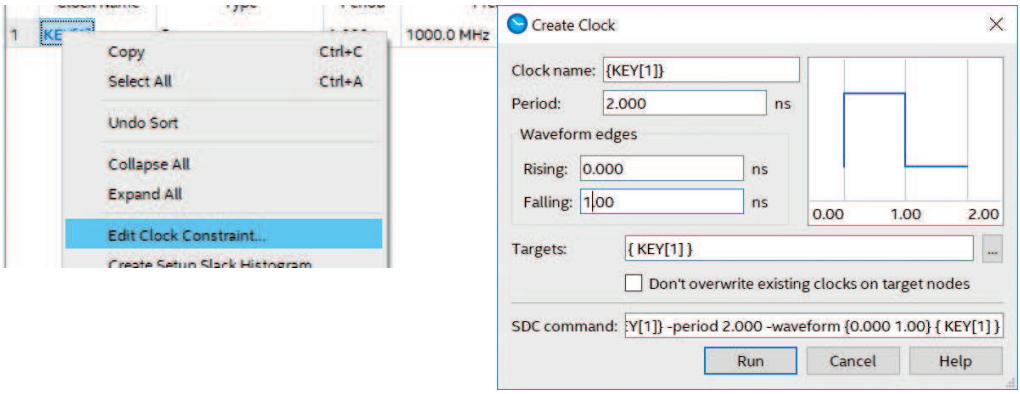


Рис. 3.17 Редактирование ограничений Clock Constraint

Path Summary | Statistics | Data Path | Waveform

	Property	Value
1	From Node	timeQuest_wrapper:comb_3|SW_reg[3]
2	To Node	timeQuest_wrapper:comb_3|LED_reg[1]
3	Launch Clock	KEY[1]
4	Latch Clock	KEY[1]
5	Data Arrival Time	4.405
6	Data Required Time	4.774
7	Slack	0.369

Рис. 3.18 Результаты успешного выполнения временно́го анализа после изменения Clock Constraint

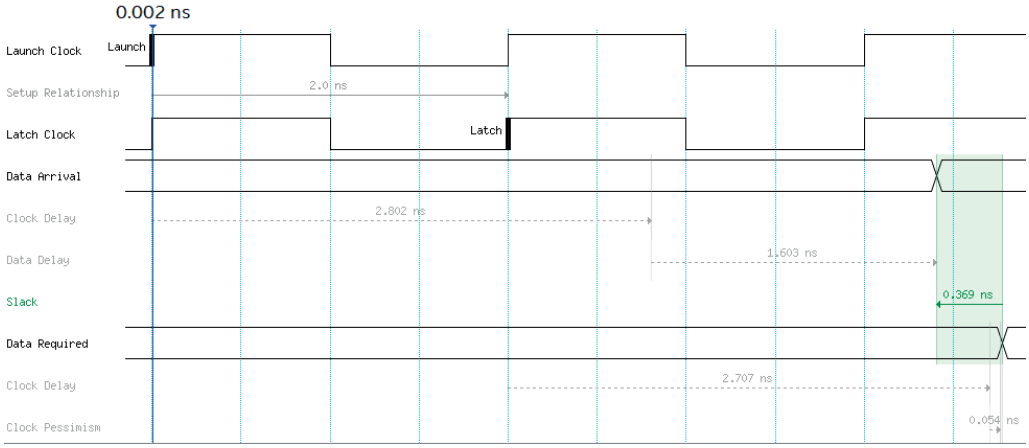


Рис. 3.19 Формы сигналов после изменения Clock Constraint

С помощью **TimeQuest** можно детально изучить путь прохождения сигнала и задержки в нем. На рис. 3.20 подробно описан путь прохождения тактового сигнала и задержек на этом пути, а на рис. 3.21 – путь прохождения данных между регистрами. Данные появляются на выходе регистра **SW_reg** в момент времени **3,544 нс**, а в момент **4,311 нс** появляются на выходе тестируемого модуля и по-

ступают на вход регистра **LED_reg**. Таким образом, полная задержка модуля комбинационной логики без модуля-оболочки равна **4,311 – 3,544 = 0,767 нс**.

Аналогичным образом можно проанализировать задержки во всех рассмотренных выше шифраторах. Соответствующие результаты вместе с информацией о количестве используемых элементов приведены в табл. 3.2 и 3.3. Из таблиц следует, что наименьшее количество элементов используется в шифраторах, описанных с использованием оператора непрерывного присваивания **assign**, они же имеют наименьшие задержки.

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	0.000	0.000					launch edge time
2	⌄ 2.802	2.802					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_A7	KEY[1]
3	0.000	0.000	RR	IC	1	IOIBUF_X49_Y54_N29	KEY[1]~input|i
4	0.853	0.853	RR	CELL	13	IOIBUF_X49_Y54_N29	KEY[1]~input|o
5	2.292	1.439	RR	IC	1	FF_X50_Y53_N19	comb_3|SW_reg[3]|clk
6	2.802	0.510	RR	CELL	1	FF_X50_Y53_N19	timeQuest_wrapper:comb_3|SW_reg[3

Рис. 3.20 Clock Delay в Data Arrival Path

3	⌄ 4.405	1.603					data path
1	3.009	0.207		uTco	1	FF_X50_Y53_N19	timeQuest_wrapper:comb_3|SW_reg[3
2	3.009	0.000	FF	CELL	2	FF_X50_Y53_N19	comb_3|SW_reg[3]|q
3	3.544	0.535	FF	IC	1	LCCOMB_X50_Y53_N20	comb_3|comb_13|binary_out[1]~2|dat
4	3.946	0.402	FF	CELL	1	LCCOMB_X50_Y53_N20	comb_3|comb_13|binary_out[1]~2|cor
5	4.180	0.234	FF	IC	1	LCCOMB_X50_Y53_N30	comb_3|comb_13|binary_out[1]~3|dat
6	4.311	0.131	FF	CELL	1	LCCOMB_X50_Y53_N30	comb_3|comb_13|binary_out[1]~3|cor
7	4.311	0.000	FF	IC	1	FF_X50_Y53_N31	comb_3|LED_reg[1]|d
8	4.405	0.094	FF	CELL	1	FF_X50_Y53_N31	timeQuest_wrapper:comb_3|LED_reg[1

Рис. 3.21 Data Delay в Data Arrival Path

Таблица 3.2 Сравнение неприоритетных шифраторов

Название модуля шифратора	Задержка, нс	Общее количество логических элементов	Максимальная частота, МГц
b1_enc_assign	0.767	8	500
b2_enc_if	2.012	18	285
b3_enc_case	2.161	20	285

Таблица 3.3 Сравнение приоритетных шифраторов

Название модуля шифратора	Задержка, нс	Общее количество логических элементов	Максимальная частота, МГц
b4_pri_enc_if	0.846	12	500
b5_pri_enc_assign	1.158	10	500
b12_anybit_enc	1.252	13	450

Схемы, реализованные с использованием ветвления (**if**, **case**) и тернарного оператора (**?:**), более понятны. Поэтому их лучше поддерживать, нежели схемы, составленные после анализа карт Карно или любой другой оптимизации (**b1_enc_assign**). В первом случае достигается лучшая производительность, в другом – умеренная производительность и большие затраты ресурсов, но при этом высокая удобочитаемость и более простая поддержка кода. Оба подхода используются на практике – все зависит от задач, которые стоят перед разработчиком. Кроме того, синтезирующие **САПР** постоянно эволюционируют, и качество оптимизации растет, поэтому первый подход представляется более перспективным. Некоторые аспекты оптимизации, выполняемой **САПР**, рассмотрены в разделе 3.4.

Дополнительное задание для самостоятельной работы

Исследуйте работу, синтезируйте и реализуйте на **FPGA**-плате все приведенные в данном разделе примеры шифраторов.

3.3 Дешифраторы

Дешифратор (**декодер**, **decoder**), как и шифратор, является кодирующим устройством. Дешифратор выполняет действие, обратное действию шифратора. При этом если шифраторы делятся на приоритетные и неприоритетные, то у дешифраторов такого деления нет. В классическом понимании дешифратор преобразовывает входную двоичную величину в унарный код на выходе. Одним из применений дешифратора является выбор одного из многих устройств (например, ячеек памяти, устройств ввода-вывода), каждое из которых имеет уникальный адрес. Адрес подается на вход дешифратора, в результате чего активируется только один разряд выходного сигнала, который выбирает соответствующее устройство. Другим распространенным применением дешифратора является реализация произвольной логической функции (или нескольких функций). Такое применение возможно, поскольку дешифратор генерирует все возможные минтермы входных сигналов. В этом случае, кроме самого дешифратора, используется логический элемент **ИЛИ**, на входы которого подаются соответствующие минтермы. Дешифраторы часто применяются для построения мультиплексоров, в которых унарный код на выходе дешифратора выбирает активированный канал мультиплексора.

Дешифраторами также называют устройства, преобразующие входной код в другой, понятный устройствам, которые подсоединены к выходу дешифратора. Например, управление семисегментным индикатором выполняется с помощью дешифратора, только вместо унарного кода он формирует специальный семиразрядный код (или восьмиразрядный, если используется десятичная точка), который определяет, какие сегменты индикатора будут включены для отображения того или иного символа.

3.3.1 Дешифратор с использованием оператора case

Классическим способом описания дешифратора (как и шифратора) является использование оператора **case**, который размещен внутри блока **always**. В каждой ветке оператора **case** выполняется блокирующее присваивание (**=**) выходному

сигналу необходимого значения (листинг 3.9). Результаты синтеза и моделирования блока приведены на рис. 3.22 и 3.23.

```
module b6_4bit_dec_case (
   input [3:0] binary_in,
   output reg [15:0] decoder_out,
   input enable
);
always @ (*)
   begin
      decoder_out = 0;
      if (enable) begin
         case (binary_in)
            4'h0 : decoder_out = 16'h1;
            4'h1 : decoder_out = 16'h2;
            4'h2 : decoder_out = 16'h4;
            4'h3 : decoder_out = 16'h8;
            4'h4 : decoder_out = 16'h10;
            4'h5 : decoder_out = 16'h20;
            4'h6 : decoder_out = 16'h40;
            4'h7 : decoder_out = 16'h80;
            4'h8 : decoder_out = 16'h100;
            4'h9 : decoder_out = 16'h200;
            4'hA : decoder_out = 16'h400;
            4'hB : decoder_out = 16'h800;
            4'hC : decoder_out = 16'h1000;
            4'hD : decoder_out = 16'h2000;
            4'hE : decoder_out = 16'h4000;
            4'hF : decoder_out = 16'h8000;
         endcase
      end
   end
endmodule
```

Листинг 3.9 Дешифратор с использованием оператора case

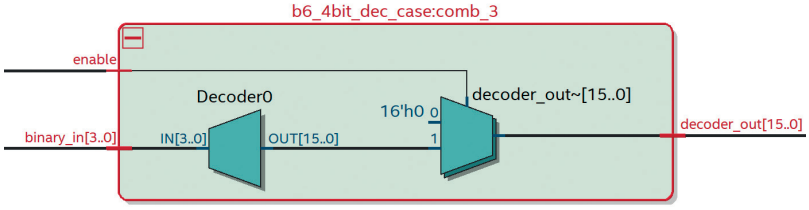


Рис. 3.22 Структура дешифратора с использованием оператора case в RTL Viewer

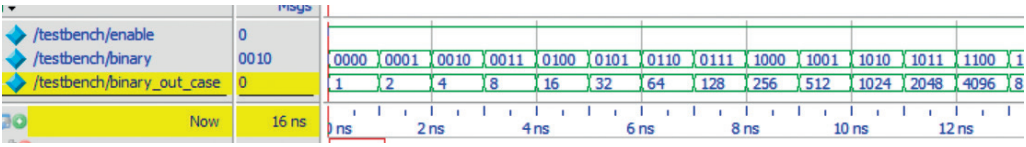


Рис. 2.23 Моделирование дешифратора с использованием оператора case

3.3.2 Дешифратор с использованием оператора сдвига

Поскольку на выходе классического дешифратора формируется унарный код, логическая единица которого расположена в позиции, номер которой определяется входным двоичным кодом, такой шифратор можно описать с помощью оператора сдвига (<<), который перемещает логическую единицу в необходимую позицию. Соответствующее описание (листинг 3.10), структура (рис. 3.24) и результаты моделирования (рис. 3.25) такой реализации дешифратора приведены ниже.

```
module b7_4bit_dec_assign_shift (
   input [3:0] binary_in,
   output wire [15:0] decoder_out,
   input enable
);
   assign decoder_out = (enable) ? (1 << binary_in) : 16'b0 ;
endmodule
```

Листинг 3.10 Дешифратор с использованием оператора сдвига

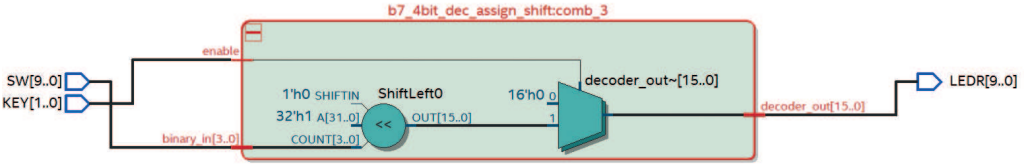


Рис. 3.24 Структура дешифратора с использованием оператора сдвига в RTL Viewer

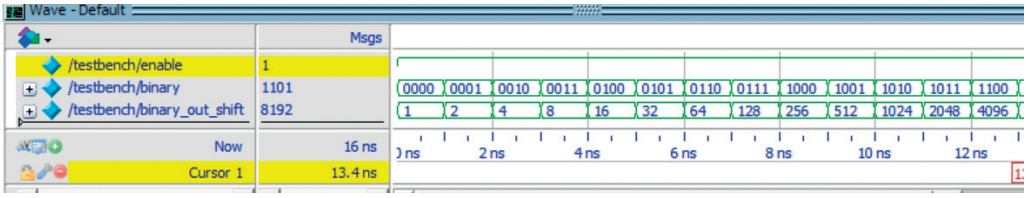


Рис. 3.25 Моделирование дешифратора с использованием оператора сдвига

3.3.3 Дешифратор с использованием оператора непрерывного присваивания assign

Существует еще один подход к проектированию дешифраторов, полностью аналогичный подходу описания шифратора, применяемому в разделе 3.2.1. Он состоит в том, что для всех элементов выходного массива сигналов используются

операторы непрерывного присваивания и явно заданы соответствующие булевы функции.

```
module b8_4bit_dec_assign_logic(in, out, enable);
   input enable;
   input [3:0] in;
   output [15:0] out;

   assign out[0] = !in[0] & !in[1] & !in[2] & !in[3] & enable;
   assign out[1] = in[0] & !in[1] & !in[2] & !in[3] & enable;
   assign out[2] = !in[0] & in[1] & !in[2] & !in[3] & enable;
   assign out[3] = in[0] & in[1] & !in[2] & !in[3] & enable;
   assign out[4] = !in[0] & !in[1] & in[2] & !in[3] & enable;
   assign out[5] = in[0] & !in[1] & in[2] & !in[3] & enable;
   assign out[6] = !in[0] & in[1] & in[2] & !in[3] & enable;
   assign out[7] = in[0] & in[1] & in[2] & !in[3] & enable;
   assign out[8] = !in[0] & !in[1] & !in[2] & in[3] & enable;
   assign out[9] = in[0] & !in[1] & !in[2] & in[3] & enable;
   assign out[10] = !in[0] & in[1] & !in[2] & in[3] & enable;
   assign out[11] = in[0] & in[1] & !in[2] & in[3] & enable;
   assign out[12] = !in[0] & !in[1] & in[2] & in[3] & enable;
   assign out[13] = in[0] & !in[1] & in[2] & in[3] & enable;
   assign out[14] = !in[0] & in[1] & in[2] & in[3] & enable;
   assign out[15] = in[0] & in[1] & in[2] & in[3] & enable;
endmodule
```

Листинг 3.11 Дешифратор с использованием оператора непрерывного присваивания assign

Отметим, что представление такого дешифратора в **RTL Viewer** (рис. 3.26) состоит из небольшого количества элементарных логических элементов (**И**, **ИЛИ**).

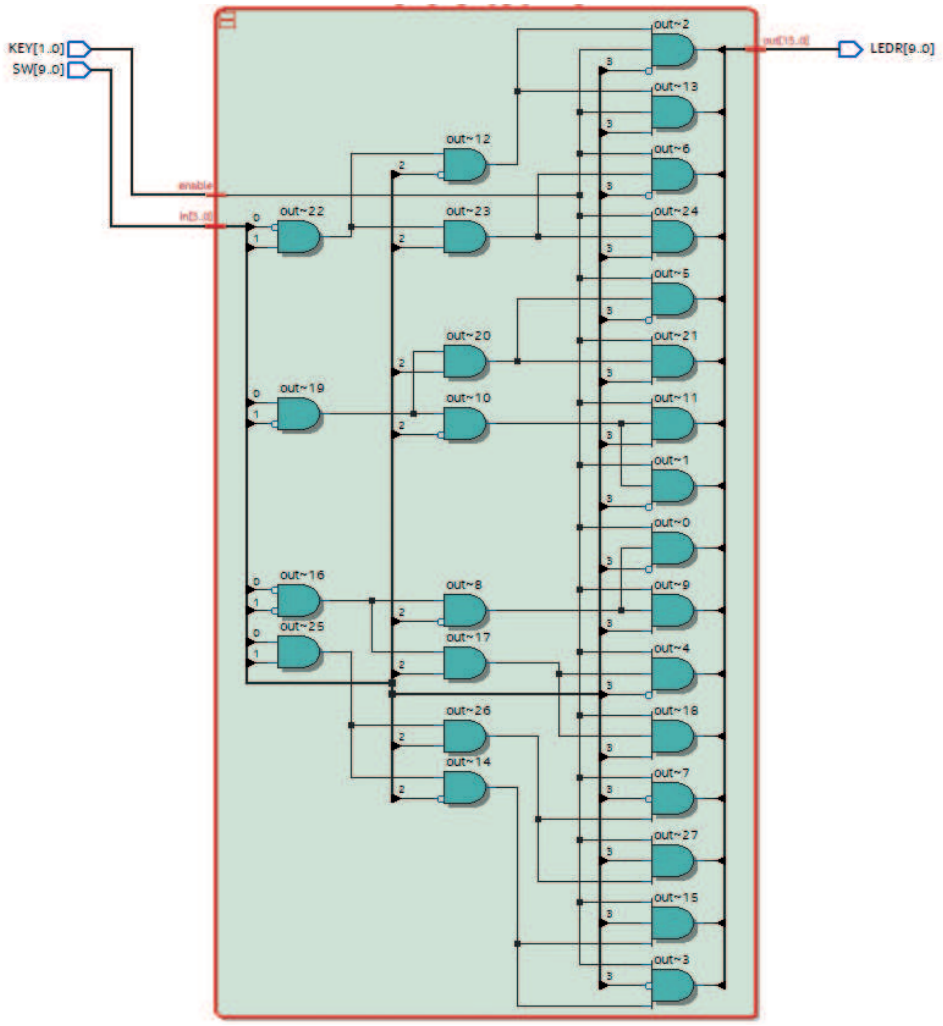


Рис. 3.26 Структура дешифратора с использованием оператора непрерывного присваивания assign в RTL Viewer

3.3.4 Дешифратор для учебного MIPS-совместимого процессора

При проектировании цифровых устройств часто необходимо по одному входному коду формировать несколько управляющих сигналов. В качестве примера можно рассмотреть дешифраторы команд **MIPS**-совместимого процессора, рассмотренного в книге **«Цифровая схемотехника и архитектура компьютера» Дэвида М. Харриса, Сары Л. Харрис**[1]. Основной дешифратор по полю **opcode** выполняемой команды формирует управляющие сигналы процессора, а дешифратор **АЛУ** использует поле **funct** команды и выходы основного дешифратора для генерации управляющих сигналов **АЛУ**. Ниже приведено описание упрощен-

1 *Харрис Д. М., Харрис С. Л.* Цифровая схемотехника и архитектура компьютера. 2-е изд. 2013.

ных вариантов этих дешифраторов (листинг 3.12), результаты синтеза (рис. 3.27) и моделирования (рис. 3.28). Для таких дешифраторов обычно используют оператор case.

```
module b9_6bit_dec_mips_maindec(input wire [5:0] op,
   output wire memtoreg, memwrite,
   output wire branch, alusrc,
   output wire regdst, regwrite,
   output wire jump,
   output wire [1:0] aluop);
   reg [8:0] controls;

   assign {regwrite, regdst, alusrc, branch, memwrite,
      memtoreg, jump, aluop} = controls;
   always @(*) begin
      case(op)
         6'b000000: controls <= 9'b110000010; // RTYPE
         6'b100011: controls <= 9'b101001000; // LW
         6'b101011: controls <= 9'b001010000; // SW
         6'b000100: controls <= 9'b000100001; // BEQ
         6'b001000: controls <= 9'b101000000; // ADDI
         6'b000010: controls <= 9'b000000100; // J
         default: controls <= 9'bxxxxxxxxx; // illegal op
      endcase
   end
endmodule

module b9_6bit_dec_mips_aludec(input wire [5:0] funct,
   input wire [1:0] aluop,
   output reg [2:0] alucontrol);

   always @(*) begin
      case(aluop)
         2'b00: alucontrol <= 3'b010; // add (for lw/sw/addi)
         2'b01: alucontrol <= 3'b110; // sub (for beq)
         default: case(funct) // R-type instructions
            6'b100000: alucontrol <= 3'b010; // add
            6'b100010: alucontrol <= 3'b110; // sub
            6'b100100: alucontrol <= 3'b000; // and
            6'b100101: alucontrol <= 3'b001; // or
            6'b101010: alucontrol <= 3'b111; // slt
            default: alucontrol <= 3'bxxx; // ???
      endcase
   endcase
   end
endmodule
```

Листинг 3.12 Дешифраторы (основной и АЛУ) процессора MIPS

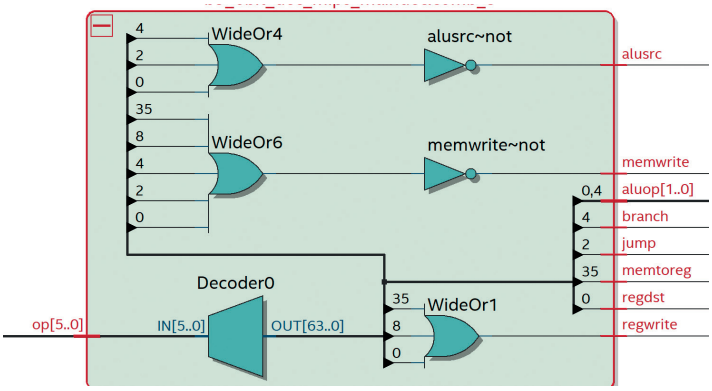


Рис. 3.27 Схема основного дешифратора процессора MIPS в RTL Viewer

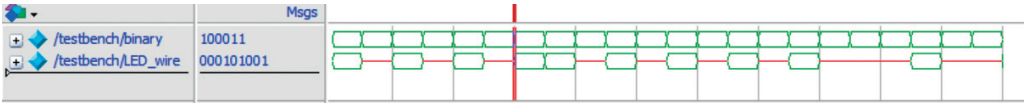


Рис. 3.28 Моделирование работы основного дешифратора процессора MIPS

3.3.5 Параметрический дешифратор

Количество входных разрядов параметрического дешифратора определяется единственным параметром. Для изменения количества разрядов не нужно переписывать код, а достаточно изменить только этот параметр. В данном примере используется оператор **generate** (листинг 3.13). Он генерирует набор независимых аналогичных по структуре аппаратных блоков. Результат синтеза дешифратора приведен на рис. 3.29.

```
module b13_custombit_dec (
   binary_in,
   decoder_out,
   enable
);
   parameter IN_SIZE = 4;
   parameter OUT_SIZE = 1<<IN_SIZE;

   input wire [IN_SIZE-1:0] binary_in;
   output wire [OUT_SIZE-1:0] decoder_out;
   input enable;

   genvar i;
   generate
      for (i=0; i<OUT_SIZE; i=i+1) begin : gen
         assign decoder_out[i] = binary_in==i & enable ? 1'b1 : 1'b0;
      end
   endgenerate
endmodule
```

Листинг 3.13 Параметрический дешифратор

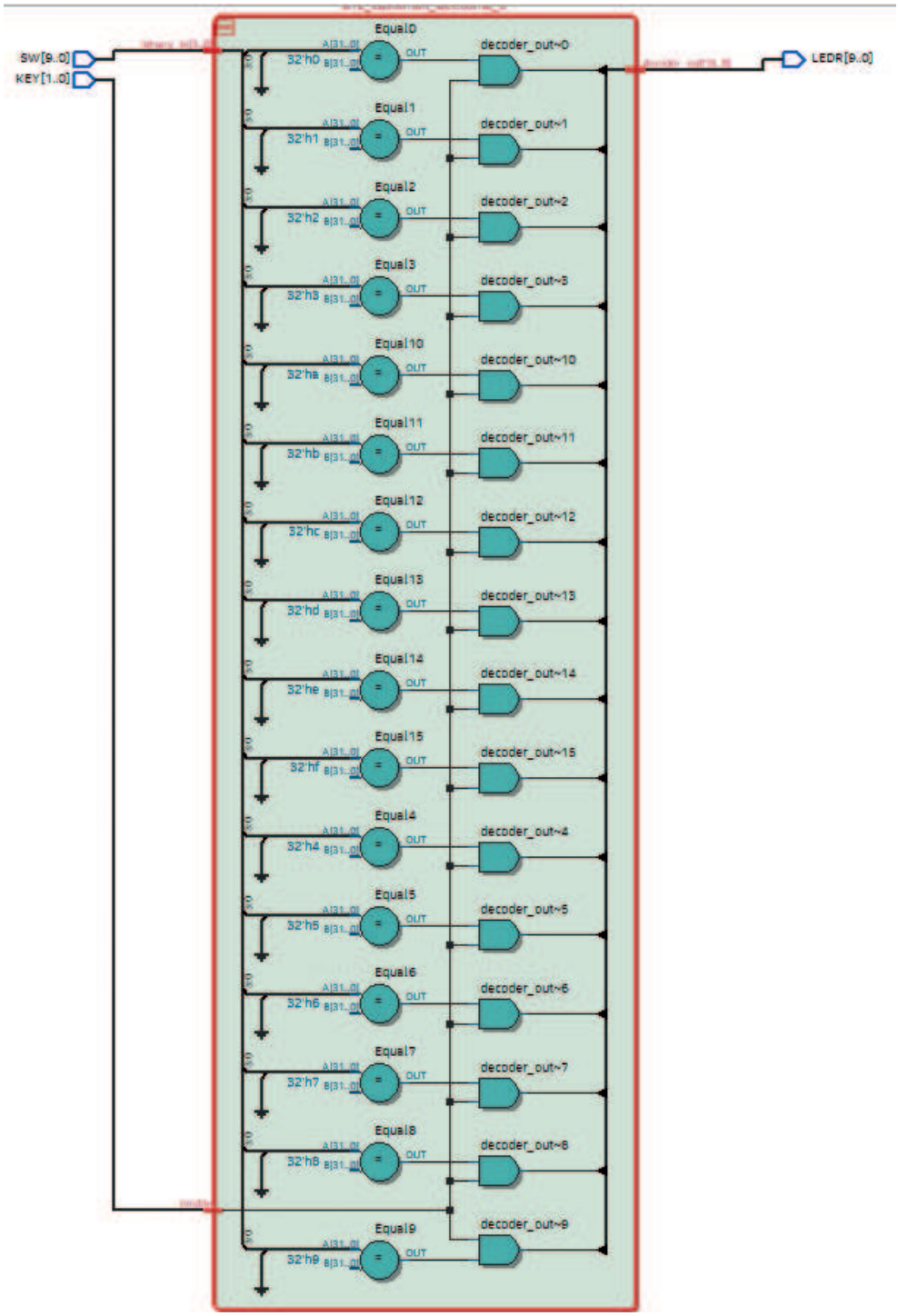


Рис. 3.29 Параметрический дешифратор

Параметрический дешифратор можно реализовать с использованием оператора сдвига, пример такого дешифратора приведен в листинге 3.14. Его принцип работы аналогичен принципу работы непараметрического дешифратора (листинг 3.10).

```
module b7_anybit_dec_assign_shift (
   binary_in, decoder_out, enable
);
   parameter IN_SIZE = 4;
   parameter OUT_SIZE = 2<<IN_SIZE;

   input wire [IN_SIZE-1:0] binary_in;
   output wire [OUT_SIZE-1:0] decoder_out;
   input enable;
   assign decoder_out = (enable) ? (1 << binary_in) : 0 ;
endmodule
```

Листинг 3.14 Параметрический дешифратор на базе оператора сдвига

3.3.6 Сравнение дешифраторов

Характеристики рассмотренных выше дешифраторов приведены в табл. 3.4. Минимальное количество логических элементов и максимальную рабочую частоту имеет дешифратор с использованием оператора непрерывного присваивания, в котором для элемента массива выходного сигнала записаны соответствующие логические функции.

Анализ схем в **RTL Viewer** показывает, что модули **b8_4bit_dec_assign_logic** и **b13_custombit_dec** имеют простую реализацию и поэтому характеризуются высоким быстродействием. В этих модулях явно описаны громоздкие логические функции, которые затем реализованы с использованием простых логических элементов. В реализациях с применением операторов **case** и **shift** используются высокоуровневые (и более наглядные) конструкции языка описания аппаратуры, а **САПР** выполняет сложные действия при синтезе, что приводит к большим затратам ресурсов и к снижению быстродействия по сравнению с описанием с помощью логических функций.

Таблица 3.4 Сравнение дешифраторов

Название модуля дешифратора	Задержка, нс	Общее количество логических элементов	Максимальная частота, МГц
b6_4bit_dec_case	0,990	21	400
b7_4bit_dec_assign_shift	0,805	15	500
b8_4bit_dec_assign_logic	0,795	14	500
b13_custombit_dec	0,825	14	500

Дополнительное задание для самостоятельной работы

Исследуйте работу, синтезируйте и реализуйте на **FPGA**-плате параметрический дешифратор с использованием оператора сдвига (листинг 3.14).

3.4 Оптимизация при синтезе

При синтезе **САПР** выполняет оптимизацию разрабатываемого устройства. Критерии оптимизации могут быть различными и зависят от требований, предъявляемых к конечному изделию. Часто необходимо увеличить быстродействие, иногда уменьшить потребление энергии, количество элементов или удешевить схему. Эти действия выполняются компилятором **Quartus Prime**, поэтому от настроек процесса компиляции зависят параметры спроектированного электронного устройства. Для открытия окна изменения настроек (рис. 3.30) необходимо нажать **Ctrl**+**Shift**+**E** или щелкнуть правой клавишей мышки на проекте и выбрать пункт меню **Settings**.

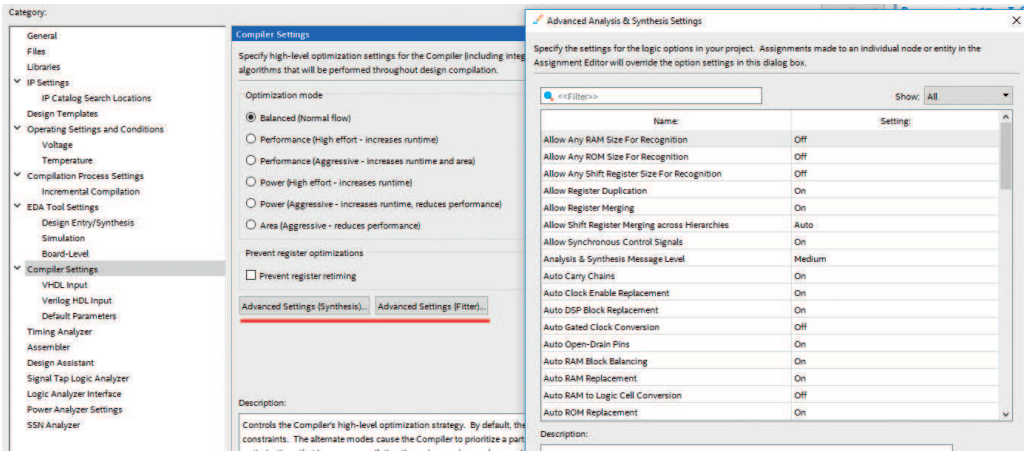


Рис. 3.30 Параметры оптимизации в настройках компилятора

На рис. 3.30 представлено множество параметров, определяющих ход процесса оптимизации. Компания **Intel FPGA** добавила в **САПР Quartus Prime** несколько «помощников», упрощающих настройку параметров оптимизации. Для того чтобы воспользоваться одним из них, необходимо выбрать меню **Tools → Advisors** (рис. 3.31).

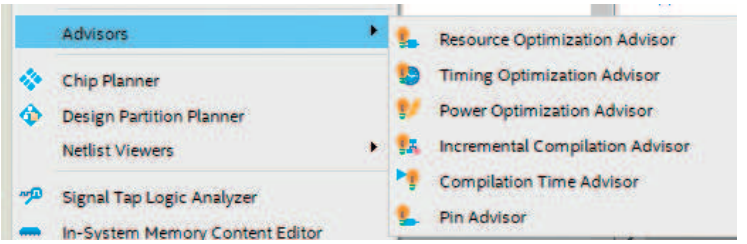


Рис. 3.31 Меню Advisors

После выбора необходимого **Advisor'а САПР Quartus Prime** предложит установить настройки, которые рекомендуется использовать.

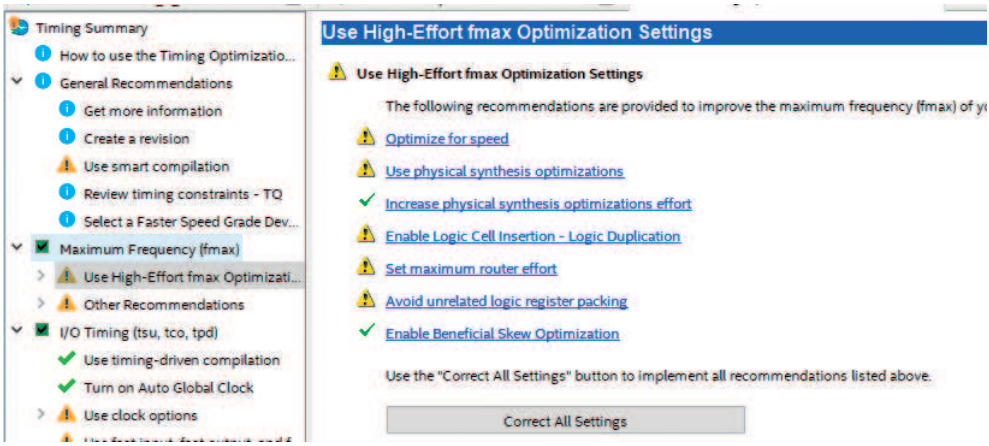


Рис. 3.32 Использование Timing Optimization Advisor

3.4.1 Режимы оптимизации

Перед ознакомлением с данным разделом необходимо ознакомиться с разделом 3.2.7, где дано подробное описание, как анализировать временны́е характеристики проектируемого устройства.

Согласно рис. 3.30, существует несколько стандартных профилей оптимизации при компиляции и синтезе. Их можно разделить на несколько категорий:

- сбалансированные настройки (**Balanced mode**);
- оптимизация по быстродействию (**Performance mode**);
- оптимизация по энергопотреблению (**Power mode**);
- оптимизация по используемой площади кристалла (**Area mode**).

Все выполняемые процедуры оптимизации влияют на скорость выполнения компиляции и синтеза средствами **САПР** и на параметры разрабатываемого устройства. В зависимости от выбранного режима оптимизации созданное устройство будет иметь различные временны́е характеристики и энергопотребление, а также будет использовать различное количество логических элементов.

Для примера рассмотрим модуль **b2_enc_if** (листинг 3.1). Оставим настройки оптимизации по умолчанию, т. е. будем использовать профиль оптимизации **Balanced mode**. После выполнения стандартного временного анализа необходимо открыть раздел **Summary of Paths** и выбрать первый путь (он является критическим), после чего в контекстном меню выбрать **Locate Path → Locate in Chip Planner** (рис. 3.33).

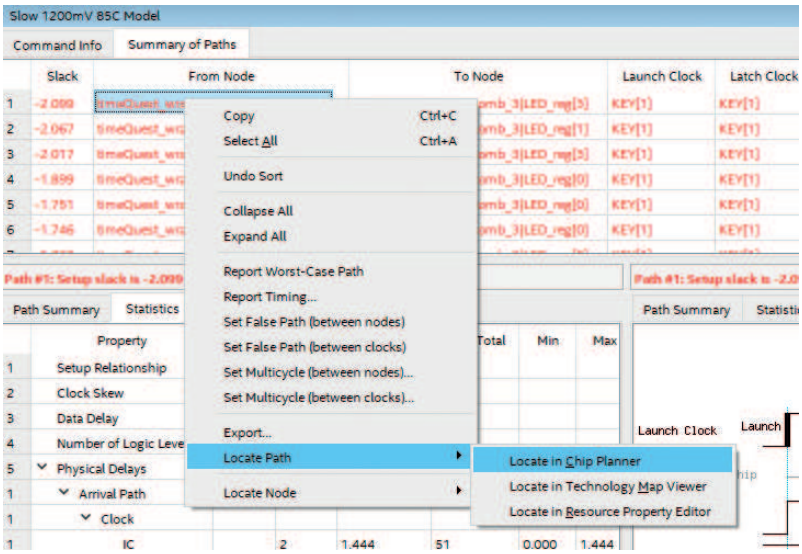


Рис. 3.33 Просмотр физического расположения пути в TimeQuest

Отобразится структура **FPGA** матрицы в **Chip Planner** (рис. 3.34). В верхней части рисунка красным цветом выделены используемые модулем ячейки матрицы. В нижней части красным цветом выделена вкладка со словом **Timing**, которая позволяет выбрать зону обзора. Дважды кликнув мышкой на выбранном пути, можно отобразить граф прохождения сигнала (рис. 3.35).

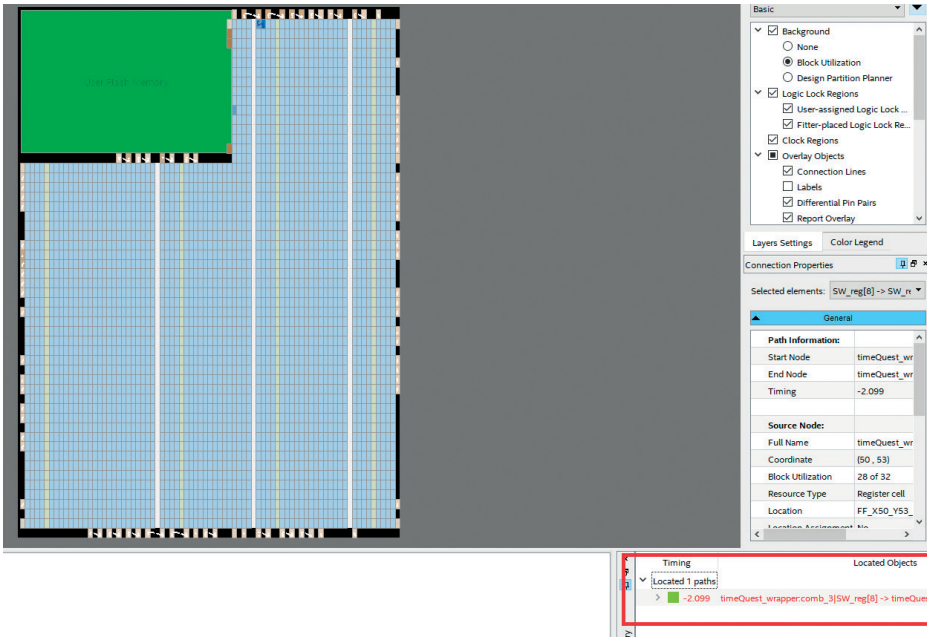


Рис. 3.34 Просмотр структуры пути (Locate in Chip Planner)

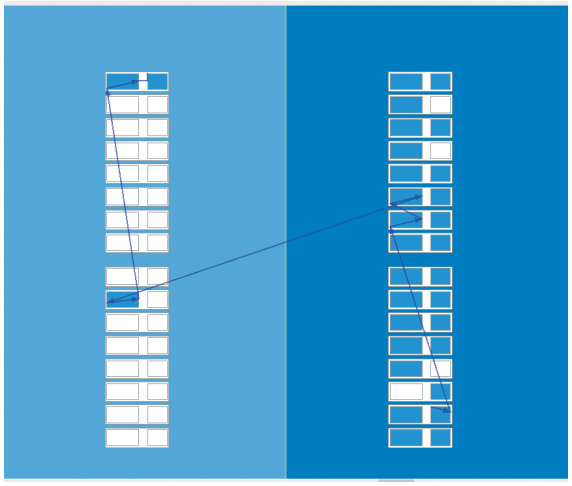


Рис. 3.35 Граф пути прохождения сигнала в Chip Planner

В соответствии с рис. 3.35 сигнал проходит весьма длинный путь между двумя группами ячеек. Если выбрать путь **Arrival Clock** (рис. 3.36), то можно отследить, насколько длинный путь проходит тактовый сигнал от контакта микросхемы до регистра. При проектировании с использованием **FPGA** задержки в межсоединениях элементов часто ограничивают быстродействие всего устройства.

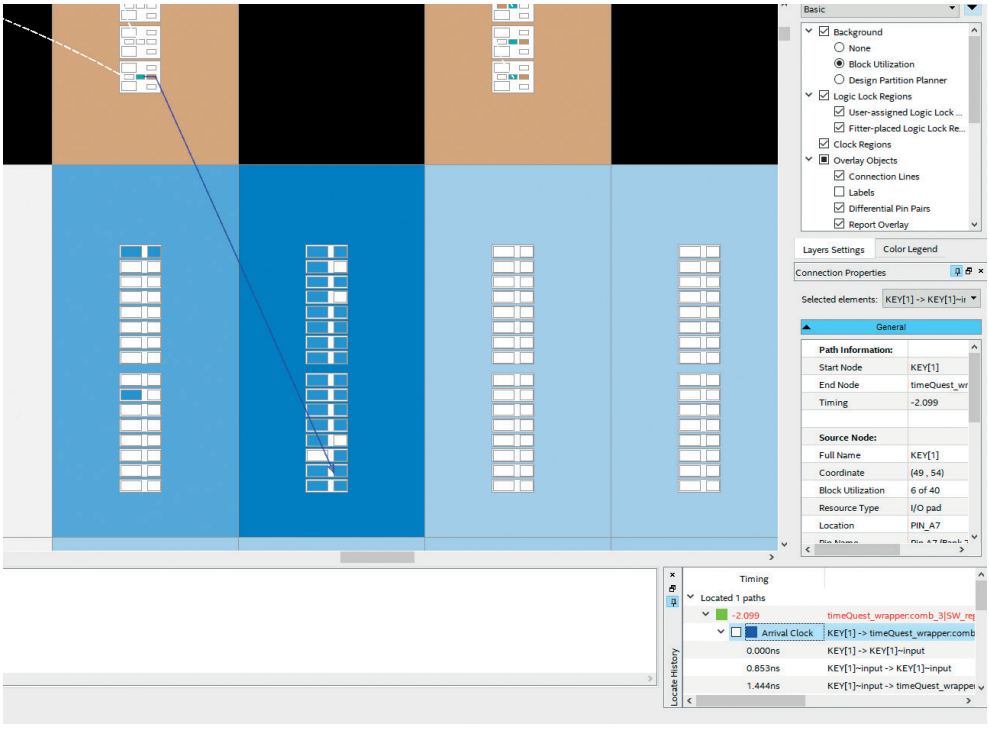


Рис. 3.36 Просмотр пути прохождения тактового сигнала в Chip Planner

Повторим синтез с другими настройками оптимизации. Для этого нужно вернуться к настройкам на рис. 3.30 и выбрать, например, профиль оптимизации по быстродействию **Performance** (**Aggressive**), сохранить проект и скомпилировать его вновь. На рис. 3.37 отражено, что в этом случае путь прохождения рассматриваемого сигнала локализован в одной группе ячеек, что повысило быстродействие всего устройства.

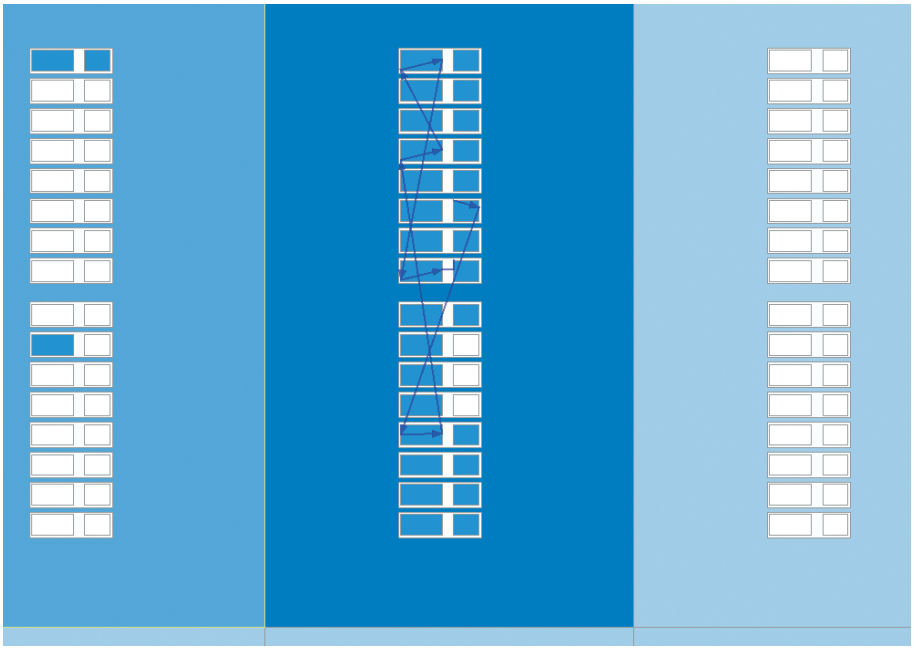


Рис. 3.37 Путь прохождения сигнала в Chip Planner при оптимизации Performance (Aggressive)

Процесс компиляции проекта в **САПР Quartus Prime** состоит из нескольких этапов. После того как исходный код проверен и синтезирован компилятором, запускается процесс **проецирования проекта на ресурсы кристалла** (**mapping**), который реализует проектируемую схему с использованием ресурсов, доступных в конкретном чипе. После этого начинается процесс **размещения** (**fitting**), который размещает используемые элементы в кристалле и трассирует соединения между ними. Средствами **САПР** можно отобразить структуру схемы после выполнения каждого из этих этапов. Для этого необходимо выбрать меню **Tools → Netlist Viewers → Technology Map Viewer** (**Post-Mapping** или **Post-Fitting** соответственно).

Анализ разрабатываемой схемы после выполнения размещения позволяет сделать вывод о том, что существует некоторое отличие в соединении элементов при использовании профилей оптимизации **Balanced** и **Performance** (рис. 3.38). Данные изменения произошли на этапе размещения, так как структура схемы после этапа проецирования на ресурсы кристалла не зависит от профиля оптимизации (рис. 3.39).

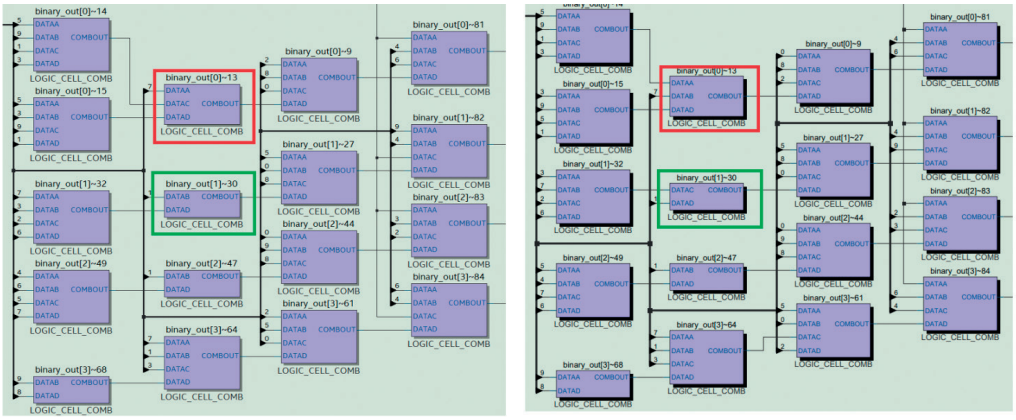


Рис. 3.38 Разные уровни оптимизаций (Post-Fitting): Performance (Aggressive) – слева и Balanced (Normal flow) – справа

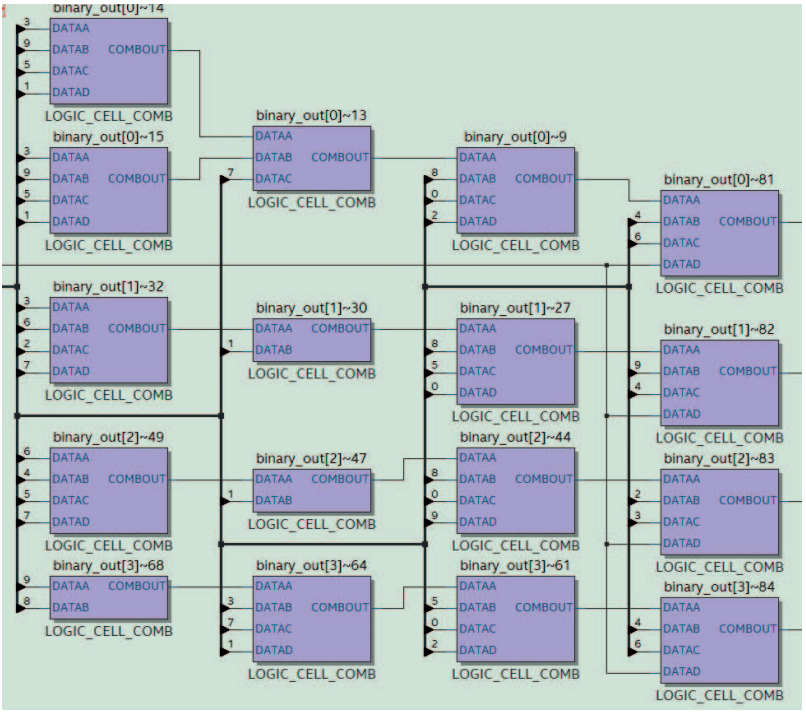
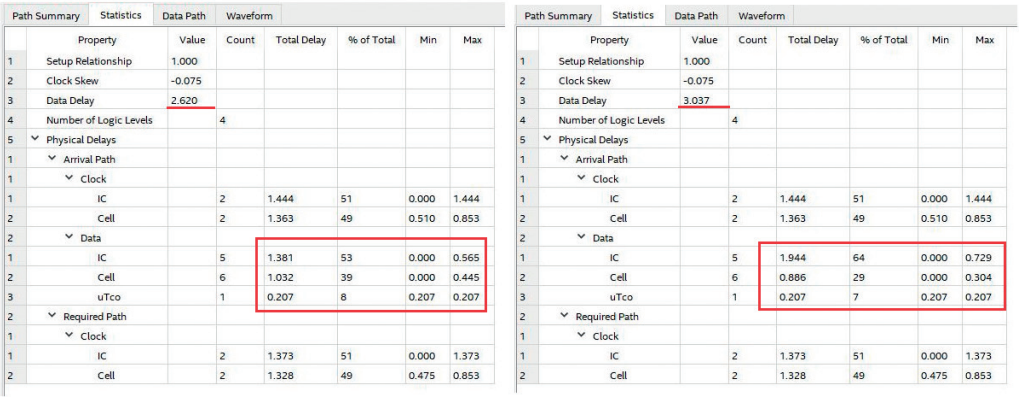


Рис. 3.39 Post Mapping. Изменения при различных профилях оптимизации не произошли

Отметим также, что результаты компиляции проекта и его размещения могут каждый раз отличаться даже без изменения профиля оптимизации. Это происходит потому, что процесс размещения является емкой по времени задачей. Компилятор не может перебрать все возможные варианты размещения проекта на кристалле и найти лучшее решение из-за огромной вычислительной сложности данной задачи. Поэтому компилятор прекращает поиск после определенного ко-

личества итераций (ограничений по времени компиляции) или по достижении требуемых ограничений по ресурсам и быстродействию. Каждый раз результаты компиляции могут быть разными.

На рис. 3.40 приведены результаты временно́го анализа при использовании различных профилей оптимизации. Разница во времени прохождения сигнала достигает **0,417 нс**, что весьма существенно. На рис. 3.41 отображены составляющие задержки прохождения сигнала между компонентами при использовании двух профилей оптимизации. Уменьшение длины пути приводит к уменьшению задержки и повышению быстродействия разрабатываемого устройства. Временны́е диаграммы, приведенные на рис. 3.42 и 3.43, подтверждают улучшение скорости прохождения сигнала после оптимизации **Performance**.



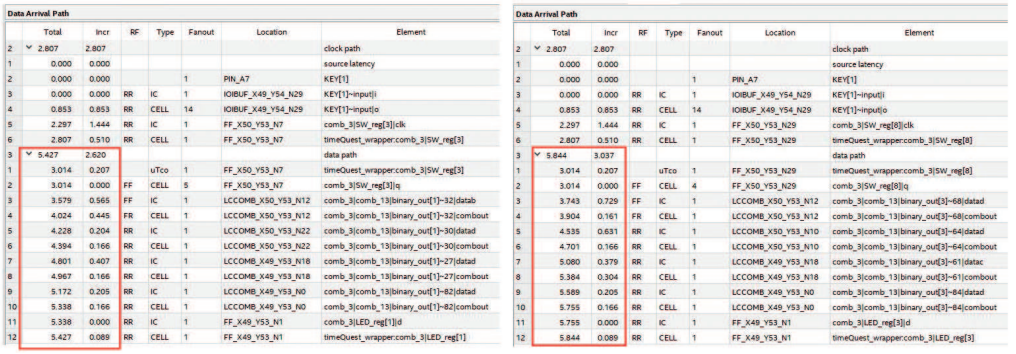
Path Summary | Statistics | Data Path | Waveform

	Property	Value	Count	Total Delay	% of Total	Min	Max
1	Setup Relationship	1.000					
2	Clock Skew	-0.075					
3	Data Delay	2.620					
4	Number of Logic Levels		4				
5	Physical Delays						
1	Arrival Path						
1	Clock						
1	IC		2	1.444	51	0.000	1.444
2	Cell		2	1.363	49	0.510	0.853
2	Data						
1	IC		5	1.381	53	0.000	0.565
2	Cell		6	1.032	39	0.000	0.445
3	uTco		1	0.207	8	0.207	0.207
2	Required Path						
1	Clock						
1	IC		2	1.373	51	0.000	1.373
2	Cell		2	1.328	49	0.475	0.853

Path Summary | Statistics | Data Path | Waveform

	Property	Value	Count	Total Delay	% of Total	Min	Max
1	Setup Relationship	1.000					
2	Clock Skew	-0.075					
3	Data Delay	3.037					
4	Number of Logic Levels		4				
5	Physical Delays						
1	Arrival Path						
1	Clock						
1	IC		2	1.444	51	0.000	1.444
2	Cell		2	1.363	49	0.510	0.853
2	Data						
1	IC		5	1.944	64	0.000	0.729
2	Cell		6	0.886	29	0.000	0.304
3	uTco		1	0.207	7	0.207	0.207
2	Required Path						
1	Clock						
1	IC		2	1.373	51	0.000	1.373
2	Cell		2	1.328	49	0.475	0.853

Рис. 3.40 Результаты временно́го анализа при использовании различных профилей оптимизаций: Performance (Aggressive) – слева и Balanced (Normal flow) – справа



Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
2	2.807	2.807					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_A7	KEY[1]
3	0.000	0.000	RR	IC	1	IOIBUF_X49_Y54_N29	KEY[1]~input|i
4	0.853	0.853	RR	CELL	14	IOIBUF_X49_Y54_N29	KEY[1]~input|o
5	2.297	1.444	RR	IC	1	FF_X50_Y53_N7	comb_3|SW_reg[3]|clk
6	2.807	0.510	RR	CELL	1	FF_X50_Y53_N7	timeQuest_wrapper:comb_3|SW_reg[3]
3	5.427	2.620					data path
1	3.014	0.207		uTco	1	FF_X50_Y53_N7	timeQuest_wrapper:comb_3|SW_reg[3]
2	3.014	0.000	FF	CELL	5	FF_X50_Y53_N7	comb_3|SW_reg[3]|q
3	3.579	0.565	FF	IC	1	LCCOMB_X50_Y53_N12	comb_3|comb_13|binary_out[1]~32|datab
4	4.024	0.445	FR	CELL	1	LCCOMB_X50_Y53_N12	comb_3|comb_13|binary_out[1]~32|combout
5	4.228	0.204	RR	IC	1	LCCOMB_X50_Y53_N22	comb_3|comb_13|binary_out[1]~30|datad
6	4.394	0.166	RR	CELL	1	LCCOMB_X50_Y53_N22	comb_3|comb_13|binary_out[1]~30|combout
7	4.801	0.407	RR	IC	1	LCCOMB_X49_Y53_N18	comb_3|comb_13|binary_out[1]~27|datad
8	4.967	0.166	RR	CELL	1	LCCOMB_X49_Y53_N18	comb_3|comb_13|binary_out[1]~27|combout
9	5.172	0.205	RR	IC	1	LCCOMB_X49_Y53_N0	comb_3|comb_13|binary_out[1]~82|datad
10	5.338	0.166	RR	CELL	1	LCCOMB_X49_Y53_N0	comb_3|comb_13|binary_out[1]~82|combout
11	5.338	0.000	RR	IC	1	FF_X49_Y53_N1	comb_3|LED_reg[1]|d
12	5.427	0.089	RR	CELL	1	FF_X49_Y53_N1	timeQuest_wrapper:comb_3|LED_reg[1]

Data Arrival Path

	Total	Incr	RF	Type	Fanout	Location	Element
2	2.807	2.807					clock path
1	0.000	0.000					source latency
2	0.000	0.000			1	PIN_A7	KEY[1]
3	0.000	0.000	RR	IC	1	IOIBUF_X49_Y54_N29	KEY[1]~input|i
4	0.853	0.853	RR	CELL	14	IOIBUF_X49_Y54_N29	KEY[1]~input|o
5	2.297	1.444	RR	IC	1	FF_X50_Y53_N29	comb_3|SW_reg[8]|clk
6	2.807	0.510	RR	CELL	1	FF_X50_Y53_N29	timeQuest_wrapper:comb_3|SW_reg[8]
3	5.844	3.037					data path
1	3.014	0.207		uTco	1	FF_X50_Y53_N29	timeQuest_wrapper:comb_3|SW_reg[8]
2	3.014	0.000	FF	CELL	4	FF_X50_Y53_N29	comb_3|SW_reg[8]|q
3	3.743	0.729	FF	IC	1	LCCOMB_X50_Y53_N12	comb_3|comb_13|binary_out[3]~68|datad
4	3.904	0.161	FR	CELL	1	LCCOMB_X50_Y53_N12	comb_3|comb_13|binary_out[3]~68|combout
5	4.535	0.631	RR	IC	1	LCCOMB_X50_Y53_N10	comb_3|comb_13|binary_out[3]~64|datad
6	4.701	0.166	RR	CELL	1	LCCOMB_X50_Y53_N10	comb_3|comb_13|binary_out[3]~64|combout
7	5.080	0.379	RR	IC	1	LCCOMB_X49_Y53_N18	comb_3|comb_13|binary_out[3]~61|datac
8	5.384	0.304	RR	CELL	1	LCCOMB_X49_Y53_N18	comb_3|comb_13|binary_out[3]~61|combout
9	5.589	0.205	RR	IC	1	LCCOMB_X49_Y53_N0	comb_3|comb_13|binary_out[3]~84|datad
10	5.755	0.166	RR	CELL	1	LCCOMB_X49_Y53_N0	comb_3|comb_13|binary_out[3]~84|combout
11	5.755	0.000	RR	IC	1	FF_X49_Y53_N1	comb_3|LED_reg[3]|d
12	5.844	0.089	RR	CELL	1	FF_X49_Y53_N1	timeQuest_wrapper:comb_3|LED_reg[3]

Рис. 3.41 Временны́е характеристики пути Data Arrival Path при использовании разных профилей оптимизаций: Performance (Aggressive) – слева и Balanced (Normal flow) – справа

Результаты синтеза при использовании различных профилей оптимизации сведены в табл. 3.5. Таким образом, профиль **Performance** оправдывает свое название. Также неплохие результаты достигаются при использовании профиля **Power**.

Применение рассмотренных профилей оптимизации сказывается на реализации проектируемого устройства в конкретном чипе **ПЛИС**. Они позволяют эффективно использовать ограниченные ресурсы чипа и повысить максимальную частоту работы конечного устройства. Важно понимать, что рассматриваемая в данном разделе оптимизация не заменяет оптимизацию исходного кода проекта, которую должен выполнить разработчик.

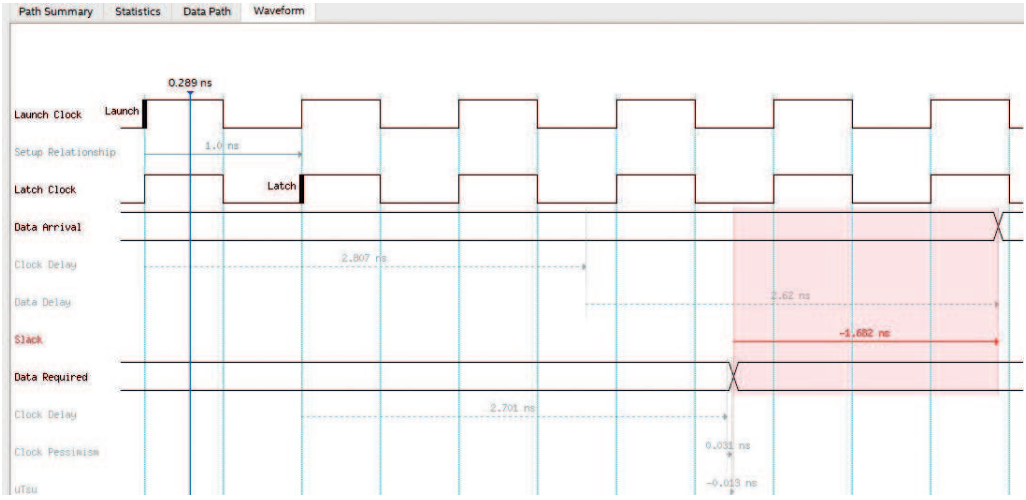


Рис. 3.42 Временны́е диаграммы при использовании профиля оптимизации Performance (Aggressive)

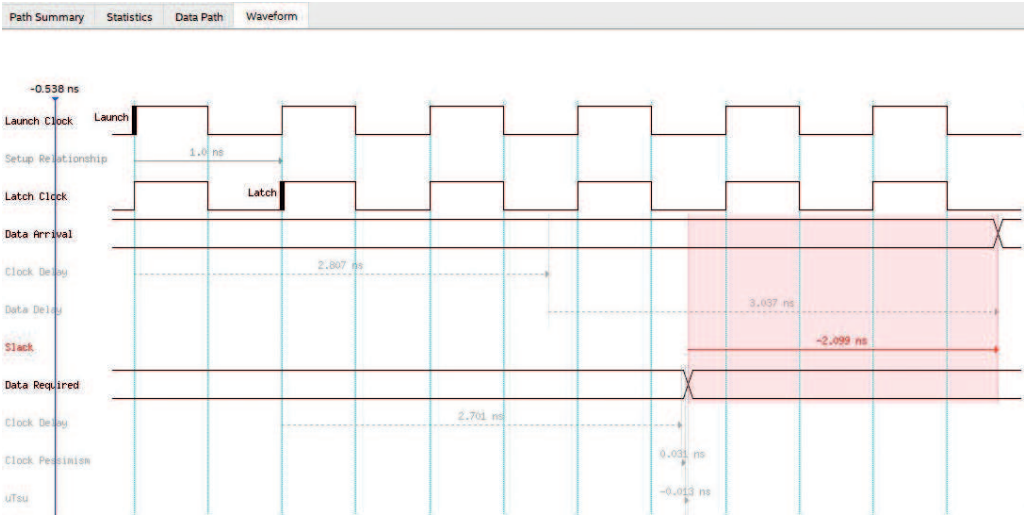


Рис. 3.43 Временны́е диаграммы при использовании профиля оптимизации Balanced (Normal flow)

Таблица 3.5 Сравнение профилей оптимизации для модуля b2_enc_if

Тип оптимизации	Slack, нс	Data Delay, нс	Total Delay IC, нс	Total Delay CELL, нс
Balanced (Normal flow)	–2,099	3,037	1,944	0,886
Performance (High effort)	–1,821	2,792	1,602	0,983
Performance (Aggressive)	–1,682	2,620	1,381	1,032
Power (High effort)	–1,730	2,701	1,228	1,266
Power (Aggressive)	–2,145	3,116	1,739	1,170
Area (Aggressive)	–2,099	3,037	1,944	0,886

Дополнительное задание для самостоятельной работы

Синтезируйте описанные выше параметрические шифраторы и дешифраторы с различными настройками профилей оптимизации и сравните параметры полученных устройств. Опишите, как отличаются пути прохождения сигналов и чем это обусловлено.

3.5 Упражнения

3.5.1 Основное задание

Выполните все дополнительные задания данной главы.

3.5.2 Задания для самостоятельной работы

1. Разработайте дешифратор для семисегментного индикатора. Обеспечьте возможность вывода на индикатор всех шестнадцатеричных цифр.
2. Разработайте конвертор **кода Грея** в унарный код.
3. Разработайте конвертор **кода Грея** в двоичный код.
4. Разработайте модуль дешифратора **6в64** с использованием приведенных в данной главе примеров модулей меньшей разрядности.
5. Разработайте дешифратор с использованием оператора **if**.
6. Разработайте конвертор унарного кода в **код Грея**.
7. Разработайте конвертор **кода Грея** в двоичный код.
8. Разработайте модули дешифратора **5в32** с использованием оператора сдвига и оператора **case**. Сравните временны́е характеристики этих модулей.
9. Разработайте модуль приоритетного шифратора **32в5** и проведите тестирование его работы.
10. Разработайте модуль для отображения двухразрядных десятичных чисел на двух семисегментных индикаторах.

11. Проведите анализ различных профилей оптимизации и разработайте несколько вариантов шифратора **16в4** и дешифратора **4в16**. Сравните получившиеся результаты при различных профилях оптимизации. Проанализируйте и опишите получившиеся результаты.

3.5.3 Контрольные вопросы

1. Опишите, что такое шифратор. Каковы различия между приоритетным и неприоритетным шифраторами?
2. Опишите, как реализовать неприоритетный шифратор с помощью операторов **assign**, **if** и **case**. В чем их различия? Какой способ лучше?
3. Опишите, как реализовать приоритетный шифратор с помощью операторов **assign** и **if**. В чем их различия? Какой способ лучше?
4. Что характеризуют такие параметры, как задержка распространения ($\mathbf{t_{pd}}$) и задержка реакции ($\mathbf{t_{CD}}$)?
5. Что такое критический путь? Почему следует стремиться сократить критические пути в комбинационной части цифровых схем?
6. Опишите предназначение дешифратора и его реализацию на **Verilog**. Приведите несколько различных вариантов.
7. Как осуществлять оптимизацию при синтезе в **САПР Quartus Prime**?
8. Опишите различные профили оптимизации в **САПР Quartus Prime**.

Александр Романов, Юрий Панчул

Цифровой синтез: практический курс

Глава 4. Мультиплексор, демультиплексор и селектор. Построение иерархических модулей

Содержание

*Глава содержит базовые сведения о различных способах реализации мультиплексоров, демультиплексоров и селекторов, об особенностях грамотного «стиля разработки кода» (**coding style**) на **Verilog**, а также приводятся примеры возможных ошибок, которые разработчики могут допускать, и пути их исправления. Вводится понятие модульности, приведен пример использования директив компилятора, а также показано, как создавать параметризированные модули с помощью параметров.*

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime** (есть студенческая версия);*
- *пакет **ModelSim Altera Edition** или программы **Icarus Verilog** и **GTKWave**;*
- *отладочная плата компании **Terasic DE10Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах[1] к данной книге).*

4.1 Проектирование мультиплексоров

Мультиплексором (**MUX**) называют комбинационное логическое устройство, предназначенное для управления передачей данных от нескольких источников на один выходной канал. В соответствии с определением мультиплексор должен иметь один выход и два типа входов (информационный и адресный). Код, поступающий на адресный вход, определяет, какой из информационных входов в данный момент подключен к выходу. Если количество адресных входов мультиплексора равно **n**, то максимально возможное количество его информационных входов будет $\mathbf{2^n}$. Такой мультиплексор называют **полным**, а если информационных входов меньше – мультиплексор **неполный**.

Следует хорошо понимать, что мультиплексор является комбинационным устройством, а не последовательностным. Это значит, что изменение сигналов на входе мультиплексора непосредственно влечет изменение на выходе (через время, определяемое задержкой распространения сигнала).

4.1.1 Однобитный мультиплексор 2в1

Рассмотрим различные способы создания мультиплексоров на примере двухвходового однобитного мультиплексора. Мультиплексор имеет два однобитных информационных входа **d0** и **d1**, один адресный вход **sel** и однобитный выход, на который коммутируются **d0** или **d1** в зависимости от **sel**.

Поскольку устройство комбинационное, то его всегда можно описать в виде комбинационной функции в базисе «**И**», «**ИЛИ**», «**НЕ**». В рассматриваемом примере функция имеет следующий вид:

[1] https://github.com/RomeoMe5/DDLM.

$$y = (sel \ \& \ d1) \mid ((\sim sel) \ \& \ d0).$$

Пример простейшей реализации мультиплексора на языке **Verilog** приведен ниже:

```
module b1_mux_2_1_comb
(
   input d0,
   input d1,
   input sel,
   output y
);
   assign y = (sel & d1) | ((~sel) & d0);
endmodule
```

Листинг 4.1 Комбинационный мультиплексор 2в1

Если количество входов мультиплексора и их разрядность больше, чем в приведенном примере, то синтез комбинационной функции, описывающей мультиплексор, становится нетривиальной задачей. Поэтому мультиплексоры обычно реализуют другими способами.

Одно из возможных решений – использование тернарного оператора (оператора выбора). Оно продемонстрированно в следующем примере:

```
module b1_mux_2_1_sel
(
   input d0,
   input d1,
   input sel,
   output y
);
   assign y = sel ? d1 : d0;
endmodule
```

Листинг 4.2 Мультиплексор 2в1 на основе тернарного оператора

Этот же мультиплексор можно реализовать и с помощью условного оператора (**if**):

```
module b1_mux_2_1_if
(
   input d0,
   input d1,
   input sel,
   output reg y
);
   always@(*)
      begin
         if(sel)
            y = d1;
         else
```

```
            y = d0;
      end
endmodule
```

Листинг 4.3 Мультиплексор 2в1 на основе условного оператора

Наиболее предпочтительной реализацией является использование оператора множественного выбора (**case**), поскольку он позволяет проще всего описывать сложные мультиплексоры для большого количества входов:

```
module b1_mux_2_1_case
(
   input d0,
   input d1,
   input sel,
   output reg y
);
   always@(*)
      begin
         case (sel)
            0: y = d0;
            1: y = d1;
         endcase
      end
endmodule
```

Листинг 4.4 Мультиплексор 2в1 на основе оператора множественного выбора

Предлагаемые реализации мультиплексоров выполняют одну и ту же функцию, в чем можно убедиться, проведя моделирование:

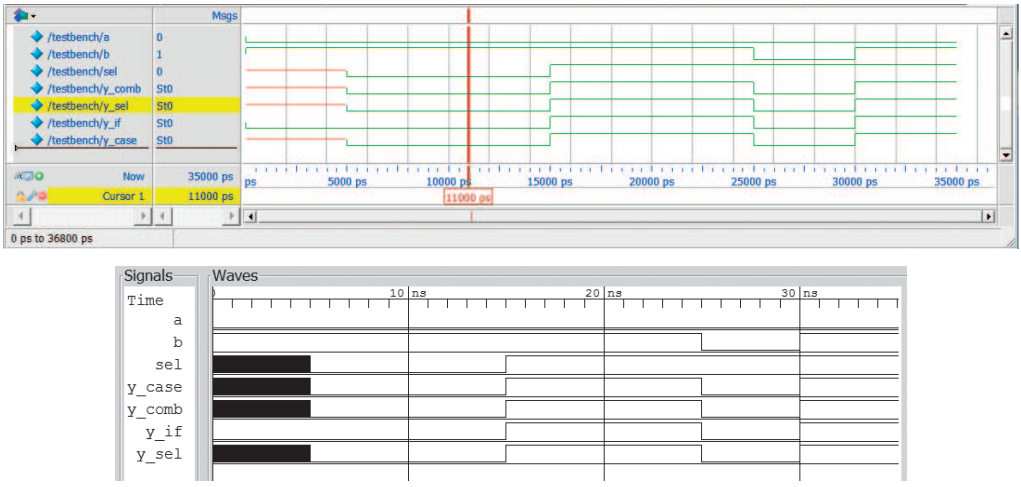


Рис. 4.1 Результаты моделирования различных реализаций однобитных мультиплексоров 2в1

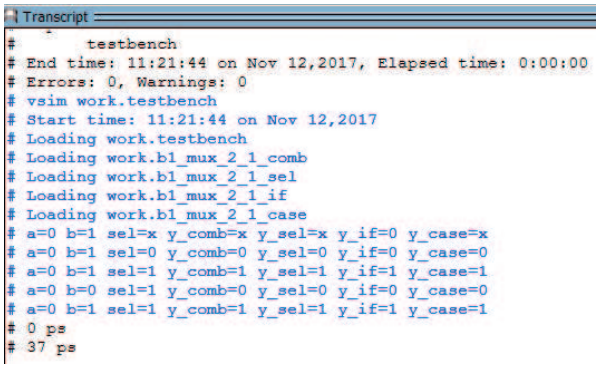


Рис. 4.1 (Продолжение)

Содержимое тестбенча (тестового окружения) приведено ниже:

```
`timescale 1 ns / 100 ps
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
module testbench;
   // input and output test signals
   reg a;
   reg b;
   reg sel;
   wire y_comb;
   wire y_sel;
   wire y_if;
   wire y_case;

   // creating the instance of the module we want to test
   b1_mux_2_1_comb b1_mux_2_1_comb (a, b, sel, y_comb);
   b1_mux_2_1_sel b1_mux_2_1_sel (a, b, sel, y_sel);
   b1_mux_2_1_if b1_mux_2_1_if (a, b, sel, y_if);
   b1_mux_2_1_case b1_mux_2_1_case (a, b, sel, y_case);

   initial
      begin
         a = 1'b0;
         b = 1'b1;
         #5;
         sel = 1'b0; // sel change to 0; a -> y
         #10;
         sel = 1'b1; // sel change to 1; b -> y
         #10
         b = 1'b0; // b change; y changes too. sel == 1'b1
         #5
         b = 1'b1;
         #5; // pause
```

```
      end
   // do at the beginning of the simulation
   // print signal values on every change
   initial
      $monitor("a=%b b=%b sel=%b y_comb=%b y_sel=%b y_if=%b y_case=%b",
           a, b, sel, y_comb, y_sel, y_if, y_case);
   // do at the beginning of the simulation
   initial
      $dumpvars; //iverilog dump init
endmodule
```

Листинг 4.5 Тестбенч для моделирования различных реализаций однобитных мультиплексоров 2в1

Различия заключаются лишь в том, что мультиплексор на основе оператора **if** интерпретирует неопределенный сигнал **sel (1'bx)** как **0** и соответственно коммутирует **0**-й вход на выход, когда в остальных случаях на выходе – неопределенное состояние.

Несмотря на то что мультиплексоры работают одинаково, их **RTL**-реализация (схемотехническое представление после конвертации в **RTL**-код) в **Quartus Prime** отличается:

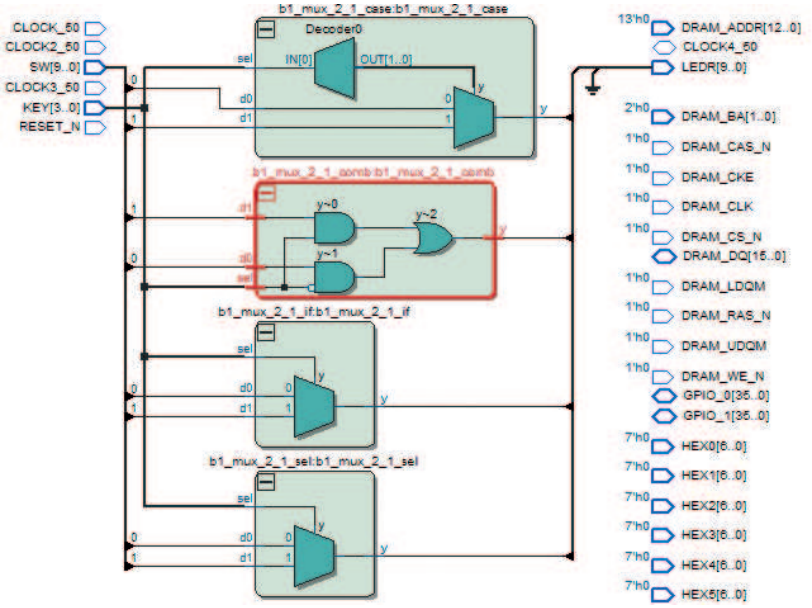


Рис. 4.2 RTL-представление различных реализаций однобитных мультиплексоров 2в1

Комбинационный мультиплексор представляется в виде соединения логических элементов, а **case** – в виде мультиплексора, на вход которого поступает декодированный сигнал **sel**. **IF** и **SELECT** реализации мультиплексоров представляются одинаково в виде мультиплексора.

Тем не менее логика работы всех реализаций мультиплексоров одинакова, а при синтезе любой из мультиплексоров будет реализован одинаково как часть комбинационного блока.

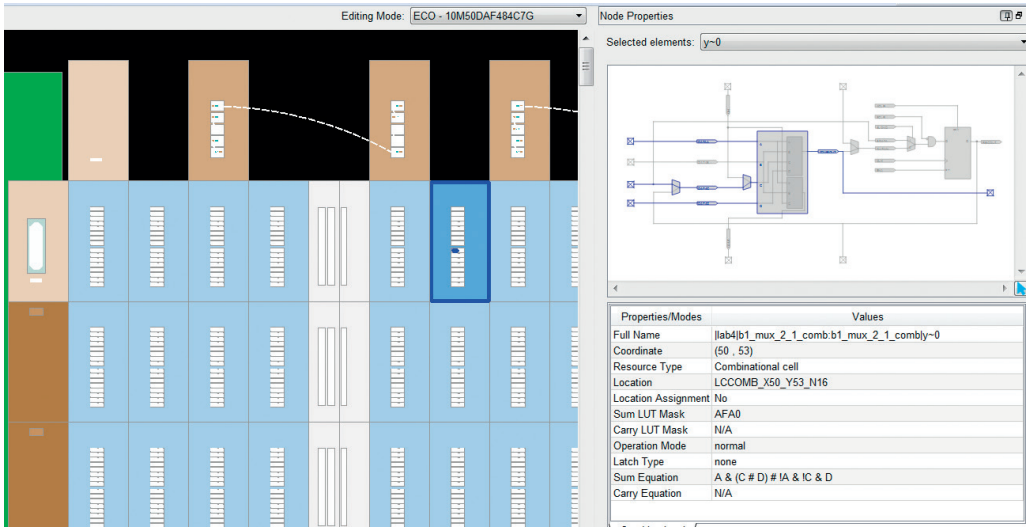


Рис. 4.3 Мультиплексор 2в1 в Chip Planner Quartus Prime

Дополнительное задание для самостоятельной работы

Исследуйте работу и проведите моделирование еще одной версии однобитного мультиплексора, реализованного с помощью операции конкатенации:

```
module b1_mux_2_1_concate
(
   input d0,
   input d1,
   input sel,
   output y
);
   wire [1:0] dataIn;
   assign dataIn = {d1,d0};
   assign y = dataIn[sel];
endmodule
```

Листинг 4.6 Мультиплексор 2в1, реализованный с помощью операции конкатенации

4.1.2 Двухбитный мультиплексор 2в1

В рассматриваемых выше примерах входы мультиплексора являлись однобитными, если же их размерность выше (например, 2), следует учитывать, что такие входы являются шинами.

В случае реализации мультиплексоров, использующих операторы **if** или **case**, сложностей не возникает, поскольку размеры входов и выходов просто увеличиваются:

```
module b2_mux_2_1_sel
(
   input [1:0] d0,
   input [1:0] d1,
   input sel,
   output [1:0] y
);
   assign y = sel ? d1 : d0;
endmodule

module b2_mux_2_1_if
(
   input [1:0] d0,
   input [1:0] d1,
   input sel,
   output reg [1:0] y
);
   always@(*)
      begin
         if(sel)
            y = d1;
         else
            y = d0;
      end
endmodule

module b2_mux_2_1_case
(
   input [1:0] d0,
   input [1:0] d1,
   input sel,
   output reg [1:0] y
);
   always@(*)
      begin
         case (sel)
            0: y = d0;
            1: y = d1;
         endcase
      end
endmodule
```

Листинг 4.7 Двухбитный мультиплексор 2в1, реализованный с помощью различных операций

С комбинационным мультиплексором существует опасность сделать грубую ошибку, которая сделает работу мультиплексора некорректной.

```
module b2_mux_2_1_comb_incorrect
(
   input [1:0] d0,
   input [1:0] d1,
   input sel,
   output [1:0] y
);
   assign y = (sel & d1) | ((~sel) & d0);
endmodule
```

Листинг 4.8 Некорректный комбинационный двухбитный мультиплексор 2в1

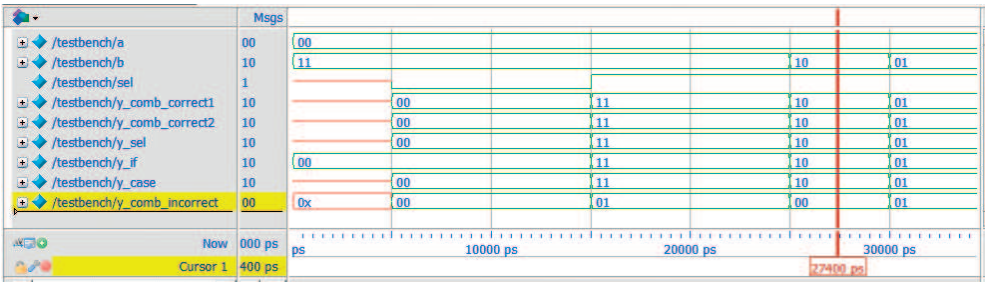


Рис. 4.4 Результаты моделирования двухбитного мультиплексора 2в1

Следует отметить, что в данном примере используются побитовые операции, в связи с чем надо следить за размерностью участвующих в них операндов, чтобы не получить неожиданный результат.

Примеры правильной реализации комбинационного мультиплексора приведены ниже:

```
module b2_mux_2_1_comb_correct1
(
   input [1:0] d0,
   input [1:0] d1,
   input sel,
   output [1:0] y
);
   assign y[0] = (sel & d1[0]) | ((~sel) & d0[0]);
   assign y[1] = (sel & d1[1]) | ((~sel) & d0[1]);
endmodule

module b2_mux_2_1_comb_correct2
(
   input [1:0] d0,
   input [1:0] d1,
   input sel,
```

```
   output [1:0] y
);
   wire [1:0] select;
   assign select = {2{sel}};
   assign y = (select & d1) | (~select & d0);
endmodule
```

Листинг 4.9 Правильный комбинационный двухбитный мультиплексор 2в1

Дополнительное задание для самостоятельной работы

Синтезируйте вышеупомянутые реализации мультиплексоров в **Quartus Prime**. Рассмотрите их представление в **RTL Viewer**, объясните, в чем состоит ошибка в модуле **b2_mux_2_1_comb_incorrect** (листинг 4.8).

4.1.3 Двухбитный мультиплексор 4в1

Когда количество входов увеличивается, увеличивается также и ширина входного сигнала **select**. Размер такого сигнала зависит от количества информационных входов и может определяться уравнением: $N_{sel} = [\log_2 N]$.

Такие мультиплексоры легче всего реализуются с помощью оператора **case**:

```
module b2_mux_4_1_case
(
   input [1:0] d0, d1, d2, d3,
   input [1:0] sel,
   output reg [1:0] y
);
   always @(*)
      case (sel)
         2'b00: y = d0;
         2'b01: y = d1;
         2'b10: y = d2;
         2'b11: y = d3;
      endcase
endmodule
```

Листинг 4.10 Двухбитный мультиплексор 4в1, реализованный с помощью оператора case

Ниже представлена еще одна версия двухбитного мультиплексора 4в1, реализованного на основе тернарного оператора:

```
module b2_mux_4_1_sel
(
   input [1:0] d0, d1, d2, d3,
   input [1:0] sel,
   output [1:0] y
);
```

```
   assign y = sel [1] ? (sel [0] ? d3 : d2)
        : (sel [0] ? d1 : d0);
endmodule
```

Листинг 4.11 Двухбитный мультиплексор 4в1, реализованный с помощью условного оператора

4.1.4 Иерархический подход при проектировании цифровых систем

Использование тернарного оператора с увеличенным количеством входов приводит к трудночитаемому коду. Улучшить читаемость кода можно путем использования иерархического подхода, в котором более сложные модули построены на более простых модулях. Этот метод является общим и применяется не только для мультиплексоров.

Рассмотрим следующую реализацию двухбитного мультиплексора 4в1, построенного на основе двухбитного мультиплексора 2в1 (листинг 4.2), разработанного в разделе 4.1.1:

```
module b2_mux_4_1_block
(
   input [1:0] d0, d1, d2, d3,
   input [1:0] sel,
   output [1:0] y
);
   wire [1:0] w01, w23;
   b2_mux_2_1_sel mux0(.d0(d0), .d1(d1), .sel(sel[0]), .y(w01));
   b2_mux_2_1_sel mux1(.d0(d2), .d1(d3), .sel(sel[0]), .y(w23));
   b2_mux_2_1_sel mux2(.d0(w01), .d1(w23), .sel(sel[1]), .y(y));
endmodule
```

Листинг 4.12 Двухбитный модульный мультиплексор 4в1

Результаты моделирования всех представленных вариантов мультиплексоров совпадают:

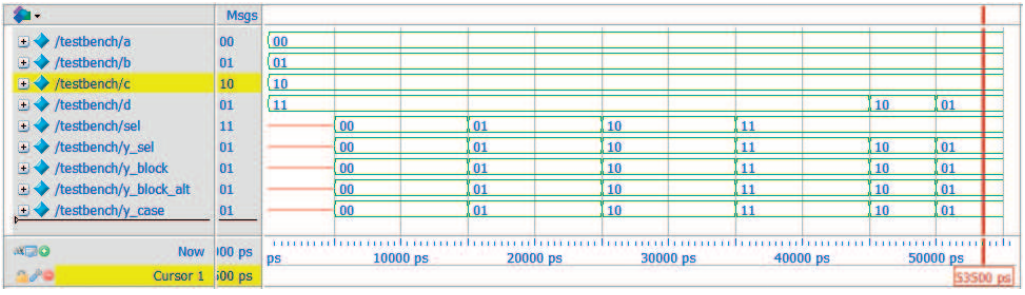


Рис. 4.5 Результаты моделирования различных реализаций мультиплексоров 4в1

Результаты синтеза различных реализаций мультиплексоров существенно различаются:

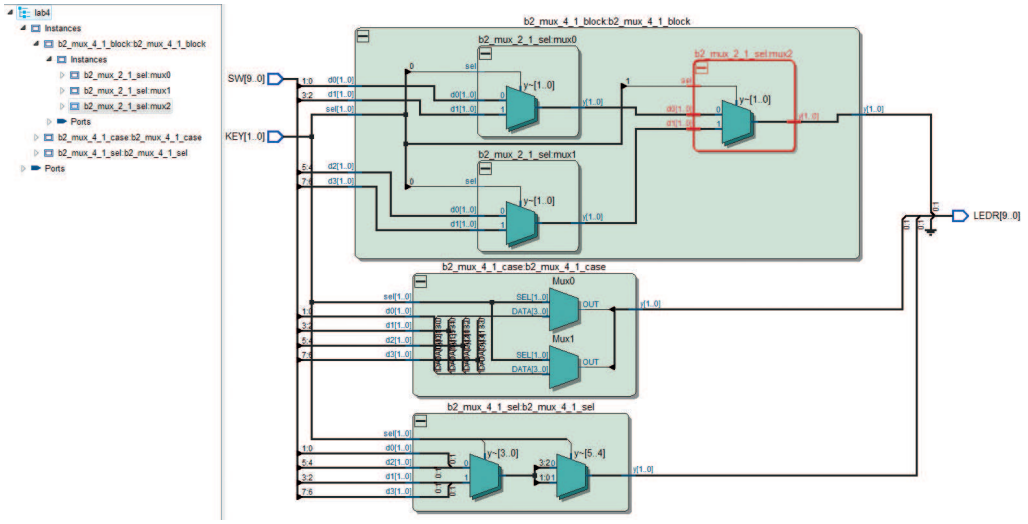


Рис. 4.6 RTL-схема различных реализаций мультиплексоров 4в1

Еще один способ улучшения кода, специфичный для мультиплексоров, – это их реализация с использованием оператора case. Можно построить еще одну альтернативную реализацию модульного двухбитного мультиплексора 4в1 на основе однобитного мультиплексора 4в1:

```
module b1_mux_4_1_case
(
   input d0, d1, d2, d3,
   input [1:0] sel,
   output reg y
);
   always @(*)
      case (sel)
         2'b00: y = d0;
         2'b01: y = d1;
         2'b10: y = d2;
         2'b11: y = d3;
      endcase
endmodule

module b2_mux_4_1_block_alt
(
   input [1:0] d0, d1, d2, d3,
   input [1:0] sel,
   output [1:0] y
);
```

```
   b1_mux_4_1_case hi(.d0(d0[1]), .d1(d1[1]), .d2(d2[1]), .d3(d3[1]),
      .sel(sel), .y(y[1]));
   b1_mux_4_1_case lo(.d0(d0[0]), .d1(d1[0]), .d2(d2[0]), .d3(d3[0]),
      .sel(sel), .y(y[0]));
endmodule
```

Листинг 4.13 Альтернативная реализация двухбитного модульного мультиплексора 4в1

Обратите внимание, что в этой реализации он представляет собой два мультиплексора 4в1, которые работают параллельно на **первой** стадии, в отличие от предыдущей схемы, где информационные сигналы проходят через **две** стадии. Это хорошо иллюстрируется **RTL**-представлением, полученным в **Quartus Prime**:

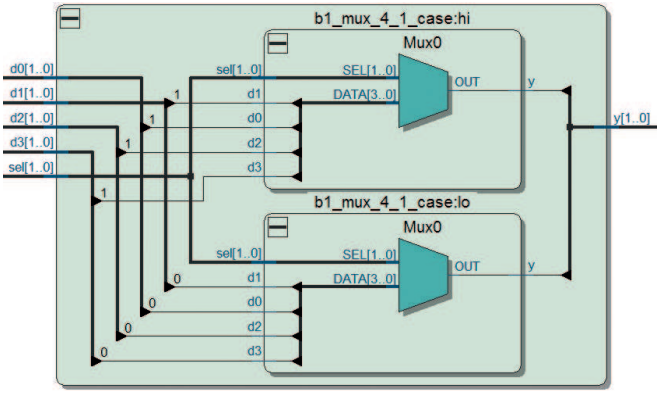


Рис. 4.7 RTL-схема модульного мультиплексора 4в1

Минимизация количества стадий в общем случае положительно сказывается на скорости работы мультиплексора, поскольку это приводит к уменьшению длины путей прохождения сигналов. При этом в синтезирующих **САПР** обычно происходит оптимизация схемы и ее привязка к ресурсам кристалла. Поэтому итоговые схемы мультиплексоров, реализованных различными методами, с большой вероятностью будут одинаковыми.

Дополнительное задание для самостоятельной работы

Проанализируйте полученные в **Quartus Prime RTL**-схемы мультиплексоров 4в1. Объясните, какая реализация, по вашему мнению, является наиболее быстрой; какая занимает меньше всего пространства на чипе; какой подход проще всего масштабировать.

4.1.5 Неполный мультиплексор 3в1

До сих пор все реализации мультиплексоров были полными, то есть количество входных информационных сигналов соответствовало количеству комбинаций, возможных на адресном входе. Рассмотрим реализацию неполного мультиплексора с тремя информационными входами (соответственно, адресный вход будет двухбитным). Вход **d0** соответствует комбинационному адресу **2'b00, d1 – 2'b01** и **d2 – 2'b10**.

Опишем это устройство с помощью оператора **case**:

```
module b2_mux_3_1_case_latch
(
   input [1:0] d0, d1, d2,
   input [1:0] sel,
   output reg [1:0] y
);
   always @(*)
      case (sel)
         2'b00: y = d0;
         2'b01: y = d1;
         2'b10: y = d2;
      endcase
endmodule
```

Листинг 4.14 Мультиплексор 3в1 с защелкой

Результаты моделирования мультиплексора 3в1:

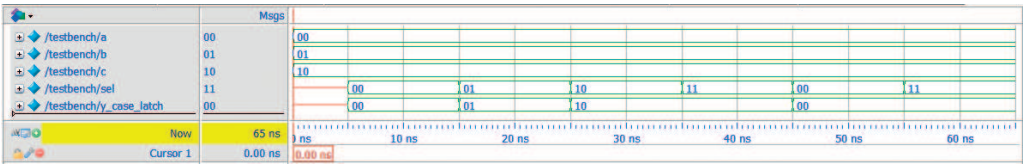


Рис. 4.8 Результаты моделирования мультиплексора 3в1

Для комбинаций адресных входов **2'b00**, **2'b01** и **2'b10** мультиплексор ведет себя так, как ожидалось, коммутируя соответствующую информацию с входов на выход. Поведение мультиплексора с неразрешенным комбинационным адресным входом **2'b11** следующее: устройство сохраняет свое предыдущее состояние. Это приводит к появлению регистра защелки (**latch**) на выходе, что обычно является недопустимой ситуацией.

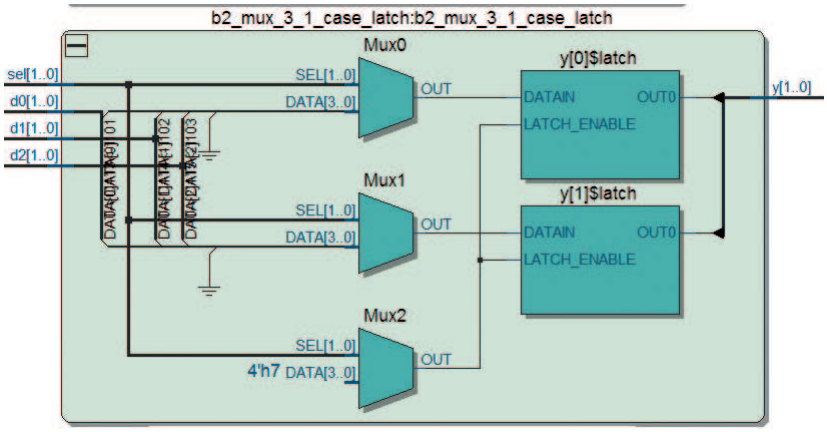


Рис. 4.9 Мультиплексор 3в1 с защелкой в RTL-схеме

Это распространенная ошибка при использовании оператора **case** с неполным набором комбинаций; та же проблема будет возникать в неполном мультиплексоре, реализованном с помощью операторов **if**.

Данная ошибка исправляется путем добавления строки **default**, описывающей выбор по умолчанию, как в следующем примере:

```
module b2_mux_3_1_case_correct
(
   input [1:0] d0, d1, d2,
   input [1:0] sel,
   output reg [1:0] y
);
   always @(*)
      case (sel)
         2'b00: y = d0;
         2'b01: y = d1;
         default: y = d2;
      endcase
endmodule
```

Листинг 4.15 Мультиплексор 3в1 без защелки

Когда комбинации **2'b10** и **2'b11** поступают на адресный вход, сигнал с входа **d2** будет переключен на выход.

Другое решение: установить на выходе состояние *2'bxx* в случае неразрешенных комбинаций на входе.

```
module b2_mux_3_1_casex_correct
(
   input [1:0] d0, d1, d2,
   input [1:0] sel,
   output reg [1:0] y
);
   always @(*)
      case (sel)
         2'b00: y = d0;
         2'b01: y = d1;
         2'b10: y = d2;
         default: y=2'bxx;
      endcase
endmodule
```

Листинг 4.16 Мультиплексор 3в1 с x-состоянием

Результаты моделирования мультиплексора 3в1:

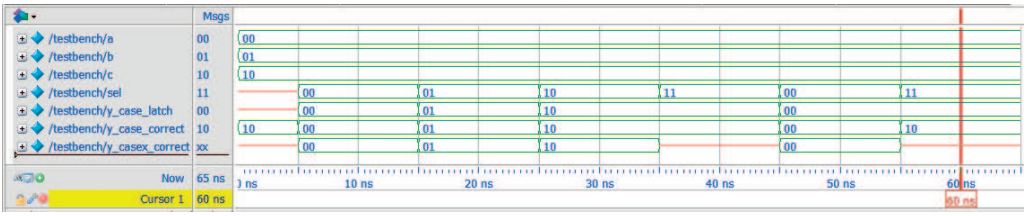


Рис. 4.10 Результаты моделирования мультиплексора 3в1

Схематически такие мультиплексоры синтезируются без регистра защелки:

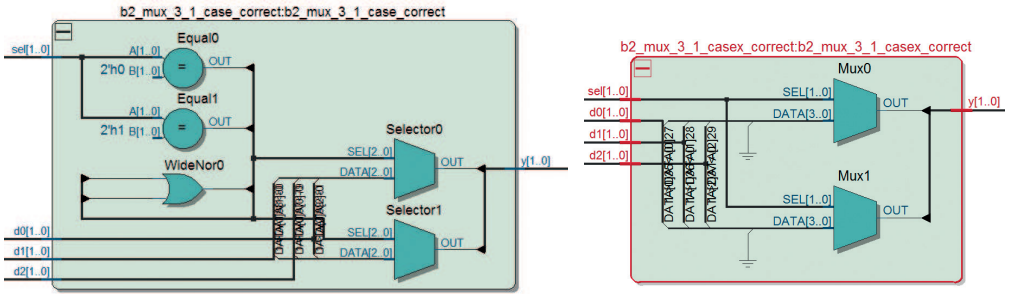


Рис. 4.11 Мультиплексоры 3в1 в RTL-схеме

Часто разработчики используют директиву

```
// synopsys full_case parallel_case
```

Однако ее применения правильнее избегать из-за возможных неприятных последствий, описанных в статье C. E. Cummings[1].

Дополнительное задание для самостоятельной работы

Разработайте мультиплексор 3в1 с использованием оператора **if**, чтобы он был реализован с помощью регистра защелки, а затем исправьте его, удалив регистр защелки. Представьте результаты моделирования и синтеза в **RTL** и объясните их. Загрузите результаты в **Technology map viewer** и дайте пояснения тому, что получится.

4.1.6 Разработка логических функций на мультиплексорах

Из предыдущего изложения может сложиться ошибочное мнение, что мультиплексор – это всегда комбинационное устройство, состоящее из логических элементов. На самом деле это не всегда так, поскольку мультиплексор сам может быть отдельным элементом библиотеки стандартных ячеек для проектирования **ASIC** и реализован на транзисторном уровне. Эта идея хорошо проиллюстрирована в книге «**CMOS VLSI design: a circuits and systems perspective**» (раздел 1.4.8 Multiplexers)[2], где показано, как построить мультиплексоры небольшой разрядности на транзисторном уровне.

[1] *Cummings C. E.* full_case parallel_case, the Evil Twins of Verilog Synthesis. SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings.

[2] *Weste N. H., Harris D.* CMOS VLSI design: a circuits and systems perspective. Pearson Education India, 2015.

Отсюда также следует еще одна идея – мультиплексор сам может быть «строительным блоком» для реализации логических элементов и даже функций. Например, мультиплексор 4в1, описание которого дано в разделе 4.1.3, имеет **четыре** информационных входа и **два** управляющих. Если на первые три информационных входа **d0**, **d1**, **d2** подать состояние логической единицы, а на последний **d3** – логического нуля, то получим логический элемент **И-НЕ**, входами которого являются **sel[0]** и **sel[1]**, а выходом – **y**. Рассматривая мультиплексор как таблицу истинности, где его управляющие входы соответствуют комбинациям входных переменных таблицы, а на информационных входах стоят соответствующие значения самой функции, на мультиплексорах большой разрядности можно строить логические функции любой сложности. Более подробно с примерами и иллюстрациями данный подход можно изучить в книге «**Цифровая схемотехника и архитектура компьютера**»[1] в разделе «Логика на мультиплексорах».

Идея создания логических элементов из мультиплексоров широко распространена в сообществе разработчиков цифровых систем, а задачка «Реализуйте такой-то логический элемент на двухвходовом мультиплексоре» является типичным вопросом в интервью для соискателей работы в этой сфере[2]. И это неудивительно, поскольку мультиплексоры находят широчайшее применение в цифровом синтезе и являются важными элементами в структуре арифметических блоков (глава 5), в цифровых автоматах (главы 8, 9), при реализации софт-процессоров (глава 11) и т. д. Особая роль отводится мультиплексорам для реализации совместного использования и распределения ресурсов в арифметических операциях при разработке систем цифровой обработки сигналов (данный подход хорошо продемонстрирован в книге «**A Verilog HDL Primer**»[3] в главе 4 «Оптимизация моделей»).

Дополнительное задание для самостоятельной работы

Изучите примеры в книге «**Цифровая схемотехника и архитектура компьютера**», после чего разработайте таблицу истинности и изобразите условную схему подключения мультиплексора 4в1, который реализует логический элемент **исключающее ИЛИ**. Опишите логическую функцию $y = \overline{A} \wedge B \wedge C \vee A \wedge \overline{B} \wedge \overline{C}$ с помощью мультиплексора 8в1.

4.2 Демультиплексор

Демультиплексор выполняет функцию, обратную **мультиплексору**, – коммутирует входной сигнал на нужный выход, номер которого задается селектором. На остальных выходах устанавливается значение **0**.

---

1 *Дэвид М. Харрис, Сара Л. Харрис*. Цифровая схемотехника и архитектура компьютера. М.: ДМК Пресс, 2017.

2 *Johnson T.* Digital logic, RTL Verilog: interview questions: a practical study guide for design engineers. 2015.

3 *Bhasker J.* A Verilog HDL Primer. Star Galaxy Publishing. 1999.

Существует много реализаций демультиплексоров. Элегантный пример демультиплексора приведен ниже (представляет собой сдвиг бита входных данных на соответствующую позицию в шине, состоящей из выходных портов):

```
module b1_demux_1_4_shift
(
   input din,
   input [1:0] sel,
   output reg dout0,
   output reg dout1,
   output reg dout2,
   output reg dout3
);
   always @(*)
      {dout3, dout2, dout1, dout0}= din<<sel
endmodule
```

Листинг 4.17 Демультиплексор 1в4 с одноразрядным сдвигом

Наиболее естественной реализацией демультиплексора является использование оператора **case**:

```
module b1_demux_1_4_case
(
   input din,
   input [1:0] sel,
   output reg dout0,
   output reg dout1,
   output reg dout2,
   output reg dout3
);
   always @(*)
      case (sel)
         2'b00:
         begin
            dout0 = din;
            dout1 = 0;
            dout2 = 0;
            dout3 = 0;
         end
         2'b01:
         begin
            dout0 = 0;
            dout1 = din;
            dout2 = 0;
            dout3 = 0;
         end
         2'b10:
```

```
        begin
           dout0 = 0;
           dout1 = 0;
           dout2 = din;
           dout3 = 0;
        end
        2'b11:
        begin
           dout0 = 0;
           dout1 = 0;
           dout2 = 0;
           dout3 = din;
        end
     endcase
endmodule
```

Листинг 4.18 Однобитный демультиплексор 1в4, реализованный с помощью оператора case

Особенно такой подход удобен при реализации масштабирования портов с использованием оператора **parameter**:

```
module bn_demux_1_4_case
#(parameter DATA_WIDTH=2)
(
   input [DATA_WIDTH-1:0] din,
   input [1:0] sel,
   output reg [DATA_WIDTH-1:0] dout0,
   output reg [DATA_WIDTH-1:0] dout1,
   output reg [DATA_WIDTH-1:0] dout2,
   output reg [DATA_WIDTH-1:0] dout3
);
```

Листинг 4.19 N-битный параметризованный демультиплексор, реализованный с помощью оператора case

Задавая значение параметра **DATA_WIDTH** (в коде листинга 4.20 это выглядит как **#(2)** для двухбитной реализации), можно получить демультиплексор с необходимой размерностью входов и выходов.

```
module lab4
(
   input [ 1:0] KEY,
   input [ 9:0] SW,
   output [ 9:0] LEDR
);
`ifdef CASE
   bn_demux_1_4_case #(2) bn_demux_1_4_case (.din(SW[1:0]),
      .sel(KEY[1:0]),
```

```
   .dout0(LEDR[1:0]), .dout1(LEDR[3:2]), .dout2(LEDR[5:4]),
      .dout3(LEDR[7:6]));
`else
   b2_demux_1_4_block b2_demux_1_4_block (.din(SW[1:0]),
      .sel(KEY[1:0]),
   .dout0(LEDR[1:0]), .dout1(LEDR[3:2]), .dout2(LEDR[5:4]),
      .dout3(LEDR[7:6]));
`endif
endmodule
```

Листинг 4.20 Пример модуля с конкретизацией параметров и директивами условной компиляции

Также возможно использование блочного подхода для синтеза демультиплексоров с большей размерностью портов, как это было показано в разделе 4.1.3.

```
module b2_demux_1_4_block
(
   input [1:0] din,
   input [1:0] sel,
   output [1:0] dout0,
   output [1:0] dout1,
   output [1:0] dout2,
   output [1:0] dout3
);

   b1_demux_1_4_case dmux0 (.din(din[0]), .sel(sel), .dout0(dout0[0]),
      .dout1(dout1[0]), .dout2(dout2[0]), .dout3(dout3[0]));
   b1_demux_1_4_shift dmux1 (.din(din[1]), .sel(sel), .dout0(dout0[1]),
      .dout1(dout1[1]), .dout2(dout2[1]), .dout3(dout3[1]));
endmodule
```

Листинг 4.21 Двухбитный модульный демультиплексор 1в4

При этом модуль может включать не только однотипные модули. Это хорошо проиллюстрировано на рис. 4.12, где представлена **RTL**-схема двухбитного демультиплексора на основе демультиплексоров, приведенных в листингах 4.17 и 4.18.

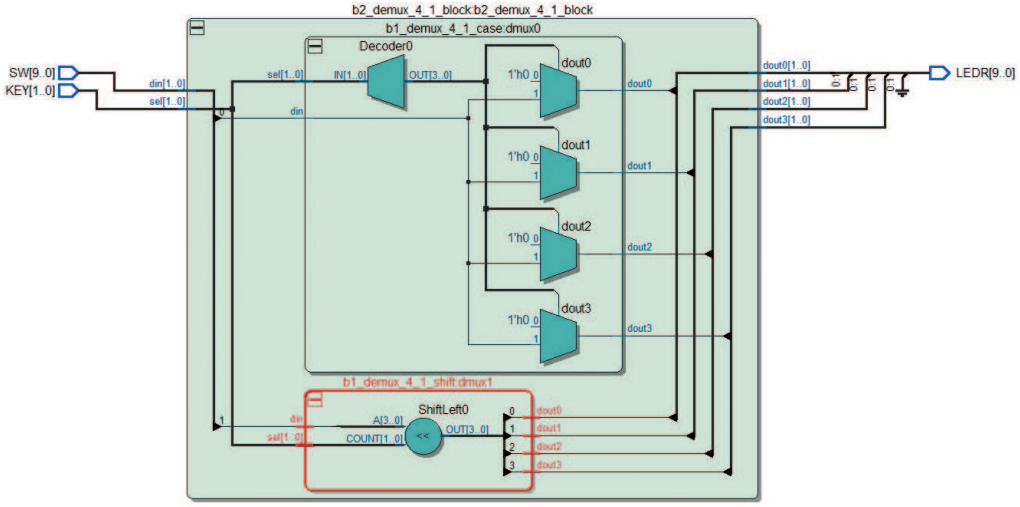


Рис. 4.12 Двухбитный модульный демультиплексор 1в4 в виде RTL-схемы

Результаты моделирования реализаций восьмибитного параметризированного демультиплексора и модульного демультиплексора, иллюстрирующие однотипную логику их поведения, приведены на рис. 4.13.

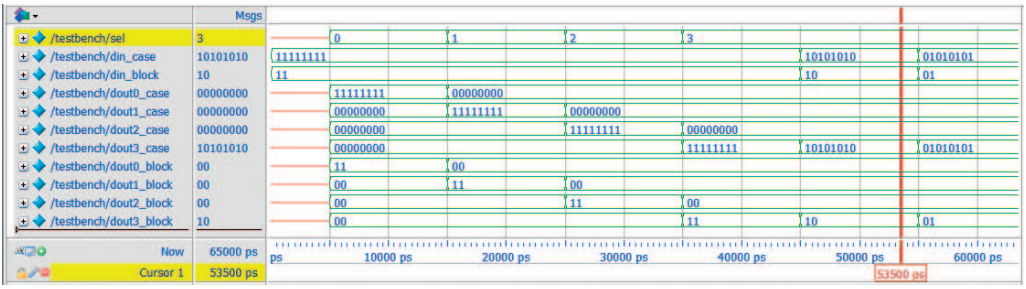


Рис. 4.13 Результаты моделирования демультиплексоров 1в4

Следует также обратить внимание на использование директив условной компиляции, которые по аналогии с языком **C/C++** позволяют «включать» и «выключать» некоторые фрагменты кода (в данном примере, из-за недостатка пинов на плате **DE10-Lite**, отладка двух модулей возможна только отдельно путем комментирования/раскомментирования строчки **`define CASE** в коде).

Дополнительное задание для самостоятельной работы

Измените значение параметра **DATA_WIDTH** в модуле демультиплексора на значения **4**, **8**, **16**, **32**, **64**, **128**. Оцените изменение потребляемых ресурсов после синтеза, опишите и объясните полученные зависимости.

4.3 Селектор

Селектором называют вариант мультиплексора, в котором адресный вход уже декодирован в единичное (**one-hot**) представление (например, из **3'b101** в **8'b00100000**). Данный специализированный вариант мультиплексора применяется для устройств с высокой частотой, где требуется разделить процессы декодирования и выбора из большого количества входов на несколько циклов. В этом случае обычный мультиплексор может быть реализован как последовательно соединенные декодер и селектор.

Пример реализации селектора и результаты его моделирования приведены ниже:

```
module bn_select_8_1_case
#(parameter DATA_WIDTH=8)
(
   input [DATA_WIDTH-1:0] d0,
   input [DATA_WIDTH-1:0] d1,
   input [DATA_WIDTH-1:0] d2,
   input [DATA_WIDTH-1:0] d3,
   input [DATA_WIDTH-1:0] d4,
   input [DATA_WIDTH-1:0] d5,
   input [DATA_WIDTH-1:0] d6,
   input [DATA_WIDTH-1:0] d7,
   input [7:0] sel,
   output reg [DATA_WIDTH-1:0] y
);

   always @(*)
      case (sel)
         8'b00000001: y=d0;
         8'b00000010: y=d1;
         8'b00000100: y=d2;
         8'b00001000: y=d3;
         8'b00010000: y=d4;
         8'b00100000: y=d5;
         8'b01000000: y=d6;
         8'b10000000: y=d7;
         default: y={DATA_WIDTH{1'bx}};
      endcase
endmodule
```

Листинг 4.22 N-битный селектор 8в1

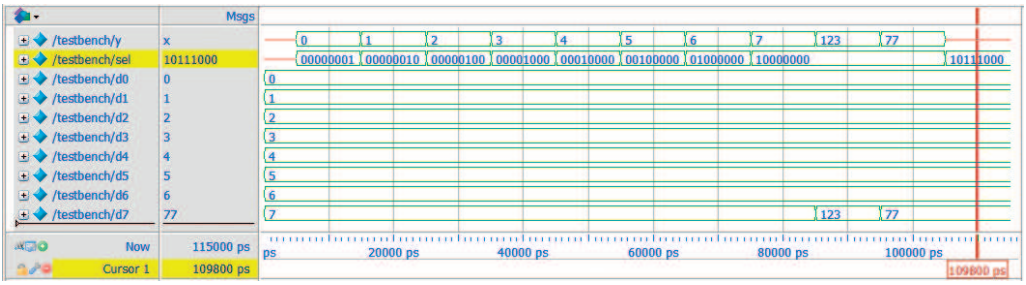


Рис. 4.14 Результаты моделирования N-битного селектора 8в1

Следует обратить внимание на необходимость предусмотреть значение по умолчанию (**default**) в операторе **case** для избежания появления регистра защелки.

Дополнительное задание для самостоятельной работы

Предложите другой вариант селектора, синтезируйте его и проведите моделирование. Сравните свою реализацию с предлагаемой в данном мануале.

4.4 Полностью параметризированный мультиплексор и селектор. Конструкция generate

В предыдущих примерах мы управляли размером входных портов с помощью оператора **parameter**, но было бы хорошо управлять и их количеством. В синтаксисе **Verilog-2001** данный подход реализуется созданием одного большого входа данных, объединяющего в себе несколько входов меньшего размера, как это сделано в примере, приведенном ниже:

```
module bn_mux_n_1_generate
#( parameter DATA_WIDTH = 8,
   parameter SEL_WIDTH = 2)
(
   input [((2**SEL_WIDTH)*DATA_WIDTH)-1:0] data,
   input [SEL_WIDTH-1:0] sel,
   output [DATA_WIDTH-1:0] y
);
wire [DATA_WIDTH-1:0] tmp_array [0:(2**SEL_WIDTH)-1];
genvar i;
generate
   for(i=0; i<2**SEL_WIDTH; i=i+1)
   begin: gen_array
      assign tmp_array[i] = data[((i+1)*DATA_WIDTH)-1:(i*DATA_WIDTH)];
   end
endgenerate
   assign y = tmp_array[sel];
endmodule
```

Листинг 4.23 Полностью параметризированный мультиплексор

Здесь **DATA_WIDTH** – это размер информационного входного сигнала, а **SEL_WIDTH** – размер адресного входного сигнала. То есть для 4-битного мультиплексора 8в1 размер информационного входного сигнала будет 32-разрядным, а размер адресного входного сигнала **sel** будет равен 3 битам (**data0** – это **data[3:0]**, **data1** – **data[7:4]** и т. д.).

Для описания работы такого мультиплексора удобно использовать конструкцию **generate**, позволяющую описать повторяющиеся блоки, – в данном случае мы разбиваем общий вход на массив шин шириной **DATA_WIDTH**, одна из которых коммутируется на выход в соответствии со значением на входе **sel**.

Здесь необходимо также отметить следующий момент код:

```
assign tmp_array[i] = data[((i+1)*DATA_WIDTH)-1:(i*DATA_WIDTH)];
```

не соответствует стандарту **Verilog-95** и может быть не понятен синтезирующей САПР. Выход из этой ситуации возможен путем применения оператора конкатенации следующим способом:

```
assign tmp_array[i] = {data[i*DATA_WIDTH+DATA_WIDTH-1],
data[i*DATA_WIDTH+DATA_WIDTH-2], …, data[i*DATA_WIDTH]};
```

Но тогда такой код теряет свою универсальность относительно **DATA_WIDTH** и становится слишком громоздким. Из этой ситуации можно выйти, добавив еще один вложенный цикл, где каждому биту **tmp_array** будет присваиваться соответствующий бит отдельно:

```
for(j=0; j<DATA_WIDTH; j=j+1)
   assign tmp[j] = data[i*DATA_WIDTH+DATA_WIDTH-1-j];
assign tmp_array[i] = tmp;
```

К счастью, современные **САПР** в массе своей уже поддерживают оператор частичного выбора (**part select**). Более того, начиная со стандарта **Verilog-2001** в языке появилась еще сокращенная запись такого оператора (**-:, +:**), которая позволяет исходную строку записать следующим образом:

```
assign tmp_array[i] = data[((i+1)*DATA_WIDTH)-1 -: DATA_WIDTH];
```

Более подробно об этом написано в книге «**Verilog–2001: A Guide to the New Features of the Verilog® Hardware Description Language**»[1].

Полученная конструкция полностью параметризируемого мультиплексора (листинг 4.23) в **RTL**-схеме синтезируется в виде набора мультиплексоров, которые коммутируют соответствующие биты входов на выход (рис. 4.15).

1 *Sutherland S.* Verilog-2001: A Guide to the New Features of the Verilog® Hardware Description Language (Vol. 652). Springer Science & Business Media, 2012.

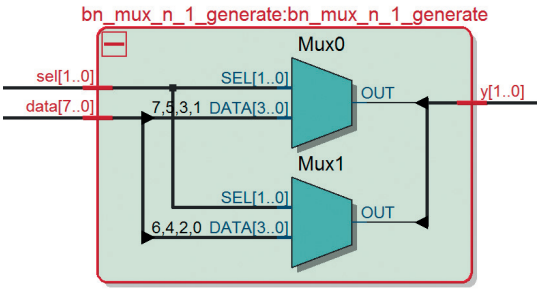


Рис. 4.15 Полностью параметризованный мультиплексор в виде RTL-схемы

Еще более красивый пример использования конструкции **generate** можно продемонстрировать на примере полностью параметризированного селектора:

```
module bn_selector_n_1_generate
#( parameter DATA_WIDTH = 8,
   parameter INPUT_CHANNELS = 2)
(
   input [(INPUT_CHANNELS*DATA_WIDTH)-1:0] data,
   input [INPUT_CHANNELS-1:0] sel,
   output [DATA_WIDTH-1:0] y
);
genvar i;
generate
   for(i=0;i<INPUT_CHANNELS;i=i+1)
   begin: gen_array
      assign y = sel[i] ? data[((i+1)*DATA_WIDTH)-1:(i*DATA_WIDTH)] :
         {DATA_WIDTH{1'bx}};
   end
endgenerate
endmodule
```

Листинг 4.24 Полностью параметризированный селектор

Следует обратить внимание на то, что в данном модуле предусмотрена обработка ситуации, когда вход **sel** некорректный (тогда на выходе будет состояние **'x**). Здесь также следует оговориться, что множество параллельных присваиваний **'x** может не поддерживаться синтезирующим САПР'ом. В такой ситуации каждый случай следует рассматривать индивидуально.

Используя конструкцию **generate**, можно создавать множество повторяющихся блоков и даже размножать модули, об этом более подробно рассказано в главе 10.

Дополнительное задание для самостоятельной работы

Ответьте на вопрос, почему нельзя модуль полностью параметризированного мультиплексора (листинг 4.23) реализовать по аналогии с селектором (листинг 4.24) следующим образом:

```
assign y = data[((sel+1)*DATA_WIDTH)-1:(sel*DATA_WIDTH)];
```

4.5 Некоторые хитрости при создании селекторов и мультиплексоров

Рассмотрим еще несколько примеров хитростей, которые можно использовать при создании мультиплексоров и селекторов.

4.5.1 Селектор, созданный на логических элементах И и ИЛИ

Как уже было сказано, селектор – это **one-hot** версия мультиплексора, удобная для применения в проектах, где требуется высокая частота функционирования модуля. Обычно селекторы создают на основе оператора **full case**, но существует также подход с использованием логических операций **И** и **ИЛИ**. Рассмотрим пример такого селектора, состоящего из **DATA_WIDTH*4**-битного информационного входного сигнала, одного 4-битного адресного входного сигнала и одного **DATA_WIDTH**-битного выходного сигнала:

```
module bn_selector_4_1_and_or
#( parameter DATA_WIDTH = 32)
(
   input [(DATA_WIDTH*4)-1:0] data,
   input [3:0] sel,
   output [DATA_WIDTH-1:0] y
);
   wire [(DATA_WIDTH*4)-1:0] mask;
   wire [(DATA_WIDTH*4)-1:0] masked_data;

   assign mask = {{DATA_WIDTH{sel[3]}},
      {DATA_WIDTH{sel[2]}},
      {DATA_WIDTH{sel[1]}},
      {DATA_WIDTH{sel[0]}}};
   assign masked_data = data & mask;
   assign y = masked_data[DATA_WIDTH*4-1:DATA_WIDTH*3]|
      masked_data[DATA_WIDTH*3-1:DATA_WIDTH*2]|
      masked_data[DATA_WIDTH*2-1:DATA_WIDTH]|
      masked_data[DATA_WIDTH-1:0];
endmodule
```

Листинг 4.25 И-ИЛИ селектор

Получившаяся **RTL**-схема селектора представлена ниже. Этот селектор полностью реализован с помощью логических элементов:

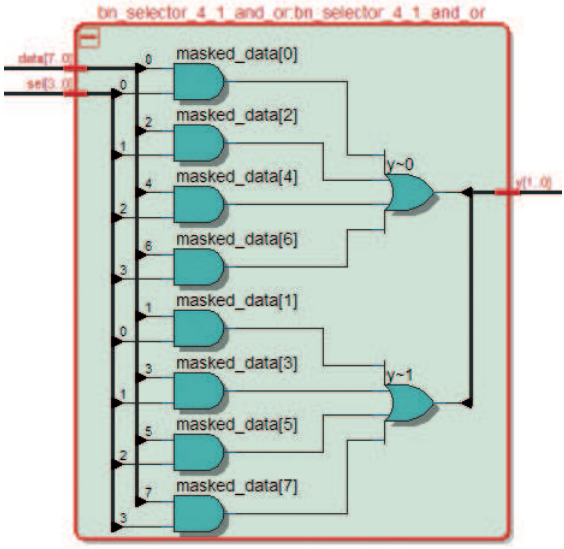


Рис. 4.16 И-ИЛИ селектор в виде RTL-схемы

4.5.2 Распределенный селектор

В некоторых задачах данные, поступающие на входные порты селектора, могут приходить с различной задержкой, в случае когда на различных путях распространения данных (**data flow**) находятся разные комбинационные блоки с различной задержкой (например, в **АЛУ**) (рис. 4.17). Другой пример – цепочка из комбинационных блоков, данные с которой снимаются с одного из ее блоков, в зависимости от управляющего сигнала (рис. 4.18). Обычно на выходе таких блоков стоит селектор, который коммутирует нужное значение на выход.

Задержки блоков могут значительно отличаться между собой. Предположим, все однотипные блоки проиндексированы в порядке возрастания их задержки **dN > di > d3 > d2 > d1**. Селектор в зависимости от разрядности данных и количества входов также будет иметь определенную задержку, обозначенную **s1**.

В этом случае рассматриваемые блоки будут иметь различную задержку сигналов в зависимости от того, по какому пути они проходят. Данное явление называется гонками (состязаниями) сигналов и устраняется введением синхронной логики путем защелкивания результирующего сигнала на триггере на выходе блока или с помощью логических методов путем добавления избыточной логики, как, например, соседнее кодирование.

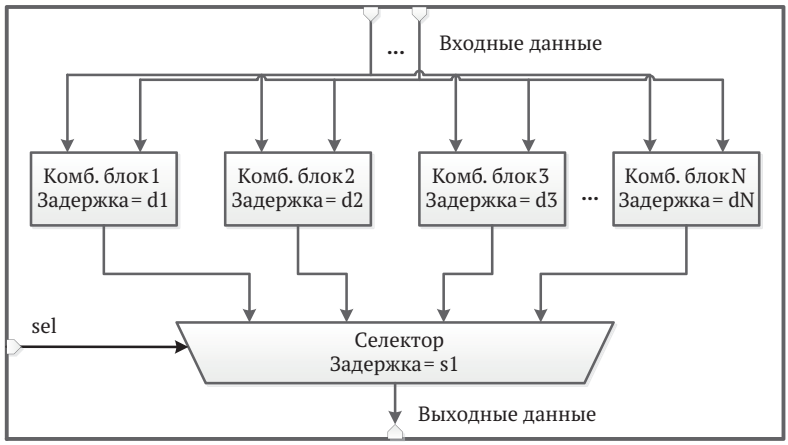


Рис. 4.17 Пример АЛУ

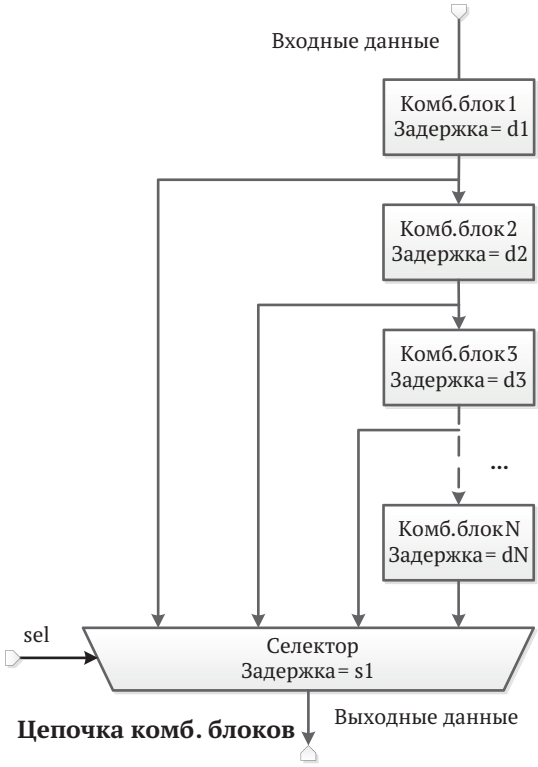


Рис. 4.18 Пример конвейера комбинационных блоков

Наибольшая задержка схемы **АЛУ** равна сумме задержки на селекторе и блоке с самой большой задержкой:

$$dALU_1 = s1 + dN,$$

а в случае с цепочкой комбинационных блоков наибольшая задержка будет равна сумме задержек всех блоков и селектора:

$$dPipline_1 = s1 + \sum_{i=1}^{N} di.$$

Существует прием по улучшению характеристик таких модулей и уменьшению их наибольшей задержки за счет разбиения селектора на селекторы меньших размеров путем размещения их параллельно комбинационным модулям, как приведено на рисунках ниже.

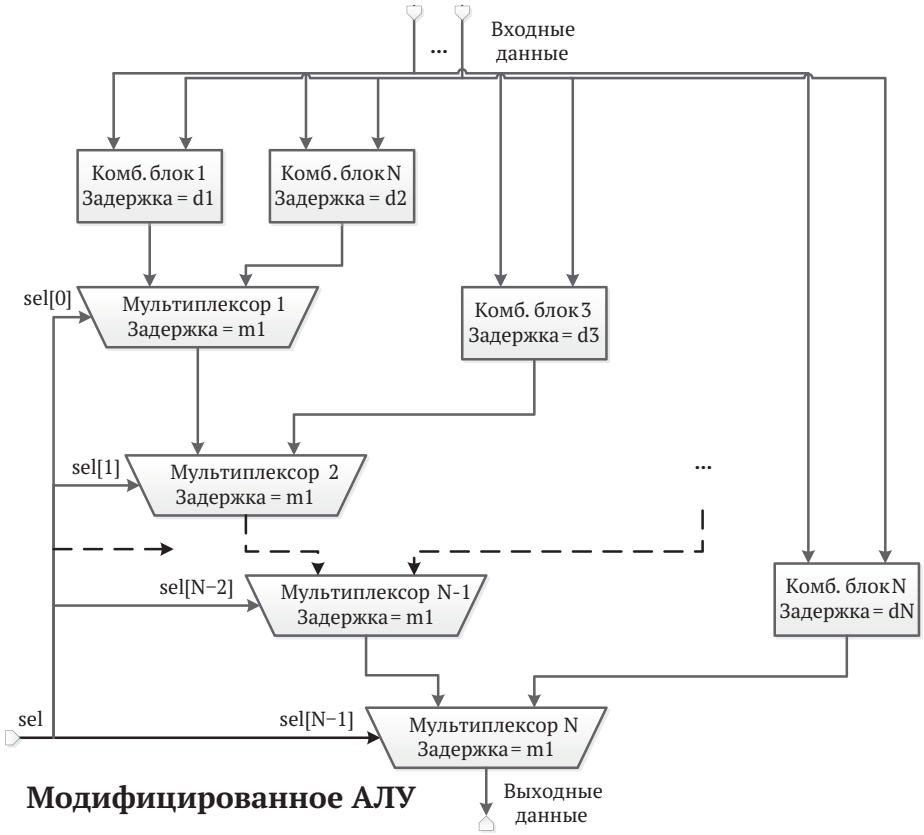


Рис. 4.19 Пример модифицированного АЛУ

Рис. 4.20 Пример улучшенной цепочки комбинационных блоков

Поскольку размеры этих селекторов меньше, чем у исходного селектора, обозначим их задержку как **s0**, и при этом она будет меньше (**s0 < s1**).

В таком случае если **s0 << d1**, то наибольшая задержка **АЛУ** будет:

$$dALU_0 = s0 + dN < dALU_1,$$

а в случае с цепочкой комбинационных блоков:

$$dPipline_0 = s0 + \sum_{i=1}^{N} di < dPipeline_1.$$

То есть часть задержки на большом селекторе распределяется на этапы выше параллельно с комбинационными блоками.

4.5.3 Пример использования распределенного мультиплексора «из жизни»

Мультиплексоры широко распространены в различных применениях. В том числе без них не обходятся процессоры. В настоящее время активно развивается направление по разработке суперскалярных процессоров, где несколько арифметических операций могут выполняться параллельно.

Рассмотрим пример: имеется **N функциональных узлов**, у которых входные данные кладутся в регистры (**операнды**). Сами устройства содержат свои конвейеры и много комбинационной логики на выходе (**результаты**). При обычном подходе **результаты** должны поступать в **регистровый файл**, откуда вместе с внешними данными на следующем такте снова поступать на входы **функциональных узлов**.

Существует подход, когда добавляется быстрый обходной путь (**hot bypass**), с помощью которого данные с выходов **функциональных узлов** сразу поступают к ним на входы. При этом такой путь, скорее всего, будет критическим за счет комбинационной логики на выходе **функциональных узлов**. Чтобы уменьшить данный путь, мультиплексор разделяют на **две** стадии: верхняя коммутирует данные с **регистрового файла** и **внешних входов** (где задержка небольшая и критический путь не возникает), а нижняя – данные с верхней стадии мультиплексора и данные, пришедшие по обходному пути.

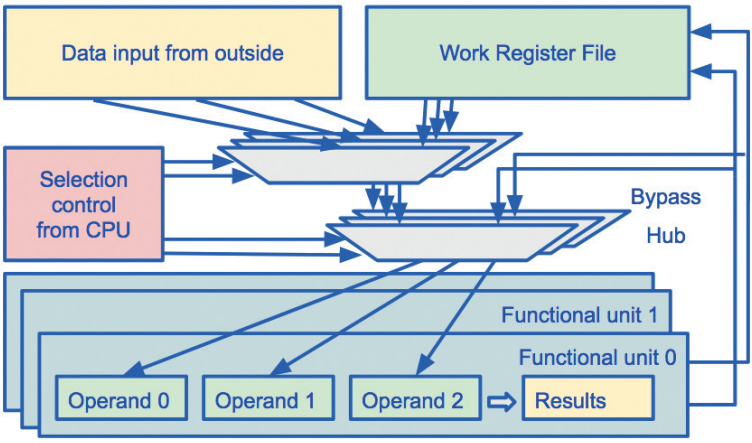


Рис. 4.21 Пример разделения мультиплексора для сокращения критического пути

Дополнительное задание для самостоятельной работы

Создайте полностью параметризированный **И-ИЛИ** селектор, используя оператор **generate**.

Реализуйте примеры **АЛУ** и конвейер на комбинационных блоках на рис. 4.17 и 4.18 на **Verilog**, исследуйте их **RTL**-схему и задержку прохождения сигналов.

Реализуйте примеры **АЛУ** и конвейер на комбинационных блоках на рис. 4.19 и 4.20 на **Verilog**, исследуйте их **RTL**-схему и задержку прохождения сигналов, сравните с предыдущим примером.

4.6 Упражнения

4.6.1 Основное задание

Выполните дополнительные задания для разделов 4.1, 4.2, 4.3, 4.4 и 4.5.

4.6.2 Задания для самостоятельной работы

Используя примеры кода из данной главы, реализуйте следующие мультиплексоры:

1. Мультиплексор 8в1, используя оператор **?:**.
2. Мультиплексор 8в1, используя оператор **if**.
3. Мультиплексор 8в1, используя оператор **case**.
4. Мультиплексор 8в1, создав два мультиплексора 4в1 и один мультиплексор 2в1. Реализовать модули мультиплексоров 4в1 и 2в1 при помощи оператора **?:**.
5. Мультиплексор 8в1, создав два мультиплексора 4в1 и один мультиплексор 2в1. Реализовать модули мультиплексоров 4в1 и 2в1 при помощи оператора **if**.
6. Мультиплексор 8в1, создав два мультиплексора 4в1 и один мультиплексор 2в1. Реализовать модули мультиплексоров 4в1 и 2в1 при помощи оператора **case**.
7. Мультиплексор 8в1, создав семь мультиплексоров 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора **?:**.
8. Мультиплексор 8в1, создав семь мультиплексоров 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора **if**.
9. Мультиплексор 8в1, создав семь мультиплексоров 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора **case**.
10. 3-битный мультиплексор 3в1, используя оператор **?:**.
11. 3-битный мультиплексор 3в1, используя оператор **if**.
12. 3-битный мультиплексор 3в1, используя оператор **case**.
13. 3-битный мультиплексор 3в1, используя два 3-битных мультиплексора 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора **?:**.
14. 3-битный мультиплексор 3в1, используя два 3-битных мультиплексора 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора мощью оператора **if**.
15. 3-битный мультиплексор 3в1, используя два 3-битных мультиплексора 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора **case**.
16. 3-битный мультиплексор 3в1, используя три 1-битных мультиплексора 3в1. Реализовать модуль мультиплексора 3в1 с помощью оператора **case**.
17. 3-битный мультиплексор 3в1, используя три 1-битных мультиплексора 3в1. Реализовать модуль мультиплексора 3в1 с помощью оператора **if**.
18. 3-битный мультиплексор 3в1, используя три 1-битных мультиплексора 3в1. Реализовать модули мультиплексора 3в1 с помощью оператора **?:**.

Разработайте таблицу истинности и изобразите условную схему подключения:

1. Мультиплексора 4в1, который реализует логический элемент **ИЛИ-НЕ**.
2. Мультиплексора 4в1, который реализует логический элемент **ИЛИ**.
3. Мультиплексора 4в1, который реализует логический элемент **И-НЕ**.
4. Мультиплексора 4в1, который реализует логический элемент **И**.
5. Мультиплексора 2в1, который реализует инвертор.

С помощью мультиплексора 8в1 опишите логическую функцию в виде условной схемы подключения:

1. $y = \overline{A} \wedge \overline{B} \wedge C \vee A \wedge \overline{B} \wedge \overline{C}$
2. $y = \overline{A} \wedge B \wedge C \vee A \wedge \overline{B} \wedge \overline{C}$
3. $y = A \wedge B \wedge C \vee A \wedge B \wedge \overline{C}$
4. $y = \overline{A} \wedge B \wedge C \vee A \wedge \overline{B} \wedge \overline{C} \vee \overline{A} \wedge B \wedge \overline{C}$
5. $y = (\overline{A} \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee \overline{C})$
6. $y = (A \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee \overline{B} \vee \overline{C})$
7. $y = \overline{A} \vee A \wedge \overline{B} \wedge \overline{C}$
8. $y = \overline{A} \oplus (B \wedge C \vee A \wedge \overline{B} \wedge \overline{C})$
9. $y = \overline{A} \oplus (B \wedge C)$
10. $y = A \oplus (B \vee C)$

4.6.3 Контрольные вопросы

1. Каким логическим устройством (комбинационным/последовательностным) является мультиплексор? Обоснуйте свой ответ.
2. Приведите три примера реализации мультиплексора 4в1 на **Verilog**. Какая реализация лучше?
3. Что из себя представляет тестбенч при разработке на **Verilog**? Для чего требуется разрабатывать тестбенчи?
4. В чем преимущества иерархического подхода при проектировании цифровых систем?
5. Каковы преимущества подхода к проектированию мультиплексоров с использованием оператора **case**?
6. Для чего используется ключевое слово default совместно с оператором case? Какие могут возникнуть проблемы при применении директивы `// synopsys full_case parallel_case`?
7. Как можно создавать логические схемы на мультиплексорах? Реализуйте логический элемент **2И** с использованием мультиплексора **2в1**.

8. Какую функцию выполняет демультиплексор? Приведите пример параметризируемого демультиплексора на **Verilog**.
9. Какова особенность селектора и в чем его отличие от мультиплексора?
10. Для чего используется конструкция **generate**? Приведите пример.
11. Для чего применяется оператор частичного выбора (**part select**)? Приведите примеры.
12. Что такое распределенный селектор, и в каких задачах он может применяться? Приведите примеры.

Михаил Шуплецов, Александр Романов

Цифровой синтез: практический курс

Глава 5. Сумматор, компаратор, устройство сдвига и АЛУ. Повышение скорости арифметических операций

Содержание

Глава содержит базовые сведения о различных способах реализации основных арифметических и логических блоков. Рассматривается построение арифметических блоков с синхронизированными входами и выходами и оценка задержки указанных блоков при помощи статического временного анализа. Кроме того, в данной главе приведено описание основных подходов к реализации комбинационных сумматоров, оптимизированных по быстродействию.

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime** (есть студенческая версия);*
- *пакет **ModelSim Altera Edition** или программы **Icarus Verilog** и **GTKWave**;*
- *отладочная плата компании **Terasic DE10-Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах[1] к данной книге).*

5.1 Арифметические комбинационные блоки: сумматор, компаратор, устройство сдвига и АЛУ

В данном разделе приводятся примеры основных арифметических и логических блоков и рассматриваются их высокоуровневые поведенческие описания. Более детально процесс проектирования указанных блоков рассмотрен в последующих разделах данной главы.

5.1.1 Сумматор

Сумматором (**Adder**) называют вычислительный блок, реализующий сложение двух **n**-разрядных двоичных чисел $x = (x_{n-1}, ..., x_0)$ и $y = (y_{n-1}, ..., y_0)$. Результат сложения часто представляют в виде **n**-разрядного двоичного числа $z = (z_{n-1}, ..., z_0)$ и дополнительного однобитного двоичного числа c_{out}, которое сигнализирует о возникновении бита переноса (переполнения), то есть ситуации, когда результирующая сумма является $(n + 1)$-битным числом. Стоит отметить, что часто сумматор имеет дополнительный вход c_{in}, который используется для учета возможного переноса, возникающего в других арифметических блоках. Наличие такого входа позволяет строить более сложные арифметические блоки из более простых.

Сумматор может быть как комбинационным, так и последовательностным устройством. В данной главе будут рассматриваться только комбинационные сумматоры, построенные из других арифметических и логических блоков.

[1] https://github.com/RomeoMe5/DDLM.

Рассмотрим одну из наиболее простых реализаций сумматора на языке **Verilog**:

```
module adder
#(
   parameter WIDTH = 8
)
(
   input [WIDTH - 1:0] x, y,
   input carry_in,
   output [WIDTH - 1:0] z,
   output carry_out
);
   assign {carry_out, z} = x + y + carry_in;
endmodule
```

Листинг 5.1 Поведенческое описание сумматора

В данном случае размерность входных аргументов **x** и **y** и выходной шины **z**, которая передает результат сложения, определяется параметром **WIDTH**. Провода **carry_in** и **carry_out** отвечают за возможный перенос на входе и, соответственно, на выходе сумматора. Для реализации непосредственно сложения используется оператор сложения, который поддерживается конструкцией **assign**. При этом шина **z** объединяется с проводом **carry_out** при помощи оператора конкатенации **{}**, а входные аргументы и результат сложения в данном случае трактуются как целые числа без знака.

Для тестирования данной реализации сумматора используется следующий тестбенч:

```
`timescale 1 ns / 100 ps
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
module testbench;
parameter WIDTH = 8;
// input and output test signals
reg c_in;
reg [WIDTH - 1:0] x, y;
wire [WIDTH - 1:0] z;
wire c_out;

// create the instances of the modules you want to test
adder #(.WIDTH(WIDTH)) adder_dut(
   .carry_in (c_in ),
   .x (x ),
   .y (y ),
   .z (z ),
   .carry_out(c_out)
);
```

```
//write test sequence
initial begin
   x = 0;
   y = 0;
   c_in = 0;
   #8
   $stop;
end

initial
   forever begin
      #1;
      x = $urandom_range(0, 2 ** WIDTH - 1);
      y = $urandom_range(0, 2 ** WIDTH - 1);

      c_in = $urandom_range(0, 1);
   end

//print signal values on every change
initial
   $monitor("Time:\t%g, x: %b, y: %b, c_in: %b, z: %b, c_out: %b",
      $time, x, y, c_in, z, c_out
   );

//configure signals dump for waveform's creation
initial begin
   $dumpfile("dump.vcd");
   $dumpvars(1); //iverilog dump init
end
endmodule
```

Листинг 5.2 Тестбенч для моделирования работы сумматора

Данный тестбенч позволяет провести моделирование работы экземпляра **8**-битного сумматора. При этом последовательность входных наборов генерируется случайно при помощи встроенной системной команды **$urandom_range**, которая возвращает псевдослучайное число из заданного диапазона. Результаты моделирования выдаются в стандартный поток вывода при помощи встроенной системной команды **$monitor** и сохраняются в специальном файле **dump.vcd** при помощи команд **$dumpfile** и **$dumpvars** для последующей визуализации в виде временных диаграмм. Стоит отметить, что использование случайных наборов наравне с применением специально подобранных тестовых наборов является удобной практикой для обнаружения типичных ошибок при создании описаний устройств.

Результирующие временные диаграммы работы **8**-битного сумматора представлены на рис. 5.1. Вид временной диаграммы может отличаться в силу случайности генерируемых входных наборов.

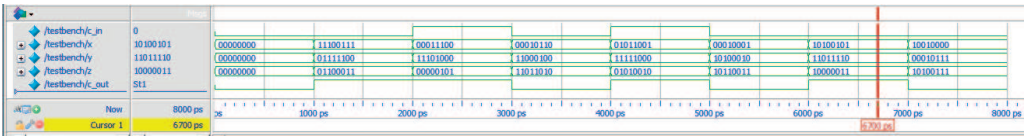


Рис. 5.1 Временная диаграмма моделирования работы 8-битного сумматора

Результаты моделирования, которые были выведены программой-симулятором в стандартный поток вывода, представлены на рис. 5.2.

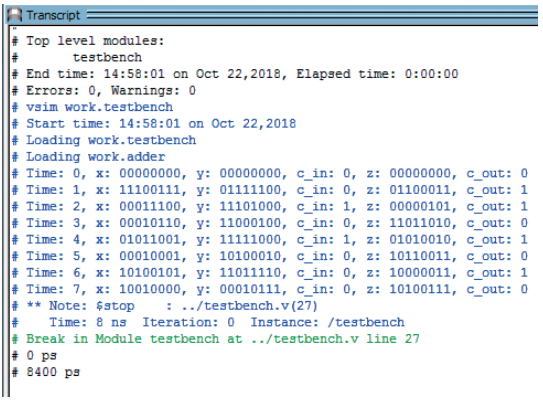


Рис. 5.2 Результаты моделирования работы 8-битного сумматора

В предложенном тестбенче было сгенерировано **8** тестовых наборов, так как задержка равна **#8**. Число случайных наборов можно легко изменить, меняя продолжительность работы первого **initial**-блока, изменяя задержку выполнения системной команды **$stop**. Также генерацию случайных наборов можно совместить с генерацией специальных наборов, которые будут покрывать какие-то особые режимы работы устройства.

Для последующего прототипирования сумматора при помощи **ПЛИС** достаточно определить схему соединения входов и выходов сумматора с доступной периферией на отладочной плате. Один из возможных вариантов описания соответствующего модуля верхнего уровня для отладочной платы **DE10-Lite** с **ПЛИС MAX10** представлен ниже:

```
module lab5
(
   input [9:0] SW,
   output [9:0] LEDR
);
   // WIDTH no greater than 4
   parameter WIDTH = 4;

   //adder
   adder #(.WIDTH(WIDTH)) i_adder
   (
      .carry_in (SW [0] ),
```

```
      .x (SW [WIDTH:1] ),
      .y (SW [2 * WIDTH:WIDTH + 1]),
      .z (LEDR[2 * WIDTH - 1:0] ),
      .carry_out(LEDR[9] )
   );

   assign LEDR[8:2 * WIDTH] = 0;
endmodule
```

Листинг 5.3 Описание модуля для тестирования сумматора на ПЛИС

В приведенном примере кода параметр **WIDTH** был введен для обеспечения удобства изменения размерности входных аргументов сумматора. На практике при разработке модуля верхнего уровня иерархии, который описывает схему соединения входов и выходов устройства с конкретной периферией отладочной платы, лучше указывать определенные значения диапазонов элементов шин проводов, привязанных к рассматриваемым элементам периферии.

При такой схеме подключения максимальная размерность входных аргументов равна **4**, так как при больших значениях размерности будет недостаточно светодиодов для отображения результатов работы модуля или переключателей для ввода входных значений.

Синтез приведенного поведенческого описания в **Quartus Prime** происходит в полностью автоматическом режиме. При этом на уровне **RTL**-описания каждый оператор сложения, используемый в конструкции **assign** для задания поведения сумматора, отображается в ячейку сумматора соответствующей размерности (рис. 5.3). Это не является оптимальной реализацией, так как на вход **CIN** сумматора **Add0** подается константа **0**, тогда как на этот вход можно было бы подать значение провода **carry_in** и обойтись одним сумматором.

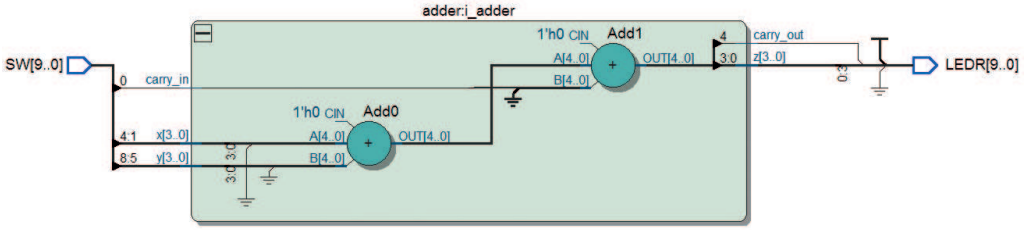


Рис. 5.3 RTL-описание 4-битного сумматора

Так как **RTL**-описание далее преобразуется в технологически зависимое описание, то в результате проводимых оптимизаций указанная неоптимальность представления устраняется. В зависимости от выбранных настроек и критериев оптимизации **Quartus Prime** ищет наиболее оптимальное технологически зависимое представление. При этом для реализации сумматоров разной размерности данные оптимальные представления могут отличаться. Например, в случае реализации однобитного сумматора наиболее оптимальная схема содержит два настраиваемых логических блока, соединенных параллельно (рис. 5.4).

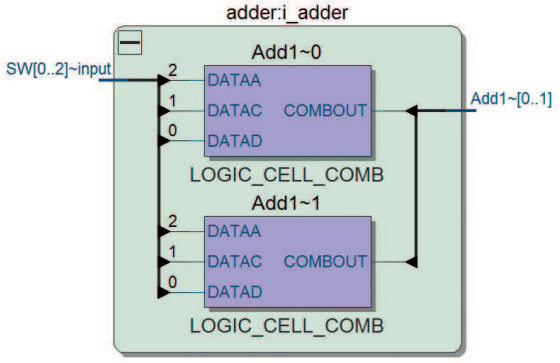


Рис. 5.4 Технологически зависимое представление однобитного сумматора, оптимизированное для ПЛИС семейства MAX10

При реализации двухбитного сумматора **Quartus Prime** выбирает совершенно другую технологически зависимую реализацию (рис. 5.5). В данном случае два настраиваемых логических блока **Add0~4** и **Add0~2** используются в арифметическом режиме, при этом для передачи сигнала переноса применяются специальные порты **CIN** и **COUT**, которые позволяют формировать быстрые цепи переноса в рамках архитектуры **ПЛИС MAX10**. Кроме того, в реализации присутствуют два блока **Add0~1** и **Add0~6**, которые играют вспомогательную роль и осуществляют связь входных и выходных сигналов проектируемого сумматора с указанными входами и выходами. Это необходимо в силу особенностей архитектуры рассматриваемой **ПЛИС**.

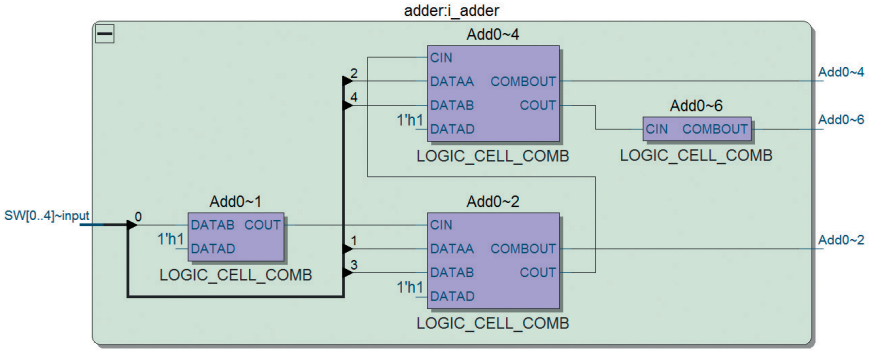


Рис. 5.5 Технологически зависимое представление двухбитного сумматора, оптимизированное для ПЛИС семейства MAX10

Более подробно исследовать особенности структурной реализации рассматриваемого сумматора можно при помощи окон **Chip Planner** и **Property Editor** программы **Quartus Prime**. В соответствии с рис. 5.6 все ячейки сумматора были размещены в рамках одного логического массива (**Logic Array Block, LAB**), при этом все ячейки в массиве расположены последовательно. Справа на рис. 5.6 изображен блок **Add0~1**, который передает сигнал **carry_in** проектируемого сумматора на выход **COUT**, при этом выход **COMBOUT** выдает константу **0**.

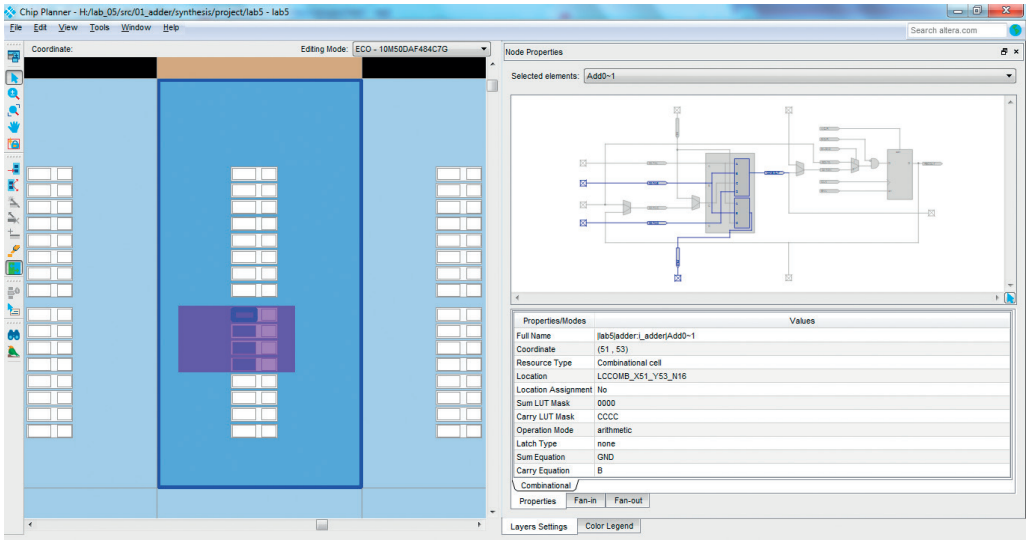


Рис. 5.6 Двухбитный сумматор и устройство ячейки Add0~1 в Chip Planner Quartus Prime

Конфигурация других блоков сумматора находится в окне **Property Editor**. Отметим, что ячейки **Add0~2** и **Add0~4** составляют ячейку полного однобитного сумматора с использованием специальных портов **CIN** и **COUT** для осуществления переноса (рис. 5.7).

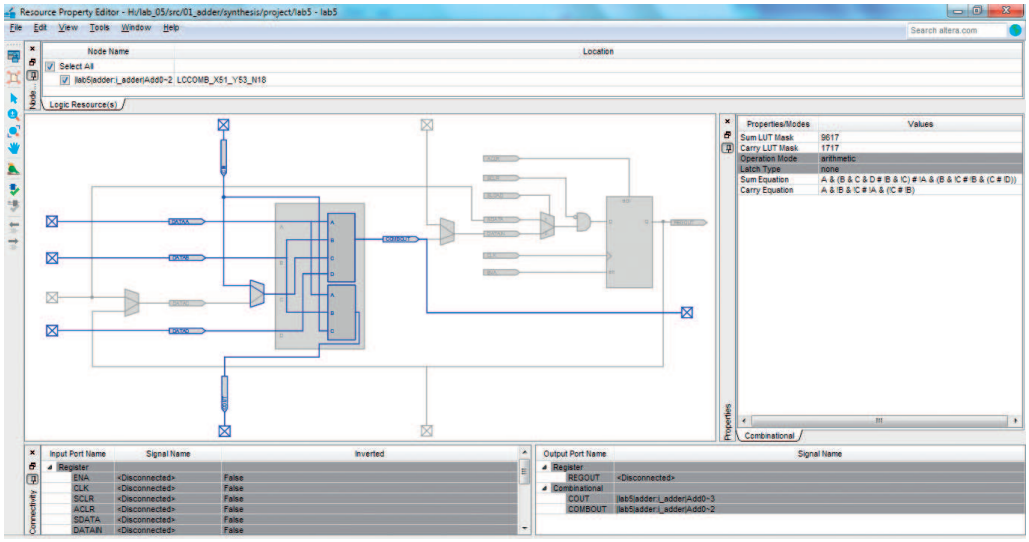


Рис. 5.7 Устройство ячейки Add0~2 двухбитного сумматора в Resource Property Editor Quartus Prime

Наконец, блок **Add0~6** осуществляет передачу значения переноса со специального входа **CIN** на стандартный выход **COMBOUT** (рис. 5.8).

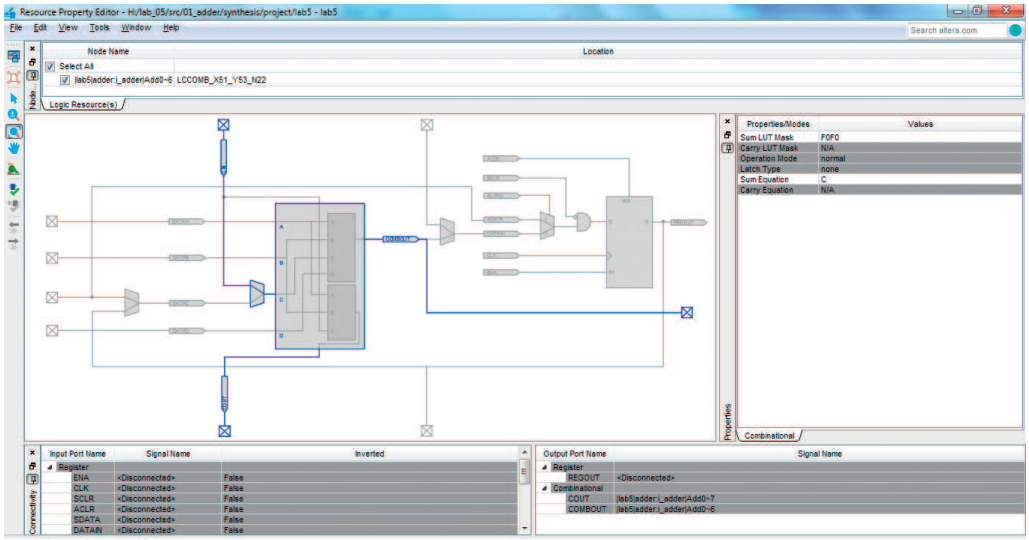


Рис. 5.8 Устройство ячейки Add0~6 двухбитного сумматора в Resource Property Editor Quartus Prime

5.1.2 Сумматор с синхронизированными входами и выходами

Наравне со сложностью, которая в основном зависит от размера схемы, одним из ключевых параметров комбинационных схем является их быстродействие (задержка срабатывания). Так как в дальнейшем в данной главе будут рассмотрены различные реализации арифметических блоков, оптимизированных по быстродействию, то в данном разделе будет рассмотрен простой пример анализа быстродействия комбинационной схемы при помощи статического временного анализа.

В качестве анализируемой комбинационной схемы возьмем сумматор (листинг 5.1). Для того чтобы провести статический временной анализ комбинационной схемы при помощи программы **TimeQuest**, интегрированной в **Quartus Prime**, требуется выполнить ряд вспомогательных действий.

Во-первых, комбинационную схему нужно превратить в последовательностную, добавив синхронизацию всех входных и выходных сигналов. Для этих целей можно использовать следующий модуль:

```
module unit_delay
# (
   parameter WIDTH = 1
)
(
   input clock,
   input [ WIDTH - 1:0 ] data_in,
   output reg [ WIDTH - 1:0 ] data_out
);
```

```
   always @ (posedge clock)
      data_out <= data_in;
endmodule
```

Листинг 5.4 Описание модуля единичной задержки

Указанный модуль реализует простейший элемент единичной задержки, который откладывает изменение значения на своем выходе до момента прихода положительного фронта синхросигнала **clock**. Для синхронизации входных сигналов вместо модуля единичной задержки зачастую удобно использовать регистр (листинг 5.5), позволяющий сохранять значение входного сигнала, обновлять его по управляющему сигналу **load** и сбрасывать по управляющему сигналу **reset** (в представленном примере регистр имеет асинхронный сигнал сброса):

```
module register
# (parameter WIDTH = 8)
(
   input clock,
   input reset,
   input load,
   input [ WIDTH - 1:0 ] data_in,
   output reg [ WIDTH - 1:0 ] data_out
);

   always @ ( posedge clock, negedge reset )
      if ( ~reset )
         data_out <= { WIDTH { 1'b0 } };
      else if ( load )
         data_out <= data_in;
endmodule
```

Листинг 5.5 Описание регистра с асинхронным сбросом

Следующий код представляет модуль сумматора, все входы и выходы которого синхронизированы при помощи элементов единичной задержки и регистров:

```
module lab5
(
   input MAX10_CLK1_50,
   input [1:0] KEY,
   input [9:0] SW,
   output [9:0] LEDR
);
   parameter WIDTH = 8;

   //wires
   wire [WIDTH - 1:0] x_bus, y_bus, z_bus;
   wire load_x, load_y;
   wire carry_in_wire;
   wire carry_out_wire;
```

```
wire clock;
wire reset;

assign clock = MAX10_CLK1_50;
assign reset = KEY[1];

//register for the arguments
register #(.WIDTH(WIDTH)) x_register
(
   .clock (clock ),
   .reset (reset ),
   .load (load_x ),
   .data_in (SW[WIDTH:1]),
   .data_out(x_bus )
);

register #(.WIDTH(WIDTH)) y_register
(
   .clock (clock ),
   .reset (reset ),
   .load (load_y ),
   .data_in (SW[WIDTH:1]),
   .data_out(y_bus )
);

//carry_in synchronization
unit_delay #(.WIDTH(1)) carry_in_register (
   .clock ( clock ),
   .data_in ( SW[ 0 ] ),
   .data_out( carry_in_wire )
);

//argument selector
assign load_x = SW[0] & ~KEY[0] ? 1'b1 : 1'b0;
assign load_y = ~SW[0] & ~KEY[0] ? 1'b1 : 1'b0;

//adder
adder #(.WIDTH(WIDTH)) i_adder
(
   .carry_in (carry_in_wire ),
   .x (x_bus ),
   .y (y_bus ),
   .z (z_bus ),
   .carry_out(carry_out_wire)
);

//output synchronization
unit_delay #(.WIDTH(WIDTH)) z_register (
   .clock (clock ),
```

```
      .data_in (z_bus ),
      .data_out(LEDR[WIDTH - 1:0])
   );

   unit_delay #(.WIDTH(1)) carry_out_register (
      .clock (clock ),
      .data_in (carry_out_wire),
      .data_out(LEDR[9] )
   );

   assign LEDR[8:WIDTH] = 0;
endmodule
```

Рис. 5.6 Описание сумматора с синхронизированными входами и выходами

Также в приведенном примере реализована дополнительная логика для обеспечения возможности загрузки значений входных аргументов.

Во-вторых, требуется указать файл, который будет содержать основные характеристики синхронизирующего сигнала и другие параметры, необходимые для проведения статического временного анализа. Наиболее удобным форматом для конфигурации данных параметров является формат проектных ограничений компании **Synopsys** (**Synopsys Design Constraints, SDC**).

Содержание подобного конфигурационного файла представлено в следующем листинге:

```
create_clock -period "50.0 MHz" [get_ports MAX10_CLK1_50]
create_clock -period "50.0 MHz" [get_ports MAX10_CLK2_50]

derive_clock_uncertainty

set_false_path -from * -to [get_ports {LEDR[*]}]
set_false_path -from [get_ports {KEY[*]}] -to [all_clocks]
set_false_path -from [get_ports {SW[*]}] -to [all_clocks]
```

Листинг 5.7 Конфигурационный файл для проведения статического временного анализа

Рассматриваемый файл содержит три основных элемента. Во-первых, при помощи команд **create_clock** специфицируется, что порты **MAX10_CLK1_50** и **MAX10_CLK2_50** являются портами, генерирующими синхронизирующий сигнал с тактовой частотой **50 MHz**. Во-вторых, команда **derive_clock_uncertainty** генерирует все неопределенности, связанные с заданными синхросигналами. В-третьих, команды **set_false_path** задают все ложные критические пути, то есть те пути, которые можно исключить из статического временного анализа. В данном случае это пути, связанные с периферийными устройствами, так как они могут повлиять на оценку времени срабатывания исследуемого блока.

По умолчанию после синтеза описания устройства в **Quartus Prime** выполняется статический временной анализ при помощи программ **TimeQuest** (рис. 5.9).

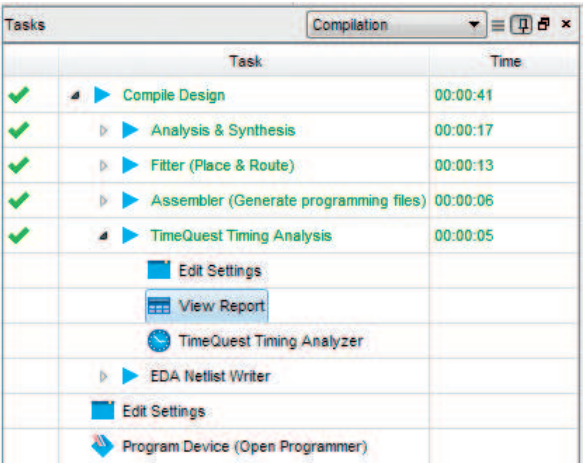


Рис. 5.9 TimeQuest в маршруте синтеза программы Quartus Prime

В результате статического временного анализа программа **TimeQuest** генерирует развернутый отчет, который содержит подробное описание временны́х характеристик синтезированной схемы. Одной из ключевых характеристик, получаемых в результате временно́го анализа, является оценка максимальной тактовой частоты для выбранной модели задержек элементов схемы. Например, на рис. 5.10 представлена оценка максимальной тактовой частоты, рассчитанная для модельных задержек элементов схемы, полученных для выбранного режима работы устройства (напряжение – **1200 mV**, температура – **85 °C**).

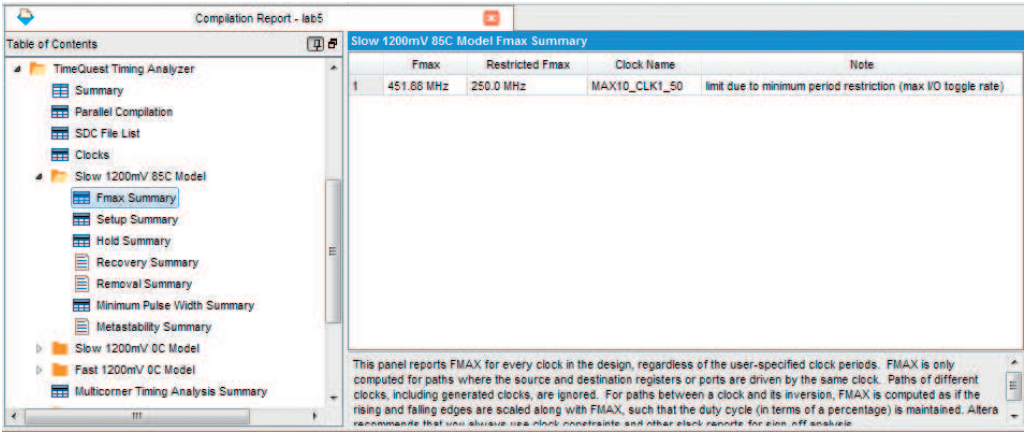


Рис. 5.10 Оценка максимальной тактовой частоты для выбранной модели задержек элементов схемы

В дополнение к отчетам, которые доступны непосредственно из программы **Quartus Prime**, программа **TimeQuest** предоставляет ряд инструментов для более детального анализа задержек. Например, доступна возможность анализировать структуру и основные параметры критических путей, на которых достигается максимальная задержка схемы (рис. 5.11).

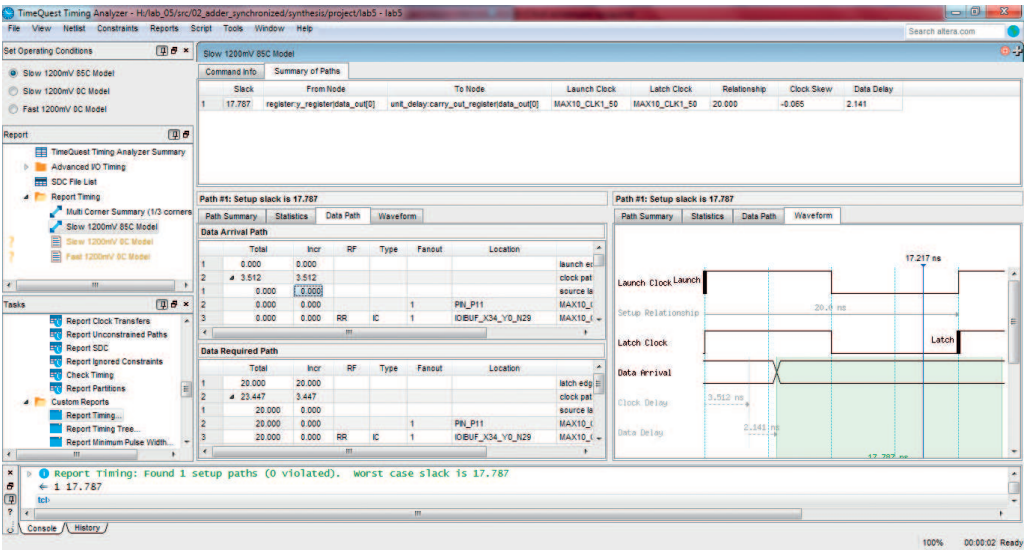


Рис. 5.11 Основной интерфейс программы TimeQuest

Интеграция программы **TimeQuest** с программой **Quartus Prime** позволяет визуализировать критические пути после решения задачи привязки логической схемы в **Technology Map Viewer** (рис. 5.12) и на уровне топологии в **Chip Planner** (рис. 5.13).

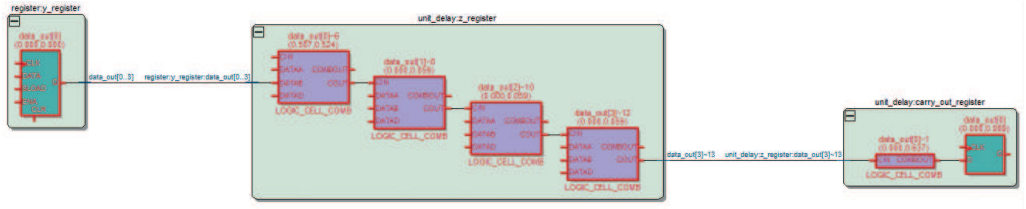


Рис. 5.12 Структура критического пути в Technology Map Viewer

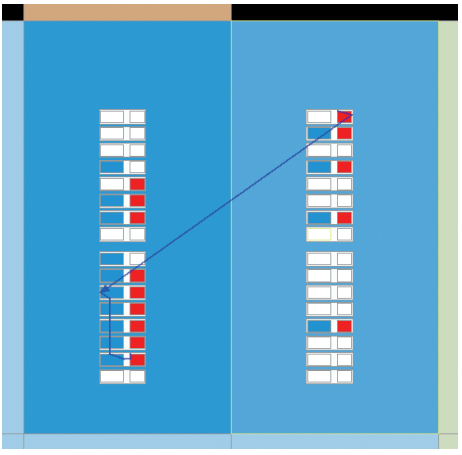


Рис. 5.13 Структура критического пути в Chip Planner

Дополнительное задание для самостоятельной работы

Используя статический временной анализ, получите оценку максимальной тактовой частоты для сумматоров различной разрядности (**8**, **16**, **32**, **64**, **128**, **256** бит). На основе указанных оценок опишите зависимость роста данного параметра от разрядности сумматора в виде функции.

5.1.3 Компаратор

Компараторами (**comparator**) называют различные логические устройства, предназначенные для сравнения чисел между собой. Чаще всего входами компаратора являются две **n**-разрядные шины, а выходом – сигнальный однобитный выход, который содержит единицу, если входные аргументы удовлетворяют выбранному соотношению сравнения, и ноль в ином случае. При этом входные аргументы трактуются как целые числа со знаком или без знака, а в качестве отношения сравнения выступает сравнение на равенство, строгое неравенство или нестрогое неравенство аргументов.

Пример поведенческого описания компаратора на языке **Verilog**, реализующего различные типы отношения сравнения, представлен в листинге ниже:

```
module comparator
#(
   parameter WIDTH = 8
)
(
   input [WIDTH - 1:0] x, y,
   output eq, neq, lt, lte, gt, gte
);
   assign eq = (x == y);
   assign neq = (x != y);
   assign lt = (x < y);
   assign lte = (x <= y);
   assign gt = (x > y);
   assign gte = (x >= y);
endmodule
```

Листинг 5.8 Поведенческое описание компаратора

Результаты моделирования рассматриваемых описаний 8-битных компараторов представлены ниже:

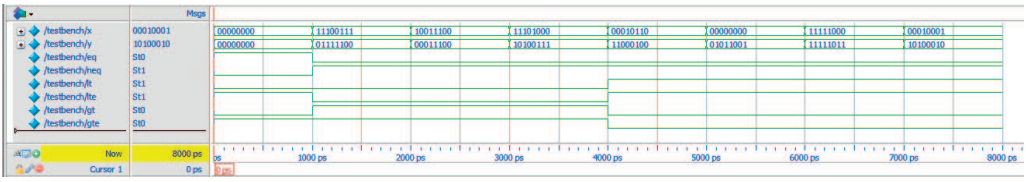


Рис. 5.14 Результаты моделирования компараторов

На рис. 5.15 дана структура **RTL**-схемы компаратора, сгенерированной в **Quartus Prime** на основе описания, представленного в листинге 5.8. Все указанные отношения сравнения в **Quartus Prime** синтезируются в виде оптимизированных реализаций на технологически зависимом уровне.

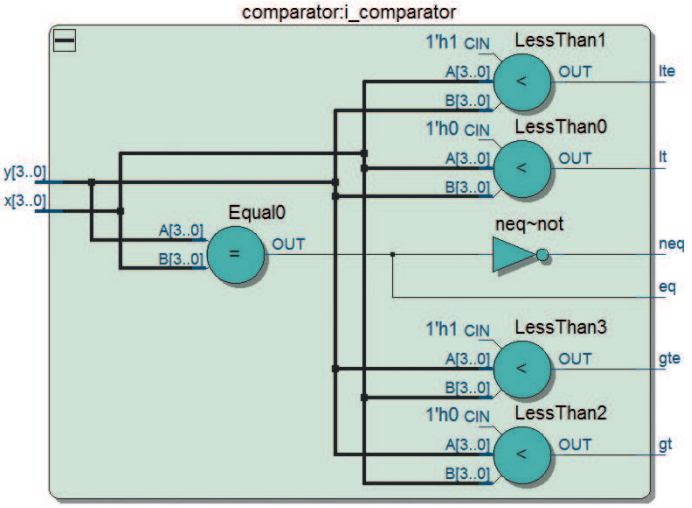


Рис. 5.15 Структура RTL-схемы компаратора

Компаратор может быть также спроектирован для сравнения вещественных чисел, но схема такого компаратора будет существенно сложнее. Еще в качестве одного из аргументов компаратора может быть использована константа для реализации различных логических блоков проверки специальных условий.

5.1.4 Устройство сдвига

Устройствами сдвига (**shifter**) и **устройствами циклического сдвига** (**barrel shifters, rotators**) являются логические блоки, представляющие собой сдвиг в адресации элементов некоторой шины данных на заданное число позиций. Дадим формальное описание: пусть $x = (x_{n-1}, ..., x_0)$ – некоторая шина данных и s – количество позиций, на которые требуется осуществить сдвиг, тогда выходом устройства сдвига является шина данных $z = (z_{n-1}, ..., z_0)$, где

$$z_i = \begin{cases} x_{i+s}, \text{ если } i < n - s \\ 0 \text{ в ином случае} \end{cases}$$

в случае, если сдвиг осуществляется вправо. Если сдвиг осуществляется влево, то шина данных определяется следующим образом:

$$z_i = \begin{cases} x_{i-s}, \text{ если } i < s - 1 \\ 0 \text{ в ином случае} \end{cases}.$$

Операции сдвига вправо и влево могут быть интерпретированы как арифметические операции целочисленного деления и умножения на 2^s соответственно.

Язык **Verilog** поддерживает операторы сдвига, поэтому рассматриваемые блоки могут быть легко описаны при помощи конструкции **assign**:

```
module left_shifter
#(
   parameter WIDTH = 8,
   //SHIFT specifies the number of bits for shamt argument
   parameter SHIFT = 3
)
(
   input [WIDTH - 1:0] x,
   input [SHIFT - 1:0] shamt,
   output [WIDTH - 1:0] z
);
   assign z = x << shamt;
endmodule
```

Листинг 5.9 Поведенческое описание блока сдвига влево

Блок сдвига вправо описывается аналогичным образом, путем замены оператора **<<** на оператор **>>**.

Результаты моделирования 8-битных блоков сдвига влево и вправо с трехбитным аргументом **shamt** представлены ниже:

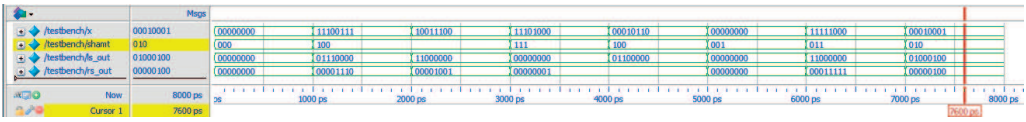


Рис. 5.16 Результаты моделирования блоков сдвига влево и вправо

Структура **RTL**-схемы, реализующей четырехбитные устройства сдвига влево и вправо, представлена на рис. 5.17.

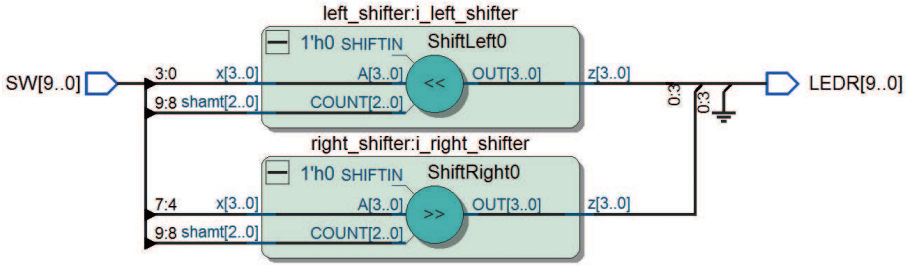


Рис. 5.17 RTL-схема четырехбитных устройств сдвига влево и вправо

В свою очередь, устройства циклического сдвига интерпретируют индексы шины данных как замкнутые в кольцо, то есть следующим индексом после $(n - 1)$ является индекс **0**. Таким образом, сдвиг индексов происходит по модулю длины шины данных. Формально результирующую шину данных $z = (z_{n-1}, ..., z_0)$ устройства циклического сдвига на s позиций можно задать следующим образом:

$$z_i = x_{(i+s) \bmod n},$$

в том случае когда сдвиг осуществляется вправо. Если сдвиг осуществляется влево, шина данных z определяется следующим образом:

$$z_i = x_{(i-s) \bmod n}.$$

В языке **Verilog** не предусмотрен специальный оператор для осуществления циклического сдвига. Тем не менее блок циклического сдвига легко можно описать при помощи обычного оператора сдвига, дублируя входную шину данных x и отбирая нужные биты после сдвига, как это сделано в примерах в листингах 5.10 и 5.11.

```
module left_rotator
#(
   parameter WIDTH = 8,
   //SHIFT specifies the number of bits for shamt argument
   parameter SHIFT = 3
)
(
   input [WIDTH - 1:0] x,
   input [SHIFT - 1:0] shamt,
   output [WIDTH - 1:0] z
);

   wire [2 * WIDTH - 1:0] temp;
   assign temp = {x, x} << shamt;
   assign z = temp[2 * WIDTH - 1: WIDTH];
endmodule
```

Листинг 5.10 Устройство циклического сдвига влево

```
module right_rotator
#(
   parameter WIDTH = 8,
   //SHIFT specifies the number of bits for shamt argument
   parameter SHIFT = 3
)
(
   input [WIDTH - 1:0] x,
   input [SHIFT - 1:0] shamt,
   output [WIDTH - 1:0] z
);

   wire [2 * WIDTH - 1:0] temp;
   assign temp = {x, x} >> shamt;
   assign z = temp[WIDTH - 1: 0];
endmodule
```

Листинг 5.11 Устройство циклического сдвига вправо

Результаты моделирования устройств циклического сдвига представлены ниже:

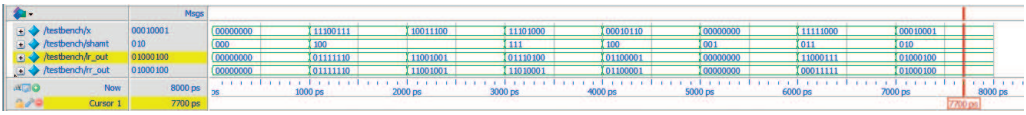


Рис. 5.18 Результаты моделирования устройств циклического сдвига влево и вправо

На рис. 5.19 представлено **RTL**-описание четырехбитных устройств циклического сдвига, синтезированных на основе **Verilog**-описаний, представленных в листингах 5.10 и 5.11. Следует отметить, что они очень похожи на описание обычных устройств сдвига: только произошло удвоение разрядности элемента сдвига – и на его выходе происходит отбор нужных проводов для формирования выходной шины.

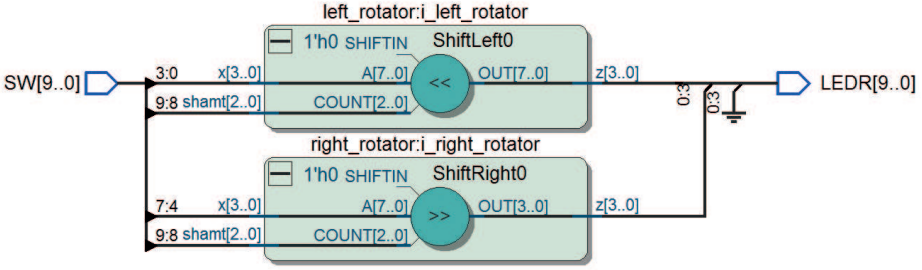


Рис. 5.19 RTL-описание 4-битных устройств циклического сдвига

5.1.5 Арифметико-логическое устройство

Арифметико-логическое устройство (**АЛУ**) представляет собой блок, позволяющий выполнять ряд арифметических и логических операций над заданным количеством аргументов. Конкретный набор операций, количество операндов и разрядность у поддерживаемых операций могут очень сильно варьироваться в зависимости от типа устройства, для которого разрабатывается **АЛУ**. Кроме того, очень часто **АЛУ** имеет ряд дополнительных выходов, которые передают специальные сигналы-индикаторы (флаги), которые обращаются в единицу при наступлении определенного события (например, при переполнении результата или обращении результата в ноль).

Следующий листинг представляет описание упрощенного **АЛУ**, поддерживающего всего **четыре** операции:

```
//ALU commands
`define ALU_AND 2'b00
`define ALU_ADD 2'b01
`define ALU_SLL 2'b10
`define ALU_SLT 2'b11

module alu
#(
   parameter WIDTH = 4,
```

```
    parameter SHIFT = 2
)
(
    input [WIDTH - 1:0] x, y,
    input [SHIFT - 1:0] shamt,
    input [ 1:0] operation,
    output zero,
    output reg [WIDTH - 1:0] result
);
    always @ (*) begin
        case (operation)
            `ALU_AND : result = x & y;
            `ALU_ADD : result = x + y;
            `ALU_SLL : result = y << shamt;
            `ALU_SLT : result = (x < y) ? 1 : 0;
        endcase
    end

    //Flags
    assign zero = (result == 0);
endmodule
```

Листинг 5.12 Verilog-описание простого арифметико-логического устройства

Рассматриваемое **АЛУ** поддерживает только **четыре** операции: конъюнкцию, сложение, сдвиг влево и проверку операндов на неравенство. Каждой операции назначен свой двоичный код, и для описания вместо самих кодов используются специальные константы, которые определены при помощи директивы **define**. Логика самого **АЛУ** реализована в блоке **always**, содержащем оператор case, который на основе кода операции присваивает нужное значение шине **result**. Кроме того, при помощи оператора **assign** реализуется флаг проверки результата операции на равенство нулю.

Результат моделирования данного **АЛУ** представлен ниже:

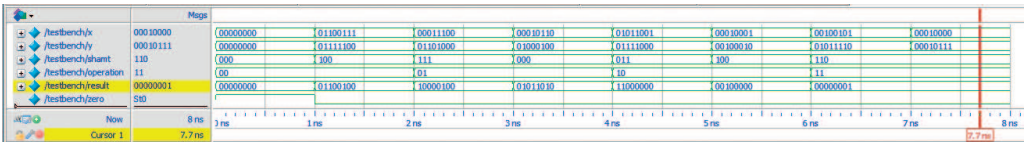


Рис. 5.20 Моделирование арифметико-логического устройства

На рис. 5.21 приведена **RTL**-схема **АЛУ**, код которого описан в листинге 5.12. Для реализации каждой поддерживаемой операции спроектирован свой отдельный блок, при этом все блоки подсоединены параллельно к входным аргументам. Такая схема позволяет всем блокам работать параллельно и независимо. Для выбора заданной операции, которую **АЛУ** должен реализовать в текущий момент времени, используется соответствующий мультиплексор.

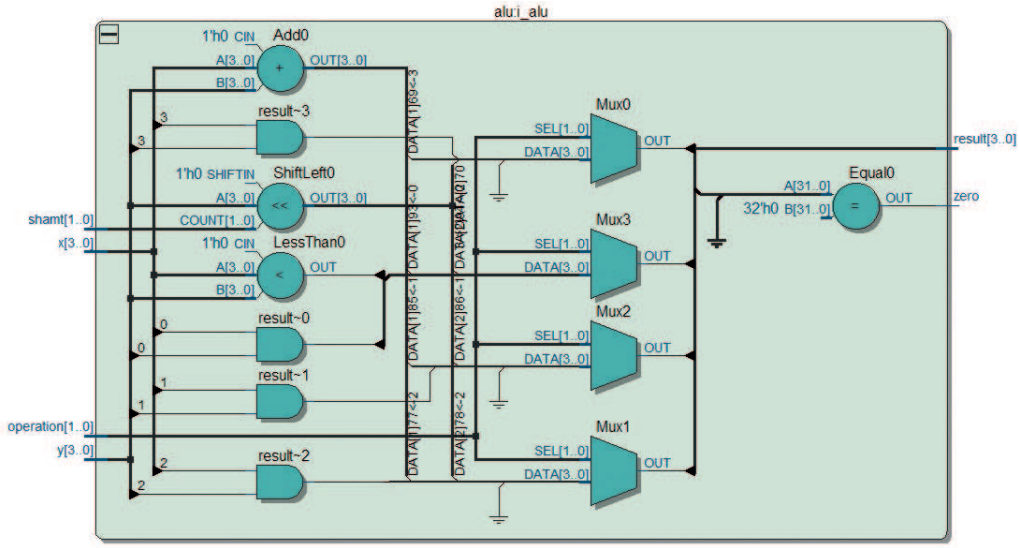


Рис. 5.21 RTL-схема простого арифметико-логического устройства

Структурная реализация **АЛУ** требует описания модулей для всех поддерживаемых операций и для мультиплексора, осуществляющего выбор результата того блока, который выполняет выбранную операцию. В случае рассмотренного ранее **АЛУ** достаточно использовать описания следующих блоков: побитового конъюнктора (листинг 5.13), сумматора (листинг 5.1), устройства сдвига (листинг 5.9) и компаратора (листинг 5.14). Описание мультиплексора возьмем из главы 4 (листинг 4.23).

```
module bitwise_and
#(
   parameter WIDTH = 8
)
(
   input [WIDTH - 1:0] x,y,
   output [WIDTH - 1:0] z
);
   assign z = x & y;
endmodule
```

Листинг 5.13 Параметрический побитовый конъюнктор

```
module slt
#(
   parameter WIDTH = 8
)
(
   input [WIDTH - 1:0] x,y,
   output [WIDTH - 1:0] z
);
```

```
   assign z = (x < y) ? 1 : 0;
endmodule
```

Листинг 5.14 Параметрический компаратор

Объединив отдельные блоки в единое структурное описание, получим следующий результат:

```
module alu_structural
#(
   parameter WIDTH = 4,
   parameter SHIFT = 2
)
(
   input [WIDTH - 1:0] x, y,
   input [SHIFT - 1:0] shamt,
   input [ 1:0] operation,
   input carry_in,
   output zero,
   output overflow,
   output [WIDTH - 1:0] result
);

   wire [4 * WIDTH - 1: 0] t; // This ALU supports 4 operations

   //Bitwise AND
   bitwise_and #( .WIDTH( WIDTH ) ) i_and
   (
      .x( x ),
      .y( y ),
      .z( t[ WIDTH - 1 : 0 ] )
   );

   //Adder
   adder #( .WIDTH( WIDTH ) ) i_adder
   (
      .x ( x ),
      .y ( y ),
      .carry_in ( carry_in ),
      .z ( t[ 2 * WIDTH - 1 : WIDTH ] ),
      .carry_out( overflow )
   );

   //Shifter
   left_shifter
   #(
      .WIDTH( WIDTH ),
      .SHIFT( SHIFT )
   )
```

```
   i_shifter
   (
      .x ( x ),
      .shamt( shamt ),
      .z ( t[ 3 * WIDTH - 1 : 2 * WIDTH ] )
   );

   //SLT
   slt #( .WIDTH( WIDTH ) ) i_slt
   (
      .x( x ),
      .y( y ),
      .z( t[ 4 * WIDTH - 1 : 3 * WIDTH ] )
   );

   //Multiplexer
   bn_mux_n_1_generate
   #(
      .DATA_WIDTH(WIDTH),
      .SEL_WIDTH(2)
   )
   i_mux
   (
      .data( t ),
      .sel ( operation ),
      .y ( result )
   );

   //Flags
   assign zero = (result == 0);
endmodule
```

Листинг 5.15 Структурное описание арифметико-логического устройства

Соответствующая **RTL**-схема представлена на следующем рисунке:

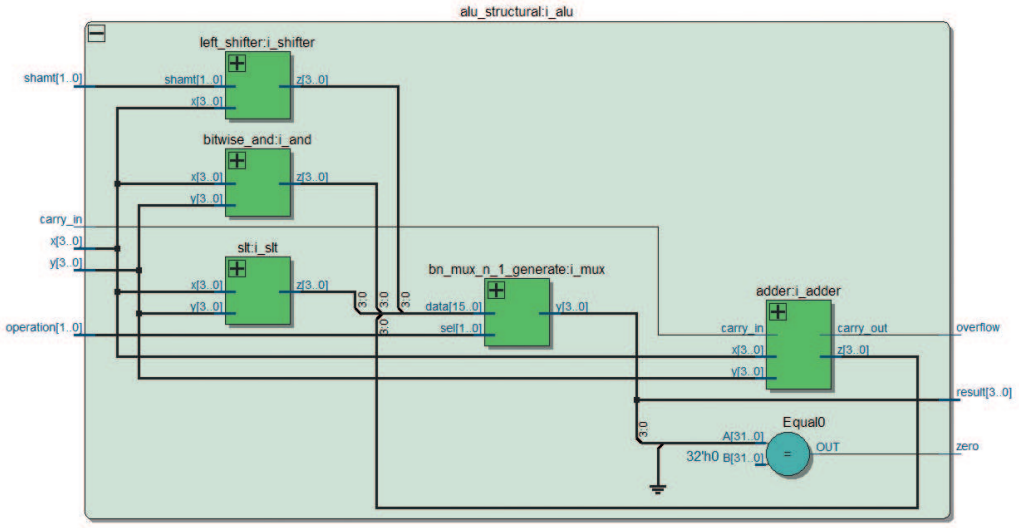


Рис. 5.22 RTL-схема структурной реализации АЛУ

Из представленной схемы следует, что все блоки, выполняющие различные операции, поддерживаемые рассматриваемым **АЛУ**, работают одновременно и генерируют свои результаты независимо. При этом мультиплексор из набора полученных результатов выбирает один, который соответствует текущей операции. Стоит также отметить, что в данном случае **АЛУ** представляет комбинационное устройство, и его сложность равна сумме сложностей блоков, реализующих поддерживаемые операции, и мультиплексора. В свою очередь, задержка такого **АЛУ** равна сумме задержки мультиплексора и максимальной задержки среди блоков, которые реализуют поддерживаемые операции. В силу модульности представленного описания любой блок может быть заменен на аналогичный по функциональности блок, но имеющий более оптимальные параметры задержки или сложности. Например, блок сумматора (листинг 5.1) может быть заменен на один из более оптимальных блоков, которые будут рассмотрены далее в этой главе.

Дополнительное задание для самостоятельной работы

Модифицируйте **АЛУ**, приведенное в листинге 5.12, таким образом, чтобы оно поддерживало четыре новые операции и имело ряд дополнительных выходов, которые реализуют флаги переполнения для арифметических операций. Создайте соответствующую структурную реализацию модифицированного **АЛУ**.

5.2 Полусумматор, полный сумматор, перенос, каскадное соединение сумматоров

Ранее в разделе 5.1.1 было рассмотрено наиболее простое поведенческое описание ячейки многобитного сумматора, которое можно использовать для синтеза сумматоров в автоматическом режиме. В данном разделе будет рассмотрена

структурная реализация многобитного сумматора на основе последовательного соединения сумматоров. Так как в этом разделе рассматриваются различные реализации сумматора, то для краткости изложения результаты моделирования в этом и последующих разделах будут опущены. При желании моделирование можно провести самостоятельно, воспользовавшись исходными файлами, тестбенчами и скриптами для моделирования в дополнительных материалах[1] к данной главе.

5.2.1 Полусумматор

Полусумматор представляет собой арифметический блок, осуществляющий сложение двух однобитных двоичных чисел **x** и **y**. В результате сложения получается двухбитное число, при этом младший бит (**z**) считается основным результатом сложения, а старший бит данного числа считается битом переноса (**carry_out**) или переполнения.

Ниже представлено наиболее простое описание ячейки полусумматора при помощи конструкции **assign**:

```
module half_adder
(
   input wire x, y,
   output wire z,
   output wire carry_out
);
   assign {carry_out, z} = x + y;
endmodule
```

Листинг 5.16 Поведенческая реализация ячейки полусумматора

Ячейку полусумматора можно довольно просто описать при помощи базовых логических вентилей. В этом случае результат сложения можно описать следующей системой булевых уравнений:

$$\begin{cases} z = x \oplus y \\ carry_{out} = x \cdot y \end{cases}.$$

Соответствующее описание на языке **Verilog** представлено в следующем листинге:

```
module half_adder_structural
(
   input wire x, y,
   output wire z,
   output wire carry_out
);
   xor(z, x, y);
   and(carry_out, x, y);
endmodule
```

Листинг 5.17 Структурная реализация ячейки полусумматора

[1] https://github.com/RomeoMe5/DDLM.

Во время синтеза модулей, представленных в листингах 5.16 и 5.17, **Quartus Prime** строит различные **RTL**-представления. В первом случае модуль отображается в ячейку сумматора, а во втором – строится соответствующее описание на уровне базовых вентилей (рис. 5.23). Следует отметить, что в рассматриваемом случае различия устраняются на последующем этапе, когда происходит привязка логической схемы к технологической библиотеке.

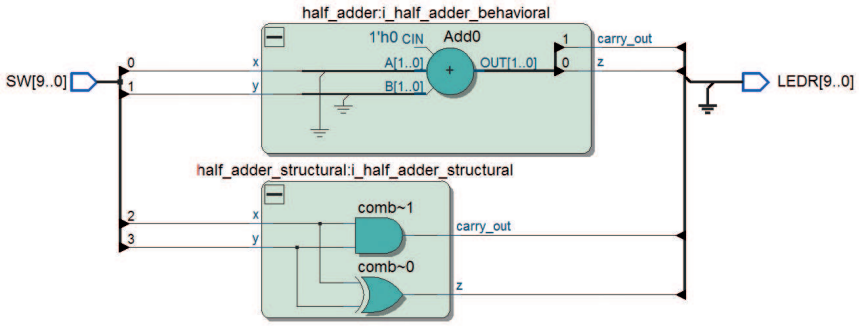


Рис. 5.23 RTL-описание ячеек полусумматора

5.2.2 Полный сумматор

В свою очередь, ячейка полного сумматора представляет арифметический блок, который осуществляет сложение двух однобитных двоичных чисел **x** и **y** с учетом возможного переноса (**carry_in**), возникшего на предыдущем этапе вычислений (например, перенос из предыдущего разряда при сложении многобитных чисел). При этом, как и в случае с ячейкой полусумматора, результатом вычислений является двухбитное число, старший бит которого считается битом переноса (**carry_out**). Листинг 5.18 представляет собой наиболее естественную реализацию ячейки полного сумматора.

```
module full_adder
(
   input wire x, y, carry_in,
   output wire z, carry_out
);
   assign {carry_out, z} = x + y + carry_in;
endmodule
```

Листинг 5.18 Поведенческое описание ячейки полного сумматора

Описание ячейки полного сумматора на уровне вентилей может быть выполнено на основе следующих булевых выражений:

$$\begin{cases} z = x \oplus y \oplus carry_{in} \\ carry_{out} = x \cdot y \vee (x \oplus y) \cdot carry_{in} \end{cases}.$$

Описание схемы на Verilog, которое реализует указанные булевы выражения, представлено ниже:

```
module full_adder_structural
(
   input wire x, y, carry_in,
   output wire z, carry_out
);
   wire [2:0] t;

   xor(t[0], x, y);
   xor(z, t[0], carry_in);
   and(t[1], x, y);
   and(t[2], t[0], carry_in);
   or (carry_out, t[1], t[2]);
endmodule
```

Листинг 5.19 Структурное описание ячейки полного сумматора

На выходе рассматриваемого блока **carry_out** реализует классическую функцию голосования, которая обращается в **1**, если два или более входных аргументов обращаются в **1**. При этом предложенная выше реализация этой функции использует подфункцию $x \oplus y$, которая является общей для этой функции и функции, реализующейся на выходе **z**, что приводит к более компактной с точки зрения вентилей реализации. Соответствующие **RTL**-описания данных вариантов реализаций ячеек полного сумматора представлены на рис. 5.24.

Отметим, что при проектировании блочных сумматоров под ячейкой полного сумматора зачастую понимают многобитный сумматор (листинг 5.1), который осуществляет сложение двоичных чисел заданной разрядности с возможным переносом. При этом указанные блоки используются в качестве основных строительных элементов для построения сумматоров большей разрядности.

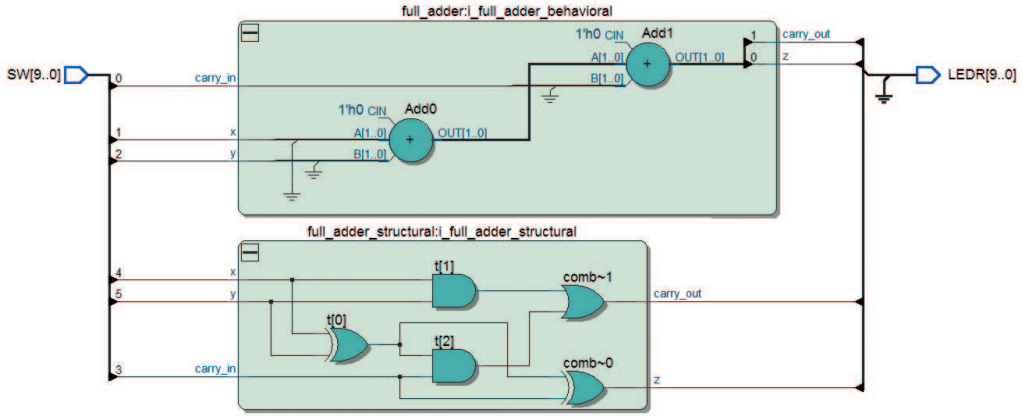


Рис. 5.24 RTL-описание ячеек полного сумматора

5.2.3 Перенос, каскадное соединение сумматоров

Рассмотрим построение многобитного сумматора на основе множества последовательно соединенных полных сумматоров. При этом выход **carry_out** очередного экземпляра модуля полного сумматора соединяется с входом **carry_in** следующего экземпляра модуля полного сумматора. Так как для такой реализации требуется объявление большого числа однотипных экземпляров модуля полного сумматора, то наиболее естественным является использование конструкции **generate**. Соответствующее описание на языке **Verilog** такого сумматора представлено ниже:

```
module ripple_carry_adder
# (
   parameter WIDTH = 8
)
(
   input carry_in,
   input [WIDTH - 1 : 0] x,
   input [WIDTH - 1 : 0] y,
   output [WIDTH - 1 : 0] z,
   output carry_out
);
   wire [WIDTH : 0] carry;

   assign carry[0] = carry_in;
   generate
      genvar i;
      for (i = 0; i <= WIDTH - 1; i = i + 1)
         begin : stage
            full_adder FA
            (
               .x (x [i] ),
               .y (y [i] ),
               .z (z [i] ),
               .carry_in (carry[i] ),
               .carry_out(carry[i + 1])
            );
      end
   endgenerate

   assign carry_out = carry[WIDTH];
endmodule
```

Листинг 5.20 Сумматор с каскадным переносом

Отметим несколько особенностей данной реализации. Параметр **WIDTH** задает количество объявляемых экземпляров модуля полного сумматора и фактически задает разрядность складываемых чисел. Вспомогательная шина **carry** используется для того, чтобы избежать краевых эффектов, которые возникают при объяв-

лении первого и последнего экземпляров модуля полного сумматора в конструкции **generate**. Блок **stage** нужен для единообразного описания всех экземпляров. При этом привязывание первого и последнего проводов шины **carry** к соответствующим портам, отвечающим за перенос, осуществляется при помощи соответствующих конструкций **assign** в начале и конце описания модуля.

На рис. 5.25 представлено **RTL**-описание, полученное в результате синтеза данного модуля сумматора в **Quartus Prime** в случае, когда параметр **WIDTH** равен **4**.

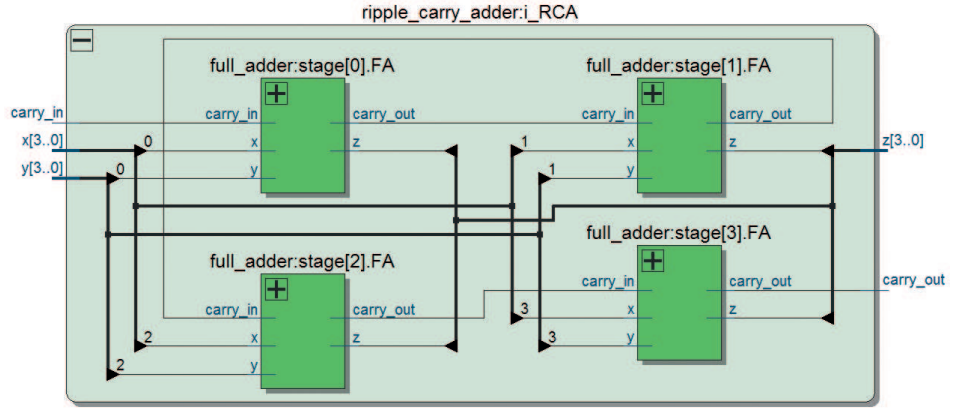


Рис. 5.25 RTL-описание 4-битного сумматора с каскадным переносом

Необходимо отметить, что из-за возникающих границ между экземплярами модулей полного сумматора в результирующей иерархии модулей, а также из-за особенностей алгоритмов, решающих задачу привязки логической схемы к библиотеке, результирующая технологически зависимая схема не будет задействовать шины, осуществляющие быстрый перенос. На следующем рисунке представлено соответствующее технологически зависимое описание:

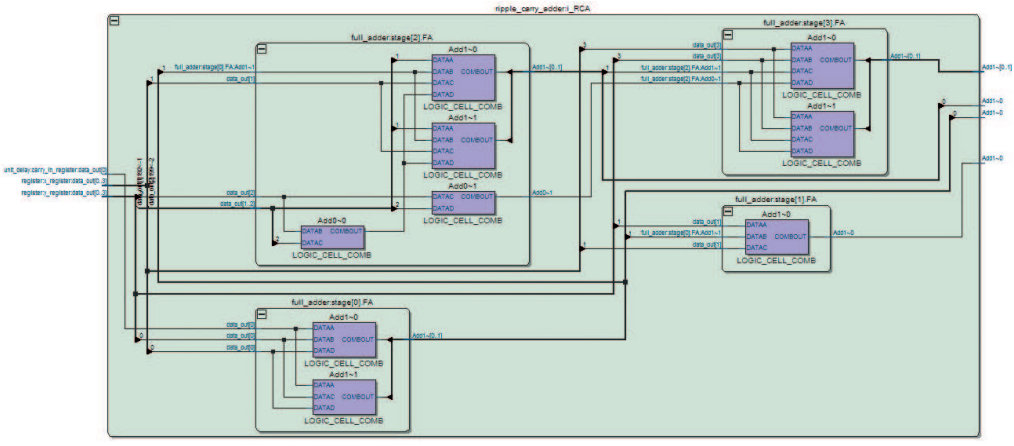


Рис. 5.26 Технологически зависимое описание 4-битного сумматора с каскадным переносом для ПЛИС MAX10

Оптимизировать реализацию сумматора с каскадным переносом можно за счет использования более крупных блоков, которые осуществляют сложение сразу нескольких битов исходных аргументов вместо однобитных блоков полного сумматора. Соответствующее описание блочного сумматора с каскадным переносом представлено в листинге 5.21.

```
module group_ripple_carry_adder
# (
   parameter BLOCK_COUNT = 2,
   parameter WIDTH = 2
)
(
   input carry_in,
   input [BLOCK_COUNT * WIDTH - 1 : 0] x,
   input [BLOCK_COUNT * WIDTH - 1 : 0] y,
   output [BLOCK_COUNT * WIDTH - 1 : 0] z,
   output carry_out
);
   wire [BLOCK_COUNT : 0] carry;

   assign carry[0] = carry_in;
   generate
      genvar i;
      for (i = 0; i <= BLOCK_COUNT - 1; i = i + 1)
         begin : stage
            adder #(.WIDTH(WIDTH)) FA
            (
               .x (x [WIDTH * (i + 1) - 1 : WIDTH * i]),
               .y (y [WIDTH * (i + 1) - 1 : WIDTH * i]),
               .z (z [WIDTH * (i + 1) - 1 : WIDTH * i]),
               .carry_in (carry[i] ),
               .carry_out(carry[i + 1] )
            );
         end
   endgenerate
   assign carry_out = carry[BLOCK_COUNT];
endmodule
```

Листинг 5.21 4-битный блочный сумматор с каскадным переносом

Отметим некоторые особенности этой реализации. Как и в листинге 5.20, в данном случае для описания сумматора используется конструкция **generate**, при этом внутри этого блока объявляются экземпляры модуля **adder** (листинг 5.1), разрядность которых определяется параметром **WIDTH**. За количество блоков отвечает параметр **BLOCK_COUNT**, который также определяет разрядность вспомогательной шины **carry**. Структура 4-битного сумматора, составленного из блоков разрядности **2**, представлена на рис. 5.27.

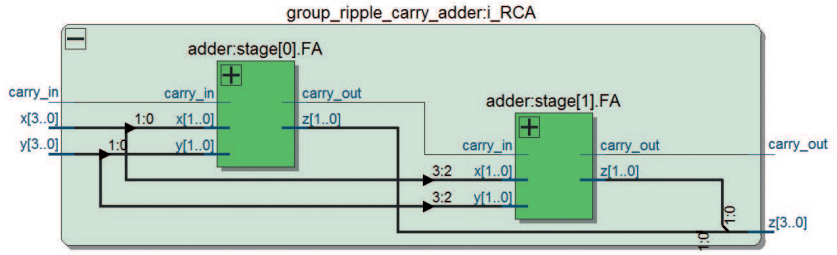


Рис. 5.27 Структура 4-битного блочного сумматора

На рис. 5.28 представлено технологически зависимое описание рассматриваемого блочного сумматора. Отметим, что такая реализация сумматора требует меньше настраиваемых логических блоков, уменьшает длину критического пути и позволяет задействовать цепи быстрого переноса в **ПЛИС MAX10**.

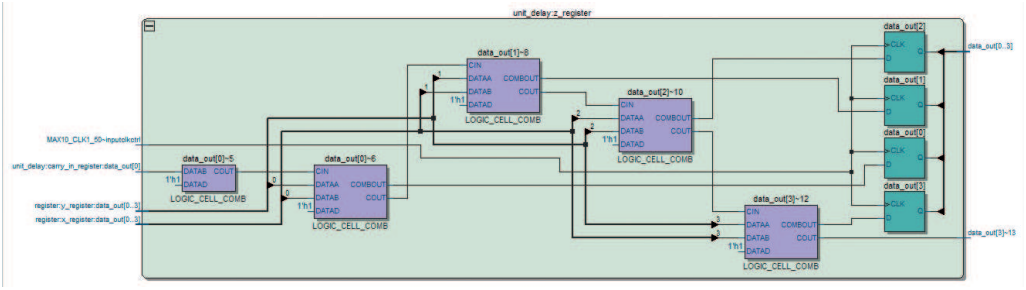


Рис. 5.28 Технологически зависимое описание 4-битного блочного сумматора с каскадным переносом для ПЛИС MAX10

В заключение стоит сказать, что n-разрядный сумматор с последовательным переносом имеет сложность $O(n)$, так как для реализации сложения одной пары битов требуется константное число логических элементов, и с точностью до мультипликативной константы указанная сложность является оптимальной. В свою очередь, задержка срабатывания в данной схеме также растет линейно с ростом разрядности сумматора, так как сигнал переноса последовательно вычисляется от самого младшего разряда к самому старшему разряду.

Дополнительное задание для самостоятельной работы

Реализуйте блочный сумматор с каскадным переносом разрядности **16**, **32**, **64** и **128** бит. Варьируя значения параметров разрядности блока (**WIDTH**) и количества блоков (**BLOCK_COUNT**), найдите их оптимальное соотношение с точки зрения задержки реализуемого блока, то есть такое соотношение, на котором достигается максимальная тактовая частота. Объясните полученные результаты.

5.3 Повышение скорости арифметических операций на примере сумматоров разных типов

Ранее были представлены описания сумматора, которые имеют задержку, растущую линейно с ростом разрядности аргументов. Естественно, такой рост задерж-

ки не удовлетворяет проектам с высокой разрядностью аргументов, требующим высокой частоты работы схемы. В данном разделе будут рассмотрены различные подходы к построению сумматоров, оптимизированных по быстродействию. Будут показаны приемы, которые, используя специальные схемы расчета сигнала переноса значения в следующий разряд, позволяют снизить задержку срабатывания до логарифмической зависимости.

5.3.1 Построение сумматора с ускоренным групповым переносом

Основная идея, позволяющая существенно снизить задержку срабатывания сумматора, заключается в том, что вместо последовательной схемы расчета сигналов переноса (**carry**) используются специальные схемы, которые позволяют ускорить процесс вычисления этих сигналов и сделать его независимым от вычисления выходных значений сумматора. Для указанных целей введем ряд вспомогательных сигналов, которые характеризуют состояние сигналов переноса.

Пусть $x = (x_{n-1}, ..., x_0)$ и $y = (y_{n-1}, ..., y_0)$ – входные аргументы **n**-разрядного сумматора, $z = (z_{n-1}, ..., z_0)$ – **n**-разрядный результат сложения, а $c = (c_n, ..., c_1, c_0)$ – шина сигналов переноса. При этом c_{i+1}, $i = 0, ..., n$ – это перенос, возникающий в результате сложения битов x_i и y_i. Тогда введем следующие сигналы:

- $g_i = x_i,\ i = 0,...,(n-1)$, обращается в **1**, если при сложении битов x_i и y_i возникает перенос (**generates**);
- $p_i = x_i \oplus y_i,\ i = 0,...,(n-1)$, обращается в **1**, если при сложении битов x_i и y_i значение переноса $c_{i+1} = c_i$, то есть значение переноса передается или распространяется (**propagates**) от предыдущего разряда к следующему.

Стоит отметить, что в литературе зачастую встречается немного другой набор вспомогательных сигналов. Например, вместо сигналов p_i используются сигналы

$$a_i = x_i \vee y_i = x_i \cdot y_i \vee (x_i \oplus y_i) = g_i \vee p_i,$$

характеризующие тот факт, что перенос в следующий разряд активен, то есть был сформирован в текущем разряде или передан из предыдущего разряда. Использование сигналов a_i продиктовано меньшей сложностью реализации дизъюнкции по сравнению с элементами **суммы по модулю 2** в ряде технологических библиотек. В случае **ПЛИС** указанные функции реализуются с одинаковой сложностью, поэтому для большей наглядности будут использоваться сигналы p_i.

Отметим, что все реализации сумматора, которые представлены в данном и в последующих разделах этой главы, имеют следующую общую структуру, состоящую из следующих трех блоков:

1) блок вычисления вспомогательных сигналов;
2) блок вычисления сигналов переноса;
3) блок вычисления результирующих значений суммы.

В листинге 5.22 представлено описание сумматора, имеющего приведенную выше структуру. В конструкции **generate** реализуются сигналы g_i и p_i, формирующие

первый блок, и для их передачи используются вспомогательные шины **generate_wire** и **propagate_wire**. Кроме того, в этой конструкции реализуются элементы выходного вектора **z**, формирующие третий блок рассматриваемой структуры. Так как выполняются следующие равенства:

$$z_i = x_i \oplus y_i \oplus c_i = p_i \oplus c_i,$$

то для реализации вектора **z** используются вспомогательная шина **propagate_wire** и шина **carry**. Наконец, экземпляр модуля **carry_lookahead_generator**, формирующий второй блок рассматриваемой структуры, применяет значения вспомогательных шин **generate_wire** и **propagate_wire** для расчета элементов шины **carry**.

```
module carry_lookahead_adder
# (
   parameter WIDTH = 8
)
(
   input carry_in,
   input [WIDTH - 1 : 0] x,
   input [WIDTH - 1 : 0] y,
   output [WIDTH - 1 : 0] z,
   output carry_out
);

   wire [WIDTH : 0] carry;
   wire [WIDTH - 1 : 0] generate_wire, propagate_wire;

   carry_lookahead_generator #(.WIDTH(WIDTH)) i_CLG
   (
      .carry_in (carry_in ),
      .generate_in (generate_wire ),
      .propagate_in (propagate_wire),
      .carry (carry ),
      .group_propagate(),
      .group_generate ()
   );

   generate
      genvar i;
         for (i = 0; i <= WIDTH - 1; i = i + 1)
         begin : stage
            //Create generate and propagate signals
            assign generate_wire [i] = x[i] & y[i];
            assign propagate_wire[i] = x[i] ^ y[i];
            //Calculate sum (z bus)
            assign z[i] = carry[i] ^ propagate_wire[i];
         end
   endgenerate
```

```
   assign carry_out = carry[WIDTH];
endmodule
```

Листинг 5.22 Описание сумматора с предварительным вычислением переноса

Структура сумматора, описанного в листинге 5.22, для случая **WIDTH = 4** представлена на рис. 5.29. Так как сложность реализации первого и третьего блоков растет линейно с ростом разрядности складываемых чисел, то можно утверждать, что общая сложность $L_{CLA}(n)$ данной реализации задается следующим равенством:

$$L_{CLA} = L_{CLG}(n) + O(n),$$

где $L_{CLG}(n)$ – сложность второго блока, который осуществляет вычисление сигналов переноса.

В свою очередь, все сигналы на выходах первого и третьего блоков могут быть вычислены независимо и параллельно. При этом по рис. 5.29 следует, что для вычисления каждого сигнала требуется конечное число аргументов. Поэтому задержка, возникающая в этих блоках, является константной и не зависит от разрядности сумматора. Таким образом, общая задержка срабатывания $D_{CLA}(n)$ данной реализации описывается соотношением:

$$D_{CLA} = D_{CLG}(n) + O(1),$$

где $D_{CLG}(n)$ – задержка срабатывания второго блока.

Таким образом, в данной структуре ключевым блоком, определяющим задержку и сложность итоговой схемы, является второй блок. Рассмотрим далее примеры оптимальных по задержке реализаций данного блока.

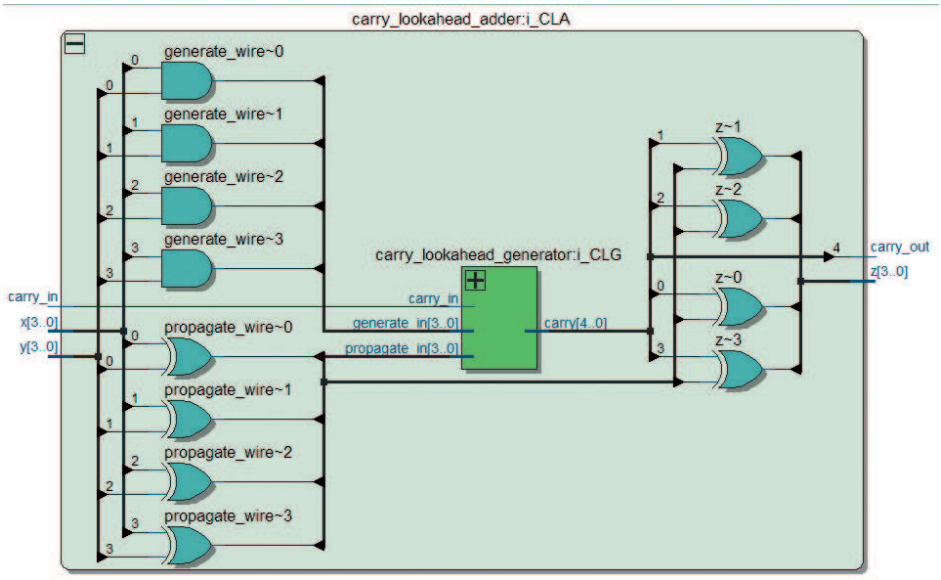


Рис. 5.29 Общая схема сумматора с предварительным вычислением переноса

Все элементы c_i, $i = 1, ..., n$, вектора c, хранящего информацию о значениях сигналов переноса, можно вычислить с использованием вспомогательных сигналов g_i и p_i при помощи следующей рекуррентной формулы:

$$c_{i+1} = g_i \vee p_i \cdot c_i$$

Эта формула говорит о том, что перенос в следующий разряд ($C_{i+1} = 1$) возникает тогда, когда наступает хотя бы одно из следующих событий:

1) перенос порождается в текущем разряде ($g_i = 1$);
2) перенос возник в предыдущем разряде ($c_i = 1$) и передается в следующий разряд ($p_i = 1$).

Реализация блока предварительного вычисления переноса, построенная на основе описанного выше рекуррентного соотношения, представлена в листинге 5.23.

```
module carry_lookahead_generator
# (
   parameter WIDTH = 8
)
(
   input carry_in,
   input [WIDTH - 1 : 0] generate_in, propagate_in,
   output [WIDTH : 0] carry,
   output group_generate, group_propagate
);
   wire [WIDTH - 1 : 0] g_temp;
   wire [WIDTH - 2 : 0] p_temp;

   assign carry[0] = carry_in;
   generate
      genvar i;
         for (i = 0; i <= WIDTH - 1; i = i + 1)
         begin : stage
            assign carry[i + 1] = generate_in[i] | propagate_in[i] & carry[i];
            case(i)
               WIDTH - 1 : assign g_temp[i] = generate_in[i];
               default : begin
                  assign p_temp[i] = & propagate_in[WIDTH - 1 : i + 1];
                  assign g_temp[i] = p_temp[i] & generate_in[i];
               end
            endcase
         end
   endgenerate
   assign group_propagate = & propagate_in;
   assign group_generate = | g_temp;
endmodule
```

Листинг 5.23 Описание блока вычисления переноса на основе рекуррентных формул

Кроме того, данный модуль содержит два дополнительных выхода, которые вычисляют дополнительные вспомогательные сигналы группового распространения сигнала переноса (**group_propagate**) и группового сигнала формирования переноса (**group_generate**).

Следующий рисунок иллюстрирует структуру соответствующей RTL-схемы:

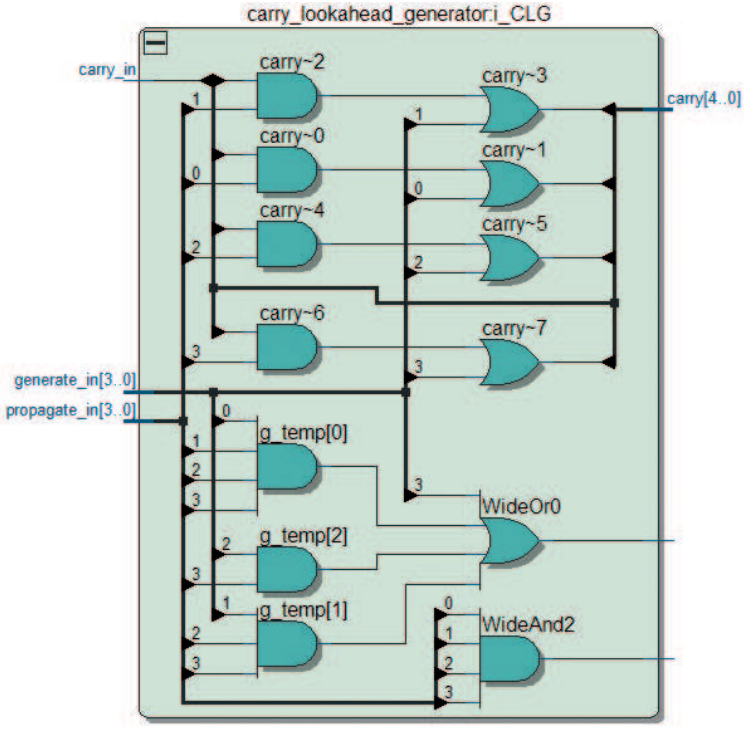


Рис. 5.30 RTL-схема блока последовательного предварительного вычисления сигналов переноса

Блок предварительного вычисления переноса, описание которого получено на основе рекуррентного соотношения, будет иметь линейную задержку. Для оптимизации задержки указанного блока требуется развернуть рекурсию, что позволяет получить явные формулы для вычисления сигналов переноса. Выпишем указанные формулы для нескольких первых значений i:

$$c_1 = g_0 \vee p_0 c_{in};$$

$$c_2 = g_1 \vee p_1 c_1 = g_1 \vee p_1 (g_0 \vee p_0 c_{in}) = g_1 \vee p_1 g_0 \vee p_1 p_0 c_{in}.$$

Таким образом, для произвольного значения параметра i явная формула будет иметь следующий вид:

$$c_{i+1} = g_i \vee p_i g_{i-1} \vee p_i p_{i-1} g_{i-2} \vee \ldots \vee p_i p_{i-1} \ldots p_0 c_{in}.$$

Последнюю формулу можно записать более компактно, группируя соответствующие слагаемые:

$$c_{i+1} = \bigvee_{j=0}^{i}\left(\bigwedge_{k=j+1}^{i} p_k\right) gi \vee \left(\bigwedge_{k=0}^{i} p_k\right) c_{in}.$$

Пусть $P_{[i,j]} = \bigwedge_{k=j}^{i} p_k$, тогда предыдущее выражение можно записать следующим образом:

$$c_{i+1} = \bigvee_{j=0}^{i} P_{[j+1,i]} g_j \vee P_{[0,i]} c_{in}.$$

Подведем итог: для описания явных формул для сигналов переноса c_i, $i = 1, \ldots, (n - 1)$, нужно сначала описать сигналы, которые задаются вспомогательными формулами $P_{[b, a]}$, для всех интервалов $[b, a]$, $0 \leqslant a \leqslant b \leqslant n + 1$. Введем нумерацию интервалов, чтобы впоследствии можно было все выходы вспомогательных формул $P_{[b, a]}$ сгруппировать в одну шину. Каждому интервалу $[b, a]$ поставим в соответствие целое число $\upsilon_{[b, a]}$, которое определяется по следующей формуле:

$$\upsilon_{[b,a]} = \frac{(a+1)a}{2} + b.$$

При такой нумерации все интервалы, имеющие одинаковое значение правой границы a, упорядочены по возрастанию левой границы b, при этом указанные группы интервалов также упорядочены по возрастанию правой границы a. Приведем пример данной нумерации интервалов для случая $n = 1$:

$$\upsilon_{[0,0]} = 0,\ \upsilon_{[1,0]} = 1,\ \upsilon_{[2,0]} = 2,$$
$$\upsilon_{[1,1]} = 2,\ \upsilon_{[2,1]} = 3,$$
$$\upsilon_{[2,2]} = 5.$$

Учитывая введенную нумерацию интервалов, блок генерации сигналов переноса можно описать при помощи языка **Verilog** следующим образом:

```
module carry_lookahead_generator
# (
   parameter WIDTH = 8
)
(
   input carry_in,
   input [WIDTH - 1 : 0] generate_in, propagate_in,
   output [WIDTH : 1] carry,
   output group_generate, group_propagate
);

   //Temporary wires for the carry calculation
   wire [((WIDTH + 1) * WIDTH) / 2 - 1 : 0] p_temp, g_temp;
   wire [WIDTH - 1 : 0] c_temp, pg_temp;

   generate
      genvar i, j;
      for (i = 0; i <= WIDTH - 1; i = i + 1)
      begin : stage
```

```
        //Calculate carry
        for (j = 0; j <= i; j = j + 1)
        begin : block
           assign p_temp[((i + 1) * i) / 2 + j] = & propagate_in[i : j];
           case(j)
              0 : assign g_temp[((i + 1) * i) / 2 + j] = generate_in[i];
              default : assign g_temp[((i + 1) * i) / 2 + j] =
                 p_temp[((i + 1) * i) / 2 + j] & generate_in[j - 1];
           endcase
        end
        assign c_temp [i] = p_temp[((i + 1) * i) / 2] & carry_in;
        assign pg_temp[i] = |g_temp[((i + 2)*(i + 1))/2 - 1:((i + 1)*i)/2];

        assign carry [i + 1] = pg_temp[i] | c_temp[i];
     end
   endgenerate
   assign group_propagate = p_temp [(WIDTH * (WIDTH - 1)) / 2];
   assign group_generate = pg_temp[WIDTH - 1];
endmodule
```

Листинг 5.24 Описание блока генерации сигналов переноса на основе явных формул

Поясним некоторые особенности предложенного описания. Для реализации операций конъюнкции и дизъюнкции множества элементов шин используются соответствующие операторы свертки | и **&** (**reduction operators**). При этом в описании используется ряд вспомогательных шин, которые применяются для передачи значений промежуточных вспомогательных формул. Шина **p_temp** передает значения вспомогательных формул $P_{[i,\ j]}$ в соответствии с введенной нумерацией, шина **g_temp** передает сигналы g_i и сигналы $P_{[j+1,\ i]}g_i$, шина **c_temp** – сигналы $P_{[0,\ i]}c_{in}$. В свою очередь, шина **pg_temp** передает результат дизъюнкции соответствующих слагаемых шины **g_temp**, над которым выполняется операция дизъюнкции с соответствующим элементом шины **c_temp** для получения финального значения сигнала переноса.

Структура соответствующей **RTL**-схемы, реализующей данное описание, представлена на рисунке ниже:

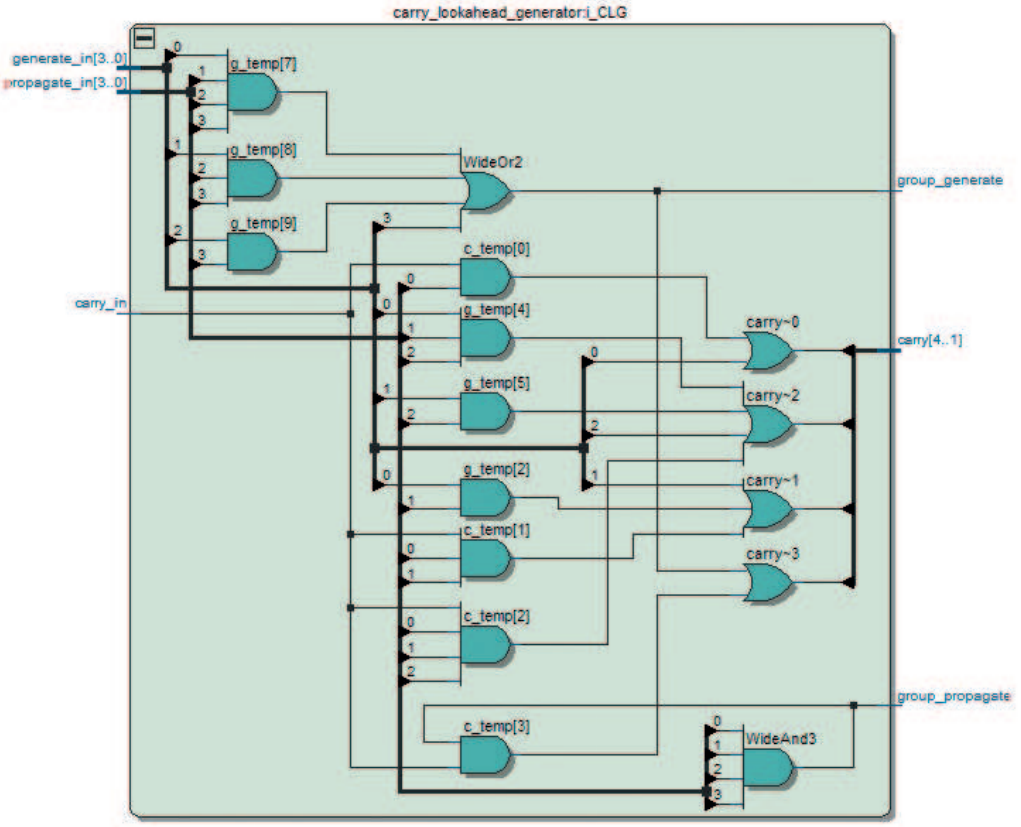


Рис. 5.31 RTL-схема блока генерации сигналов переноса, синтезированная на основе явных формул

Представленные выше формулы для явного вычисления сигналов переноса могут быть использованы для грубой оценки задержки срабатывания рассматриваемого блока. Это возможно, поскольку задержка каждой из вспомогательных формул $P_{[b,\ a]}$ составляет не более $O(\log n)$, так как результаты синтеза будут представлять сбалансированное дерево из логических элементов. Конъюнкции указанных вспомогательных формул с сигналами g_i и c_{in} приводят к константной задержке, и, наконец, внешняя дизъюнкция, которая содержит не более $n + 2$ слагаемых, увеличивает задержку еще на $O(\log n)$ единиц. В итоге задержка срабатывания такой реализации будет описываться логарифмической зависимостью.

Дополнительное задание для самостоятельной работы

Используя статический временной анализ, получите оценку максимальной тактовой частоты сумматора с предварительным вычислением переноса для разных значений разрядности (**8**, **16**, **32**, **64** и **128** бит). Опишите зависимость изменения данного параметра как функцию от разрядности исследуемого сумматора.

5.3.2 Последовательное соединение сумматоров с ускоренным групповым переносом

Рассмотренный в предыдущем разделе сумматор с ускоренным групповым переносом имеет довольно сложный для реализации блок генерации сигналов переноса, поэтому интерес представляет комбинированная реализация сумматора, предполагающая предварительное вычисление сигналов переноса для групп аргументов заданного размера.

Один из наиболее простых подходов к разработке такого сумматора основан на том, что рассматриваемые блоки с предварительным вычислением переноса соединяются последовательно. Реализация такого подхода представлена в следующем листинге:

```
module ripple_block_CLA
# (
   parameter GROUP_WIDTH = 4,
   parameter GROUP_COUNT = 2
)
(
   input carry_in,
   input [GROUP_COUNT * GROUP_WIDTH - 1 : 0] x,
   input [GROUP_COUNT * GROUP_WIDTH - 1 : 0] y,
   output [GROUP_COUNT * GROUP_WIDTH - 1 : 0] z,
   output carry_out
);
   wire [GROUP_COUNT : 0] carry;
   assign carry[0] = carry_in;
   generate
      genvar i;
      for (i = 0; i <= GROUP_COUNT - 1; i = i + 1)
      begin : stage
         carry_lookahead_adder #(.WIDTH(GROUP_WIDTH)) i_CLA
         (
            .carry_in (carry[i] ),
            .x (x[(i + 1) * GROUP_WIDTH - 1 : i * GROUP_WIDTH]),
            .y (y[(i + 1) * GROUP_WIDTH - 1 : i * GROUP_WIDTH]),
            .z (z[(i + 1) * GROUP_WIDTH - 1 : i * GROUP_WIDTH]),
            .carry_out(carry[i + 1] )
         );
      end
   endgenerate
   assign carry_out = carry[GROUP_COUNT];
endmodule
```

Листинг 5.25 Описание последовательного соединения сумматоров с ускоренным групповым переносом

На следующем рисунке представлена схема **8**-битного сумматора, полученного в результате последовательного соединения двух **4**-битных сумматоров:

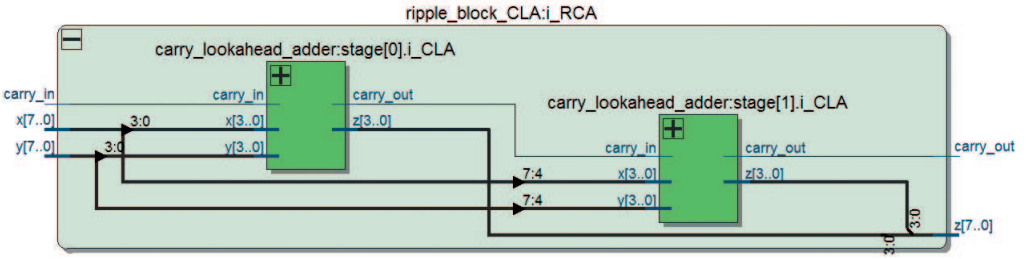


Рис. 5.32 RTL-схема двух последовательно соединенных 4-битных сумматоров с ускоренным групповым переносом

Фактически представленный сумматор является сумматором с каскадным переносом, в котором основными блоками являются сумматоры с ускоренным групповым переносом заданной разрядности. Такая реализация позволяет уменьшить суммарную сложность блоков предварительного вычисления сигналов переноса, но при этом увеличивает задержку схемы.

5.3.3 Двухуровневая реализация сумматора с ускоренным групповым переносом

Задержку схемы и ее сложность можно сбалансировать при помощи формирования многоуровневой структуры из сумматоров и блоков предварительного вычисления переноса. Такая структура предполагает наличие множества сумматоров, вычисляющих вспомогательные сигналы генерации и передачи переноса для группы аргументов. Данные сигналы впоследствии используются блоками предварительного вычисления переноса для вычисления сигналов переноса, которые подаются на соответствующие входы рассматриваемого множества сумматоров.

Следующий листинг представляет сумматор, в котором сумматоры и блоки предварительного вычисления переноса формируют двухуровневую структуру:

```
module two_level_CLA
# (
   parameter WIDTH_0 = 2,
   parameter WIDTH_1 = 2,
   parameter GROUP_COUNT = 2
)
(
   input carry_in,
   input [GROUP_COUNT * WIDTH_1 * WIDTH_0 - 1 : 0 ] x,
   input [GROUP_COUNT * WIDTH_1 * WIDTH_0 - 1 : 0 ] y,
   output [GROUP_COUNT * WIDTH_1 * WIDTH_0 - 1 : 0 ] z,
   output carry_out
);
   wire [GROUP_COUNT * WIDTH_1 - 1 : 0] p_temp, g_temp;
```

```
   wire [GROUP_COUNT * WIDTH_1 : 0] c_temp;

   assign c_temp[0] = carry_in;
   generate
      genvar i;
      genvar j;
      for (i = 0; i <= GROUP_COUNT * WIDTH_1 - 1; i = i + 1)
      begin : stage
         carry_lookahead_adder #(.WIDTH(WIDTH_0)) i_CLA
         (
            .carry_in (c_temp[i] ),
            .x (x[(i + 1) * WIDTH_0 - 1 : i * WIDTH_0]),
            .y (y[(i + 1) * WIDTH_0 - 1 : i * WIDTH_0]),
            .z (z[(i + 1) * WIDTH_0 - 1 : i * WIDTH_0]),
            .group_propagate(p_temp[i] ),
            .group_generate (g_temp[i] ),
            .carry_out ()
         );
      end

      for (j = 0; j <= GROUP_COUNT - 1; j = j + 1)
      begin : clg_stage
         carry_lookahead_generator #(.WIDTH(WIDTH_1)) i_CLG
         (
            .carry_in (c_temp[WIDTH_1 * j] ),
            .generate_in (g_temp[WIDTH_1 * (j + 1) - 1 : WIDTH_1 * j]),
            .propagate_in (p_temp[WIDTH_1 * (j + 1) - 1 : WIDTH_1 * j]),
            .carry (c_temp[WIDTH_1 * (j + 1) : WIDTH_1 * j + 1]),
            .group_propagate(),
            .group_generate ()
         );
      end
   endgenerate
   assign carry_out = c_temp[GROUP_COUNT * WIDTH_1];
endmodule
```

Листинг 5.26 Описание двухуровневого сумматора с предварительным групповым вычислением переноса

На рис. 5.33 представлена структура основных блоков 8-битного сумматора, синтезированного на основе листинга 5.26 (для случая, когда все блоки имеют разрядность **2** и на втором уровне располагается всего два блока предварительного вычисления переноса).

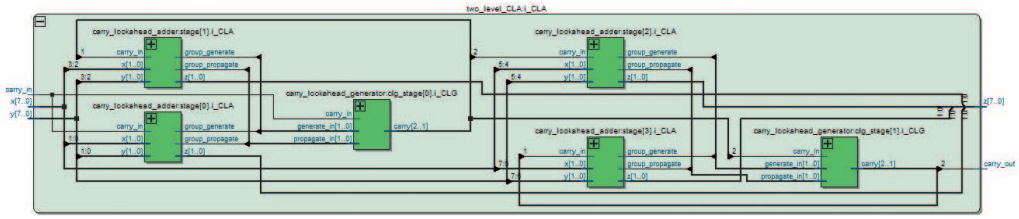


Рис. 5.33 Структура 8-битного двухуровневого сумматора с предварительным групповым вычислением переноса

Стоит отметить, что число уровней в рассматриваемой структуре можно увеличить, формируя иерархию из блоков предварительного вычисления переноса. Таким образом, можно балансировать задержку и сложность схемы при использовании такого подхода к реализации сумматоров.

5.3.4 Префиксный сумматор

Префиксные сумматоры представляют собой еще один класс реализаций сумматоров, которые базируются на идее предварительного вычисления сигналов переноса. При этом считается, что вектор сигналов переноса представляет собой результат применения специального префиксного оператора к вспомогательным сигналам g_i и p_i.

Введем ряд вспомогательных определений, которые необходимы для описания процесса разработки префиксного сумматора. Для произвольного интервала $[b, a]$, $0 \leqslant a \leqslant b \leqslant n+1$, введем сигнал $G_{[b,a]}$, который обращается в единицу, если при сложении всех битов из указанного интервала возник перенос, и в ноль – в противоположном случае. Фактически сигналы $G_{[b,a]}$ являются обобщением сигналов g_i. То есть согласно данному определению:

$$c_{i+1} = G_{[i,0]}.$$

Пользуясь введенными ранее сигналами $P_{[b,a]}$, формулы для вычисления сигналов переноса выразим следующим образом:

$$c_{i+1} = G_{[i,j]} \vee P_{[i,j]} c_j,\ j \leqslant i.$$

Перенос возникает в одном из двух случаев: либо перенос был сгенерирован на интервале от j до i, либо сигнал переноса был сгенерирован сигналом c_j и передан на указанном интервале.

Аналогичным образом можно задать формулы для расчета сигналов $P_{[b,a]}$ и $G_{[b,a]}$ на основе пересекающихся интервалов меньшей длины. Для произвольных и d_1, d_2, $a < d_1 \leqslant d_2 < b$, верны следующие соотношения:

$$P_{[b,a]} = P_{[b,d1]} \cdot P_{[d2,a]},$$

$$G_{[b,a]} = G_{[b,d1]} \vee P_{[b,d1]} \cdot G_{[d2,a]}.$$

Справедливость первого определяется тем фактом, что, для того чтобы перенос передавался на всем интервале $[b, a]$, он должен передаваться и на интервале

$[d_2, a]$, и на интервале $[b, d_1]$. Для сигнала генерации переноса $G_{[b, a]}$, как и прежде, возможны два случая: либо сигнал был сгенерирован на интервале $[b, d_1]$, либо он был сгенерирован на интервале $[d_2, a]$ и был передан от d_2 до b. Так как интервал $[b, d_2]$ вложен в интервал $[b, d_1]$, то вместо $P_{[b, d2]}$ можно использовать $P\,[b, d_1]$.

Сформулируем определение префиксного оператора, который будет использоваться для вычисления всех сигналов переноса. В общем случае, для произвольного ассоциативного оператора $\circ$ и множества переменных $x_0, x_1 \ldots, x_{n-1}$, префиксным называется оператор, который порождает вектор длины **n** из следующих элементов:

$$z_i = x_i \circ x_{i-1} \circ \ldots \circ x_1 \circ x_0,\ i = 0, \ldots, n - 1.$$

Пусть теперь $x_i = (g_i, p_i)$, а в качестве ассоциативного оператора $\circ$ будет использован оператор, который по паре кортежей (g_L, p_L) и (g_R, p_R) вычисляет кортеж (g_{out}, p_{out}), элементы которого определяются следующим образом:

$$g_{out} = g_L \vee p_L g_R,$$

$$p_{out} = p_L \cdot p_R.$$

Тогда, исходя из введенных ранее соотношений для расчета сигналов $P_{[b,\ a]}$ и $G_{[b, a]}$ на основе пересекающихся подынтервалов, соответствующий префиксный оператор порождает вектор из следующих элементов:

$$z_i = (G_{[i, 0]}, P_{[i, 0]}).$$

Таким образом, первый элемент $G_{[i, 0]}$ в результирующих кортежах z_i задает сигналы переноса c_{i+1}. В итоге если описать блок, реализующий введенный выше ассоциативный оператор, то блок предварительного вычисления сигналов переноса префиксного сумматора может быть построен при помощи специальной сети из указанных блоков, которые реализуют соответствующий префиксный оператор.

Существует целый ряд различных сетей, которые позволяют реализовать рассматриваемый префиксный оператор. Рассмотрим сеть под названием **Сумматор Скланского** (**Sklansky adder**[1]), которая позволяет построить блок предварительного вычисления сигналов переноса, имеющий логарифмическую задержку. Идея построения данной сети основана на том, что все необходимые интервалы

1 *Sklansky J.* Conditional-sum addition logic // IRE Transactions on Electronic computers, 1960, (2), 226–231.

могут быть получены на основе последовательного деления отрезков пополам. Такую сеть можно изобразить в виде следующего графа:

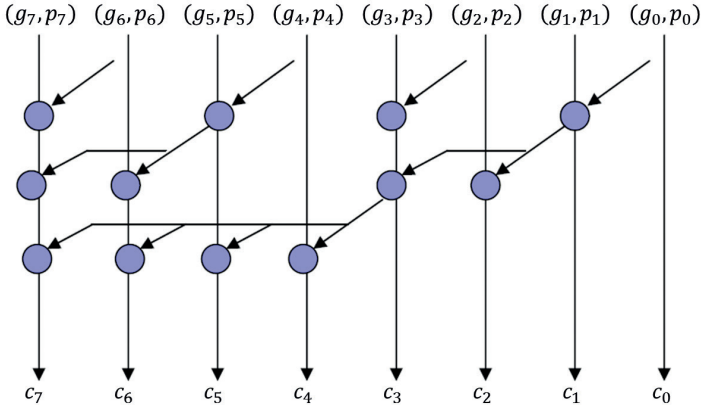


Рис. 5.24 Граф префиксной сети

Вершины данного графа представляют введенный выше ассоциативный оператор, а ребра графа указывают, как эти элементы связаны между собой.

Рассмотрим описание данной сети на языке **Verilog**. Модуль, реализующий ассоциативный оператор, представлен ниже:

```
module gp_cell
(
   input wire g_left, g_right, p_left, p_right,
   output wire g_out, p_out
);
   assign g_out = g_left | p_left & g_right;
   assign p_out = p_left & p_right;
endmodule
```

Листинг 5.27 Описание ассоциативного оператора префиксной сети

В свою очередь, саму префиксную сеть данной структуры можно описать следующим параметрическим модулем:

```
module prefix_carry_generator
# (
   parameter LEVELS = 3,
   parameter WIDTH = 2 ** LEVELS
)
(
   input carry_in,
   input [ WIDTH - 1 : 0 ] generate_in, propagate_in,
   output [ WIDTH - 1 : 0 ] carry,
   output carry_out
);
```

```
   // Sklansky adder
   wire [WIDTH : 0] g_temp [LEVELS : 0];
   wire [WIDTH : 0] p_temp [LEVELS : 0];

   assign g_temp[0] = {generate_in, carry_in};
   assign p_temp[0] = {propagate_in, carry_in};

   generate
      genvar i, j;
      for (i = 0; i <= LEVELS - 1; i = i + 1)
      begin : stage
         for (j = 0; j <= WIDTH; j = j + 1)
         begin : block
            if( (j / 2 ** i) % 2 == 1 )
               gp_cell i_gp_cell
               (
                  .g_left (g_temp[i][j] ),
                  .g_right(g_temp[i][(j / (2**i)) * (2**i) - 1]),
                  .p_left (p_temp[i][j] ),
                  .p_right(p_temp[i][(j / (2**i)) * (2**i) - 1]),
                  .g_out (g_temp[i+1][j] ),
                  .p_out (p_temp[i+1][j] )
               );
            else
            begin
               assign g_temp[i+1][j] = g_temp[i][j];
               assign p_temp[i+1][j] = p_temp[i][j];
            end
         end
      end
   endgenerate

   assign carry = g_temp[LEVELS][WIDTH-1:0];
   assign carry_out = g_temp[LEVELS][WIDTH]|p_temp[LEVELS]
      [WIDTH]&g_temp[LEVELS][WIDTH-1];
endmodule
```

Листинг 5.28 Описание префиксной сети

В данном случае описание сети происходит по слоям. Шины **p_temp** и **g_temp** используются для того, чтобы передавать значения с одного слоя на другой, а основной **generate**-блок отвечает за генерацию всех ассоциативных операторов и связывание проводов между соседними уровнями. Сигнал **carry_out** реализуется при помощи дополнительного оператора **assign**, чтобы не нарушать и не усложнять общую структуру описания префиксной сети. Соответствующая **RTL**-схема, полученная по данному описанию, представлена на рис. 5.35.

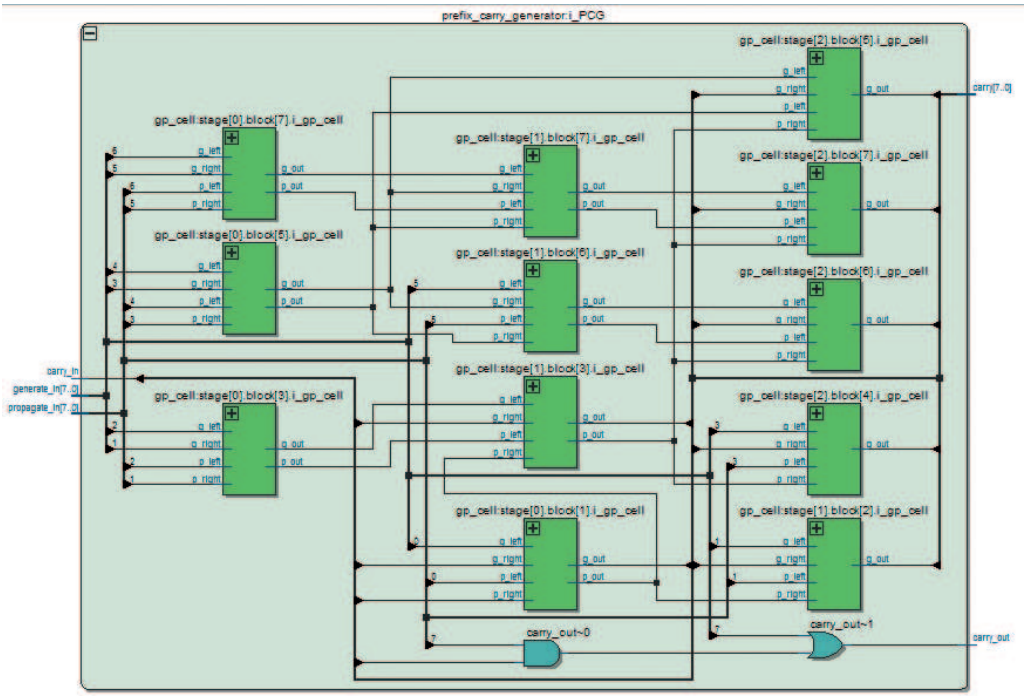


Рис. 5.35 RTL-схема префиксной сети

Дополнительное задание для самостоятельной работы

Используя статический временной анализ, опишите зависимость максимальной тактовой частоты префиксного сумматора как функцию от разрядности исследуемого устройства. Сравните это поведение с поведением аналогичных функций для сумматора с последовательным вычислением переноса и для блочного сумматора с предварительным вычислением переноса.

5.4 Упражнения

5.4.1 Основное задание

Выполните дополнительные задания для разделов 5.1.2, 5.1.5, 5.2.3, 5.3.1 и 5.3.4.

5.4.2 Задания для самостоятельной работы

Используя примеры кода из данного пособия, реализуйте следующие комбинационные блоки:

1. Устройство сдвига заданной разрядности (**8**, **16** и **32** бита) на основе мультиплексоров.
2. Устройства циклического сдвига заданной разрядности (**8**, **16** и **32** бита) на основе мультиплексоров.

3. Блочный сумматор с **пропуском переноса** (**Carry-Skip Adder**) заданной разрядности (**8**, **16**, **32** и **64** бит) и заданным размером блока (**2**, **4** и **8** бит).
4. Блочный сумматор с **пропуском переноса**, имеющий настраиваемое число блоков **BLOCK_COUNT** и размер блока **BLOCK_SIZE**.
5. Блочный сумматор с **выбором переноса** (**Carry-Select Adder**) заданной разрядности (**8**, **16**, **32** и **64** бит) и заданным размером блока (**2**, **4** и **8** бит).
6. Блочный сумматор с **выбором переноса**, имеющий настраиваемое число блоков **BLOCK_COUNT** и размер блока **BLOCK_SIZE**.
7. Параллельный префиксный сумматор **Когге–Стоуна** (**Kogge-Stone Adder**[1]) заданной разрядности (**8**, **16** и **32** бит).
8. Параллельный префиксный сумматор **Когге–Стоуна**, имеющий настраиваемое число уровней префиксной сети при помощи параметра **LEVELS**.
9. Параллельный префиксный сумматор **Брента–Кунга** (**Brent-Kung Adder**[2]) заданной разрядности (**8**, **16** и **32** бит).
10. Параллельный префиксный сумматор **Брента–Кунга**, имеющий настраиваемое число уровней префиксной сети при помощи параметра **LEVELS**.
11. Параллельный префиксный сумматор **Ладнера–Фишера** (**Ladner-Fischer Adder**[3]) заданной разрядности (**8**, **16** и **32** бит).
12. Параллельный префиксный сумматор **Ладнера–Фишера**, имеющий настраиваемое число уровней префиксной сети при помощи параметра **LEVELS**.
13. Блок умножения двух бинарных чисел заданной разрядности (**8**, **16**, **32** и **64** бита) на основе алгоритма «умножения в столбик».
14. Блок умножения двух бинарных чисел заданной разрядности (**8**, **16**, **32** и **64** бита) на основе **дерева Уоллеса** (**Wallace Tree Multiplier**[4]).
15. Блок умножения двух бинарных чисел заданной разрядности (**8**, **16**, **32** и **64** бита) на основе **алгоритма Бута** (**Booth Multiplier**[5]).

5.4.3 Контрольные вопросы

1. Что такое комбинационный сумматор? Приведите примеры реализаций.
2. С помощью каких средств **Quartus Prime** можно проанализировать технологически зависимую и структурную реализацию блоков проекта?

---

[1] *Kogge P. M., Stone H. S.* A parallel algorithm for the efficient solution of a general class of recurrence equations // IEEE transactions on computers, 1973, 100 (8), 786–793.

[2] *Brent R. P., Kung H. T.* A regular layout for parallel adders // IEEE transactions on Computers, 1982, (3), 260–264.

[3] *Ladner R. E., Fischer M. J.* Parallel prefix computation // Journal of the ACM (JACM), 1980, 27 (4), 831–838.

[4] *Wallace C. S.* A suggestion for a fast multiplier // IEEE Transactions on electronic Computers, 1964, (1), 14–17.

[5] *Booth A. D.* A signed binary multiplication technique // The Quarterly Journal of Mechanics and Applied Mathematics, 1951, 4 (2), 236–240.

3. Опишите пример сумматора с синхронизированными входами и выходами. Для чего он использовался в данной главе?
4. Опишите, для чего нужен статический временной анализ. Как осуществляется статический временной анализ в **Quartus Prime**?
5. Опишите назначение компаратора и приведите примеры его реализации.
6. Опишите назначение устройства сдвига и приведите пример его реализации. Каких типов бывают устройства сдвига?
7. АЛУ – назначение и пример реализации. Почему АЛУ обычно реализуют как комбинационное устройство?
8. Иерархический подход к проектированию сумматоров различной размерности. Как осуществляется перенос и каскадное соединение сумматоров при иерархическом подходе?
9. Как осуществляется построение сумматоров с ускоренным групповым переносом?
10. Опишите, как выполняется двухуровневая реализация сумматора с ускоренным групповым переносом.
11. Опишите, как реализуется префиксный сумматор, в чем его отличие от других типов сумматоров.

Александр Телятников, Александр Романов

Цифровой синтез: практический курс

Глава 6. Последовательная логика. Счетчики и сдвиговые регистры

Содержание

В главе рассмотрена реализация последовательностной логики на примере счетчиков и сдвиговых регистров. Работа сдвиговых регистров демонстрируется на примере различных вариаций «бегущих огоньков» для одномерного массива светодиодов, расположенного на плате, и для двумерного массива с использованием внешней матрицы светодиодов. Приводятся примеры использования счетчиков для генерации звука и измерения расстояния.

*Также рассмотрен принцип работы сдвигового регистра с линейной обратной связью. Такие регистры применяются для вычисления циклического избыточного кода (**CRC**), как средства обнаружения ошибок в сетях и устройствах хранения информации и для генерации псевдослучайных множеств, которые успешно используются в области криптографии.*

Требования к аппаратным и программным средствам

Для выполнения практических работ понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime** (есть студенческая версия);*
- *пакет **ModelSim Altera Edition** или программы I**carus Verilog** и **GTKWave**;*
- *отладочная плата компании **Terasic DE10-Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах[1] к данной книге).*

6.1 Последовательностные устройства

На заре эпохи цифрового проектирования одним из основных подходов при разработке цифровых микросхем был асинхронный. И большинство цифровых схем тогда были построены соответственно, а самыми яркими представителями того времени стали мейнфреймы: **ILLIAC II**, **Atlas** и **MU5**. Внутренние связи в асинхронных схемах осуществляются с помощью передачи последовательности сигналов, направленных от приемника к передатчику. После завершения операции приемник должен подтвердить успешное выполнение задачи. Эта процедура носит название «подтверждение установления связи».

В процессе развития технологий сложность цифровых схем многократно возрастала. Поэтому процесс верификации асинхронных схем стал очень сложной задачей, требующей много сил и времени. Тогда разработчики вынуждены были искать новые пути решения для организации цифровых систем и обратили внимание на синхронный метод проектирования, который основывается на хранении состояний схемы в регистрах, а изменение данных происходит в определенные дискретные моменты времени. Подобный принцип детерминированной

[1] https://github.com/RomeoMe5/DDLM.

работы цифровых систем стал широко распространенным, и в настоящее время почти все современные цифровые устройства являются синхронными.

Комбинационные схемы используются для реализации относительно простых электрических схем, которые выполняют несложные вычисления. Выходы таких схем однозначно определены их входами. Чтобы схема была более сложной, например выполняла последовательные операции, она должна иметь определенную форму памяти, называемую состоянием. Схемы с состоянием называются последовательностными (секвенциональными), поскольку они определяют порядок выполнения состояний. Информация о состояниях обычно хранится в регистрах. Также существуют последовательностные схемы, которые для хранения используют другие элементы, такие как защелки, но их применяют редко и используют в особых случаях.

6.1.1 Модель Хаффмана для последовательностных устройств

Модель Хаффмана – это удобная абстракция для описания последовательностных схем и преобразования их в **HDL**-код. Она состоит из четырех составных частей: входы, выходы, комбинационная логика и элементы хранения состояний, реализованные при помощи **D-триггеров** (рис. 6.1).

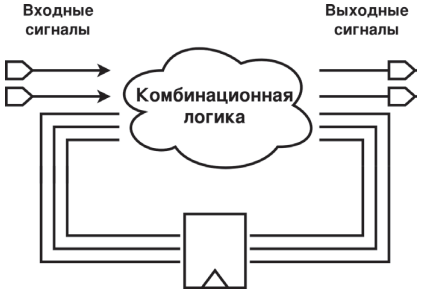


Рис. 6.1 Модель Хаффмана

Эту структуру можно описать с помощью **HDL**-кода. Достаточно всего лишь два блока **always**, один из которых реализует комбинационную логику, а второй – хранит и запоминает состояние системы.

```
module huffman_model
(
   input clock,
   input rst_n,
   input [...] i,
   output [...] o
);
   reg [...]    d,q;

   always@*
      d = comb_func(i,q);

   always@(posedge clock or negedge rst_n)
```

```
   begin
      if(!rst_n)
         q <= some_initial_state;
      else
         q <= d;
   end
endmodule // huffman_model
```

Листинг 6.1 Принципиальная модель Хаффмана

В приведенном примере в блоках **always** используются два разных типа присвоений: блокирующее присвоение, которое обозначается оператором =, и неблокирующее присвоение, которое обозначается <=. Ранее, в главе 1, мы уже использовали непрерывное присвоение = с оператором **assign** и при прямом объявлении переменной типа wire.

Для простоты понимания различия между присвоениями можно описать так: блокирующие присвоения (=) вступают в силу немедленно, а неблокирующие присвоения (<=) откладывают назначения на более поздний срок. Блокирующее присвоение используется в комбинационных блоках **always**, чтобы описать комбинационную логику или избежать появления защелок после проведения операций синтеза. Использование же неблокирующих присвоений в тактируемых блоках **always** необходимо, чтобы избежать проблем, связанных с возможностью появления **гонок сигналов** (r**ace condition**). В начале 90-х годов эти правила легли в основу методологии межрегистровых передач – **Register Transfer Level** (**RTL**). Прежде чем сформулировать эти правила, продемонстрируем разницу между различными типами присвоений и рассмотрим принципы работы ядра симулятора на примере симуляции кода на **Verilog**.

6.2 Последовательностные присвоения

6.2.1 Блокирующие и неблокирующие присвоения

В **Verilog** существует **два** типа процедурных присвоений: блокирующие и неблокирующие.

Блокирующее присвоение выглядит следующим образом:

variable = { [delay] | [event] } expression.

Операция присвоения состоит из нескольких вспомогательных процедур:

1) вычисляется значение выражения **expression**;
2) результат вычисления присваивается переменной **variable**;
3) присвоение может выполняться с временной задержкой **delay** или при возникновении определенного события **event**.

Блокирующее присвоение происходит строго в порядке объявления. Выполнение последующего присвоения не начинается, если не завершилась процедура вы-

числения текущего процесса. Такое поведение схоже с объявлением переменных в языках программирования **C**, **C++**, **Python** и др. В разделе 6.5 этот процесс рассмотрен более детально.

Неблокирующее присвоение имеет вид:

variable <= { [delay] | [event] } expression.

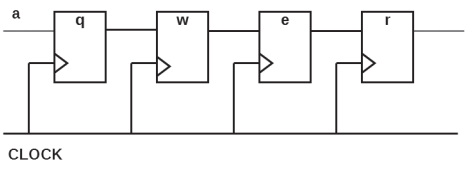
Процесс неблокирующего присвоения выполняется иначе:

1) происходит вычисление значения, находящегося справа от оператора;
2) результат вычисления сохраняется во внутренней переменной;
3) симулятор планирует выполнить присвоение в определенный момент системного времени;
4) если необходимо выполнить несколько неблокирующих присвоений, вычисление значения правой части одного выражения не будет блокировать вычисление другого;
5) неблокирующее присвоение в блоке **begin-end** выполняется независимо от порядка расположения операций.

6.2.2 Порядок присвоений

Рассмотрим пример, демонстрирующий, как порядок присвоений может повлиять на результат синтеза (рис. 6.2).

```
always@(posedge clock)
   begin
      r = e;
      e = w;
      w = q;
      q = a;
   end
```




```
always@(posedge clock)
   begin
      q = a;
      w = q;
      e = w;
      r = e;
   end
```

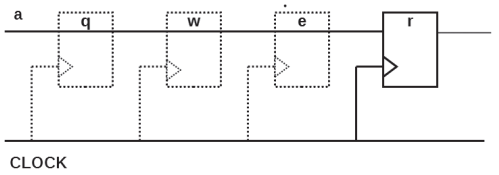


Рис. 6.2 Результаты синтеза в зависимости от порядка блокирующих присвоений

Получившийся результат наглядно демонстрирует, что порядок присвоений в блоке **always** значительно изменяет функционал схемы. Если требуется избегать ситуаций неопределенности и не зависеть от порядка присвоений, в приведенном примере необходимо использовать неблокирующие присвоения. Ниже представлены две разные части кода, которые используют неблокирующие присвоения. Результатом синтеза является идентичная принципиальная схема (рис. 6.3).

```
always@(posedge clock)
   r <= e;
always@(posedge clock)
   q <= a;
always@(posedge clock)
   w <= q;
always@(posedge clock)
   e <= w;
```

```
always@(posedge clock)
   begin
      q <= a;
      w <= q;
      e <= w;
      r <= e;
   end
```

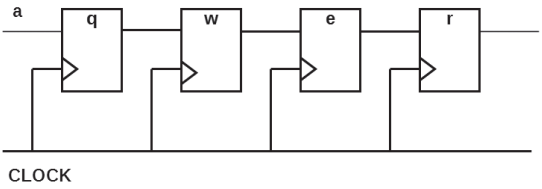


Рис. 6.3 Результаты синтеза в зависимости от порядка неблокирующих присвоений

6.2.3 Сравнение двух типов присвоения

Рассмотрим пример:

```
module blocking_assign;
   reg op1;
   reg op2;
   reg [7:0] inc_cmd;
   reg [1:0] rev_cmd;
   real acl;
   integer cnt;

   initial begin
      $display("==========Let's start %m module==========\n");
      $monitor("@%0t ns\t op1=1'b%b\t op2=1'b%b\t inc_cmd=0x%h\t rev_
         cmd=0x%h\t acl=%f\t cnt=%0d\n", $time,op1,op2,inc_cmd,
         rev_cmd,acl,cnt);
   end
      initial begin
         op1 = 1'b1;
         op2 = 1'b0;
         cnt = 0;
         inc_cmd = 8'hFF;
         rev_cmd = inc_cmd;
         acl = 2.28;
         inc_cmd[3:0] = #10 4'b1001;
         rev_cmd[1] = #11 1'b0;
         acl = #12 1.2;
         cnt = cnt + 3;
         op1 = 1'b0;
```

```
        op2 = 1'b1;
    end
endmodule // blocking_assign
```

Листинг 6.2 Пример использования блокирующих присвоений

Проанализируем, как будет выполняться моделирование схемы, описанной в листинге 6.2. Для начала необходимо определить значения переменных в нулевой момент времени. Для этого запустим симуляцию данного модуля без графического интерфейса. Все присвоения выполняются последовательно в порядке, указанном в примере. Величина условных единиц может быть установлена при помощи директивы симулятора **`timescale**. Данные действия позволяют отследить моменты времени, в которые происходят присвоения. В момент времени **10** изменяется значение регистра **inc_cmd[3:0]**. Затем, по прошествии **11** временных единиц, изменяется значение старшего бита регистра **rev_cmd**. С момента начала симуляции прошло время, равное **21** единице. По прошествии еще **12** единиц времени, в момент, когда с начала симуляции прошло время, равное **33**, происходит присвоение сразу нескольких переменных: **acl**, **cnt**, **op1**, **op2**. Результаты симуляции сведены в таблицу:

Таблица 6.1 Результаты симуляции модуля blocking_assign

Временные единицы	Переменные					
	op1	op2	inc_cmd	rev_cmd	acl	cnt
0	1'b1	1'b0	8'hFF	2'b10	2.28	0
10	1'b1	1'b0	8'hF9	2'b10	2.28	0
21	1'b1	1'b0	8'hF9	2'b01	2.28	0
33	1'b0	1'b1	8'hF9	2'b01	1.20	3

Внесем изменения в рассматриваемый пример. Заменим все операции блокирующего присвоения на операции неблокирующего присвоения и запустим симуляцию модуля заново. Часть примера, которая была изменена:

```
// Предыдущий код ...
   inc_cmd[3:0] <= #10 4'b1001;
   rev_cmd[1] <= #11 1'b0;
   acl <= #12 1.2;
   cnt <= cnt + 3;
   op1 <= 1'b0;
   op2 <= 1'b1;
```

Листинг 6.3 Изменения в коде из листинга 6.2

Значения переменных в начальный момент времени остаются такими же, как в предыдущем примере. Вычислим значения всех выражений, расположенных справа от оператора присвоения. Во время работы симулятор планирует выполнить каждое присвоение в определенный момент времени, и лишь только затем

происходит присвоение значения левой части выражения. После запуска модуля на симуляцию получены следующие результаты:

Таблица 6.2 Результаты симуляции модуля nonblocking_assign

Временные единицы	Переменные					
	op1	**op2**	**inc_cmd**	**rev_cmd**	**acl**	**cnt**
0	1'b0	1'b1	8'hFF	2'b10	2.28	3
10	1'b0	1'b1	8'hF9	2'b10	2.28	3
11	1'b0	1'b1	8'hF9	2'b01	2.28	3
12	1'b0	1'b1	8'hF9	2'b01	1.20	3

Таким образом, по сравнению с предыдущим примером время присвоения переменных изменилось. Но можно заметить, что значения регистров **op1**, **op2**, **cnt** остались без изменений. Данный факт объясняется далее.

6.3 Циклы симуляции

Чтобы правильно понимать, как работают различные типы присвоений и как их выполняет **Verilog**-симулятор, следует обратиться к разделу «**The stratified event queue**» стандарта **IEEE Verilog Standart**[1]. В нем **Verilog** определяется с точки зрения дискретно-событийной модели выполнения. Эта модель выполняет набор определенных процессов, из которых состоит проект. К ним относятся **примитивы**, **initial**- и **always**-блоки, непрерывные присвоения, асинхронные функции и присвоения в процедурных блоках. Основная функция этих процессов заключается в том, что они могут реагировать на изменения своих входных данных для получения выходных.

С точки зрения симулятора все события, которые происходят в схеме, можно разделить на два типа: события обновления и события оценки. **Событие обновления** включает в себя любые изменения значений переменных или уровня напряжения в цепи. **Событие оценки** происходит, если событие обновления выполняется и входит в список чувствительности процесса. Выполнение этих событий обычно происходит в различные моменты времени. Чтобы точно следовать установленному порядку, все события помещаются в специальную очередь событий. Ядро симулятора выполняет процессы в свое системное время, что отличается от времени, которое можно наблюдать на циклограмме в **САПР**. Очередь событий можно условно разделить на несколько основных групп:

- **активная очередь событий**. В ней выполняются: непрерывные присвоения, блокирующие присвоения, вычисление правых частей неблокирующих присвоений, системные функции $display, обновление выходных данных примитивов. Все события в этой очереди происходят в произвольном порядке;

[1] 1364–2005 – IEEE Standard for Verilog Hardware Description Language in IEEE Standards. IEEE, 2006. DOI: 10.1109/IEEESTD.2006.99495.

- **неактивная очередь событий**. В данной группе выполняются присвоения с временно́й задержкой **#0**. Использование нулевых задержек может служить решением проблем, связанных с возникновением гонок сигналов, например между модулем и тестом. В настоящее время такой подход является устаревшим и не рекомендуется к использованию, поскольку может приводить к возникновению конфликтов синхронизации между модулями проекта;
- **неблокирующая очередь событий**. В ней происходит обновление значений левой части неблокирующих присвоений;
- **наблюдающая очередь событий**. Служит для обновления всех переменных. В ней также выполняются системные функции **$monitor** и **$strobe**.

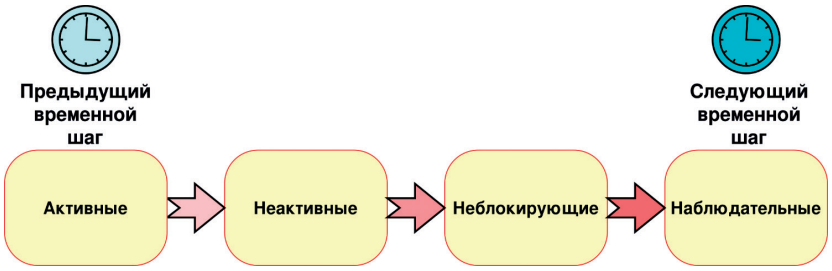


Рис. 6.4 Порядок событий во время симуляции

Вернемся к рассматриваемому примеру (табл. 6.2) и проанализируем состояния регистра **op1**. При запуске симуляции в нулевой момент времени выполняются блокирующее присвоение (**op1 = 1'b1**) и вычисление значения правой части неблокирующего присвоения (**op1 <= 1'b0**). Затем происходит обновление значения левой части неблокирующего присвоения в группе. Конечным значением регистра **op1** является результат выполнения неблокирующего присвоения, поскольку данное изменение переменной происходило позднее в очереди событий. Аналогичные рассуждения можно провести для регистров **op2** и **cnt**.

6.4 Порядок правильного использования блокирующих и неблокирующих присвоений

Рассмотрим следующий пример:

```
module comb_nblk
(
   input [7:0] q,w,e,
   output reg [8:0] result
);
   reg [8:0] qw_sum,qe_sum;

   always@(q or w or e or qw_sum
      or qe_sum)
```

```
module comb_blk
(
   input [7:0] q,w,e,
   output reg [8:0] result
);
   reg [8:0] qw_sum,qe_sum;

   always@(q or w or e)
```

```
      begin
         qw_sum <= q + w;
         qe_sum <= q + e;
         result <= qw_sum | qe_sum;
      end
endmodule // comb_nblk

module comb_nblk
(
   input [7:0] q,w,e,
   output reg [8:0] result
);
   reg [8:0] qw_sum,qe_sum;

   always@(q or w or e or qw_sum
      or qe_sum)
      begin
         qw_sum <= q + w;
         qe_sum <= q + e;
         result <= qw_sum | qe_sum;
      end
endmodule // comb_nblk
```

```
      begin
         qw_sum = q + w;
         qe_sum = q + e;
         result = qw_sum | qe_sum;
      end
endmodule // comb_blk

module comb_blk
(
   input [7:0] q,w,e,
   output reg [8:0] result
);
   reg [8:0] qw_sum,qe_sum;

   always@(q or w or e)
      begin
         qw_sum = q + w;
         qe_sum = q + e;
         result = qw_sum | qe_sum;
      end
endmodule // comb_blk
```

Листинг 6.4 Два сумматора и логический элемент OR

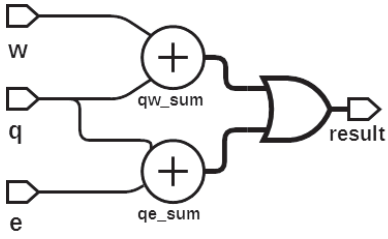


Рис. 6.5 Принципиальная схема из двух сумматоров и логического элемента OR

Эти модули реализуют одинаковую принципиальную схему: два сумматора и логический элемент **ИЛИ** (рис. 6.5). Результаты симуляции и синтеза таких реализаций одинаковы. Отличие между двумя подходами состоит во времени симуляции. Рассмотрим причины этого явления.

При использовании блокирующих присвоений переменные **qw_sum** и **qe_sum** являются временными, поэтому все действия цикла симулятор выполняет во время «активной» очереди событий. Но если использовать неблокирующее присвоение, для сохранения функциональной модели необходимо добавить **qw_sum** и **qe_sum** в список чувствительности блока **always**. Это важно сделать, для того чтобы после обновления левой части выражения блок **always** сработал снова и установил правильное значение переменной на выходе. В итоге симулятор тратит больше времени и ресурсов для обработки информации. Тем самым снижается скорость работы инструмента симуляции.

6.4.1 Защелки (latch), и почему их рекомендуется избегать

При работе с **ИС** и **ПЛИС** не рекомендуется использовать защелки. Причина данного ограничения следующая: это приводит к появлению трудностей при проведении верификации, временного анализа и при использовании технологий тестопригодного проектирования. Кроме того, многие технологические библиотеки не содержат элементов защелок. К тому же анализ цепей с защелками с помощью инструментов **САПР** превратится в емкий и кропотливый процесс.

Рассмотрим следующие примеры:

```
module if_else_latch
(
   input en,q,w,
   output reg result
);
   always@*
      if(en)
         result = q & w;
endmodule // if_else_latch
```

```
module if_else_latch_fix
(
   input en,q,w,
   output reg result
);
   always@*
      begin
         // Another method
         // result = q | w;
         if(en)
            result = q & w;
         else
            result = q | w;
      end
endmodule // if_else_latch_fix
```

Листинг 6.5 Возникновение защелок при использовании конструкции if-else

```
module case_latch
(
   input [1:0] sel,
   input q,w,
   output reg result
);
   always@*
      begin
         case(sel)
            2'b00: result = q | w;
            2'b01: result = q & w;
            2'b10: result = q ^ w;
         endcase
      end
endmodule // case_latch
```

```
module case_latch_fix
(
   input [1:0] sel,
   input q,w,
   output reg result
);
   always@*
      begin
         // Another method
         // result = q & ~w;
         case(sel)
            2'b00: result = q | w;
            2'b01: result = q & w;
            2'b10: result = q ^ w;
            default: result = q & ~w;
         endcase
      end
endmodule // case_latch_fix
```

Листинг 6.6 Возникновение защелок при использовании оператора case

Слева размещаются примеры кода, которые приводят к возникновению защелок. Справа приведены правильные реализации той же логики, но без возникновения защелок. При необходимости описать комбинационную логику внутри блока **always** следует указывать все возможные состояния выходных сигналов. В примере, где используется конструкция **if-else**, можно сначала объявить значение переменной по умолчанию, а после изменять ее. В случае использования оператора **case** следует указать значение по умолчанию с помощью зарезервированного слова **default** или объявить это значение заранее, как показано в примере. Также следует внимательно анализировать системные сообщения инструмента синтеза: возникновение ошибки типа «**Inferring latch**» (добавлена защелка) говорит о том, что в списке цепей появилась защелка.

6.4.2 Комбинационная и последовательностная логика

В данном разделе дается объяснение, почему необходимо разделять блоки **always** с комбинационной и последовательностной логикой и почему в комбинационном блоке используют блокирующие присвоения, а в другом – неблокирующие. Рассмотрим следующий пример:

```
module one_dff
(
   input q,w,e,
   input clock,
   output reg result
);
   reg tmp;

   always@(posedge clock)
   begin
      tmp = q & w;
      if(e)
         tmp = q | w;
      result <= tmp;
   end
endmodule // one_dff
```

```
module unwanted_dff
(
   input q,w,e,
   input clock,
   output reg result
);
   reg tmp;

   always@(posedge clock)
      begin
         if(e)
            tmp = q | w;
         result <= tmp;
      end
endmodule // unwanted_dff
```

Листинг 6.7 Появление «неявного» триггера

Допустим, необходимо использовать временную переменную и не сохранять ее значение между фронтами тактового сигнала. Полученная в результате электрическая схема будет представлять собой **один D-триггер** и комбинационную логику на его входе. Если же во время разработки **RTL**-модели не указать значение **tmp** по умолчанию, в итоге инструмент синтеза создаст **два D-триггера**. При этом синтезатор не сообщит об ошибке. На заключительном этапе маршрута проектирования наш проект попадет на фабрику для изготовления. Полученная **ИС** может стать основой для управления критических узлов в автоматизированных системах. Масштаб последствий может быть огромным: от финансовых затрат на изготовление нового образца микросхемы до техногенных катастроф.

Избежать подобных проблем довольно просто: необходимо разделять логику на **комбинационную** и **последовательностную** и записывать ее в отдельные блоки **always** (листинг 6.8).

```
module two_blocks
(
   input q,w,e,
   input clock,
   output reg result
);
   reg tmp;

   always@*
   begin
      tmp = q & w;
      if(e)
         tmp = q | w;
   end

   always@(posedge clock)
      result <= tmp;
endmodule // two_blocks
```

Листинг 6.8 Разбиение на последовательностную и комбинационную логику

Кроме того, имена реальных и временных переменных можно разделять при помощи префиксов, например **r_state** и **state**. Рассмотрим, почему в **always**-блоке, который тактируется по фронту сигнала, используются неблокирующие присвоения.

```
// Bad example
always@(posedge clock)
   mac = f(mac);
always@(posedge clock)
   res = mac;
```

```
// Good example
always@(posedge clock)
   mac <= f(mac);
always@(posedge clock)
   res <= mac;
```

Листнг 6.9 Возникновение «гонок» сигналов

Как было указано ранее, в зависимости от особенностей симулятора порядок выполнения блокирующих присвоений может отличаться. В приведенном примере подобная ситуация может привести к состоянию неоднозначности. Такое состояние в **Verilog** называется **состоянием гонки**. В итоге результаты до и после синтеза могут кардинально отличаться.

Решение данной проблемы – использование неблокирующего присвоения. Значение переменной **mac** будет вычислено в момент времени перед фронтом тактового сигнала, а значит, все процессы, которые используют эту переменную, однозначно определены.

6.5 Счетчики

Счетчики являются одними из базовых элементов при проектировании цифровых схем. Счетчик последовательно увеличивает или уменьшает значение в момент, когда происходит событие из списка чувствительности.

6.5.1 Простой счетчик с асинхронным сбросом

Рассмотрим пример простого счетчика:

```
module simple_counter
#( parameter WIDTH=8 )
(
   input clk,
   input rst_n,
   output reg [WIDTH-1:0] cnt);

   always@(posedge clk or negedge rst_n)
   begin
      if(!rst_n)
         cnt <= {WIDTH{1`b0}};
      else
         cnt <= cnt + 1`b1;
   end
endmodule // simple_counter
```

Листинг 6.10 8-битный счетчик с асинхронным сбросом

Для хранения значений счетчика в рассматриваемом примере используется **8**-битный регистр. Следовательно, счет идет в диапазоне значений от **0** до **255** (в формате **unsigned**). Данный счетчик управляется с помощью сигнала асинхронного сброса (**rst_n**). Как только значение счетчика достигает максимума, при следующей операции инкремента оно устанавливается в **0**. Запустив симуляцию модуля, можно убедиться в функциональной корректности его работы:

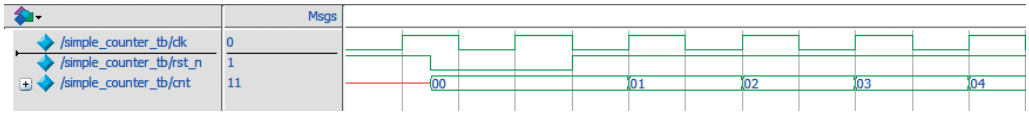


Рис. 6.6 Результаты моделирования модуля simple_counter

RTL-представление рассматриваемого счетчика в **Quartus Prime** на этапе предварительного синтеза имеет следующий вид:

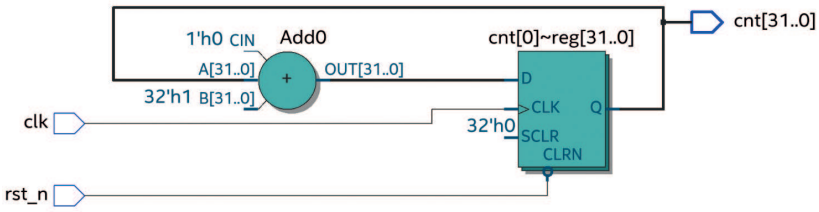


Рис. 6.7 Результаты предварительного синтеза в САПР Quartus Prime

Часто при разработке сложных систем на кристалле или прототипировании на **ПЛИС** требуется использовать готовые функциональные блоки, поскольку на разработку и верификацию с нуля простых функциональных блоков уходит большое количество времени и сил. Чтобы сократить эти издержки, обычно применяют уже готовые **сложнофункциональные** (**СФ**) **блоки**, или иначе **мегафункции** (рис. 6.8).

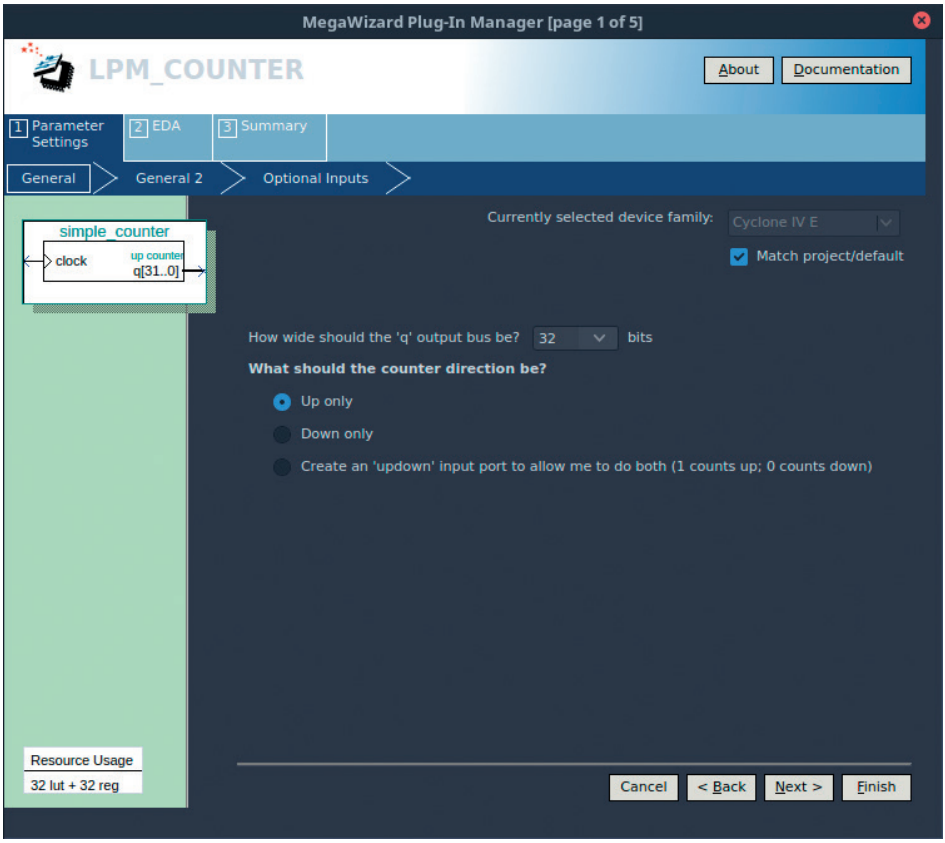
Условно их можно разделить на **две** категории: программные и аппаратные. Программные **СФ**-блоки – это **RTL**-модель блока или уже готовый список цепей для выбранного технологического процесса. Также сюда можно отнести **СФ-блоки** для верификации. Готовые к производству блоки, например, **GDSII**-формата являются аппаратными. Часто дистрибьюторы **САПР** включают в свои продукты некоторые простые **СФ**-блоки для свободного применения.

Для **ПЛИС** тоже существуют собственные **СФ-блоки**. В **САПР Quartus Prime** блок простого счетчика называется **LPM_COUNTER**.

Использование встроенных **СФ-блоков** позволяет оптимизировать работу на этапах синтеза, трассировки, размещения и оптимизации ресурсов. Но их применение не гарантирует полного преимущества над специализированным проектированием. Часто поведенческое описание, которое реализует специальный алгоритм, работает быстрее и потребляет меньше ресурсов, чем схема, состоящая из готовых блоков. Помимо этого, инструмент синтеза может объединять эквивалентные логические узлы воедино для сокращения количества элементов. Подобные оптимизации позволяют сократить количество логических элементов, тем самым уменьшить количество потребляемой мощности при проектировании микросхем.

Рассмотрим приведенные доводы на примере усложнения арифметического блока путем добавления в счетчик следующих функций:

- возможность инициализации начального значения счетчика;
- возможность выбрать направление счета.


MegaWizard Plug-In Manager [page 1 of 5]
LPM_COUNTER
About
Documentation
1 Parameter Settings
2 EDA
3 Summary
General
General 2
Optional Inputs
simple_counter
clock
up counter
q[31..0]
Currently selected device family:
Cyclone IV E
Match project/default
How wide should the 'q' output bus be?
32
bits
What should the counter direction be?
Up only
Down only
Create an 'updown' input port to allow me to do both (1 counts up; 0 counts down)
Resource Usage
32 lut + 32 reg
Cancel
< Back
Next >
Finish

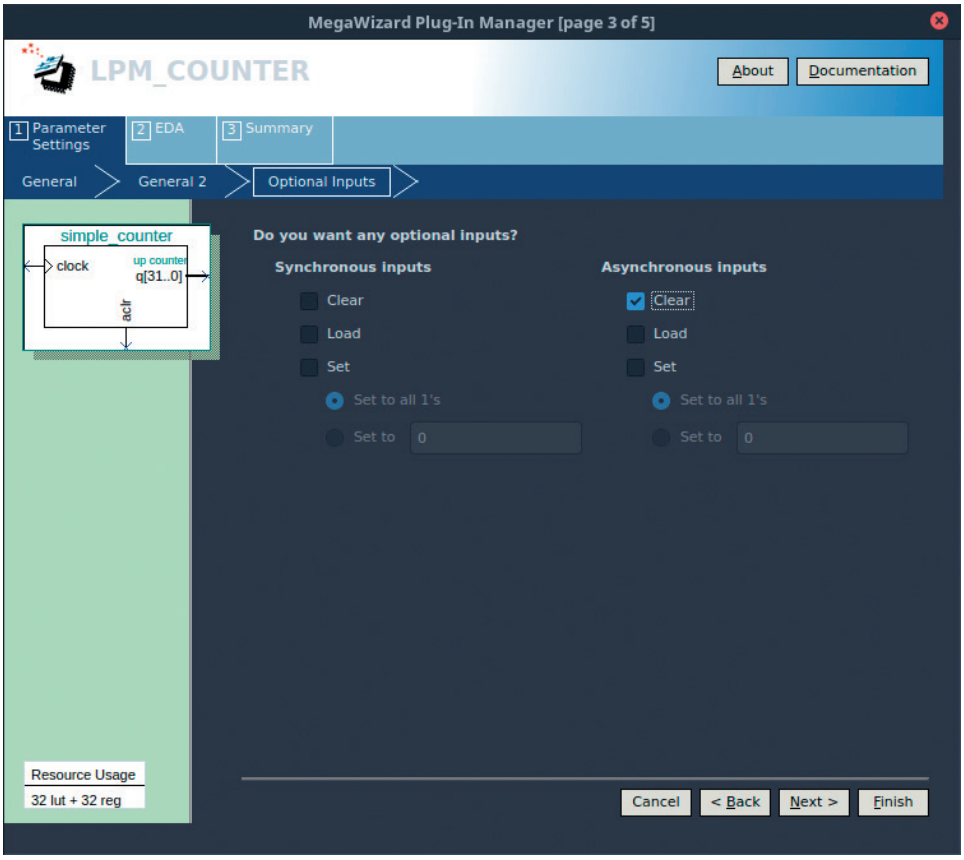


Рис. 6.8 Создание счетчика средствами САПР Quartus Prime

6.5.2 Счетчик с предустановкой

Счетчик с предустановкой (с загрузкой) имеет дополнительные входные сигналы: **load** – сигнал разрешения установки значения, **data_load** – входные данные. Если сигнал **load** активен, то значение счетчика устанавливается сигналом **data_load**. Таким образом, можно установить требуемое начальное значение для дальнейшей работы счетчика.

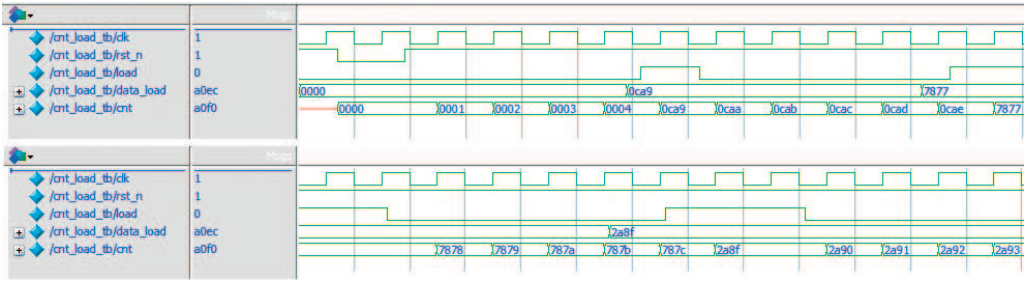


Рис. 6.9 Результаты моделирования с предустановкой

Дополнительное задание для самостоятельной работы

Разработайте **RTL**-модель и тест для рассмотренного счетчика. При возникновении сложностей обратитесь к дополнительным материалам к данной главе. Сравните полученные результаты.

6.5.3 Счетчик с управляемым направлением счета

Счетчик с управляемым направлением счета использует дополнительный входной сигнал **up_down** (или два отдельных сигнала **up** и **down**). Если сигнал **up_down** имеет высокий логический уровень, происходит последовательное увеличение счетчика, в противном случае – уменьшение.

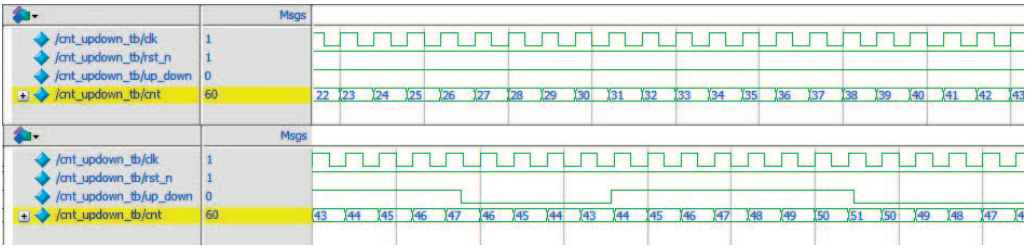


Рис. 6.10 Результаты моделирования счетчика с управляемым направлением счета

Дополнительное задание для самостоятельной работы

Разработайте **RTL**-модель и тест для рассмотренного счетчика. При возникновении сложностей обратитесь к дополнительным материалам к данной главе. Сравните полученные результаты.

6.5.4 Делитель частоты

Счетчики можно использовать для уменьшения частоты входного тактового сигнала. Рассмотрим пример:

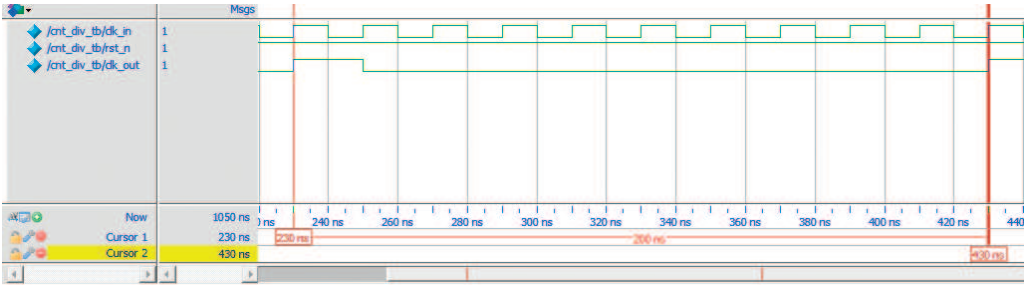


Рис. 6.11 Результаты моделирования делителя частоты

На приведенном рисунке основная тактовая частота делится на **10**. Величину деления можно изменять с помощью параметра **DIV_CNT**.

Полученный сигнал нельзя напрямую использовать для тактирования других модулей или внешних устройств, поскольку из-за использования комбинационной

логики на выходе будет возникать дополнительная задержка, что может привести к нарушению правил синхронного проектирования. Если же все-таки необходимо создать тактовый сигнал с помощью счетчика, то следует:

- добавить ограничения для инструмента временного анализа;
- добавить в проект специальный модуль **BUFG** для **ПЛИС** фирмы **Xilinx** или **global** для **ПЛИС** фирмы **Intel FPGA**. Эти модули используют ресурсы глобальных цепей тактирования, поэтому время распространения сигнала в них меньше.

```
// Предыдущий код ….
global glob_net
(
   .in ( clock_slow ),
   .out ( clock_glob )
);
```

Листинг 6.11 Пример использования модуля global

Для того чтобы включить **clock_slow** в список глобальных сигналов, на вход буфера необходимо подать искомый сигнал, а на выходе получить сигнал **clock_glob**, который будет использоваться для тактирования внутренних модулей в **ПЛИС**. Количество используемых буферов строго ограничено и зависит от конкретной модели чипа.

Помимо способа, приведенного в примере (листинг 6.11), для использования глобальных ресурсов **ПЛИС** можно задать настройки в **Assignment Editor**. В поле **To** устанавливается сигнал для подключения, в **Assignment Name** – параметр **Global Signal**, в поле **Value** – значение **Global Clock**, а значение поля **Enable** – **Yes**. Чтобы убедиться в успешности работы в **Assignment Editor**, необходимо запустить сборку проекта повторно и в качестве проверки обратиться к вкладке **Fitter** в меню отчетов. В каталоге **Resource Section** будет находиться отчет **Global&Other Fast Signals**, в котором появится глобальный сигнал.

Приведенный выше подход позволяет избежать множества проблем, связанных со статическим временным анализом. Пример использования модуля **global** можно также встретить в дополнительных материалах к разделу 6.7.

Дополнительное задание для самостоятельной работы

Проведите симуляцию модуля **cnt_div**, файлы с описанием которого находятся в дополнительных материалах[1] к данной практической работе. Определите частоту выходного сигнала **clk_out**. Измените код так, чтобы коэффициент заполнения сигнала составлял **20 %**, **50 %**. Опишите полученные результаты.

6.5.5 Широтно-импульсная модуляция

Широтно-импульсная модуляция (**ШИМ**) используется для регулирования подаваемой к нагрузке мощности путем изменения времени полезной работы сиг-

[1] https://github.com/RomeoMe5/DDLM.

нала. Это простой процесс, который может быть реализован с помощью последовательной логики. Например, если нужно отрегулировать уровень яркости светодиодной лампы, проконтролировать скорость шаговых двигателей или использовать сервоприводы, самый простой способ сделать это – с помощью **ШИМ**.

Разработаем регулятор **ШИМ**. Для этого необходимо установить длительность выходного сигнала. Его значение в условных единицах будет зависеть от тактовой частоты. Затем запускается простой счетчик и сравнивается текущее значение счетчика с установленным значением длительности импульса. В зависимости от результата на выходе модуля устанавливается соответствующий логический уровень сигнала.

Дополнительное задание для самостоятельной работы

Проанализируйте работу модуля **pwm** из дополнительных материалов[1] к данной практической работе. Определите, в каких случаях светодиод будет светить ярче. Определите коэффициент заполнения выходного сигнала **pwm_out** для каждого случая. Проанализируйте временные диаграммы на рис. 6.12. Разработайте собственный модуль **ШИМ**. Сравните полученные результаты.

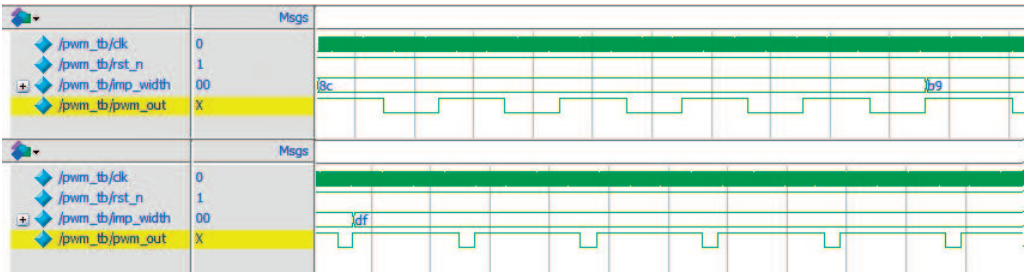


Рис. 6.12 Результаты моделирования ШИМ-регулятора

6.5.6 Счетчик Грея

Код Грея, названный в честь **Франка Грея**, предназначен для защиты информации от ошибок в электромеханических переключателях или датчиках угла поворота. Наиболее распространенным является двоичный **код Грея**, хотя его применение возможно и для других систем счисления. По сравнению с бинарным представлением чисел, значения соседних чисел в коде Грея отличаются всего на **один** разряд. Таким образом, при последовательной передаче значений исключается вероятность приема заведомо ложной информации. Помимо этого, **код Грея** используется при работе с указателями чтения и записи в асинхронных **FIFO**.

Алгоритм преобразования двоичного числа в **код Грея** очень прост. Необходимо сложить по **модулю 2** исходное число и то же самое число, но уже сдвинутое на **один** разряд вправо. Для выполнения обратной операции значение, представленное в коде Грея, необходимо сложить по **модулю 2** последовательно со всеми числами, которые получаются при сдвиге вправо на один разряд, пока слагаемое не станет равно **0**.

[1] https://github.com/RomeoMe5/DDLM.

Пример работы **счетчика Грея** приведен на рис. 6.13. Каждое соседнее значение счетчика отличается друг от друга всего лишь на один разряд.

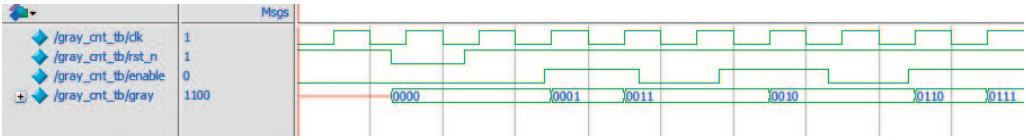


Рис. 6.13 Результаты моделирования 4-битного счетчика Грея

Дополнительное задание для самостоятельной работы

Разработайте **RTL**-модель и тест для **счетчика Грея**. При возникновении сложностей обратитесь к дополнительным материалам к данной главе. Проанализируйте полученные результаты.

6.6 Сдвиговые регистры

Сдвиговый регистр – это один из основных элементов последовательной логики, который состоит из **D-триггеров**. Сдвиговые регистры отличаются по типу чтения и записи входных или выходных данных:

- последовательностный вход – последовательностный выход;
- последовательностный вход – параллельный выход;
- параллельный вход – последовательностный выход;
- параллельный вход – параллельный выход;
- другие комбинации.

Наиболее распространенные задачи, которые решают с помощью сдвигового регистра:

- преобразование потока последовательной информации в параллельный код, и наоборот;
- обнаружение маскированной последовательности.

На рис. 6.14 приведена общая схема реализации сдвигового регистра.

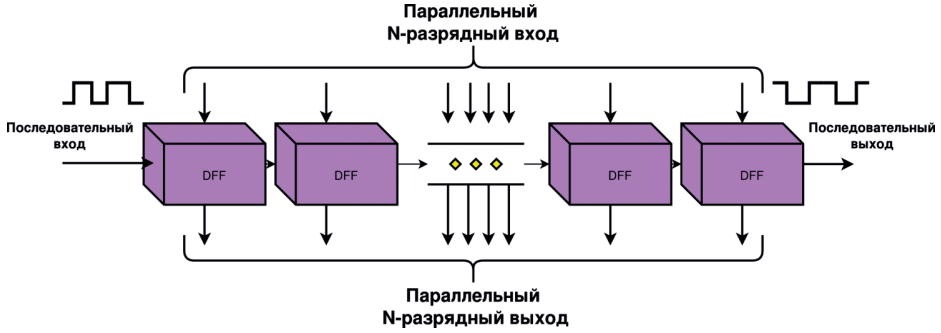


Рис. 6.14 N-битный сдвиговый регистр

Сдвиговые регистры широко используются в цифровой электронике, в том числе, например, в таких протоколах связи, как **SPI**, **UART**, **I2C** и др.

6.6.1 Сдвиговый регистр с сигналом разрешения

Рассмотрим пример работы **8**-битного сдвигового регистра с последовательностным вводом данных и сигналом разрешения сдвига (рис. 6.15). До момента появления сигнала разрешения (**shift_en**) в регистре хранилось значение **0x1A**. После того как сигнал разрешения стал активен (**1**), происходит синхронный сдвиг значений регистра вправо. Значение, которое было вытеснено из регистра, является сигналом последовательного выхода (**serial_out**). Все это происходит до тех пор, пока сигнал разрешения сдвига не станет неактивен (**0**).

Дополнительное задание для самостоятельной работы

Разработайте **RTL**-модель и тест сдвигового регистра, который бы соответствовал приведенному примеру. При возникновении сложностей обратитесь к дополнительным материалам к данной главе. Проанализируйте полученные результаты.

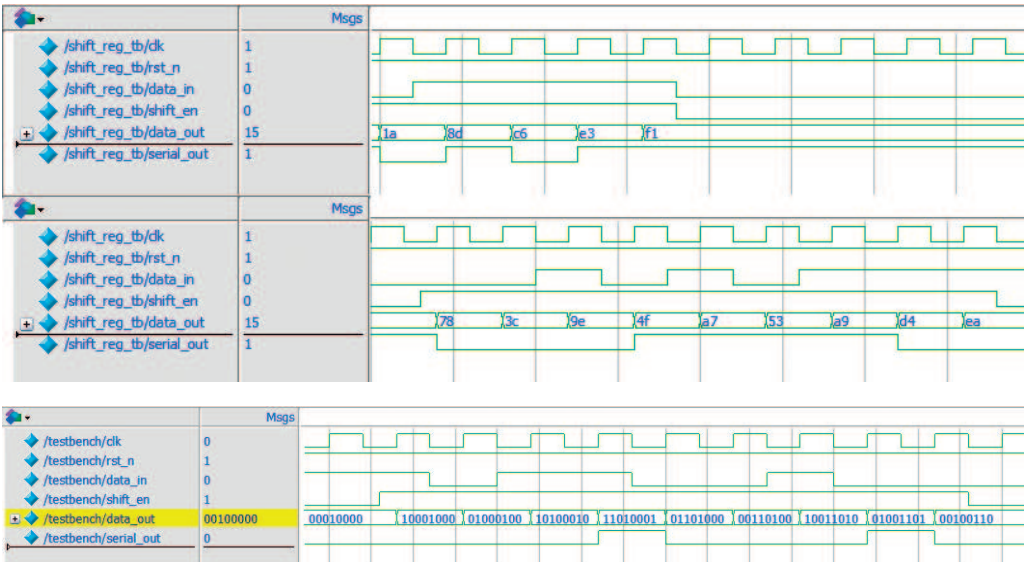


Рис. 6.15 Результаты моделирования сдвигового регистра с разрешением сдвига

В **Quartus Prime** есть **СФ-блок**, который реализует сдвиговый регистр с различными параметрами конфигурации. Данный блок можно найти в каталоге **СФ-блоков** под названием **LPM_SHIFTREG**. Применение **СФ-блоков** позволяет провести улучшенную оптимизацию во время синтеза. Особенно это заметно, если проект занимает более 50 % ресурсов **ПЛИС** или если необходимо обеспечить, чтобы проект удовлетворил требованиям быстродействия либо энергопотребления. Использование готовых **СФ-блоков** от фирмы-производителя может решить проблему.

Сдвиговый регистр может также использовать **RAM**-блоки **ПЛИС**. Для этого в **Quartus Prime** существует специализированный **СФ-блок ALTSHIFT_TAPS**. Его применение в проекте может значительно сэкономить количество используемых ресурсов **ПЛИС**.

6.6.2 Регистр сдвига с линейной обратной связью

Другим распространенным типом регистра сдвига является **регистр сдвига с линейной обратной связью** (**РСЛОС**). Принцип его работы прост. На рис. 6.15 представлен пример **8**-битного **РСЛОС Фибоначчи**. Входной бит на входе регистра получается путем работы комбинационной схемы, обычно состоящей из элементов **исключающие ИЛИ**/**исключающие ИЛИ-НЕ**. Для того чтобы регистр работал корректно, начальное значение, записанное в регистр, не должно быть нулевым. Сдвиговый регистр с обратной связью часто используют для генерации псевдослучайных чисел.

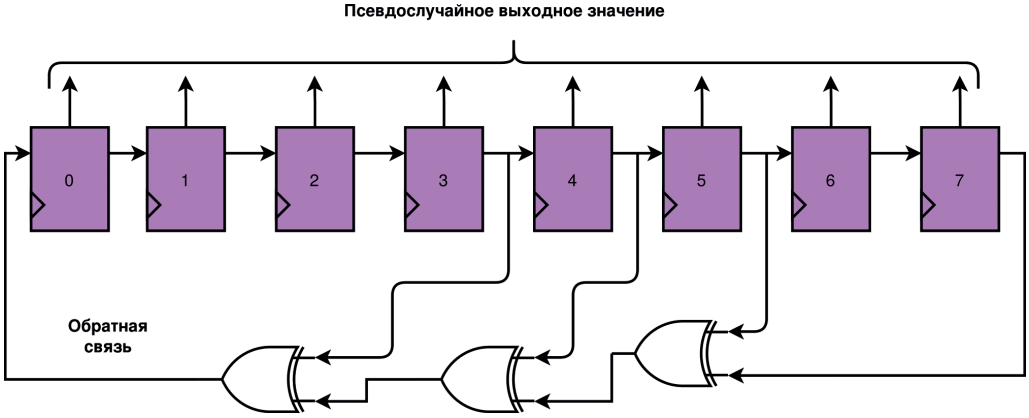


Рис. 6.16 РСЛОС Фибоначчи

Генератор псевдослучайных чисел используется в качестве одного из главных блоков алгоритмов шифрования и криптографии. Построение систем на основе **РСЛОС** стало широко распространенным из-за простоты реализации в аппаратуре. Последовательность битов на выходе генератора имеет линейную сложность, поэтому, применяя алгоритм **Берлекэмпа–Мэсси** по поиску кратчайшего **РСЛОС**, можно воссоздать исходный **РСЛОС**. Поэтому, чтобы увеличить сложность системы, используют объединение нескольких **РСЛОС** различной длины.

На основе **РСЛОС** построено большое количество разновидностей потоковых шифров. **Генератор Геффа**, **тактируемый генератор**, **пороговый генератор**, **суммирующий генератор** и **генератор с нелинейной функцией** – примеры криптографически устойчивых генераторов по сравнению с обычным генератором на **РСЛОС**.

Цифровые устройства используют различные протоколы для передачи информации по каналам связи. Одной из важнейших задач передачи является сохранение целостности данных. Корректирующие коды, которые используют для об-

наружения ошибок, позволяют нивелировать влияние сторонних воздействий. Передатчик перед отправкой добавляет дополнительный код в конец исходного сообщения. Приемное устройство после получения объединенного сообщения проверяет дополнительные биты по заранее известному алгоритму, который сигнализирует о целостности переданного сообщения.

6.6.3. Циклический избыточный код

Циклический избыточный код (**CRC**) – один из самых распространенных алгоритмов обнаружения ошибок. Он применяется в протоколах связи **Ethernet**, **USB**, **CAN**, **Modbus**. При расчете **CRC** используется полиномиальная арифметика по **модулю 2**. Операции сложения и вычитания представляют собой логическую функцию исключающего **ИЛИ**. Операции умножения и деления реализуются с помощью логического сдвига. Данная особенность **CRC** позволила аппаратной реализации алгоритма поиска ошибок стать распространенной в цифровых технологиях.

CRC можно найти из следующего выражения:

$$M(x) = G(x) \times Q(x) + R(x),$$

где $M(x)$ – полином, степени которого определяют биты передаваемой информации;

$G(x)$ – порождающий полином, используется в качестве делителя исходного сообщения $M(x)$. Степень полинома должна быть больше 0 и меньше степени $M(x)$;

$Q(x)$ – это частное от деления полинома $M(x)$ на $G(x)$;

$R(x)$ – это остаток от деления полиномов, двоичное представление которого и есть **CRC**. Степень полученного полинома не может быть больше m, где $m + 1$ – степень полинома $G(x)$.

Допустим, необходимо передать **1** байт данных (**0xA9** или **8'b10101001**) для аутентификации **USB**-токена. Сообщение будет передаваться через последовательностный интерфейс, начиная со старшего бита. В качестве порождающего полинома будем использовать $G(x) = x^5 + x^2 + 1$ (**0x25** или **6b'100101**). Алгоритм деления прост: если старший бит $M(x) = 1$, тогда выполняется операция **XOR** с $G(x)$, если же старший бит равен **0**, тогда $M(x)$ сдвигается вправо на **1** бит. Полученный остаток от деления – **0x18** (**5'b11000**). Это и есть **CRC**-код.

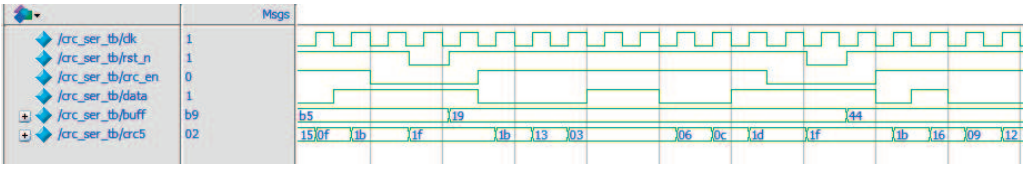


Рис. 6.17 Результаты моделирования модуля вычисления CRC для USB-токена

Подобный алгоритм можно реализовать с использованием **РСЛОС**. Но если необходимо обработать большой поток информации, процедура побитового вычисления займет слишком много времени. Поэтому, чтобы ускорить процесс поиска **CRC**, объем обрабатываемых данных за один период следует увеличить, но он не должен превышать значение степени $G(x)$. Пример реализации алгоритма **CRC-32**, используемого в **IEEE 802.3**, **MPEG-2**, **PNG**, **POSIX**, **RAR**, **ZIP**, приведен в приложении (раздел 6.8.4). Размер обрабатываемых данных за период – **1** байт.

Перед началом вычисления **CRC-32** необходимо установить все значения регистра сдвига в логическую единицу. В момент, когда сигнал разрешения активен, происходит побайтное вычисление контрольной суммы. Промежуточные результаты вычисления хранятся в исходном **РСЛОС**. После вычисления последнего байта в сообщении сдвиговый регистр содержит значение искомого **CRC**-кода.

Чтобы проверить результат вычисления, производится обработка данных по такому же алгоритму, только в конец исходного сообщения добавляется полученный ранее **CRC**-код (точнее, остаток к делителю). В результате в каждом разряде **РСЛОС** должен получиться **0** (рис. 6.18), поскольку осуществляется деление без остатка.

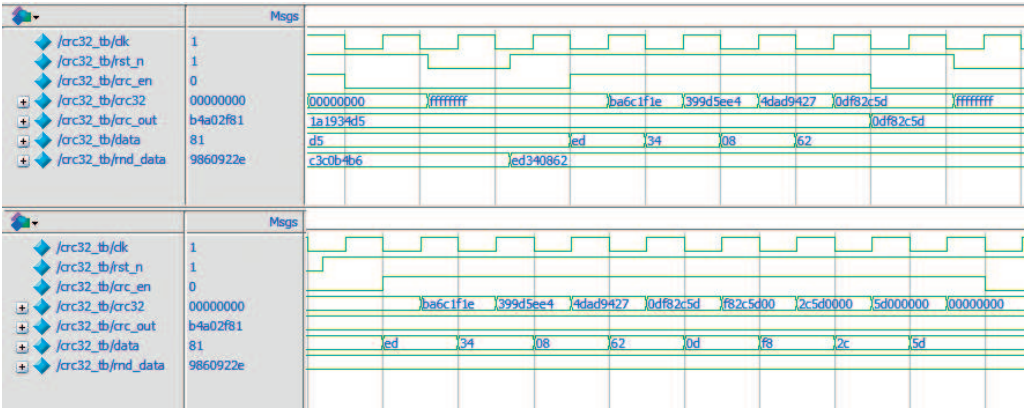


Рис. 6.18 Результаты моделирования модуля вычисления CRC-32

РСЛОС используются для реализации алгоритмов скремблирования и дескремблирования. **Скремблирование** – это процесс шифрования информации, после которого данные выглядят случайными. Процесс скремблирования служит для улучшения спектральных параметров передаваемого сигнала и защиты информации от несанкционированного доступа.

6.7 Примеры организации взаимодействия цифровых систем с простыми периферийными модулями

Изучение основ последовательностной логики было бы неполным без объединения полученных знаний с практическими задачами, например подключаемыми модулями (**матрица светодиодов, ультразвуковой датчик hc-sr04, пьезоэлектрический излучатель**). Ниже дано несколько примеров задач, в решении

которых требуется применять рассмотренные в данной практической работе модули.

6.7.1 Матрица светодиодов

Первый пример представляет собой генерацию различных фигур на экране светодиодной матрицы с использованием сдвиговых регистров. Подавая различные уровни напряжения на входные контакты матрицы, можно включать или выключать светодиоды матрицы. Исходные файлы проекта можно найти в дополнительных материалах к практической работе. После того как будут более подробно изучены последующие главы, советуем обязательно вернуться к данному разделу и разработать свой собственный драйвер для отображения текстовых символов или изображений.

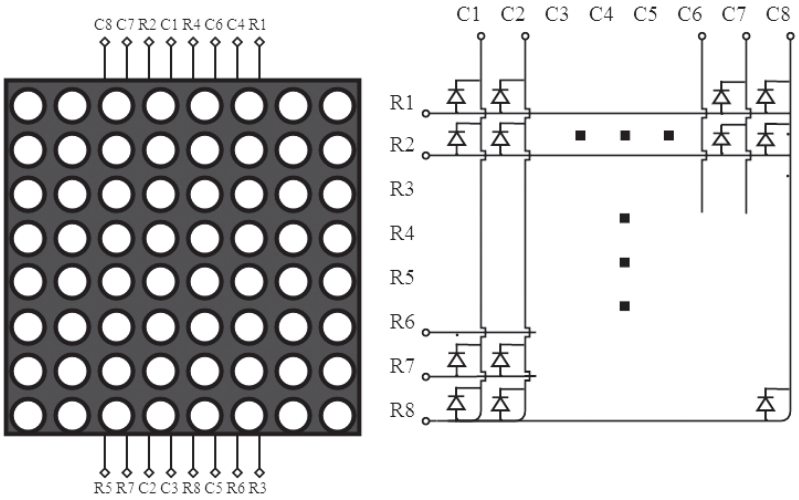


Рис. 6.19 Матрица светодиодов

6.7.2 Пьезоэлектрический излучатель

Еще один интересный проект, который можно сделать, – работа с электромеханическим звукоизлучателем. Его легко найти на материнской плате компьютера. Принцип работы излучателя достаточно прост: изменяя частоту входного напряжения, можно генерировать звуки различной тональности. Объединив звуки в гаммы, можно составить мелодию. В дополнительных материалах[1] к практической работе уже реализованы **три** мелодии. Попробуйте распознать эти музыкальные композиции. Перед запуском проекта в **Quartus Prime** укажите номер необходимой мелодии в макросе. Некоторые файлы проекта написаны на языке **SystemVerilog**. Реализуйте эти файлы на **Verilog** и сравните результат. Обратите внимание, что минимальная используемая длительность ноты – одна четвертная.

[1] https://github.com/RomeoMe5/DDLM.

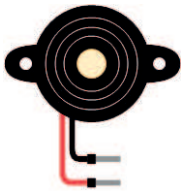


Рис. 6.20 Пьезоэлектрический излучатель

Модифицируйте проект. Добавьте несколько октав для выбранной тональности. Реализуйте несколько любимых мелодий. Добавьте аккомпанемент к мелодии, используя больше подключаемых модулей.

6.7.3 Ультразвуковой дальномер

Ультразвуковой дальномер HC-SR04 предназначен для измерения расстояния между местом расположения датчика и препятствием. В основе работы устройства лежит следующий принцип: датчик посылает ультразвуковую волну и, после того как она отразилась от препятствия, регистрирует время задержки между началом отправления и приемом, что позволяет рассчитать расстояние до объекта. Такой тип датчика не подходит для измерения расстояний до звукопоглощающих объектов, а ультразвук, созданный датчиком, распространяется в узком секторе в заданном направлении.

Для запуска датчика на стороне пользователя необходимо сгенерировать импульс продолжительностью около **10 мкс** и подать на вход **Trig**. После отправления датчик посылает серию из **8** импульсов и устанавливает активный уровень сигнала **Echo**. Как только волна возвращается обратно, датчик изменяет уровень сигнала **Echo**, который и становится неактивным. Зная длительность импульса **Echo**, можно рассчитать расстояние до объекта. Для удобства использования модуля полученное значение расстояния можно вывести на семисегментный дисплей.

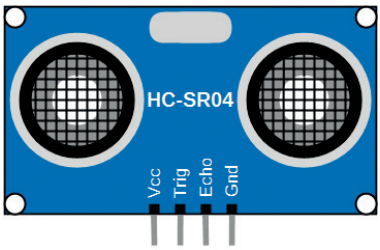


Рис. 6.21 Ультразвуковой дальномер

Приведенные примеры – далеко не полный список периферийных устройств, с которыми можно работать, обладая лишь базовыми знаниями о последовательной логике. Для лучшего понимания материала рекомендуется самостоятельно выполнить все работы, а затем сравнить полученные результаты с исходным кодом из дополнительных материалов[1] к данной практической работе.

[1] https://github.com/RomeoMe5/DDLM.

6.8 Упражнения

6.8.1 Основное задание

Выполните все дополнительные задания данной главы.

6.8.2 Задания для самостоятельной работы

В качестве основы для выполнения задач необходимо использовать проекты из разделов 6.5.1 и 6.6.1. Для демонстрации значений счетчика и сдвигового регистра используйте семисегментный дисплей и светодиоды, которые расположены на отладочной плате с **ПЛИС**.

1. Измените проект счетчика так, чтобы цифры на семисегментном дисплее отображались в десятичном формате без знака (0, 1, 2, …, 9, 10, …).
2. Измените проект счетчика так, чтобы цифры на семисегментном дисплее отображались в десятичном формате со знаком (…, –10, –9, …, –2, –1, 0, 1, 2, …, 9, 10, …).
3. Измените проект счетчика так, чтобы цифры каждого разряда на семисегментном дисплее отображались одинаково (пример: 0000, 1111, 2222, …, AAAA, …).
4. Измените проект счетчика так, чтобы цифры четных разрядов на семисегментном дисплее увеличивались или уменьшались в зависимости от нажатия на кнопки, а цифры нечетных разрядов оставались неизменны (пример: 0000, 1010, 2020, …, A0A0, …).
5. Измените проект счетчика так, чтобы цифры нечетных разрядов на семисегментном дисплее увеличивались или уменьшались в зависимости от нажатия на кнопки, а цифры четных разрядов оставались неизменны (пример: 0000, 0101, 0202, …, 0A0A, …).
6. Измените проект счетчика так, чтобы цифры двух произвольных разрядов инкрементировались, а цифры других разрядов декрементировались.
7. Измените проект счетчика так, чтобы цифры различных разрядов на семисегментном дисплее увеличивались или уменьшались с различной скоростью (пример: 0135, 026A, 039F, 04C5).
8. Измените проект счетчика так, чтобы цифры на семисегментном дисплее отображались с различной скоростью в зависимости от нажатия на кнопки.
9. Измените проект счетчика так, чтобы инкремент изменялся (1, 2, 3) в зависимости от нажатой кнопки.
10. Измените проект счетчика так, чтобы по нажатии на кнопки на семисегментном дисплее отображалось значение в коде Грея.
11. Измените проект счетчика так, чтобы увеличение счетчика происходило: раз в 1 секунду, раз в 2 секунды, раз в 0,25 секунды, раз в 0,01 секунды.

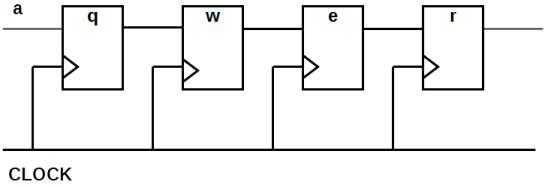
12. Измените проект счетчика так, чтобы отображаемые значения являлись элементами числовой последовательности Фибоначчи (пример: 0, 1, 1, 2, 3, 5, …).
13. Измените проект сдвигового регистра так, чтобы при нажатии кнопки изменялось направление сдвига. Для демонстрации используйте светодиоды на плате.
14. Измените проект сдвигового регистра так, чтобы при нажатии различных кнопок изменялась скорость сдвига. Для демонстрации используйте идентичные сегменты на семисегментном дисплее.
15. Измените проект сдвигового регистра так, чтобы реализовать кольцевой регистр. Данные следует выводить на светодиоды.
16. Измените проект сдвигового регистра так, чтобы с помощью кнопки вводить значения для последовательного сдвига, а на семисегментном дисплее отображать значение в десятичном и шестнадцатеричном форматах данных.
17. Измените проект сдвигового регистра так, чтобы происходил сдвиг двух разрядов одновременно. Данные вводятся последовательно, при помощи кнопки. При демонстрации используйте светодиоды и семисегментный дисплей на плате.
18. Измените проект сдвигового регистра так, чтобы реализовать генератор случайных чисел. Для демонстрации используйте семисегментный дисплей.
19. Измените проект ГСЧ для счетчика с загрузкой.
20. Измените проект ГСЧ так, чтобы после нажатия кнопки 1 спустя случайное время загорался светодиод. После зажигания светодиода необходимо как можно быстрее нажать на кнопку 2. Полученное время реакции вывести на семисегментный дисплей.
21. Изменить проект сдвигового регистра так, чтобы последовательностные значения счетчика выводились в виде бегущей строки на семисегментном дисплее (пример: 0000, 0001, 0012, …, 7891, 8910, 9101, …).
22. Измените проект сдвигового регистра и счетчика так, чтобы по нажатии кнопки 1 значение счетчика сохранялось в сдвиговом регистре (не менее трех значений). По нажатии кнопки 2 данные регистра последовательно выводятся на семисегментный дисплей.

6.8.3 Контрольные вопросы

1. В чем основные отличия последовательностных устройств от комбинационных?
2. Почему сложно разрабатывать асинхронные схемы, а в современной электронике большие асинхронные цифровые схемы практически не используются?
3. Опишите **модель Хаффмана** для последовательностных устройств.
4. Каково различие между блокирующим и неблокирующим присвоением?

5. К чему приведет изменение порядка операций в блоке **always**?

```
always@(posedge clock)
   begin
      r = e;
      e = w;
      w = q;
      q = a;
   end
```



6. В каких случаях следует использовать блокирующее присвоение, а в каких – неблокирующее?
7. Опишите порядок выполнения циклов симуляции, определенный в **IEEE Verilog Standart**.
8. В результате чего при синтезе возникают защелки (**latch**), и чем они опасны?
9. Для чего нужны счетчики? Какими они бывают? Опишите на **Verilog** пример простого счетчика.
10. Для каких задач используются делители частоты?
11. Опишите, что такое **широтно-импульсная модуляция** (**ШИМ**) и для чего она применяется.
12. Как строится счетчик Грея? В чем его особенность? Опишите пример счетчика Грея на **Verilog**.
13. Для чего нужны сдвиговые регистры? Каковы виды сдвиговых регистров? Приведите пример сдвигового регистра на **Verilog**.
14. Что такое циклический избыточный код (**CRC**)? Приведите области применения.
15. Приведите примеры периферийных устройств, используемых с отладочными платами **ПЛИС**; опишите процесс их подключения.

6.8.4 Приложение к главе

```
// Filename : crc32.v
// Description :
   x^32+x^26+x^23+x^22+x^16+x^12+x^11+x^10+x^8+x^7+x^5+x^4+x^2+x^1+x^0
// Algorithm : CRC-32 MPEG-2
// Polynomial : 0x04C11DB7
// Init value : 0xFFFFFFFF
// RefIn : False
// RefOut : False
// Checkout : 0x376E6E7
module crc32
(
   input clk,
```

```
   input rst_n,
   input [7:0] data,
   input crc_en,
   output reg [31:0] crc32
);
   always@(posedge clk or negedge rst_n)
      if(!rst_n)
         crc32 <= 32'hFFFFFFFF;
      else begin
         crc32 <= crc_en ? crc32_byte(crc32,data) : crc32;
      end

   function [31:0] crc32_bit;
      input [31:0] crc;
      input data;
      begin
         crc32_bit = crc << 1;
         crc32_bit[0] = crc[31] ^ data;
         crc32_bit[1] = crc[31] ^ data ^ crc[0];
         crc32_bit[2] = crc[31] ^ data ^ crc[1];
         crc32_bit[4] = crc[31] ^ data ^ crc[3];
         crc32_bit[5] = crc[31] ^ data ^ crc[4];
         crc32_bit[7] = crc[31] ^ data ^ crc[6];
         crc32_bit[8] = crc[31] ^ data ^ crc[7];
         crc32_bit[10] = crc[31] ^ data ^ crc[9];
         crc32_bit[11] = crc[31] ^ data ^ crc[10];
         crc32_bit[12] = crc[31] ^ data ^ crc[11];
         crc32_bit[16] = crc[31] ^ data ^ crc[15];
         crc32_bit[22] = crc[31] ^ data ^ crc[21];
         crc32_bit[23] = crc[31] ^ data ^ crc[22];
         crc32_bit[26] = crc[31] ^ data ^ crc[25];
      end
   endfunction // crc32_bit

   function [31:0] crc32_byte;
      input [31:0] crc;
      input [7:0] data;
      integer i;
      begin
         crc32_byte = crc;
         for(i=0;i<8;i=i+1) begin
            crc32_byte = crc32_bit(crc32_byte,data[7-i]);
         end
      end
   endfunction // crc32_byte
endmodule // crc32
```

Сергей Иванец, Александр Романов

Цифровой синтез: практический курс

Глава 7. Память: регистровый файл и стек

Содержание

*Память является одним из основных блоков любой вычислительной системы – как персонального компьютера, так и различных мобильных устройств. В главе приведена информация об основных типах памяти, применяемой в цифровых системах, а также даны основы работы со встроенной памятью на примере встроенной памяти **ПЛИС**. Изложение касается только полупроводниковой памяти и не относится к вопросам построения оптических или магнитных накопителей информации.*

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime** (есть студенческая версия);*
- *пакет **ModelSim Altera Edition** или программы **Icarus Verilog** и **GTKWave**;*
- *отладочная плата компании **Terasic DE10-Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах[1] к данной книге).*

7.1 Полупроводниковые устройства памяти

Любой элемент цифровой системы, сохраняющий свое состояние, можно рассматривать как определенную форму запоминающего устройства. В главе 2 был рассмотрен **D-триггер** – схема, в которой хранится один бит данных. Также был рассмотрен регистр как совокупность **D-триггеров** (рис. 7.1). Типичная ширина регистра составляет **8**, **16**, **32** или **64** бита, но можно написать **HDL**-код для регистра любой ширины, необходимой для проекта. Так, например, для работы с аудио в DVD могут потребоваться достаточно широкие регистры – до **24** бит.

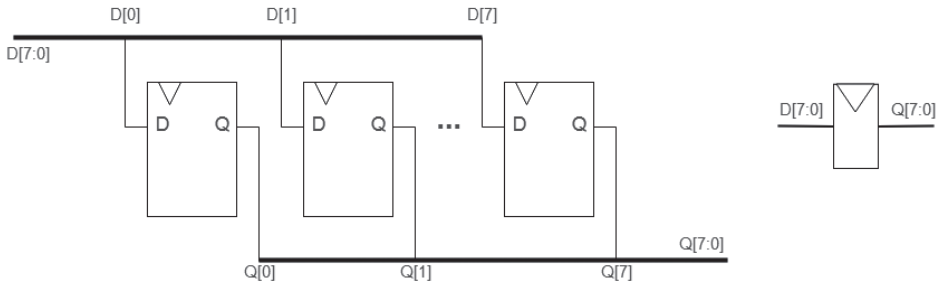


Рис. 7.1 8-битный регистр

Регистр может хранить только одно слово данных. Для хранения нескольких слов данных, например пакета 24-битных выборок аудиосигнала, уже требуется мас-

[1] https://github.com/RomeoMe5/DDLM.

сив памяти. В таком случае применяют **регистровые файлы**, представляющие собой несколько регистров с дополнительными цепями (рис. 7.2).

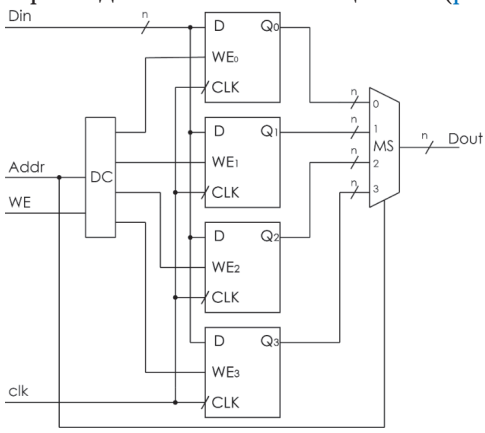


Рис. 7.2 Регистровый файл

Регистровый файл представляет собой несколько регистров с общими сигналами управления, адреса и данных. На рис. 7.2 приведена схема регистрового файла, в котором регистры имеют общий тактовый сигнал и общий вход данных. Для выбора одного из регистров используют дешифратор, на вход которого подают адрес. Этот же адрес подается на управляющие входы мультиплексора. К входам данных мультиплексора подсоединены выходы регистров.

Работа регистрового файла происходит следующим образом:

1. Для записи данных на линии адреса и данных подают адрес нужного регистра и его содержимое.
2. На вход **WE** подают сигнал разрешения записи, который формирует на входах **WE** регистров сигнал разрешения записи.
3. При поступлении переднего фронта тактового сигнала в нужный регистр произойдет запись данных. Так как на входы управления выходного мультиплексора поступают те же сигналы адреса, то на выходе появятся данные с выбранного регистра.
4. Для чтения данных необходимо снять сигнал разрешения записи и установить адрес нужного регистра. Этот адрес подключит выход мультиплексора к нужному входу.

Проблема регистрового файла, как и любых схем хранения данных на основе D-триггеров, заключается в том, что такие схемы синтезируются очень большими и занимают значительную площадь кристалла. Это связано с тем, что для реализации одного триггера потребуется более **30** транзисторов.

Более экономным решением является использование для хранения данных бистабильных ячеек статической памяти, которые требуют всего 6 транзисторов. Такие элементы памяти организовывают в виде **массива ячеек памяти**. На

рис. 7.3 приведена схема четырех строк однобитных элементов хранения. Каждая строка имеет уникальный адрес. Например, для доступа к нижней строке сохраненных данных необходимо использовать значение **2'b11**, которое подается на входы адреса **A[1:0]**. С помощью этого адреса внутри модуля памяти адресный дешифратор выберет нужную строку накопителя. Значение, хранящееся в данной строке, будет выведено на выходные линии **RD[3:0]**. В массиве имеется набор однобитных ячеек, которые воспринимаются снаружи как ячейка памяти определенного размера. На рис. 7.3 это **4** бита.

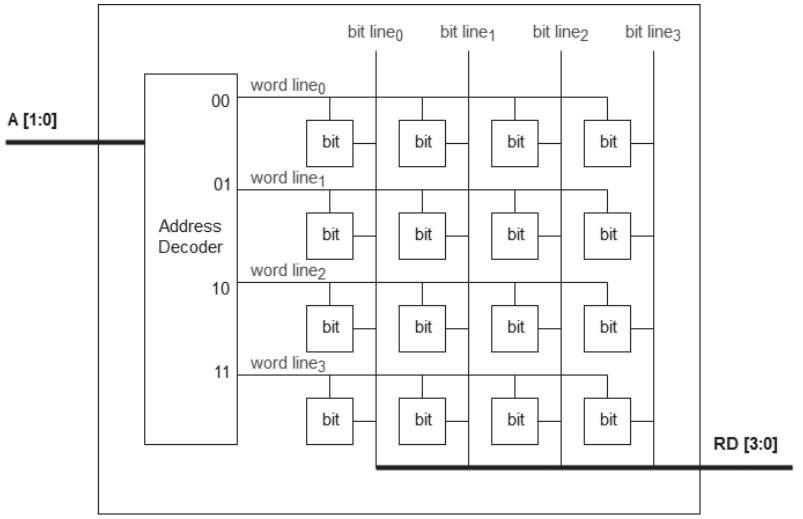


Рис. 7.3 Массив ячеек памяти

Приведенное выше графическое изображение модуля является основой **статической памяти**, или **ОЗУ** (**оперативного запоминающего устройства**). Английское название оперативной памяти носит исторические причины и в переводе значит «память с произвольным доступом» – **RAM** (**Random Access Memory**).

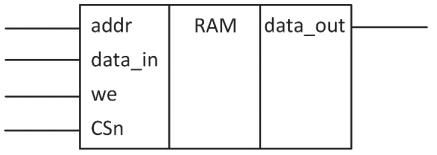


Рис. 7.4 Однопортовое ОЗУ

На рис. 7.4 приведена типичная микросхема статической памяти, которая имеет такие сигналы:

1) **адрес** (**addr**) – на этих линиях задается адрес ячейки, с которой сейчас будет производиться считывание или запись;

2) **входные** (**data_in**) и **выходные** (**data_out**) **данные**. По этим линиям данные от внешнего устройства подаются в ячейку, адрес которой выбран на шине адреса. Обычно в микросхемах памяти, которые выпускаются как отдельные

устройства, шина данных делается двунаправленной, т. е. одни и те же выводы микросхемы используются для чтения и записи информации. Это делается для того, чтобы уменьшить количество выводов в корпусе и уменьшить размеры корпуса. В памяти, которая реализуется внутри микросхем (**ПЛИС** или **ASIC**), делают отдельные линии для записи и для чтения;

3) **сигнал разрешения записи** (**we**) – предназначен для управления записью/чтением. При активном уровне сигнала (в нашем случае это высокий уровень сигнала) в модуль памяти производится запись. При неактивном – чтение. В некоторых случаях такой сигнал обозначается так: $\overline{W}/R$. В подобных схемах запись проводится при низком уровне сигнала, а чтение – при высоком;

4) **сигнал выбора кристалла** (**SCn**) – сигнал, который присутствует в микросхемах памяти и предназначен для отключения микросхемы и перевода ее выходов в z-**состояние**. В рассматриваемых в данной главе примерах активный уровень сигнала принят за низкий.

Основные режимы работы памяти – это чтение, запись и хранение (полный список режимов памяти зависит от технологии памяти).

1. **Режим чтения** памяти (рис. 7.5). При чтении внешнее устройство (мастер) указывает адрес строки, содержащей данные. Для проведения чтения на вход чтения-записи устанавливается сигнал чтения. После задержки (называемой временем доступа к адресу, $\mathbf{t_{AA}}$) память выставляет на линии данных информацию, хранящуюся в указанной ячейке. Часто микросхема памяти имеет дополнительные входы: разрешение работы (**enable**), выбор микросхемы (**chip select**) и т. д. В этом случае для чтения данных из памяти будет необходимо наличие активных уровней сигнала на этих входах.

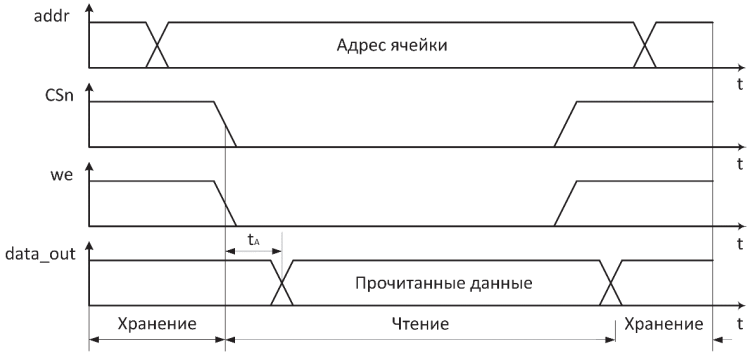


Рис. 7.5 Режим чтения из памяти

2. Режим записи (рис. 7.6). В этом режиме устанавливаются те же сигналы, что и при чтении, – адрес и дополнительные сигналы. Сигнал чтения-записи меняется на сигнал «запись», а на входы данных подают данные, которые необходимо записать. Через некоторое время ($\mathbf{t_A}$ или $\mathbf{t_{cs}}$) происходит запись необходимых данных в микросхему.

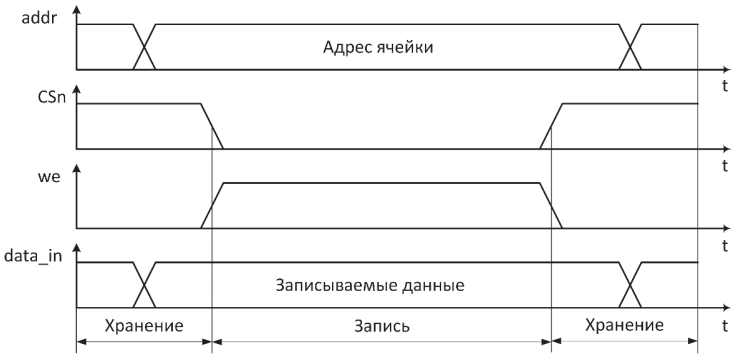


Рис. 7.6 Режим записи в память

3. В **режиме хранения** микросхема просто хранит данные и не производит обмен с другими устройствами.

4. Для микросхем динамической памяти существует также дополнительный режим – режим **регенерации памяти**, во время которого микросхема не реагирует на внешние сигналы и производит регенерацию данных с помощью контроллера регенерации, встроенного в микросхему.

7.2 Двухпортовая память

Описанная выше память имеет один набор шин адреса, данных и управления для работы с одним устройством. **Многопортовая память** имеет несколько наборов шин для одновременной работы с несколькими устройствами. Наиболее часто встречается двухпортовая память, которая позволяет производить одновременный доступ к содержимому памяти из двух различных портов (рис. 7.7). Существует несколько вариантов двухпортовой памяти:

1. **Простая двухпортовая память** (**simple dual-port memory**). Здесь один порт предназначен для записи, а второй – для чтения. То есть на первый порт подается сигнал адреса и данные для записи, а также сигнал записи. На второй порт подается сигнал адреса для чтения и сигнал чтения. На выходах данных этого порта будут существовать данные в соответствии с сигналами чтения. То есть у памяти такого типа существует два входа адреса, раздельные линии данных для чтения и для записи, а также раздельные линии управления чтением-записью. В простейшем случае оба порта работают с одинаковой частотой. В более сложном случае каждый порт работает со своей собственной частотой.

2. **Полная двухпортовая память** (**true dual-port memory**). Здесь каждый порт содержит линии адреса и данных. Каждый порт может проводить операции чтения и записи. В простом случае порты синхронизируются с той же частотой, и оба порта работают синхронно. В более сложном случае каждый оснащен собственным входом тактовой частоты, и они могут иметь разные разрядности данных.

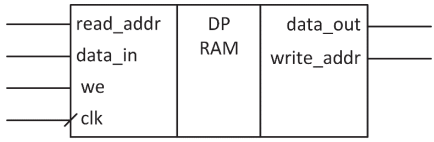


Рис. 7.7 Двухпортовая память

Возможно несколько вариантов работы двухпортовой памяти:

1. **Одновременная запись и чтение разных ячеек памяти**. То есть один порт производит запись данных в память, а второй – чтение данных из памяти. При этом порты обращаются к разным ячейкам памяти. В этом случае мы можем воспринимать двухпортовую память как два различных модуля памяти, которые не оказывают взаимного влияния друг на друга.

2. **Чтение одной и той же ячейки из разных портов**. Ситуация подобна описанной выше – порты не оказывают взаимного влияния.

3. Запись информации в память одним портом, в то время как другой порт производит чтение данных из этой же ячейки. Этот режим носит название «**чтение во время записи**» (**Read-During-Write Mode**). Поведение блока памяти определяется тем, как ведет себя в данной ситуации микросхема **ПЛИС**, на которой производится реализация данного модуля. Здесь возможны такие варианты поведения памяти:

   - «**старые данные**» (**old data**) – на выходе будут видны те данные, которые были записаны в эту ячейку ранее. Новые данные появятся на выходе только в следующем цикле чтения;
   - «**новые данные**» (**new data**) – на выходе будут видны те данные, которые записываются в ячейку в текущем такте;
   - «**не важно**» (**don't care**) – в этом случае производитель не гарантирует на выходе какого-то определенного значения (старого или нового содержимого ячейки). Такая ситуация наиболее вероятна в том случае, когда каждый из портов работает на своей тактовой частоте.

4. **Одновременная запись данных в одну и ту же ячейку**. Такая ситуация порождает конфликт, исход которого неясен, т. к. в микросхеме не заложен механизм разрешения данного конфликта. В этом случае разработчику необходимо описать поведение блока памяти в такой ситуации и поместить данный блок в свой проект.

Наиболее важным параметром памяти является ее информационная емкость. **Информационная емкость** – это объем данных, которые могут быть сохранены в устройстве. Емкость памяти выражается в битах или байтах. Другим параметром является **ширина данных, хранимых в памяти**: количество битов в каждой ячейке (например, **16** бит).

Информационная емкость памяти определяется по формуле:

$$\text{Емкость памяти} = \text{Ширина шины данных} \times 2^{\text{Ширина шины адреса}}.$$

Например, для микросхемы памяти, у которой есть **10** линий адреса и объем одной ячейки равен **8** битам, информационная емкость будет равна

$$8 \times 2^{10} = 8 \times 1024 = 1 \text{ кбайт.}$$

Самый распространенный **способ классификации памяти** – это разделение на **оперативные запоминающие устройства** (**ОЗУ**) и **постоянные запоминающие устройства** (**ПЗУ**). С самого начала появления полупроводниковой памяти и до конца XX века (времени изобретения флеш-памяти) эти два вида памяти разделялись по двум признакам – энергонезависимость и скорость доступа во время чтения и записи. **Энергонезависимая** (**ПЗУ**) память хранит информацию при выключении питания, **энергозависимая** (**ОЗУ**) теряет информацию при выключении питания. Скорость доступа к ячейке для **ОЗУ** одинакова как в режиме чтения, так и в режиме записи, а для **ПЗУ** скорость записи была намного ниже, чем скорость чтения. Но сейчас эти границы стираются, и можно встретить **ОЗУ**, которое содержит блок питания для сохранения информации, **ПЗУ** типа флеш-памяти, в котором скорость записи ненамного ниже скорости чтения.

Оперативная память, **ОЗУ** (**Random Access Memory**, **RAM**) – это память с произвольным доступом. Она разделяется на два вида: статическая и динамическая.

Статическая оперативная память (**Static RAM**, **SRAM**). Слово «статическая» означает, что такая память хранит информацию до тех пор, пока на нее подано питание, но для нее не требуются дополнительные операции для поддержания актуального значения содержимого ячеек памяти. Память этого вида является основой для кеш-памяти процессов. В микроконтроллерах встроенная статическая память может быть единственным доступным видом ОЗУ. Именно этот вид памяти будет использоваться в данной работе в качестве основного строительного материала для различных видов памяти.

Динамическая оперативная память (**Dynamic Random Access Memory**, **DRAM**). Особенностью такого вида памяти является то, что для хранения информации используется конденсатор. Поскольку емкость конденсатора мала (единицы пикофарад), то его необходимо периодически перезаряжать. Процесс перезаряда конденсатора называется регенерацией. Поэтому данный вид памяти называется «динамическим». Современные чипы **DRAM** (**SDRAM** и особенно **DDRx**) являются достаточно сложными и требующими для управления специального устройства – контроллера динамической памяти. Разработка на языке описания аппаратуры контроллера динамической памяти является достаточно сложной задачей и может служить темой для индивидуального задания.

Постоянные запоминающие устройства (**ПЗУ**, **Read Only Memory – ROM**). **ПЗУ** – это энергонезависимая память, т. е. она сохраняет информацию при выключении питания. Обычно этот тип памяти используется для хранения программы загрузки микропроцессорной системы – первой программы, которая запускается после включения питания (ее основная задача – проверить аппаратное обеспечение и скопировать в **ОЗУ** следующий программный код начальной загрузки). Можно выделить несколько типов **ПЗУ**, с которыми вы можете встретиться в настоящее время: **PROM**, **EPROM**, **EEPROM** и **флеш-память** (**Flash memory**).

Программируемые постоянные запоминающие устройства (**ППЗУ**, **Programmable Read-Only Memory** – **PROM**), иногда еще называемые однократно программируемые **ПЗУ**. Микросхема выпускается с плавкими перемычками во всех ячейках памяти. Данные записываются в ней путем пережигания этих плавких перемычек при помощи специального устройства – программатора. Пережигание возможно произвести только один раз.

Репрограммируемые постоянные запоминающие устройства (**РППЗУ**, **Erasable PROM** – **EPROM**). Информация в данных микросхемах может быть записана и стерта некоторое количество раз, которое в зависимости от типа микросхемы может колебаться от нескольких десятков до нескольких тысяч раз. Ячейка памяти в таких микросхемах состоит из одного или двух полевых транзисторов. Эти микросхемы делятся на два вида – со стиранием ультрафиолетом и электрически стираемые. Для программирования таких микросхем также необходимо использование программатора. И в первом, и во втором случаях данные микросхемы имеют под затвором дополнительный участок, называемый плавающим затвором, в котором может накапливаться заряд. Наличие или отсутствие заряда в плавающем затворе изменяет пороговое напряжение открытия полевого транзистора, что, в свою очередь, позволяет кодировать «**0**» или «**1**».

Стираемые ультрафиолетом микросхемы (**РППЗУ – УФ**) имеют в корпусе специальное окно, через которое в микросхему попадает ультрафиолетовое излучение, и заряд в плавающем затворе получает дополнительную энергию для перехода в подложку и рекомбинации.

Электрически стираемые репрограммируемые ПЗУ (**Electrically Erasable PROM** – **EEPROM**) для стирания используют специальную электрическую схему внутри микросхемы. Широко используемая сейчас **флеш-память** исторически основана на **EEPROM**, позволяет частичное удаление данных. При этом современные флеш-микросхемы не требуют дополнительного внешнего программатора и могут перепрограммироваться прямо на плате. Хотя скорость записи у современной флеш-памяти возросла, она все еще ниже, чем скорость чтения.

Память может быть классифицирована по функциям, которые она выполняет в компьютере (рис. 7.8):

1. **Регистры центрального процессора** (**CPU registers**), иногда называемые **сверхбыстрой оперативной памятью** (**СОЗУ**). Регистры процессора являются самым быстродействующим видом памяти, так как они находятся внутри процессорного ядра и работают на частотах ядра. Однако объем памяти, реализуемой на таких регистрах, очень мал – их количество редко превышает 64. Такая память является энергозависимой, и информация в ней исчезает после выключения питания. Еще одним недостатком подобного вида памяти является ее стоимость – для хранения одного бита данных необходима ячейка, представляющая собой D-триггер, для реализации которого потребуется более 30 транзисторов. Под регистрами центрального процессора подразумеваются как **регистры общего назначения** (**General Purpose Registers**, **GPR**), так и другие регистры, видимые разработчику программного обеспечения: **регистр состояния**

(**status register**), **счетчик команд** (**program counter**), регистры сопроцессоров. Некоторые внутренние регистры не видны программистам: регистры этапов конвейера, регистры работы с оперативной памятью.

2. **Кеш-память** (**cache memory**). Кеш-память процессора спроектирована как массив **статической оперативной памяти** (**Static RAM**, **SRAM**) с ассоциативным доступом. Это фактически буфер с высокой скоростью доступа между процессором и «медленной» внешней памятью. Ее стоимость не так велика, как стоимость регистров центрального процессора, так как для хранения одного бита требуется около 6 транзисторов. Однако скорость обращения к такой памяти ниже, чем к регистрам процессора. Слово «статическая» означает, что такая память хранит информацию до тех пор, пока на нее подано питание, но для нее не требуются дополнительные операции для поддержания актуального значения содержимого ячеек памяти. В микроконтроллерах встроенная статическая память может быть единственным доступным **ОЗУ**.

3. Следующий вид памяти компьютера, который наиболее часто ассоциируется с термином «память», – это **динамическая оперативная память**, или **динамическая память**. По сравнению с кеш-памятью она имеет достаточно большой объем и более низкое быстродействие. Для работы с таким видом памяти необходим специальный контроллер, который в персональном компьютере входит в состав набора микросхем для обслуживания материнской платы – чипсета. Для работы с динамической памятью выделяется специальная шина, которая позволяет проводить как высокоскоростной обмен с центральным процессором, так и обмен памяти с периферийными устройствами без участия процессора. Последний режим работы называется **прямой доступ к памяти** (**ПДП**, **Direct Memory Access** – **DMA**).

4. В терминологии микропроцессорных систем все типы памяти, о которых говорилось выше, можно также назвать первичной памятью. Еще один тип памяти, о котором мы поговорим, называется **вторичной памятью** – недорогой энергонезависимой памятью с очень медленным доступом к данным большого объема. Он также называется дисковым хранилищем, т. е. для хранения данных используются периферийные устройства, такие как **жесткие магнитные диски** (**HDD**), **SSD-диски**, использующие флеш-память, или магнитная лента (в настоящее время применяемая для резервного копирования).

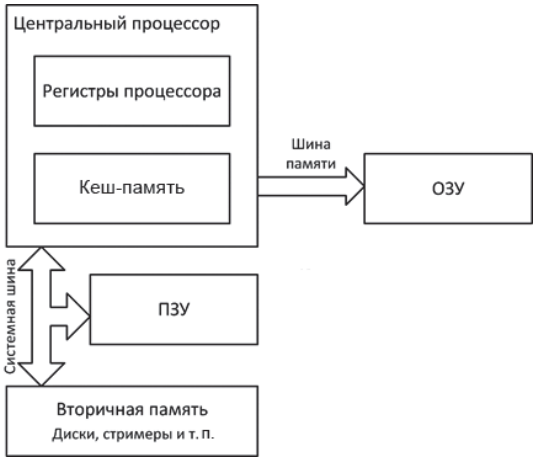


Рис. 7.8 Классификация памяти по функциям, которые она выполняет в компьютере

7.3 Иерархия памяти в компьютере. Память в микросхемах ПЛИС

Современные микросхемы **ПЛИС** содержат целую иерархию памяти, которая зависит от семейства микросхем. Наиболее простым вариантом памяти можно считать таблицу перекодировки (**Lookup таблица**, **LUT**)[1]. Она представляет собой однобитную память с четырьмя адресными входами (рис. 7.9). Пример использования таблицы перекодировки приведен в табл. 7.1. Здесь есть **4** адресных входа (**data1**–**data4**), которые определяют значение выходного сигнала (**y**). В зависимости от комбинации сигналов **data1**–**data4** можно получать различные значения на выходе **y**. Это значение может быть любой логической функцией от четырех переменных. Например, для логической функции **4-И** будет получена такая таблица.

Таблица 7.1 Пример таблицы перекодировки

data1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
data2	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
data3	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
data4	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
y	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Таблица перекодировки очень широко используется в различных семействах **ПЛИС**; так, в микросхеме **MAX10**, которую мы используем в данном практикуме, содержится 100 500 логических элементов (**Logic Elements**, **LE**), каждый из которых имеет таблицу перекодировки (рис. 7.9).

[1] Intel MAX 10 FPGA Device Overview. Intel, 2017. 14 p.

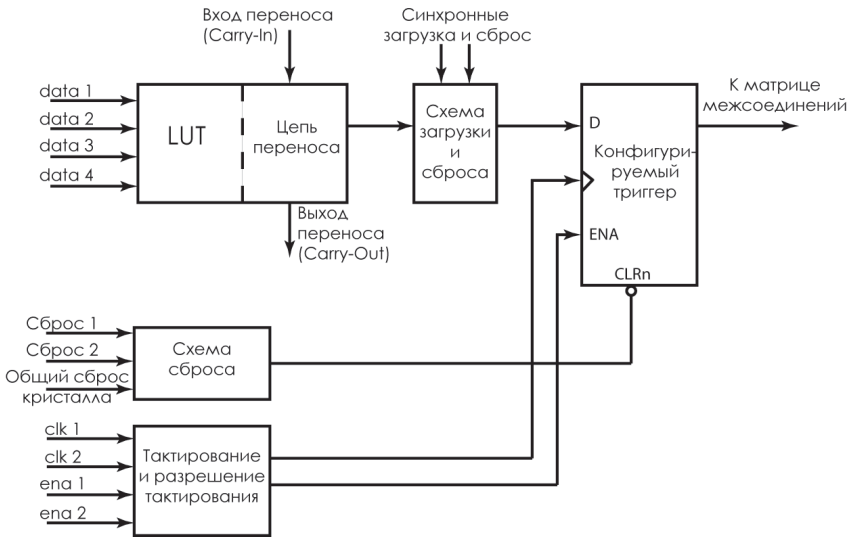


Рис. 7.9 Логический элемент микросхемы Intel FPGA MAX10

В состав логического элемента **ПЛИС** также входит программируемый регистр, который может служить основой для небольших модулей памяти.

Наиболее важным видом памяти в **ПЛИС** является встроенная память. Микросхемы семейства **MAX10** содержат блоки встроенной памяти **M9K**, объем которых составляет 9216 бит с битом контроля четности[1]. Эти блоки можно каскадировать для получения модулей памяти разного объема и разрядности. В зависимости от своих функций блок памяти может конфигурироваться в самые разные устройства: **ОЗУ** или **ПЗУ**, **одно-** или **двухпортовую память**, **регистр сдвига**, **блок FIFO**. При применении однопортовой памяти можно использовать **In-System Memory Content Editor** для контроля содержимого памяти из пакета **Quartus Prime**[2].

Основным отличием встроенных блоков памяти **ПЛИС** является их конфигурируемость, т. е. блок памяти может иметь несколько вариантов организации, объем которых не превышает объем встроенной памяти микросхемы. Блок памяти **M9K** допускает конфигурации, описанные в руководстве по использованию встроенной памяти микросхемы **Intel FPGA MAX 10**[2].

Таблица 7.2 Варианты конфигурации блока памяти M9K в микросхеме MAX10

Режим работы	Разрядность данных
Однопортовая память	×1, ×2, ×4, ×8, ×9, ×16, ×18, ×32, ×36
Простая двухпортовая память	×1, ×2, ×4, ×8, ×9, ×16, ×18, ×32, ×36
Полная двухпортовая память	×1, ×2, ×4, ×8, ×9, ×16, ×18

[1] Intel MAX 10 Embedded Memory User Guide. Intel, 2018. 58 p.

[2] Chapter 18. In-System Modification of Memory and Constants // Intel® Quartus® Prime Standard Edition Handbook Volume 1. Design and Synthesis. Intel, 2017. 1916 p.

Для того чтобы работать со встроенным блоком памяти, необходимо разработать код на языке описания аппаратуры, который будет описывать нужный вам блок памяти. Если код разработан корректно, то пакет синтеза (в нашем случае это **Quartus Prime**) распознает в этом коде описание памяти и будет использовать блоки **M9K** для последующей реализации памяти. Как правильно сделать такое описание, показано в последующих разделах этой главы. Для правильного описания модулей памяти следует пользоваться рекомендациями производителей микросхем[1].

Микросхема **ПЛИС** содержит еще один вид памяти – конфигурационную память. Она предназначена для хранения конфигурации, т. е. соединения логических элементов, памяти, умножителей, элементов ввода-вывода и других модулей микросхемы между собой. Мы используем эту память во время загрузки конфигурации с помощью интерфейса **Quartus Programmer**.

7.4 Регистровый файл

Описание регистрового файла на языке Verilog

Опишем регистровый файл, изображенный на рис. 7.2, на языке **Verilog**. Для наглядности возьмем разрядность **n**, равную **4**. Описание регистрового файла приведено в листинге 7.1. Нет смысла останавливаться на разборе кода регистра, так как его описание уже рассмотрено в главе 6. В основной части модуля этот регистр подключен с помощью такой конструкции:

```
register #(4) register0 (.ena(w_we[0]), .clk(clk), .d(SW[3:0]), .q(reg0));
```

Здесь необходимо указать имя модуля (**register**), его разрядность (**4**) и имя экземпляра данного модуля, который сейчас инициализируется (**register0**). Далее идет подключение портов подключаемого регистра к портам основного модуля.

Также в модуле описаны дешифратор 2в4 и мультиплексор. Дешифратор выбирает нужный регистр в зависимости от входного адреса (**SW[5:4]**) и переводит выходы в неактивное состояние при неактивном сигнале разрешения записи (**SW[6]**). Мультиплексор коммутирует выходы регистров в зависимости от значения входного адреса (**SW[5:4]**).

```
module register
#(
   parameter SIZE = 4
)
(
   input ena,
   input clk,
   input [SIZE - 1 : 0] d,
   output reg [SIZE - 1 : 0] q
);
```

[1] Chapter 12. Recommended HDL Coding Styles // Intel® Quartus® Prime Standard Edition Handbook Volume 1. Design and Synthesis. Intel, 2017. 1916 p.

```
   always @ (posedge clk)
   if (ena)
      q <= d;
endmodule

module lab7_1
#(
   parameter DATA_WIDTH=4,
   parameter ADDR_WIDTH=2
)
(
   input clk,
   input [6:0] SW,
   output [3:0] LEDR
);
   reg [3:0] w_we; // DC output for write enable
   reg [3:0] mux_out; // mux out
   wire [(DATA_WIDTH-1):0] reg0; // output register 0
   wire [(DATA_WIDTH-1):0] reg1; // output register 1
   wire [(DATA_WIDTH-1):0] reg2; // output register 2
   wire [(DATA_WIDTH-1):0] reg3; // output register 3

   // DC
   always @(SW[6:4])
      begin
         if (SW[6])
            case (SW[5:4])
               2'b00 : w_we = 4'b0001;
               2'b01 : w_we = 4'b0010;
               2'b10 : w_we = 4'b0100;
               2'b11 : w_we = 4'b1000;
               default : w_we = 4'b0000;
            endcase
         else
            w_we = 4'b0000;
      end

   //mux
   always @( SW[5:4], reg0, reg1, reg2, reg3)
   case (SW[5:4])
      2'b00 : mux_out = reg0;
      2'b01 : mux_out = reg1;
      2'b10 : mux_out = reg2;
      2'b11 : mux_out = reg3;
      default : mux_out = 4'b0000;
   endcase

   assign LEDR = mux_out;
```

```
   register #(4) register0 (.ena(w_we[0]), .clk(clk), .d(SW[3:0]),
      .q(reg0));
   register #(4) register1 (.ena(w_we[1]), .clk(clk), .d(SW[3:0]),
      .q(reg1));
   register #(4) register2 (.ena(w_we[2]), .clk(clk), .d(SW[3:0]),
      .q(reg2));
   register #(4) register3 (.ena(w_we[3]), .clk(clk), .d(SW[3:0]),
      .q(reg3));
endmodule
```

Листинг 7.1 Регистровый файл

Схема подключения регистрового файла к периферии отладочной платы **DE10-Lite** приведена на рис. 7.10.

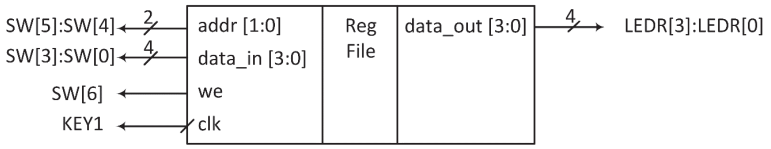


Рис. 7.10 Подключение регистрового файла к периферии отладочной платы DE10-Lite

Создание и компиляция проекта

Создайте новый проект. Исходные файлы проекта находятся в папке **1_lab7_hdl_Register_File**. Сохраните листинг 7.1 в файле **lab7_1.v**. Это файл верхнего уровня иерархии для нашего проекта. Выполните компиляцию проекта.

Результат компиляции проекта в **RTL Viewer** приведен на рис. 7.11.

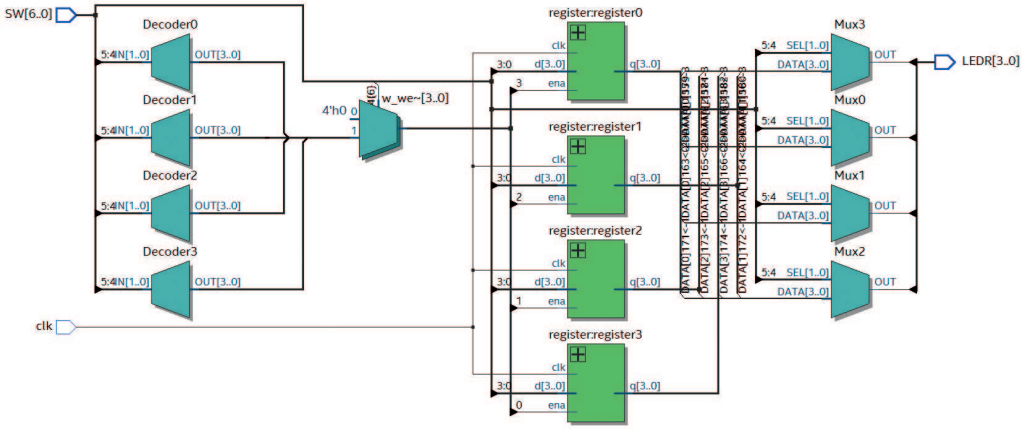


Рис. 7.11 Регистровый файл в RTL Viewer

В результате компиляции было получено 4 регистра. Выходы регистров коммутируются на выход с помощью мультиплексора, что и было описано в коде выше. Реализация дешифратора сделана на **четырех** дешифраторах (**Decoder0–**

Decoder3) и мультиплексоре, который коммутирует на входы разрешения записи регистров один из пяти четырехразрядных наборов – четыре выхода дешифраторов и все нули для ситуации запрета записи.

Назначение выводов. Компиляция проекта

Назначьте выводы в соответствии с табл. 7.3.

Табл. 7.3 Соответствие выводов для проекта lab 7_1

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
LEDR[3]	Output	PIN_B10	3.3-V LVTTL
LEDR[2]	Output	PIN_A10	3.3-V LVTTL
LEDR[1]	Output	PIN_A9	3.3-V LVTTL
LEDR[0]	Output	PIN_A8	3.3-V LVTTL
SW[6]	Input	PIN_A13	3.3-V LVTTL
SW[5]	Input	PIN_B12	3.3-V LVTTL
SW[4]	Input	PIN_A12	3.3-V LVTTL
SW[3]	Input	PIN_C12	3.3-V LVTTL
SW[2]	Input	PIN_D12	3.3-V LVTTL
SW[1]	Input	PIN_C11	3.3-V LVTTL
SW[0]	Input	PIN_C10	3.3-V LVTTL
clk	Input	PIN_A7	3.3 V Schmitt Trigger

Конфигурирование ПЛИС и проверка работы блока памяти

Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.

Теперь проверьте работоспособность проекта. Для этого выполните следующие действия:

- используйте кнопки и микропереключатели для подачи сигналов на входы и проверяйте их значения с помощью светодиодов;
- проведите запись информации в регистры, контролируя вывод информации на светодиоды;
- проведите чтение регистров, проверяя правильность записанной информации.

Тестбенч и его описание

Тестбенч для регистрового файла очень прост. Сначала проведем первоначальную установку сигналов данных, сигнала разрешения записи и выбор первого регистра:

```
SW = 7'b1001111;
```

Далее с задержкой в 20 нс проведем последовательную запись во все регистры.

```
repeat(3)
begin
   #20;
   SW[5:4] = SW[5:4]+2'b01;
end
```

Так как выбор первого регистра уже был проведен, то ближайший тактовый сигнал запишет в него данные с линии данных. В трех последующих тактах будет проведена запись в три оставшихся регистра. Для этого значение адреса увеличивается на **1** в каждом новом такте.

Затем производится сброс значения сигнала разрешения записи и проводится последовательное чтение всех регистров. Для того чтобы убедиться, что проводится чтение сохраненных данных, на входах устанавливаем новые значения.

```
SW = 7'b0000000;
repeat(3)
begin
   #20;
   SW[5:4] = SW[5:4]+2'b01;
end
```

Содержимое тестбенча следующее:

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns/1ns
module testbench;
   // input and output test signals
   reg clk;
   reg [6:0] SW;
   wire [3:0] LEDR;
   // creating the instance of the module we want to test
   // lab7_1 - module name
   // dut - instance name (‘dut’ means ‘device under test’)
   lab7_1 dut ( clk, SW, LEDR );

   initial
      begin
         // set inital values of signal
         clk = 1;
         SW = 7'b1001111;
         #5;

         repeat(3)
         begin
            #20;
            SW[5:4] = SW[5:4]+2'b01;
         end
```

```
        #40;
        SW = 7'b0000000;
        repeat(3)
        begin
           #20;
           SW[5:4] = SW[5:4]+2'b01;
        end
     end

     //every 10 ns invert clk
     always #10 clk = ~clk;

     initial
        #200 $finish;

     // do at the beginning of the simulation
     // print signal values on every change
     initial
        $monitor(«clk=%b SW=%h LEDR=%h», clk, SW, LEDR);
     // do at the beginning of the simulation
     initial
        $dumpvars; //iVerilog dump init
endmodule
```

Листинг 7.2 Тестбенч для моделирования регистрового файла

Симуляция модуля регистровой памяти

Для запуска **ModelSim** будем использовать скрипт, который содержится в файле **01_simulate_with_modelsim.bat**. Он полностью повторяет скрипт из главы 1. Здесь изменили только скрипт для работы в пакете **modelsim_script.tcl**, в котором изменили имя файла с описанием модуля памяти:

```
# compile all the Verilog sources
vlog ../testbench.v ../../lab7_1.v
```

Во всех остальных частях работы, если не указано другое, для запуска процесса симуляции необходимо заменить имя модуля в скрипте **modelsim_script.tcl**.

Рисунок 7.12 демонстрирует результаты симуляции регистрового файла в **ModelSim**.

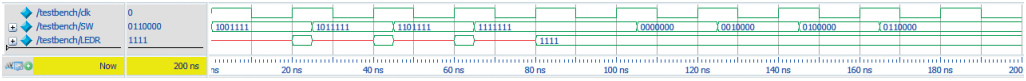


Рис. 7.12 Результат симуляции регистрового файла в ModelSim

7.5 Реализация ПЗУ с помощью таблицы перекодировки

Логический элемент **ПЛИС** содержит таблицу перекодировки – маленькое **ПЗУ** на четыре входа и один выход (рис. 7.9). Следовательно, на его базе можно реали-

зовать **ПЗУ**. Для этого необходимо описать **ПЗУ** с помощью таблицы истинности, и она будет реализована на логических элементах. Для примера реализуем функцию **4-И** с помощью таблицы перекодировки, описанной в табл. 7.1.

Описание логической функции на основе LUT на языке Verilog

Для описания логической функции на основе **LUT** на языке **Verilog** используем оператор **case**. Для этого необходимо перебрать все значения входной переменной **x** и указать значение выходной функции **y**.

```
module lab7_2
(
   input [3:0] x,
   output reg y
);
   always @(x)
      begin
         case(x)
            'b0000: y<= 0;
            'b0001: y<= 0;
            'b0010: y<= 0;
            'b0011: y<= 0;
            'b0100: y<= 0;
            'b0101: y<= 0;
            'b0110: y<= 0;
            'b0111: y<= 0;
            'b1000: y<= 0;
            'b1001: y<= 0;
            'b1010: y<= 0;
            'b1011: y<= 0;
            'b1100: y<= 0;
            'b1101: y<= 0;
            'b1110: y<= 0;
            'b1111: y<= 1;
         endcase
      end
endmodule
```

Листинг 7.3 Реализация логической функции с помощью таблицы перекодировки

Поскольку данный пример очень прост, мы не будем проводить проверку его работы на отладочном стенде. Результат компиляции в **RTL Viewer** приведен на рис. 7.13.

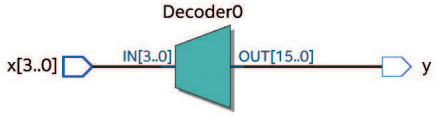


Рис. 7.13 Результат компиляции логической функции в RTL Viewer

Тестбенч и его описание

Тестбенч для разработанного модуля приведен в листинге 7.4. В листинге сначала производится начальная установка значения сигнала **x**:

```
x = 4'b0000;
```

Затем последовательно перебираются все оставшиеся значения сигнала **x** и подаются на вход **DUT**.

```
repeat(15)
begin
   #10;
   x = x + 4'b0001;
end
```

Листинг тестбенча приведен ниже.

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns/1ns
module testbench;
   // input and output test signals
   reg [3:0] x;
   wire y;
   // creating the instance of the module we want to test
   lab7_2 dut ( x, y );

initial
   begin
      // set inital values of signal
      x = 4'b0000;
   end

   initial
      #160 $finish;

      // do at the beginning of the simulation
      // print signal values on every change
   initial
      $monitor("x=%b y=%h", x, y);

   // do at the beginning of the simulation
   initial
      $dumpvars; //iVerilog dump init
endmodule
```

Листинг 7.4 Тестбенч для моделирования простой логической функции

На рис. 7.14 приведены результаты симуляции простой логической функции в **ModelSim**.

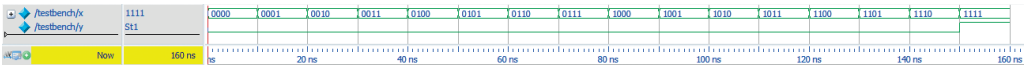


Рис. 7.14 Результат симуляции простой логической функции в ModelSim

7.6 Однопортовая память

Описание однопортового ОЗУ на языке Verilog

Для описания памяти на языке **Verilog** необходимо создать переменную типа массив, в которой будут храниться данные. Размер элемента массива должен совпадать с разрядностью ячейки памяти, т. е. если ячейка памяти хранит **4** бита, то и разрядность элементов массива должна быть равна **4**. Для этого в квадратных скобках мы укажем диапазон **[3:0]**, младший бит справа (**little endian**). Количество элементов массива задается вторым диапазоном **[0:63]** и равно количеству ячеек памяти. Ячейка памяти с нулевым адресом будет находиться в нулевом элементе массива:

```
reg [3:0] ram[0:63];
```

Имя переменной **ram** относится в целом к массиву. Для того чтобы выбрать первую ячейку, необходимо использовать такую запись: **ram[0]**. Число в квадратных скобках должно находиться в диапазоне от **0** до **63**, т. к. разрабатывается модуль памяти объемом в **64** ячейки. Это определяет необходимость в наличии 6 линий адреса, объявление которых выглядит следующим образом:

```
input [5:0] addr,
```

Далее необходимо описать линии входных и выходных данных. Разрядности этих портов должны быть одинаковы и совпадать с разрядностью ячейки в массиве, описывающем память:

```
input [3:0] data_in,
output [5:0] addr_out
```

После этого можно записать объявление массива памяти, используя разрядность шины данных памяти. Для этого необходимо возвести двойку в степень, равную разрядности шины адреса разрабатываемого блока памяти минус единица. В данном случае количество ячеек равно **64**, поэтому **2** должна быть возведена в степень, равную **6 – 1**:

```
reg [3:0] ram[0:2**6-1];
```

Рассмотрим пример простого модуля однопортовой статической оперативной памяти. Схема подключения модуля памяти к кнопкам, светодиодам и переключателям на плате **DE10-Lite** приведена на рис. 7.15. Таблица истинности модуля памяти соответствует табл. 7.4.

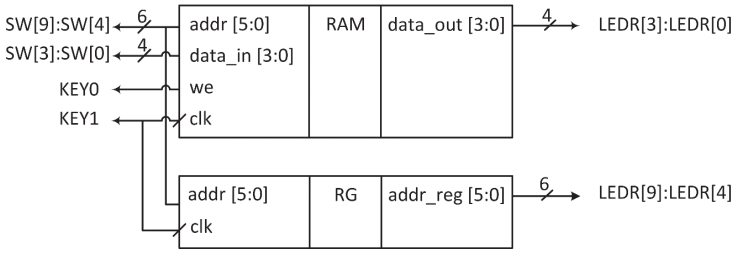


Рис. 7.15 Схема проекта

Таблица 7.4 Таблица истинности однопортового ОЗУ

data_in	we	clk	addr	data_out	Mode
X	0	0	X	Previous output data	Hold
X	0	Rising edge	Address	Output data	Read
X	1	0	X	Previous output data	Hold
Input data	1	Rising edge	Address	Input data	Write

```
module lab7_30
(
   input [3:0] data_in,
   input [5:0] addr,
   input we, clk,
   output [3:0] data_out,
   output [5:0] addr_out
);
   reg [3:0] ram[0:2**6-1];
   reg [5:0] addr_reg;
   wire w_we = ~ we;
   wire w_clk = ~ clk;
      always @ (posedge w_clk)
         begin
            if (w_we)
               //Write operation:
               ram[addr] <= data_in;
               addr_reg <= addr;
         end
         //Read operation:
         assign data_out = ram[addr_reg];
         assign addr_out = addr_reg;
endmodule
```

Листинг 7.5 Модуль простого ОЗУ

При описании модуля памяти необходимо учитывать, что на отладочной плате **DE10-Lite** при нажатии на кнопки генерируется сигнал низкого уровня. Поэтому эти сигналы должны быть инвертированы:

```
wire w_we = ~ we;
wire w_clk = ~ clk;
```

Создадим переменную, в которую будет записываться адрес с внешних линий:

```
reg [5:0] addr_reg;
```

Запись в регистр адреса производится внутри **always**-блока по переднему фронту тактовой частоты:

```
addr_reg <= addr;
```

Использование блока **always** для записи данных в регистр адреса и чтения из памяти является обязательным для того, чтобы компилятор (в нашем случае – пакет **Quartus Prime**) распознал блок памяти и поместил его во встроенный блок памяти микросхемы **ПЛИС**. Использование другого стиля описания блоков памяти может привести к тому, что модуль памяти не будет опознан компилятором именно как модуль памяти и будет реализован с помощью логических элементов. Еще не желательно использовать не поддерживаемые производителем сигналы сброса и другие сигналы управления памятью, поскольку в этом случае модуль памяти также будет реализован на логических элементах.

Работа с памятью описывается достаточно просто. При появлении переднего фронта тактового сигнала **w_clk** анализируется значение сигнала **w_we**. Если сигнал равен «**1**», то производится операция записи входных данных **data_in** по адресу, указанному на линиях **addr**:

```
always @ (posedge w_clk)
if (w_we)
   //Write operation:
   ram[addr] <= data_in;
```

Выходные данные определяются содержимым ячейки массива памяти, адрес которой задается регистром адреса:

```
assign data_out = ram[addr_reg];
```

В рассматриваемом примере выход регистра адреса специально выведен наружу, чтобы была возможность увидеть, какие значения имеют адрес и данные при работе с памятью:

```
assign addr_out = addr_reg;
```

Модуль памяти полностью описан и может быть синтезирован. Модули памяти большего размера отличаются от приведенного выше примера разрядностью шины адреса и данных. Для упрощения работы с таким модулем в дальнейшем сделаем его параметризированным. Это позволит при повторном использовании данного модуля изменять только разрядности шин адреса и данных, не меняя остальной код. Для этого определим два параметра:

```
#(parameter DATA_WIDTH=4, parameter ADDR_WIDTH=6)
```

В языке **Verilog** параметр – это символическое имя, которое представляет собой константу. Установим такие значения: **4** – для линий данных (**DATA_WIDTH**) и **6** – для линий адреса (**ADDR_WIDTH-1**).

```
input [(DATA_WIDTH-1):0] data_in,
input [(ADDR_WIDTH-1):0] addr,
input we, clk,
output [(DATA_WIDTH-1):0] data_out,
```

После определения параметров они будут видны во всем модуле. Поэтому следует использовать параметры для определения разрядностей внутренних сигналов:

```
reg [ADDR_WIDTH-1:0] addr_reg;
```

Для описания количества битов в одной ячейке запись будет выглядеть так: **[(DATA_WIDTH1):0]**. А для описания количества слов, хранящихся в памяти, запись должна быть такой: **[2 ** ADDR_WIDTH-1:0]**. Тогда декларация модуля памяти будет выглядеть следующим образом:

```
reg [DATA_WIDTH-1:0] ram[0:2**ADDR_WIDTH-1];
```

Для того чтобы видеть текущее состояние линий адреса, следует вывести их на светодиоды и дополнительный выход описать так:

```
output [(ADDR_WIDTH-1):0] addr_out
```

Полное описание модуля однопортовой памяти приведено в листинге 7.6:

```
module lab7_3
#(parameter DATA_WIDTH=4, parameter ADDR_WIDTH=6)
(
   input [(DATA_WIDTH-1):0] data_in,
   input [(ADDR_WIDTH-1):0] addr,
   input we, clk,
   output [(DATA_WIDTH-1):0] data_out,
   output [(ADDR_WIDTH-1):0] addr_out
);
   // Memory array and address register:
   reg [DATA_WIDTH-1:0] ram[0:2**ADDR_WIDTH-1];
   reg [ADDR_WIDTH-1:0] addr_reg;

   wire w_we = ~ we;
   wire w_clk = ~ clk;

   always @ (posedge w_clk)
      begin
         if (w_we) // write enable = 1
            ram[addr] <= data_in; // write input data to memory
         addr_reg <= addr; // latch address in internal register
      end
```

```
    assign data_out = ram[addr_reg]; // read data at latched address
    assign addr_out = addr_reg;
endmodule
```

Листинг 7.6 Модуль простого однопортового ОЗУ

Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 7.6 в файле **lab7_3.v**. Это файл верхнего уровня иерархии проекта. Откомпилируйте проект.

Результаты компиляции проекта в **RTL Viewer** приведены на рис. 7.16. Отметим, что был получен модуль памяти с дополнительным регистром адреса. В качестве основы для реализации памяти был взят блок **M9K**.

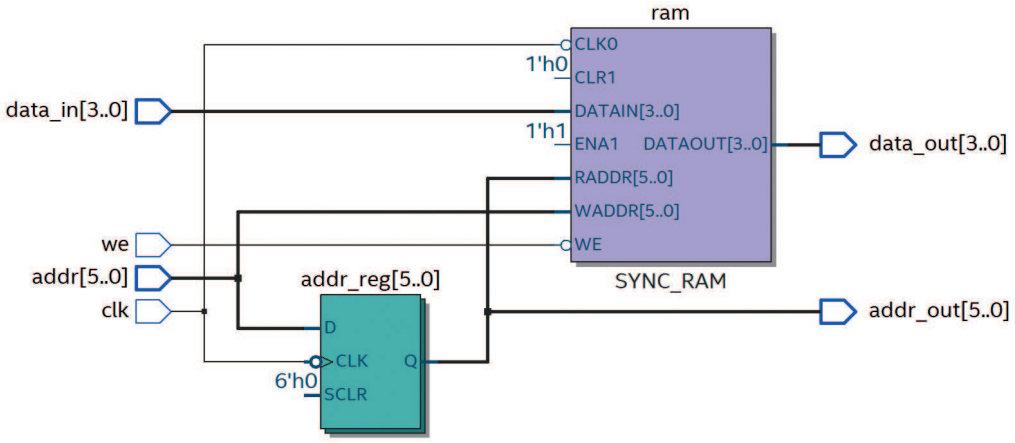


Рис. 7.16 Модуль простого однопортового ОЗУ в RTL Viewer

На рис. 7.17 приведена часть отчета компиляции, из которого видно, что для реализации разработанного модуля потребовалось использовать 256 бит памяти и 7 логических элементов.

Revision Name	lab7_1
Top-level Entity Name	lab7_1
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	7 / 49,760 (< 1 %)
Total registers	6
Total pins	22 / 360 (6 %)
Total virtual pins	0
Total memory bits	256 / 1,677,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 288 (0 %)

Рис. 7.17 Отчет компиляции однопортового ОЗУ

Назначьте выводы в соответствии с табл. 7.5.

Таблица 7.5 Соответствие выводов для проекта

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
addr[0]	Input	PIN_A12	3.3-V LVTTL
addr[1]	Input	PIN_B12	3.3-V LVTTL
addr[2]	Input	PIN_A13	3.3-V LVTTL
addr[3]	Input	PIN_A14	3.3-V LVTTL
addr[4]	Input	PIN_B14	3.3-V LVTTL
addr[5]	Input	PIN_F15	3.3-V LVTTL
addr_out[0]	Output	PIN_D13	3.3-V LVTTL
addr_out[1]	Output	PIN_C13	3.3-V LVTTL
addr_out[2]	Output	PIN_E14	3.3-V LVTTL
addr_out[3]	Output	PIN_D14	3.3-V LVTTL
addr_out[4]	Output	PIN_A11	3.3-V LVTTL
addr_out[5]	Output	PIN_B11	3.3-V LVTTL
clk	Input	PIN_A7	3.3 V Schmitt Trigger
data_in[0]	Input	PIN_C10	3.3-V LVTTL
data_in[1]	Input	PIN_C11	3.3-V LVTTL
data_in[2]	Input	PIN_D12	3.3-V LVTTL
data_in[3]	Input	PIN_C12	3.3-V LVTTL
data_out[0]	Output	PIN_A8	3.3-V LVTTL
data_out[1]	Output	PIN_A9	3.3-V LVTTL
data_out[2]	Output	PIN_A10	3.3-V LVTTL
data_out[3]	Output	PIN_B10	3.3-V LVTTL
we	Input	PIN_B8	3.3 V Schmitt Trigger

Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.

Проверьте работоспособность проекта. Для этого выполните следующие действия:

- используйте кнопки и микропереключатели для подачи сигналов на входы и проверяйте их значения с помощью светодиодов;
- проверьте таблицу истинности памяти.

Тестбенч и его описание

Первые строки тестбенча нужны для инициализации входных сигналов:

```
clk = 0;
we = 0;
addr = 6'b000000;
data_in = 4'b0000;
```

Опишем **процесс записи данных в память**. Для записи данных по определенному адресу необходимо установить значение адреса и данных, а значение сигнала разрешения записи – в «**0**».

```
#20; we = 0;
```

Такая пара значений будет выглядеть следующим образом:

```
addr = 6'b000001;
data_in = 4'b0001;
```

Для формирования подобных пар сигналов используется цикл, в котором изменяется сигнал адреса от **0** до **5** и последовательно увеличивается значение сигнала данных:

```
for (addr=0; addr < 6; addr= addr+1)
   begin
      #20;
      data_in = data_in + 1;
   end
```

Чтение данных из памяти. Для этого необходимо установить сигнал разрешения записи в **1**, а на линии адреса подать адрес ячейки, из которой будет осуществляться чтение данных. Адреса ячеек для чтения также можно формировать с помощью цикла:

```
#20; we = 1;
for (addr=0; addr < 6; addr= addr+1)
   #20;
```

Полный код тестбенча приведен в листинге 7.7:

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns/1ns
module testbench;
   // input and output test signals
   reg [3:0] data_in;
   reg [5:0] addr;
   reg we;
   reg clk;
   wire [3:0] data_out;
   wire [5:0] addr_out;

   // creating the instance of the module we want to test
   lab7_3 dut (data_in, addr, we, clk, data_out, addr_out);

   initial
      begin
         // set initial values of signal
         clk = 0;
```

```
        we = 0;
        addr = 6'b000000;
        data_in = 4'b0000;
        // write data in RAM
        #20; we = 0;
        for (addr=0; addr < 6; addr= addr+1)
           begin
              #20;
              data_in = data_in + 1;
           end
        // read data from RAM
        #20; we = 1;
        for (addr=0; addr < 6; addr= addr+1)
           #20;
     end

     // every 10 ns invert clk
     always #10 clk = ~clk;

     initial
        #300 $finish;

     // do at the beginning of the simulation
     // print signal values on every change
     initial
        $monitor("data_in=%b addr=%b we=%b clk=%b data_out=%b
           addr_out=%b", data_in, addr, we, clk, data_out, addr_out);
     // do at the beginning of the simulation
     initial
        $dumpvars;
endmodule
```

Листинг 7.7 Тестбенч для однопортового ОЗУ

На рис. 7.18 даны результаты симуляции ОЗУ в ModelSim.

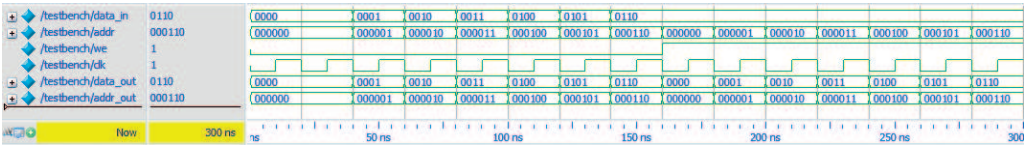


Рис. 7.18 Результаты симуляции ОЗУ в ModelSim

Дополнительное задание для самостоятельной работы

Увеличьте разрядность данных до **8**, а количество ячеек до **2048**. Представьте результаты моделирования и синтеза в **RTL** и объясните их. Загрузите результаты в **Technology Map Viewer** и дайте описание того, что получится.

7.7 Однопортовое ПЗУ

Описание однопортового ПЗУ на языке Verilog

Описание однопортового **ПЗУ** очень похоже на описание однопортового **ОЗУ**, за исключением того, что отсутствует операция записи информации в память. Поэтому в таком виде памяти нет возможности записи данных. В **ПЛИС ПЗУ** реализуется на тех же самых ресурсах, что и **ОЗУ**, т. е. выполняется эмуляция **ПЗУ**. Во время конфигурирования **ПЛИС** в **ПЗУ** производится запись данных. Реализация **ПЗУ** в микросхемах **ASIC** производится так же, как и модуля постоянной памяти.

Чтобы описать **ПЗУ**, точно так же как и для **ОЗУ**, необходимо выполнить декларацию массива, но в этот массив нужно записать информацию, которая хранится в **ПЗУ**.

```
input clk,
input [(ADDR_WIDTH-1):0] addr,
output [(DATA_WIDTH-1):0] data_out,
output [(ADDR_WIDTH-1):0] addr_out
```

Так же, как и в случае с **ОЗУ**, объявим дополнительный выход **addr_out**, который необходим для контроля состояния адресных входов на отладочной плате.

Для декларации массива **ПЗУ** используем следующую конструкцию:

```
reg [DATA_WIDTH-1:0] rom[0:2**ADDR_WIDTH-1];
```

Для первоначальной инициализации памяти необходимо иметь текстовый файл, который будет содержать прошивку ПЗУ. Для подключения его к коду используют команды **$readmemb** или **$readmemh**.

Текст в файле прошивки может содержать пробелы, комментарии. Для чтения данных из файла, записанных в двоичном коде, требуется системная функция **$readmemb**, а для чтения данных из файла, записанных в шестнадцатеричном коде, – **$readmemh**. Эти команды имеют два аргумента: имя текстового файла с данными и имя массива, в который необходимо записать эти данные.

```
initial begin
   $readmemh("../rom.txt",rom);
end
```

Данные будут записываться в массив словами слева направо, начиная с адреса **start_addr** или же начиная с левой границы массива. Последний вариант используется по умолчанию. Необходимый адрес может быть определен в файле с помощью конструкции **@hhhhhh**, где «**h**» обозначает число в шестнадцатеричной системе счисления. Использование пробела для разделения цифр в адресе не разрешается.

Блок памяти содержит всего 6 линий адреса, поэтому для его кодирования необходимы всего две цифры. Начальное значение зададим равным нулю: **@00**.

Содержание файла **rom.txt** приведено в листинге 7.8.

```
@00
1 3 7 F F E C 8 8 C E F F 7 3 1
1 3 7 F F E C 8 8 C E F F 7 3 1
1 3 7 F F E C 8 8 C E F F 7 3 1
1 3 7 F F E C 8 8 C E F F 7 3 1
```

Листинг 7.8 Содержание файла «rom.txt»

Опишем память размером 64×4. Схематическое изображение памяти, подключенной к выводам отладочной платы, приведено на рис. 7.19. Таблица истинности **ПЗУ** приведена в табл. 7.6. Код, описывающий **ПЗУ**, находится в файле **Lab7_2.v** (листинг 7.9).

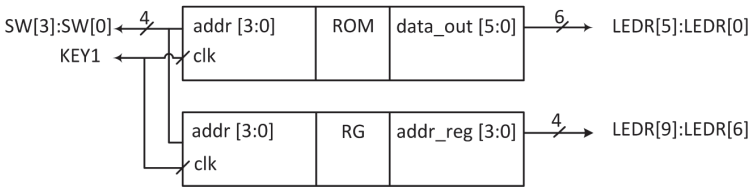


Рис. 7.19 Схема подключения ПЗУ к периферии отладочной платы DE10-Lite

Таблица 7.6 Таблица истинности ПЗУ

clk	addr	data_out	Режим
0	X	Предыдущие выходные данные	Хранение
Передний фронт	Адрес ячейки	Выходные данные	Чтение

Полный листинг с описанием ПЗУ приведен ниже.

```
module lab7_4
#(parameter DATA_WIDTH=4, parameter ADDR_WIDTH=6)
( input clk,
   input [(ADDR_WIDTH-1):0] addr,
   output [(DATA_WIDTH-1):0] data_out,
   output [(ADDR_WIDTH-1):0] addr_out
);
   // ROM array
   reg [DATA_WIDTH-1:0] rom[0:2**ADDR_WIDTH-1];
   // invert chip select signal and clk button
   wire w_clk = ~ clk;
   reg [ADDR_WIDTH-1:0] addr_reg;

   // read ROM content from file
   initial begin
      $readmemh("../rom.txt",rom);
   end

   always @ (posedge w_clk)
   begin
```

```
      addr_reg <= addr;
   end

   assign data_out = rom[addr_reg];
   assign addr_out = addr_reg;
endmodule
```

Листинг 7.9 Реализация ПЗУ

Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 7.9 в файле **lab7_4.v**. Это файл верхнего уровня иерархии проекта. Откомпилируйте проект.

Результаты компиляции проекта в **RTL Viewer** приведены на рис. 7.20. В результате компиляции был получен модуль памяти с дополнительным регистром адреса.

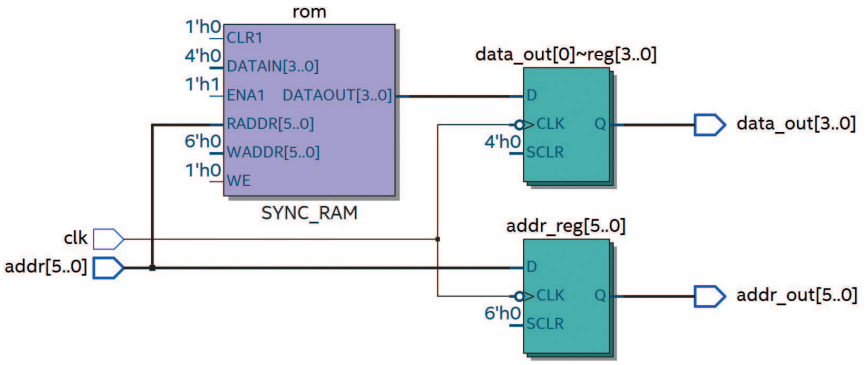


Рис. 7.20 Результат компиляции ПЗУ в RTL Viewer

Реализация **ПЗУ** в микросхемах **ПЛИС** производится на блоках **ОЗУ**, что можно отследить в отчете компиляции и в сообщениях **TCL-консоли**. Для реализации постоянной памяти использован блок **SYNC_RAM**, в котором отключены линии входных данных и адреса записи.

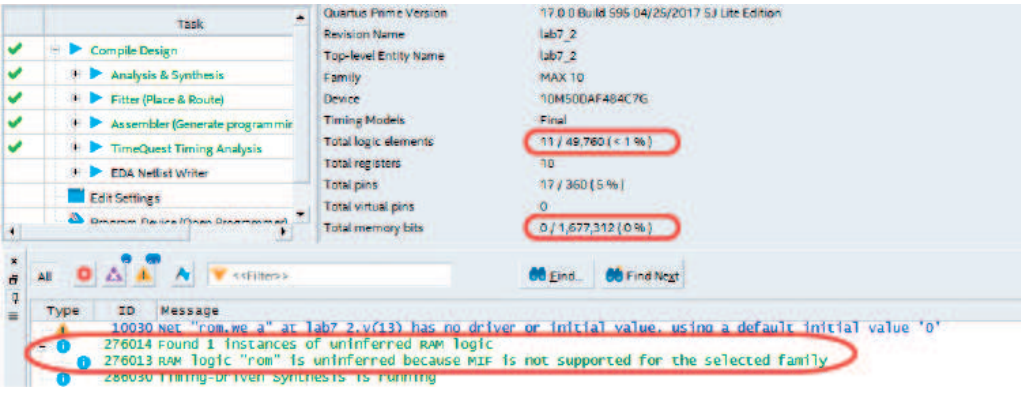


Рис. 7.21 Отчет компиляции модуля ПЗУ

Хотя результат в **RTL Viewer** показывает, что компилятор распознал разработанный модуль памяти, но результаты компиляции, которые содержатся на рис. 7.21, говорят о том, что для реализации модуля памяти не было использовано ни одного бита встроенной памяти, а вся память была реализована на логических элементах, которых потребовалось всего **11**. В отчете компиляции (**TCL-консоль**), приведенном на том же рисунке, видно, что память не была реализована, т. к. использование ***.mif** (**Memory Initialization File**) не поддерживается в данном семействе микросхем. Это значит, что при применении микросхем семейства **MAX10** нет возможности записывать в **ОЗУ** какие-то предварительные данные. Это подтверждается документом **MAX 10 Embedded Memory User Guide**.

Назначьте выводы в соответствии с табл. 7.7.

Таблица 7.7 Соответствие выводов для проекта lab 7_2

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
addr[3]	Input	PIN_C12	3.3-V LVTTL
addr[2]	Input	PIN_D12	3.3-V LVTTL
addr[1]	Input	PIN_C11	3.3-V LVTTL
addr[0]	Input	PIN_C10	3.3-V LVTTL
clk	Input	PIN_A7	3.3 V Schmitt Trigger
data_out[5]	Output	PIN_A8	3.3-V LVTTL
data_out[4]	Output	PIN_A9	3.3-V LVTTL
data_out[3]	Output	PIN_A10	3.3-V LVTTL
data_out[2]	Output	PIN_B10	3.3-V LVTTL
data_out[1]	Output	PIN_D13	3.3-V LVTTL
data_out[0]	Output	PIN_C13	3.3-V LVTTL
addr_out[3]	Output	PIN_E14	3.3-V LVTTL
addr_out[2]	Output	PIN_D14	3.3-V LVTTL
addr_out[1]	Output	PIN_A11	3.3-V LVTTL
addr_out[0]	Output	PIN_B11	3.3-V LVTTL

Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.

Проверьте работоспособность проекта. Для этого выполните следующие действия:

- используйте кнопки и микропереключатели для подачи сигналов на входы и проверьте их значения с помощью светодиодов;
- проверьте таблицу истинности памяти.

Дополнительное задание для самостоятельной работы

Самостоятельно изучите решение описанной выше проблемы (например, в обсуждении на форуме **Electronix**[1]). Реализуйте **ПЗУ** на встроенной памяти.

Тестбенч для модуля ПЗУ

Полное описание тестбенча приведено в листинге 7.10.

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns/1ns
module testbench;
   reg clk;
   reg [5:0] addr;
   wire [3:0] data_out;
   wire [5:0] addr_out;
   integer i;

   // creating the instance of the module we want to test
   lab7_4 dut (clk, addr, data_out, addr_out);

initial
   begin
      // set initial values of signal
      clk = 0;
      #20; addr = 6'b000000; // set address signals value
      for (i=0 ; i<=16; i=i+1)
         begin
            #20; addr = addr+6'b000001; // set address signals value
         end
   end

   // every 10 ns invert clk
   always #10 clk = ~clk;
      initial
         #380 $finish;
   // do at the beginning of the simulation
   // print signal values on every change
   initial
      $monitor("clk=%b addr=%b data_out=%b addr_out=%b",
         clk, addr, data_out, addr_out);
   // do at the beginning of the simulation
   initial
      $dumpvars; // iVerilog dump init
endmodule
```

Листинг 7.10 Тестбенч для ПЗУ

1 Форум electronix.ru. Инициализация ROM в MAX10 compact features. URL: https://electronix.ru/forum/index.php?showtopic=141992&hl=max10+rom.

Симуляция модуля однопортового ПЗУ

На рис. 7.22 приведены результаты симуляции разработанного **ПЗУ** в **ModelSim**.

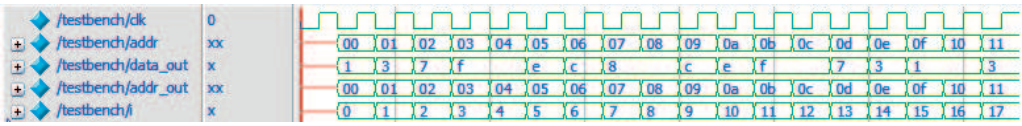


Рис. 7.22 Результаты симуляции ПЗУ в ModelSim

Дополнительное задание для самостоятельной работы

Измените целевую микросхему на любой чип из семейства **Cyclone V**. Представьте результаты компиляции и синтеза в **RTL** и объясните их. Загрузите **Technology Map Viewer** и опишите полученные результаты.

7.8 Двухпортовая память

Описание двухпортовой памяти

Рассмотрим простую двухпортовую память с одним портом для записи и одним портом для чтения. Порт для записи с описанием входных данных может быть представлен следующим образом:

```
input [(DATA_WIDTH-1):0] data_in,
input [(ADDR_WIDTH-1):0] read_addr,
```

Порт для чтения с описанием выходных данных:

```
input [(ADDR_WIDTH-1):0] write_addr,
```

Выходной порт данных:

```
output reg [(DATA_WIDTH-1):0] data_out
```

Входы тактирования и разрешения записи:

```
input we,
input clk,
```

Так же, как и в оперативной памяти, объявляем массив для данных:

```
reg [DATA_WIDTH-1:0] dp_ram [0:2**ADDR_WIDTH-1];
```

Работа модуля происходит по активному фронту сигнала **clk**. Если есть сигнал разрешения записи, то происходит запись в переменную **dp_ram** по адресу, который задается сигналом **write_addr**. При каждом переднем фронте тактовой частоты происходит чтение ячейки памяти по адресу, который задается сигналом **read_addr**.

Режимы работы двухпортовой памяти приведены в табл. 7.8. Описание на языке **Verilog** – в модуле **dual_port_RAM** из листинга 7.11.

```
module hexto7segment
( input [3:0] in_hex,
```

```
   output reg [6:0] out_7seg
);
   always @in_hex
      case (in_hex)
         4'b0000 : //Digit 0
            out_7seg = 7'b1000000;
         4'b0001 : //Digit 1
            out_7seg = 7'b1111001 ;
         4'b0010 : //Digit 2
            out_7seg = 7'b0100100 ;
         4'b0011 : //Digit 3
            out_7seg = 7'b0110000 ;
         4'b0100 : //Digit 4
            out_7seg = 7'b0011001 ;
         4'b0101 : //Digit 5
            out_7seg = 7'b0010010 ;
         4'b0110 : //Digit 6
            out_7seg = 7'b0000010 ;
         4'b0111 : //Digit 7
            out_7seg = 7'b1111000;
         4'b1000 : //Digit 8
            out_7seg = 7'b0000000;
         4'b1001 : //Digit 9
            out_7seg = 7'b0011000 ;
         4'b1010 : //Digit A
            out_7seg = 7'b0001000 ;
         4'b1011 : //Digit B
            out_7seg = 7'b0000011;
         4'b1100 : //Digit C
            out_7seg = 7'b1000110 ;
         4'b1101 : //Digit D
            out_7seg = 7'b0100001 ;
         4'b1110 : //Digit E
            out_7seg = 7'b0000110 ;
         4'b1111 : //Digit F
            out_7seg = 7'b0001110 ;
   endcase
endmodule

module dc2
(  input [1:0] a,
   output reg [3:0] dc_out
);
   always @a
      case (a)
         2'b00 : dc_out = 4'b0001;
```

```
        2'b01 : dc_out = 4'b0010;
        2'b10 : dc_out = 4'b0100;
        2'b11 : dc_out = 4'b1000;
     endcase
endmodule

module register
#( parameter SIZE = 4)
(  input ena,
   input clk,
   input rst_n,
   input [SIZE - 1 : 0] d,
   output reg [SIZE - 1 : 0] q
);
   always @ (posedge clk or negedge rst_n)
      if (! rst_n)
         q <= {SIZE {1'b0}};
      else if (ena)
         q <= d;
endmodule

module dual_port_RAM
#( parameter DATA_WIDTH=8, parameter ADDR_WIDTH=4)
(
   input we,
   input clk,
   // Read port:
   input [(DATA_WIDTH-1):0] data_in,
   input [(ADDR_WIDTH-1):0] read_addr,
   // Write port:
   input [(ADDR_WIDTH-1):0] write_addr,
   output reg [(DATA_WIDTH-1):0] data_out
);
   // Declare the array variable
   reg [DATA_WIDTH-1:0] dp_ram [0:2**ADDR_WIDTH-1];
   always @ (posedge clk)
      begin
         // Write
         if (we)
            dp_ram [write_addr] <= data_in;
         // Read
         data_out <= dp_ram [read_addr];
      end
endmodule

module lab7_5
#( parameter DATA_WIDTH=8,
```

```
   parameter ADDR_WIDTH=4
)
(  input clk,
   input rst_n,
   input [9:0] SW,
   output [7:0] HEX0,
   output [7:0] HEX1,
   output [7:0] HEX2,
   output [7:0] HEX3,
   output [7:0] HEX4,
   output [7:0] HEX5
);
   wire [3:0] w_we; //DC output for write enable
   wire [(DATA_WIDTH-1):0] data_in_reg; //data_in register output
   wire [(ADDR_WIDTH-1):0] read_addr_reg; //read_addr register output
   wire [(ADDR_WIDTH-1):0] write_addr_reg; //write_addr register output
   wire [(DATA_WIDTH-1):0] data_out; //memory output

   dc2 dc2in4 (.a(SW[9:8]), .dc_out(w_we[3:0]));
   assign HEX0[7] = ~w_we[0];
   assign HEX1[7] = ~w_we[0];
   assign HEX2[7] = ~w_we[3];
   assign HEX3[7] = ~w_we[3];
   assign HEX4[7] = ~w_we[1];
   assign HEX5[7] = ~w_we[2];

   register #(8) register_data_in (.ena(w_we[0]), .clk(clk),
      .rst_n(rst_n), .d(SW[7:0]), .q(data_in_reg));
   register #(4) register_read_addr (.ena(w_we[1]), .clk(clk),
      .rst_n(rst_n), .d(SW[3:0]), .q(read_addr_reg));
   register #(4) register_write_addr (.ena(w_we[2]), .clk(clk),
      .rst_n(rst_n), .d(SW[3:0]), .q(write_addr_reg));

   hexto7segment hexto7segment_data_in0 (.in_hex(data_in_reg[3:0]),
      .out_7seg(HEX0[6:0]));
   hexto7segment hexto7segment_data_in1 (.in_hex(data_in_reg[7:4]),
      .out_7seg(HEX1[6:0]));
   hexto7segment hexto7segment_data_out0 (.in_hex(data_out[3:0]),
      .out_7seg(HEX2[6:0]));
   hexto7segment hexto7segment_data_out1 (.in_hex(data_out[7:4]),
      .out_7seg(HEX3[6:0]));
   hexto7segment hexto7segment_read_addr (.in_hex(read_addr_reg),
      .out_7seg(HEX4[6:0]));
   hexto7segment hexto7segment_write_addr (.in_hex(write_addr_reg),
      .out_7seg(HEX5[6:0]));

      dual_port_RAM #(8,4) dual_port_RAM
         (  .we(w_we[3]),
```

```
            .clk(clk),
            .data_in(data_in_reg),
            .read_addr(read_addr_reg),
            .write_addr(write_addr_reg),
            .data_out(data_out)
        );
endmodule
```

Листинг 7.11 Код для исследования модуля двухпортового ОЗУ

Таблица 7.8 Режимы работы простой двухпортовой памяти

data_in	we	write_addr	read_addr	data_out	Mode
Input data	0	Write address	Read address	Output data	Read
Input data	1	Write address	Read address	Input data	Write

Как можно убедиться, реализация двухпортовой памяти не очень сложна. Для проверки ее работоспособности необходимо создать схему, которая позволит в ручном режиме проверить все режимы работы двухпортовой памяти и усвоить логику работы такого вида памяти. Схема проверки двухпортовой памяти приведена на рис. 7.23.

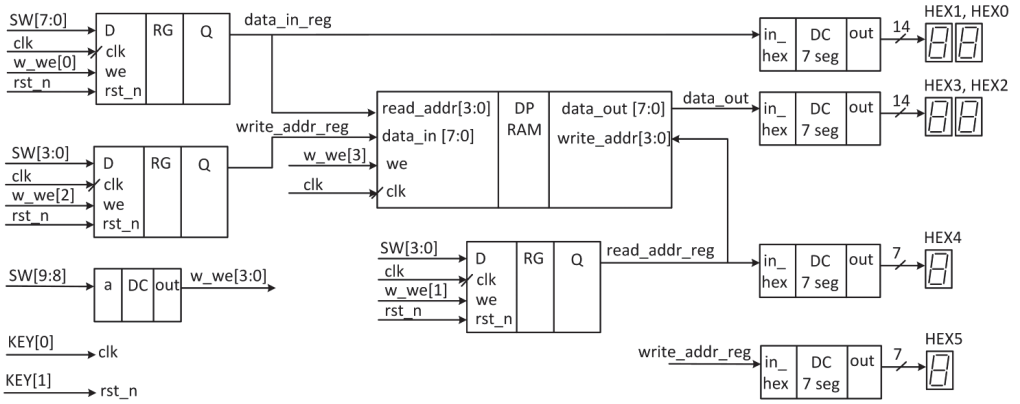


Рис. 7.23 Схема для проверки двухпортового ОЗУ на отладочной плате DE10-Lite

Для проверки работы памяти необходимо последовательно проанализировать все режимы работы памяти:

- запись в память;
- чтение из памяти;
- чтение во время записи в случае, когда адреса записи и чтения различны;
- чтение во время записи в случае, когда адреса записи и чтения одинаковы.

Схема работает следующим образом. Для проведения работы схемы во всех вышеописанных режимах необходимо, чтобы на всех входах памяти были нужные

сигналы. Для этого все входные порты снабжены регистрами, в которые сначала происходит запись нужной информации, а потом производится цикл обмена с памятью.

Для записи информации в любой из регистров необходимо на переключателях **SW[7:0]** выставить нужную информацию, на переключателях **SW[9:8]** выставить комбинацию для выбора нужного устройства и нажать кнопку **KEY0** для подачи тактового сигнала. При выборе устройства переключателями **SW[9:8]** загораются сегменты на индикаторах, которые соответствуют выбранному устройству. Назначение переключателей и их режимы работы указаны в табл. 7.9.

Табл. 7.9 Режимы работы проекта для тестирования двухпортовой памяти

Устройство	Разрядность	Переключатели данных SW[7:0]	Значение переключателей устройства SW[9:8]	Индикация
Регистр входных данных	8	SW[7:0]	00	HEX1, HEX0
Регистр адреса чтения	4	SW[3:0]	01	HEX4
Регистр адреса записи	4	SW[3:0]	10	HEX5
Память	8	–	11	HEX3, HEX2

Дешифратор семисегментного индикатора описан в модуле **hexto7segment**, дешифратор 2в4 – в модуле – **dc2**, регистр – в модуле **register** (листинг 7.11). Реализация этих элементов уже изучалась в предыдущих главах.

Для управления сегментами семисегментных индикаторов задействованы выходы дешифратора 2в4, который управляет сигналами разрешения записи регистров.

Тело основного модуля содержит подключение инстансов элементов схемы и соединение их между собой.

Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 7.11 в файле **lab7_5.v**. Это файл верхнего уровня иерархии для проекта. Откомпилируйте проект.

Результаты компиляции проекта в **RTL Viewer** приведены на рис. 7.24. Как и в случае, иллюстрируемом на рис. 7.23, компилятор для реализации компонентов памяти использовал блок **M9K**, встроенный в микросхему **MAX10**.

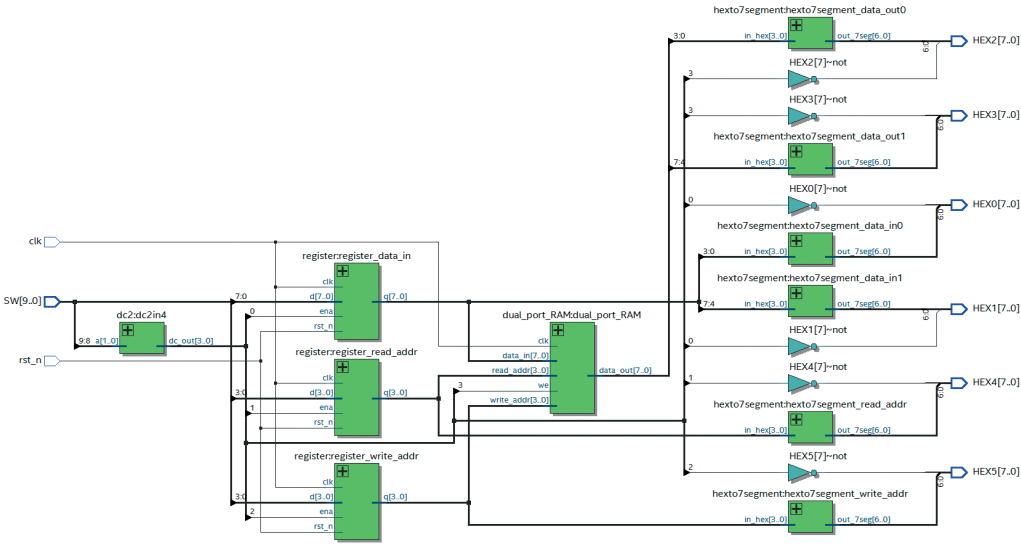


Рис. 7.24 Результат компиляции двухпортового ОЗУ в RTL Viewer

В проекте назначьте выводы в соответствии с табл. 7.10.

Таблица 7.10 Соответствие выводов для проекта lab 7_3

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
HEX0[7]	Output	PIN_D15	3.3-V LVTTL
HEX0[6]	Output	PIN_C17	3.3-V LVTTL
HEX0[5]	Output	PIN_D17	3.3-V LVTTL
HEX0[4]	Output	PIN_E16	3.3-V LVTTL
HEX0[3]	Output	PIN_C16	3.3-V LVTTL
HEX0[2]	Output	PIN_C15	3.3-V LVTTL
HEX0[1]	Output	PIN_E15	3.3-V LVTTL
HEX0[0]	Output	PIN_C14	3.3-V LVTTL
HEX1[7]	Output	PIN_A16	3.3-V LVTTL
HEX1[6]	Output	PIN_B17	3.3-V LVTTL
HEX1[5]	Output	PIN_A18	3.3-V LVTTL
HEX1[4]	Output	PIN_A17	3.3-V LVTTL
HEX1[3]	Output	PIN_B16	3.3-V LVTTL
HEX1[2]	Output	PIN_E18	3.3-V LVTTL
HEX1[1]	Output	PIN_D18	3.3-V LVTTL
HEX1[0]	Output	PIN_C18	3.3-V LVTTL
HEX2[7]	Output	PIN_A19	3.3-V LVTTL

Продолжение табл. 7.10

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
HEX2[6]	Output	PIN_B22	3.3-V LVTTL
HEX2[5]	Output	PIN_C22	3.3-V LVTTL
HEX2[4]	Output	PIN_B21	3.3-V LVTTL
HEX2[3]	Output	PIN_A21	3.3-V LVTTL
HEX2[2]	Output	PIN_B19	3.3-V LVTTL
HEX2[1]	Output	PIN_A20	3.3-V LVTTL
HEX2[0]	Output	PIN_B20	3.3-V LVTTL
HEX3[7]	Output	PIN_D22	3.3-V LVTTL
HEX3[6]	Output	PIN_E17	3.3-V LVTTL
HEX3[5]	Output	PIN_D19	3.3-V LVTTL
HEX3[4]	Output	PIN_C20	3.3-V LVTTL
HEX3[3]	Output	PIN_C19	3.3-V LVTTL
HEX3[2]	Output	PIN_E21	3.3-V LVTTL
HEX3[1]	Output	PIN_E22	3.3-V LVTTL
HEX3[0]	Output	PIN_F21	3.3-V LVTTL
HEX4[7]	Output	PIN_F17	3.3-V LVTTL
HEX4[6]	Output	PIN_F20	3.3-V LVTTL
HEX4[5]	Output	PIN_F19	3.3-V LVTTL
HEX4[4]	Output	PIN_H19	3.3-V LVTTL
HEX4[3]	Output	PIN_J18	3.3-V LVTTL
HEX4[2]	Output	PIN_E19	3.3-V LVTTL
HEX4[1]	Output	PIN_E20	3.3-V LVTTL
HEX4[0]	Output	PIN_F18	3.3-V LVTTL
HEX5[7]	Output	PIN_L19	3.3-V LVTTL
HEX5[6]	Output	PIN_N20	3.3-V LVTTL
HEX5[5]	Output	PIN_N19	3.3-V LVTTL
HEX5[4]	Output	PIN_M20	3.3-V LVTTL
HEX5[3]	Output	PIN_N18	3.3-V LVTTL
HEX5[2]	Output	PIN_L18	3.3-V LVTTL
HEX5[1]	Output	PIN_K20	3.3-V LVTTL
HEX5[0]	Output	PIN_J20	3.3-V LVTTL
SW[9]	Input	PIN_F15	3.3-V LVTTL
SW[8]	Input	PIN_B14	3.3-V LVTTL
SW[7]	Input	PIN_A14	3.3-V LVTTL
SW[6]	Input	PIN_A13	3.3-V LVTTL
SW[5]	Input	PIN_B12	3.3-V LVTTL

Окончание табл. 7.10

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
SW[4]	Input	PIN_A12	3.3-V LVTTL
SW[3]	Input	PIN_C12	3.3-V LVTTL
SW[2]	Input	PIN_D12	3.3-V LVTTL
SW[1]	Input	PIN_C11	3.3-V LVTTL
SW[0]	Input	PIN_C10	3.3-V LVTTL
clk	Input	PIN_B8	3.3 V Schmitt Trigger
rst_n	Input	PIN_A7	3.3 V Schmitt Trigger

Конфигурирование ПЛИС и проверка работы блока памяти

Выполните следующую последовательность действий:

1. Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.
2. Проверьте работоспособность проекта.
3. После загрузки схемы необходимо нажать кнопку **KEY1** для сброса всей системы. Далее можно начинать работу со стендом.
4. Рассмотрите все варианты работы со стендом:
   - для проведения **записи в память** необходимо сначала последовательно провести запись в регистры входных данных и адреса записи, а только затем провести цикл обмена с памятью. То есть для записи одного байта в память нужно провести три операции;
   - для проведения **чтения из памяти** необходимо сначала последовательно провести запись в регистр адреса чтения, а только затем провести цикл обмена с памятью. То есть для чтения одного байта из памяти следует провести две операции;
   - для проведения **чтения во время записи** необходимо сначала последовательно провести запись в регистры входных данных, адреса чтения и адреса записи. Таким образом, у нас все готово для проведения подобного обращения к памяти, а только затем нужно провести цикл обмена с памятью. То есть для записи одного байта в память следует провести три операции.
5. Используйте кнопки и микропереключатели для подачи сигналов на входы и проверьте их значения с помощью индикаторов.
6. Проверьте работу памяти в соответствии с таблицей истинности.

Тестбенч и его описание

Для удобства наблюдения сигналов в рассматриваемом примере тестбенча внутренние сигналы выведены наружу. Для этого при объявлении портов тестбенча

описаны не только входы и выходы тестируемого модуля, но и дополнительные сигналы. При объявлении портов сигналы с **10** переключателей, используемых в схеме, разбиты на две части – вход дешифратора **DC[1:0]** и входы данных регистров **SW[7:0]**.

```
reg clk;
reg rst_n;
reg [1:0] DC;
reg [7:0] SW;
wire [7:0] data_out;
wire [7:0] HEX0;
wire [7:0] HEX1;
wire [7:0] HEX2;
wire [7:0] HEX3;
wire [7:0] HEX4;
wire [7:0] HEX5;
```

Дополнительными портами являются четыре выхода дешифратора разрешения записи:

```
wire [3:0] w_we;
```

Также к ним относятся выходы всех трех внутренних регистров – адресов чтения и записи и входных данных:

```
wire [7:0] data_in_reg;
wire [3:0] read_addr_reg;
wire [3:0] write_addr_reg;
```

Присвоение сигналов дополнительным портам производится обычным непрерывным присваиванием. В левой части записываем внешний порт, а в правой внутренний узел схемы в формате **dut.имя_узла**, где **dut** – тестируемое устройство, объявленное в строке

```
lab7_5 dut (clk, rst_n, {DC,SW}, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);
```

а **имя_узла** – имя узла внутри тестируемого модуля.

```
assign w_we = dut.w_we;
assign data_in_reg = dut.data_in_reg;
assign data_out = dut.data_out;
assign read_addr_reg = dut.read_addr_reg;
assign write_addr_reg = dut.write_addr_reg;
```

Как было сделано в более ранних примерах, в первых строках тестбенча производится **инициализация входных сигналов**. При этом выбирается регистр данных и устанавливаются переключателями **SW** данные для записи **00001111**, после чего происходит начальный сброс системы:

```
clk = 1;
DC = 2'b00;
```

```
SW[7:0] = 8'b00001111;
rst_n = 0;
#10; rst_n = 1;
```

После задержки в **50 нс** на входах регистров **SW** устанавливается число **00000000** для записи в регистр адреса записи **write_addr**:

```
#50;
SW[7:0] = 8'b00000000;
```

Далее производится **три последовательные записи в память**. Для этого выполняется такая последовательность действий:

- выбирается переключателями **DC = 2'b10** регистр записи, и в следующем такте тактовой частоты проводится в него запись;
- выбирается переключателями **DC = 2'b11** модуль памяти, и в следующем такте тактовой частоты проводится запись в него;
- увеличивается адрес на единицу.

```
repeat(3)
   begin
      DC = 2'b10; //select write_addr reg
      #40; //write addr to write_addr reg
      DC = 2'b11; //select RAM
      #40; //write data to RAM
      SW = SW+8'b00000001; //set new write_addr value
   end
```

Опишем операцию **чтения данных**. Для этого на входах регистров **SW** устанавливается число **00000000** для записи в регистр адреса записи **read_addr**. Также с помощью переключателей **DC = 2'b10** выбирается регистр чтения:

```
SW[7:0] = 8'b00000000;
DC = 2'b01; //select read_addr reg
```

Далее производится три последовательные операции чтения из памяти. Для этого в каждом такте увеличивается адрес в регистре адреса на единицу.

```
repeat(3)
   begin
      #40; //read data from RAM
      SW = SW+8'b00000001; //set new write_addr value
   end
```

Последний рассматриваемый режим работы – **чтение во время записи**. В нем сначала производится запись новых данных **10101010** в регистр данных:

```
SW[7:0] = 8'b10101010;
DC = 2'b00; //set new data_in
#40;
```

Далее устанавливается адрес ячейки **0001** и тремя последовательными тактами производится запись в регистры адреса записи и чтения, а затем и в саму память:

```
SW[7:0] = 8'b00000001;
DC = 2'b10; //set new write_addr
#40; DC = 2'b01; //set new read_addr
#40; DC = 2'b11; // write data in DP RAM
```

Полное описание тестбенча приведено в листинге 7.12.

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns/1ns
module testbench;
   // input and output test signals
   reg clk;
   reg rst_n;
   reg [1:0] DC;
   reg [7:0] SW;
   wire [3:0] w_we;
   wire [7:0] data_in_reg;
   wire [3:0] read_addr_reg;
   wire [3:0] write_addr_reg;
   wire [7:0] data_out;
   wire [7:0] HEX0;
   wire [7:0] HEX1;
   wire [7:0] HEX2;
   wire [7:0] HEX3;
   wire [7:0] HEX4;
   wire [7:0] HEX5;

// creating the instance of the module we want to test
lab7_5 dut (clk, rst_n, {DC,SW}, HEX0, HEX1, HEX2, HEX3, HEX4, HEX5);

initial
   begin
      // set inital values of signal
      clk = 1;
      DC = 2'b00;
      SW[7:0] = 8'b00001111;
      rst_n = 0;
      #10; rst_n = 1;
      // write data in write_addr register
      #50;
      SW[7:0] = 8'b00000000;

      //write data to RAM and read-during-write
      repeat(3)
      begin
```

```
      DC = 2'b10; //select write_addr reg
      #40; //write addr to write_addr reg
      DC = 2'b11; //select RAM
      #40; //write data to RAM
      SW = SW+8'b00000001; //set new write_addr value
   end

   // read data from DP RAM
   SW[7:0] = 8'b00000000;
   DC = 2'b01; //select read_addr reg
   repeat(3)
   begin
      #40; //read data from RAM
      SW = SW+8'b00000001; //set new write_addr value
   end

   //read-during-write at the same addr
   SW[7:0] = 8'b10101010;
   DC = 2'b00; //set new data_in
   #40;

   SW[7:0] = 8'b00000001;
   DC = 2'b10; //set new write_addr
   #40; DC = 2'b01; //set new read_addr
   #40; DC = 2'b11; // write data in DP RAM
end

assign w_we = dut.w_we;
assign data_in_reg = dut.data_in_reg;
assign data_out = dut.data_out;
assign read_addr_reg = dut.read_addr_reg;
assign write_addr_reg = dut.write_addr_reg;

//every 20 ns invert clk
always #20 clk = ~clk;
initial
   #760 $finish;

// do at the beginning of the simulation
// print signal values on every change
initial
   $monitor("clk=%b rst_n=%b DC=%b SW[7:0]=%h HEX0=%h HEX1=%h
   HEX2=%h HEX3=%h HEX4=%h HEX5=%h w_we=%b data_in_reg=%h
   read_addr_reg=%h write_addr_reg=%h data_out=%h", clk, rst_n,
   DC, SW[7:0], HEX0, HEX1, HEX2, HEX3, HEX4, HEX5, w_we,
   data_in_reg, read_addr_reg, write_addr_reg, data_out );

// do at the beginning of the simulation
initial
```

```
    $dumpvars; //iVerilog dump init
endmodule
```

Листинг 7.12 Тестбенч двухпортовой памяти

Симуляция двухпортовой памяти

Так как работа двухпортового **ОЗУ** достаточно сложна, то результаты симуляции разделены на три части. На рис. 7.25 приведена запись в память. На рис. 7.26 дано чтение данных из памяти. На рис. 7.27 представлен режим чтения во время записи.

Перед тем как начать работать с памятью, необходимо произвести сброс регистров путем записи числа **00001111** в регистр входных данных (**data_in**). Запись в память занимает два такта: сначала производится запись в регистр адреса записи (**write_addr**), потом в следующем такте производится запись данных из регистра **data_in** в память по адресу, находящемуся в регистре **write_addr**.

Во время записи второго значения в регистр **write_addr** происходит чтение информации из памяти. Адрес для чтения находится в регистре **read_addr**. В этом регистре находится число **0000**. В ячейке по этому адресу уже есть значение **00001111**, которое и выводится на выходные линии (рис. 7.25).

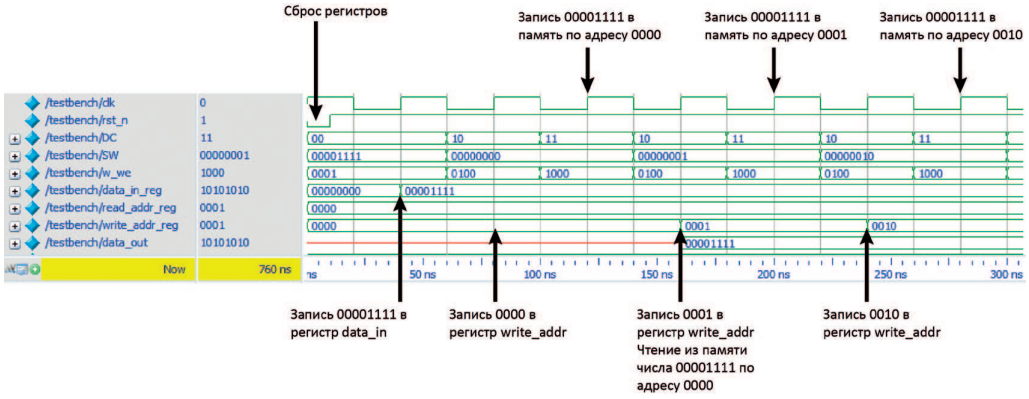


Рис. 7.25 Запись в двухпортовое ОЗУ

Для проведения чтения необходимо сначала записать значение в регистр адреса чтения **read_addr**, а затем произвести чтение данных. Так как работа регистра адреса чтения и модуля памяти происходит по одному тактовому сигналу, то можно проводить одновременно запись нового значения в регистр адреса чтения и чтение данных из памяти (рис. 7.26).

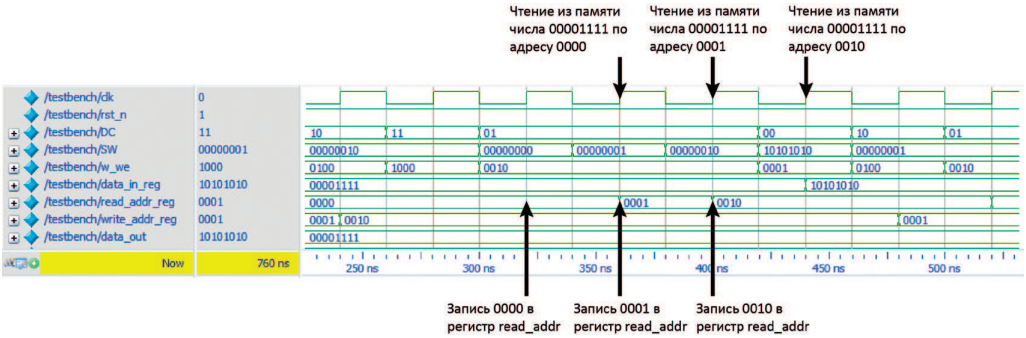


Рис. 7.26 Чтение двухпортового ОЗУ

Чтение во время записи является наиболее специфичным режимом работы двухпортового **ОЗУ**. Для иллюстрации работы этого режима памяти сначала записывается в регистр данных **data_in** число **10101010**, в регистры адреса записи **write_addr** и регистр адреса чтения **read_addr** адрес ячейки **0001**. Во время записи данных в эту ячейку на линиях выходных данных находятся старые данные **00001111**, и только в следующем такте появляются новые данные **10101010**.

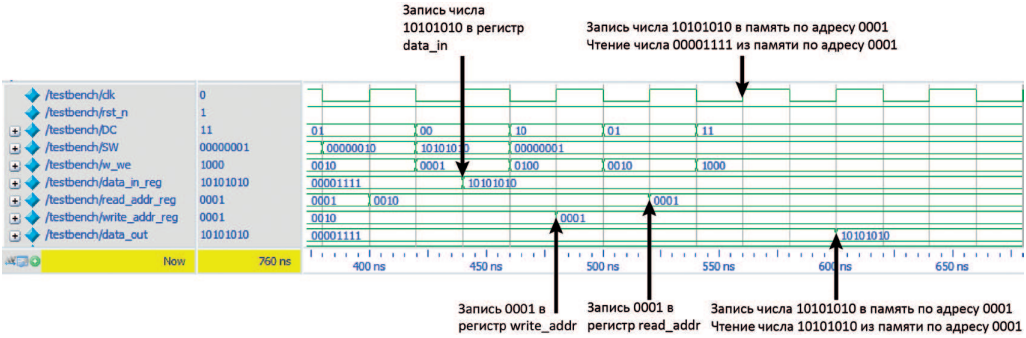


Рис. 7.27 Чтение во время записи в двухпортовом ОЗУ

Дополнительное задание для самостоятельной работы

Проверьте, как изменится размер двухпортовой памяти при увеличении разрядности шины адреса до **8**, **10**, **12**, **16** бит. Представьте результаты компиляции и синтеза в **RTL**, опишите и объясните полученные результаты.

7.9 Простой стек (LIFO)

7.9.1 Краткие сведения об организации стека

Стек – это такой вид памяти, который работает по принципу «последний вошел – первый вышел» (**Last In – First Out**, **LIFO**). В отличие от **ОЗУ**, стек не имеет адреса. Первый режим работы такого модуля – **запись в стек** (**push**). Новое слово записывается в ячейку, которая в данный момент является вершиной стека. Второй режим работы – **чтение данных из стека** (**pop**). Слово, находящееся в вершине стека, считывается и удаляется. Наиболее просто понять работу стека можно на

примере стопки тарелок – новая тарелка всегда ставится сверху, а чтобы добраться до второй тарелки, необходимо снять верхнюю. Адрес верхнего элемента стека называется вершиной стека. Самая нижняя ячейка – основание стека.

При записи в стек указатель на вершину стека увеличивается на единицу, т. е. вершина стека растет вверх (рис. 7.28). При этом старые данные в стеке остаются, а новые записываются в верхнюю ячейку.

При чтении указатель на вершину стека уменьшается на единицу, т. е. вершина стека уменьшается (рис. 7.29). При этом данные из вершины стека передаются на выход, а указатель становится на более низкую ячейку. Для того чтобы прочитать нижний элемент стека, необходимо вытолкнуть все стоящие выше данные из стека.

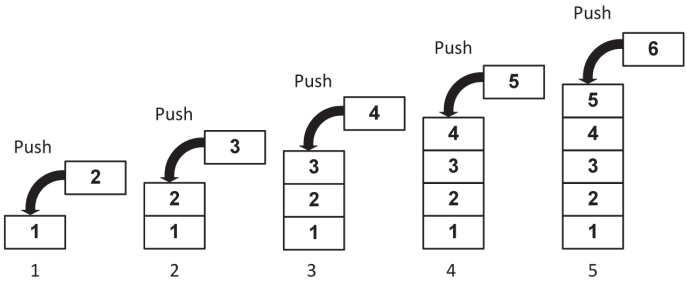


Рис. 7.28 Операция записи в стек

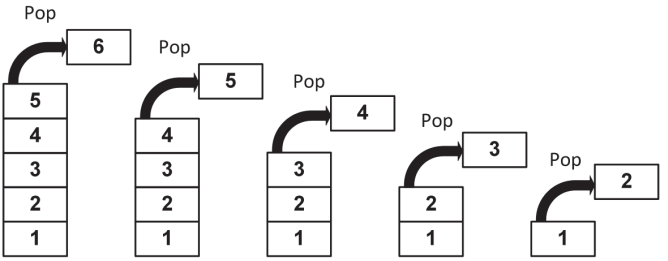


Рис. 7.29 Операция чтения из стека

В данной работе рассмотрим две реализации стека: с помощью сдвигового регистра и с помощью сдвига указателя.

7.9.2 Реализация стека с помощью сдвигового регистра

Сначала рассмотрим реализацию стека с помощью регистра сдвига. Стек состоит из четырех ячеек по **8** бит в каждой. Для определения их разрядности зададим параметры:

```
#(parameter DATA_WIDTH=8, parameter STACK_SIZE=4)
```

Точно так же, как и при описании **ОЗУ**, определим стек как массив регистров:

```
reg [DATA_WIDTH - 1:0] stack [0:STACK_SIZE - 1];
```

Операции загрузки и выгрузки в рассматриваемой реализации будут проводиться по переднему фронту тактовой частоты:

```
always @(posedge w_clk)
```

В блоке **always** сначала проверяем сигнал сброса. При сбросе устанавливаем содержимое всех ячеек стека в ноль.

```
if (reset)
   begin
      for (i = 0; i <STACK_SIZE; i = i + 1)
         stack [i] <= 0;
   end
```

Далее проведем анализ сигналов **push** и **pop**. Если сигнал **push** имеет высокий уровень, то увеличиваем значение указателя. В реальности происходит сдвиг значения всех регистров вглубь стека, а в регистр с нулевым индексом записывается входящая информация.

```
if (push)
   begin
      for (i = 0; i <STACK_SIZE - 1; i = i + 1)
         stack [i + 1] <= stack [i];
         stack [0] <= write_data;
   end
```

Затем проверим состояние сигнала **pop**. Если его значение равно единице, то производим сдвиг информации вниз. Освободившиеся регистры на вершине стека обнуляем.

```
if (pop)
   begin
      for (i = 0; i <STACK_SIZE - 1; i = i + 1)
         stack [i] <= stack [i + 1];
         stack [STACK_SIZE - 1] <= 0;
         read_data = stack [0];
   end
```

Модуль с описанием стека приведен в листинге 7.13.

```
module lab7_6
#(parameter DATA_WIDTH=8, parameter STACK_SIZE=4)
(  input clock,
   input reset,
   input push,
   input pop,
   input [DATA_WIDTH - 1:0] write_data,
   output reg [DATA_WIDTH - 1:0] read_data
);
   reg [DATA_WIDTH - 1:0] stack [0:STACK_SIZE - 1];
```

```
   // invert clk button
   wire w_clk = ~ clock;

   integer i;
   always @(posedge w_clk)
   begin
      if (reset)
         begin
            for (i = 0; i < STACK_SIZE; i = i + 1)
               stack [i] <= 0;
         end
      else if (push)
         begin
            for (i = 0; i < STACK_SIZE - 1; i = i + 1)
               stack [i + 1] <= stack [i];

               stack [0] <= write_data;
         end
      else if (pop)
         begin
            for (i = 0; i < STACK_SIZE - 1; i = i + 1)
               stack [i] <= stack [i + 1];

               stack [STACK_SIZE - 1] <= 0;
               read_data = stack [0];
         end
   end
endmodule
```

Листинг 7.13 Реализация стека с помощью сдвиговых регистров

Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 7.13 в файле **lab7_6.v**. Это файл верхнего уровня иерархии проекта. Выполните компиляцию проекта.

Результаты компиляции проекта в **RTL Viewer** приведены на рис. 7.30.

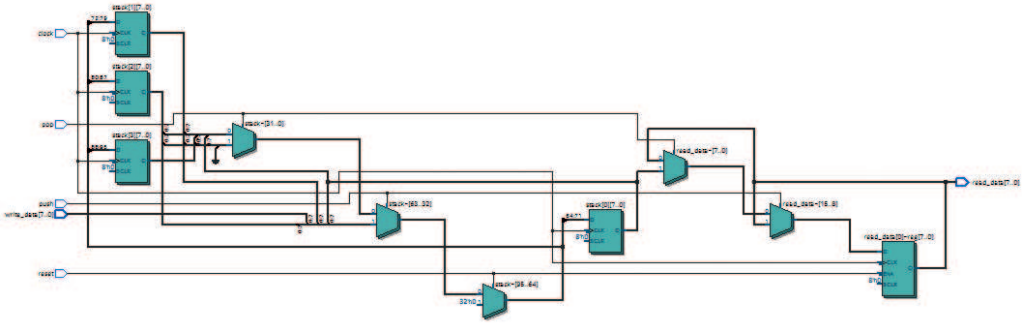


Рис. 7.30 Реализация стека в RTL Viewer

Стек был синтезирован в набор регистров, что соответствует его описанию в модуле. Когда необходимо сдвинуть информацию вверх или вниз по стеку, то данные передаются через мультиплексоры. Сигнал сброса является синхронным, и реализация сброса также выполняется с помощью мультиплексоров. По сигналу сброса входы регистров подключаются через мультиплексоры к нулю.

Назначьте выводы в проекте в соответствии с табл. 7.11.

Таблица 7.11 Соответствие выводов для проекта lab 7_4

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
clock	Input	PIN_A7	3.3 V Schmitt Trigger
pop	Input	PIN_C11	3.3-V LVTTL
push	Input	PIN_C10	3.3-V LVTTL
read_data[7]	Output	PIN_D14	3.3-V LVTTL
read_data[6]	Output	PIN_E14	3.3-V LVTTL
read_data[5]	Output	PIN_C13	3.3-V LVTTL
read_data[4]	Output	PIN_D13	3.3-V LVTTL
read_data[3]	Output	PIN_B10	3.3-V LVTTL
read_data[2]	Output	PIN_A10	3.3-V LVTTL
read_data[1]	Output	PIN_A9	3.3-V LVTTL
read_data[0]	Output	PIN_A8	3.3-V LVTTL
reset	Input	PIN_B8	3.3 V Schmitt Trigger
write_data[7]	Input	PIN_F15	3.3-V LVTTL
write_data[6]	Input	PIN_B14	3.3-V LVTTL
write_data[5]	Input	PIN_A14	3.3-V LVTTL
write_data[4]	Input	PIN_A13	3.3-V LVTTL
write_data[3]	Input	PIN_B12	3.3-V LVTTL
write_data[2]	Input	PIN_A12	3.3-V LVTTL
write_data[1]	Input	PIN_C12	3.3-V LVTTL
write_data[0]	Input	PIN_D12	3.3-V LVTTL

Конфигурирование ПЛИС и проверка работы стека

Откомпилируйте проект и загрузите конфигурацию в ПЛИС на отладочной плате. Проверьте работоспособность проекта. Для этого выполните следующие действия:

- используйте кнопки и микропереключатели для подачи сигналов на входы и проверяйте их значения с помощью светодиодов;
- проверьте таблицу истинности памяти.

Тестбенч и его описание

Для симуляции работы стека необходимо сделать шаги, приведенные ниже. Сначала требуется сгенерировать сигнал сброса и выполнить начальную установку сигналов:

```
clk = 1;
reset = 1;
push = 1;
pop = 0;
```

Затем сигнал сброса устанавливается в **0** и передаются данные для записи в стек:

```
reset = 0;
write_data = 8'b00000000;
```

Следующий шаг – переключение на режим чтения данных. Следует обратить внимание на то, что в конце, когда стек становится пустым, будет считываться нулевое значение.

```
push = 0;
pop = 1;
```

Полное описание тестбенча приведено в листинге 7.14.

```
// testbench is a module which only task is to test another module
// testbench is for simulation only, not for synthesis
`timescale 1ns/1ns
module testbench;
   // input and output test signals
   reg clk;
   reg reset;
   reg push;
   reg pop;
   reg [7:0] write_data;
   wire [7:0] read_data;

   // creating the instance of the module we want to test
   lab7_6 dut (clk, reset, push, pop, write_data, read_data);

initial
   begin
      // set inital values of signal
      clk = 1;
      reset = 1;
      push = 1;
      pop = 0;
      write_data = 8'b00000000;

      // write data in RAM
      #20; write_data = 8'b00000001;
      #10; reset = 0;
```

```
      #10; write_data = 8'b00000010;
      #20; write_data = 8'b00000011;
      #20; write_data = 8'b00000100;
      #20; write_data = 8'b00000101;
      #20; write_data = 8'b00000110;
         push = 0;
         pop = 1;
      #20; write_data = 8'b00000111;
      #20; write_data = 8'b00001000;
      #20; write_data = 8'b00001001;
      #20; write_data = 8'b00001011;
   end

   //every 10 ns invert clk
   always #10 clk = ~clk;

   initial
      #220 $finish;

   // do at the beginning of the simulation
   // print signal values on every change
   initial
      $monitor("clk=%b reset=%b push=%b pop=%b write_data=%b
         read_data=%b", clk, reset, push, pop, write_data, read_data);

   // do at the beginning of the simulation
   initial
      $dumpvars; //iVerilog dump init
endmodule
```

Листинг 7.14 Тестбенч для стека

Симуляция модуля стека

На рис. 7.31 приведены результаты симуляции разработанного модуля в **ModelSim**.

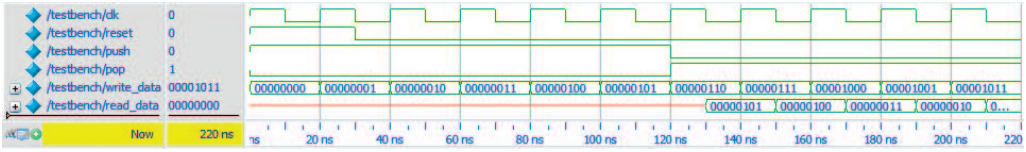


Рис. 7.31 Результат симуляции стека в ModelSim

7.9.3 Реализация стека с помощью сдвига указателя

Разработаем модуль стека, построенного на принципе увеличения или уменьшения значения указателя. В этом случае будет осуществляться изменение значение одного регистра, а не сдвиг информации во всех регистрах стека. Указатель такого стека указывает на вершину стека, и его значение изменяется по переднему фронту тактового сигнала:

```
reg [PTR_WIDTH - 1:0] pointer;
```

Когда сигнал сброса равен единице, указатель стека установлен в ноль и указывает на основание стека:

```
if (!rst_n)
   pointer <= {PTR_WIDTH {1'b0}};
```

Во время записи данных в стек (операция **push**) указатель увеличивается на единицу:

```
if (push)
   pointer <= pointer + 1;
```

Во время чтения данных из стека (операция **pop**) указатель соответственно уменьшается на единицу:

```
if (pop)
   pointer <= pointer - 1;
```

В операторе **always** анализируется проведение операции чтения с помощью оператора **case**:

```
case ( {push, pop} )
   2'b10: begin
      if (pointer < 2**PTR_WIDTH - 1) begin
         stack[pointer] <= write_data;
         pointer <= pointer + 1;
         end
      else
         stack[pointer] <= write_data;
      end

   2'b01: begin
      if (pointer != 0) begin
         read_data <= stack [pointer-1];
         pointer <= pointer - 1;
         end
      else
         read_data <= 0;
      end
   default: read_data <= stack [pointer];
endcase
```

Для упрощения кода выполняется конкатенация сигналов **push** и **pop**, что дает возможность анализировать двухбитный сигнал. Возможно четыре комбинации сигналов **push** и **pop**:

1. **Запись в стек**, когда **push == 1** и **pop == 0**. В этом случае производится запись в регистр, на который указывает указатель стека, и указатель стека увеличивается на **1**:

```
stack[pointer] <= write_data;
pointer <= pointer + 1;
```

Если указатель стека указывает на вершину стека, то его значение не изменяется, и новые данные затирают значение в верхнем регистре стека:

```
stack [pointer] <= write_data;
```

2. **Чтение из стека**, когда **push == 0** и **pop == 1**. В этом случае производится чтение регистра, который находится ниже указателя стека, и значение указателя стека уменьшается на **1**:

```
read_data <= stack [pointer-1];
pointer <= pointer - 1;
```

В том случае если указатель стека адресует основание стека, то возвращается нулевое значение:

```
read_data <= 0;
```

3. **Операции со стеком не производятся** (**push == 0** и **pop == 0**) или подаются ошибочные сигналы (**push == 1** и **pop == 1**). В этой ситуации на выход подается значение регистра, адресуемого указателем стека, а сам указатель не изменяется:

```
read_data <= stack [pointer];
```

Модуль с описанием стека приведен в листинге 7.15.

```
module lab7_7
#( parameter DATA_WIDTH=8,
   parameter PTR_WIDTH=2
)
(  input clock,
   input rst_n,
   input push,
   input pop,
   input [DATA_WIDTH - 1:0] write_data,
   output reg [DATA_WIDTH - 1:0] read_data
);
   reg [DATA_WIDTH - 1:0] stack [0:2**PTR_WIDTH - 1];
   reg [PTR_WIDTH - 1:0] pointer;

   always @(posedge clock or negedge rst_n)
      begin
         if (!rst_n)
            pointer <= {PTR_WIDTH {1'b0}};
         else case ( {push, pop} )
            2'b10: begin
               if (pointer < 2**PTR_WIDTH - 1) begin
                  stack[pointer] <= write_data;
```

```
                pointer <= pointer + 1;
             end
          else
             stack[pointer] <= write_data;
          end
       2'b01: begin
          if (pointer != 0) begin
             read_data <= stack [pointer-1];
             pointer <= pointer - 1;
             end
          else
             read_data <= 0;
          end
       default: read_data <= stack [pointer];
    endcase
  end
endmodule
```

Листинг 7.15 Реализация стека с указателем

Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 7.9 в файле **lab7_7.v**. Это файл верхнего уровня иерархии. Откомпилируйте проект.

Результаты компиляции проекта, полученные в **RTL Viewer**, приведены на рис. 7.32. Следует обратить внимание, что для реализации стека использован модуль памяти, а работа с указателем стека реализована с помощью логических элементов. Для инкремента или декремента, а также сравнения значений указателя используются сумматоры. Для сохранения значения указателя используется двухразрядный регистр.

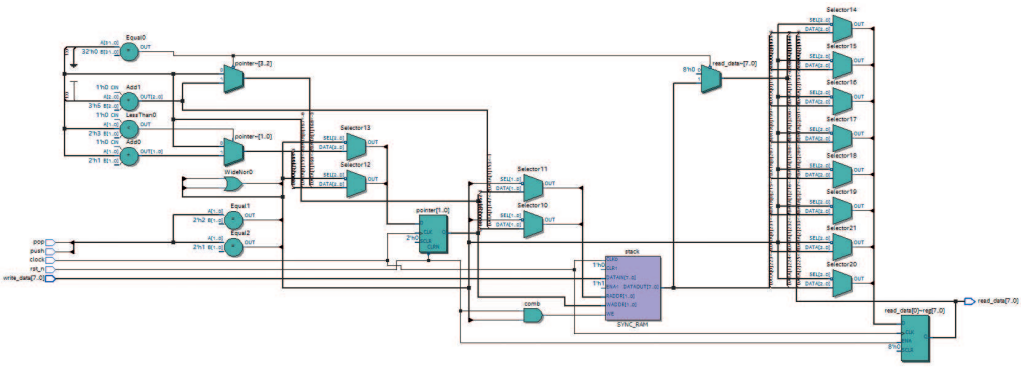


Рис. 7.32 Реализация стека в RTL Viewer

Тестбенч и его описание

Тестбенч будет таким же самым, как и в предыдущем случае. На рис. 7.33 приведены результаты симуляции в **ModelSim**.

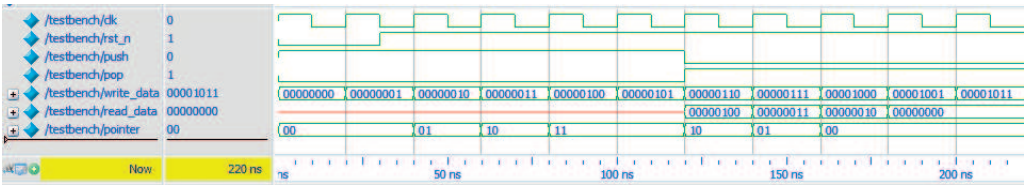


Рис. 7.33 Результат симуляции стека в ModelSim

Сравнение двух реализаций стека

Сравним приведенные выше реализации стека. Чтобы это сделать, заполните табл. 7.12.

Таблица 7.12 Сравнение двух реализаций стека

Размер стека (STACK_SIZE)	Первая реализация		Вторая реализация	
	Количество логических элементов	Количество регистров	Количество логических элементов	Количество регистров
4				
16				
128				
1024				

В первом столбце таблицы приведены значения размера стека (**STACK_SIZE**), которые необходимо использовать для каждой реализации стека. После компиляции в отчете будут получены значения общего количества логических элементов и регистров, необходимых для первой (столбцы 2 и 3) или второй (столбцы 4 и 5) реализации (рис. 7.34).

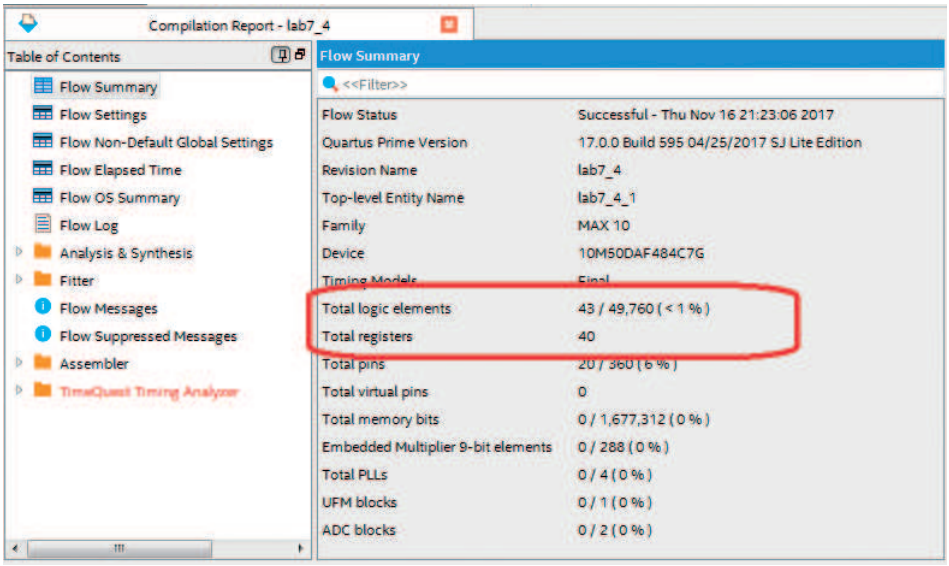


Рис. 7.34 Отчет компиляции при реализации стека

Следует отметить, что первая реализация стека во время работы будет потреблять больше энергии, т. к. при каждом обращении происходит изменение содержимого всех регистров. Во второй реализации будут изменяться регистры указателя и адресуемых регистров.

Дополнительное задание для самостоятельной работы

Проанализируйте, как изменится размер стека при увеличении глубины стека до **32**, **64**, **128**, **256** элементов. Представьте результаты компиляции и синтеза в **RTL Viewer** и объясните их. Загрузите результаты в **Technology Map Viewer**, опишите и объясните полученные результаты.

7.10 Очередь (FIFO)

Очередь (**First In First Out**, **FIFO**) – это вид памяти, работающей по принципу «первый зашел – первый вышел». Подобно стеку, она не имеет адресного ввода и, как простая двухпортовая **ОЗУ**, имеет сигналы записи и чтения. В случае записи новое слово записывается в конец очереди (хвост очереди, верхнее расположение в стековых терминах). В случае чтения слово в начале очереди (голова очереди, нижнее расположение в терминах стека) считывается и удаляется. На рис. 7.35 приведены обе операции.

Чтобы прочитать последний элемент в очереди, необходимо выполнить операцию чтения для всех предыдущих элементов. Когда происходит запись нового элемента в конец очереди, все предыдущие элементы остаются на месте. Когда происходит чтение элемента из головы очереди, другие элементы перемещаются на один шаг ближе к голове очереди, сохраняя свой порядок. В случае одновременных операций чтения и записи оба поведения должны быть реализованы, как приведено в последнем случае на рис. 7.35.

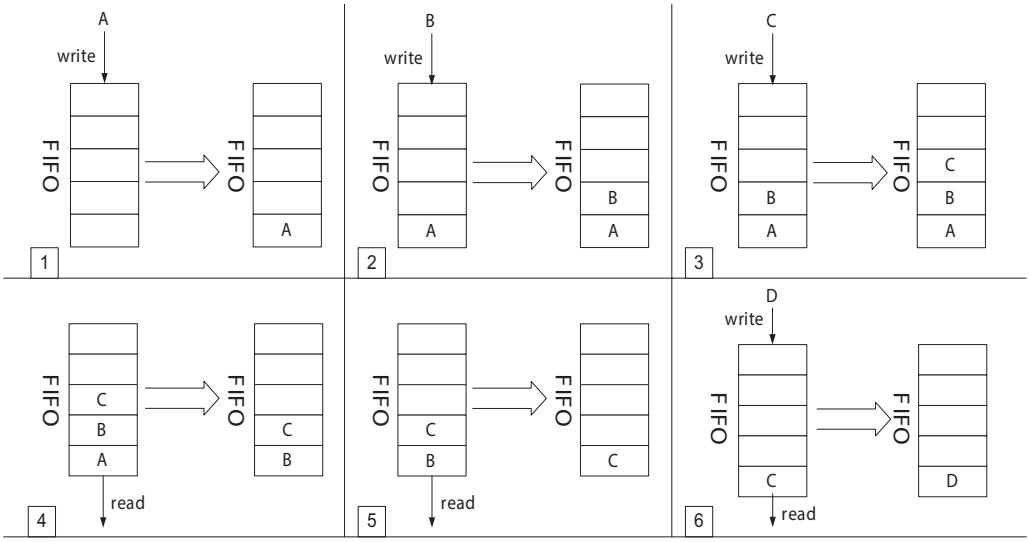


Рис. 7.35 Операции в блоке FIFO

Дополнительное задание для самостоятельной работы

Реализуйте два варианта очереди, как в случае со стеком в разделе 7.9 с глубиной до **32**, **64**, **128**, **256** элементов. Представьте результаты компиляции и синтеза в **RTL Viewer** и объясните их. Загрузите результаты в **Technology Map Viewer**, опишите и объясните полученные результаты.

7.11 Встроенная память микросхем ASIC

В этом разделе рассматривается пример простой статической памяти размером **64x6** бит, которая встраивается в специализированные микросхемы **ASIC**. При реализации в **ПЛИС** компилятор вызывает примитив встроенной памяти и передает в его описание параметры, которые извлекает из описания модуля памяти. То есть у каждого производителя **ПЛИС** существует свой параметризированный модуль памяти, в который передаются параметры, извлекаемые из кода разработчика.

Чип **ASIC** похож на чистый лист. Он не имеет ни одного из встроенных модулей памяти, доступных на **ПЛИС**. Как правило, если нужна оперативная память размером **1Kx8**, то используют уже готовые библиотечные компоненты, которые представляет фабрика – изготовитель микросхем. Такие библиотечные модули уже заранее скомпонованы, и они имеют очень высокую плотность упаковки на уровне транзисторов. Разработчик не делает разводку модуля памяти на кристалле. Берется уже готовый библиотечный компонент и подключается в коде программы на языке **Verilog**, а затем размещается на **ASIC**-чипе как часть общего проекта.

В приведенном ниже примере предположим, что для реализации **ASIC**-дизайна требуется небольшая 64-разрядная **SRAM**. Но фабрика-производитель предо-

ставляет в библиотеке меньшую память, имеющую название **SRAM_16x6** (предположим, что этот библиотечный модуль имеет входы и выходы, аналогичные тому модулю памяти, который был описан в листинге 7.9). Для реализации блока памяти нужного размера следует использовать четыре таких модуля **16x6 SRAM**. Тогда в общей сложности будет 64 шестибитных ячеек (листинг 7.16).

```
//Memory Bank x4
module SRAM_BANK_x4
(  input we, clk,
   input [1:0] page,
   input [3:0] addr,
   input [5:0] data_in,
   output reg [5:0] data_out
);
   //Individual SRAM enables:
   reg [3:0] en;
   //Output multiplexer data:
   wire [5:0] dout0, dout1, dout2, dout3;
   //Decode page bits:
   always @*
      begin: decode
         en = 4'b0000;
         en[page] = 1'b1;
      end

   //Instantiate four vendor-specific 16x6 RAM blocks:
   SRAM_16x6 M0(.data_in(data_in),.addr(addr),.clk(clk),
      .data_out(dout0), .we( en[0] ) ),
      M1(.data_in(data_in),.addr(addr),.clk(clk),
      .data_out(dout1), .we( en[1] ) ),
      M2(.data_in(data_in),.addr(addr),.clk(clk),
      .data_out(dout2), .we( en[2] ) ),
      M3(.data_in(data_in),.addr(addr),.clk(clk),
      .data_out(dout3), .we( en[3] ) );
   //Multiplex the four data outputs:
   always @*
      case (page)
         0: data_out = dout0;
         1: data_out = dout1;
         2: data_out = dout2;
         3: data_out = dout3;
      endcase
endmodule // SRAM_BANK_x4
```

Листинг 7.16 Реализация модуля памяти для ASIC

Следует обратить внимание на то, что полный адрес этого банка памяти состоит из **6** бит: **2** бита для выбора страницы и **4** бита адреса. Например, 6-битный адрес

6'b110000 будет обращаться к ячейке **0** в экземпляре **M3 SRAM**. Два старших бита для выбора страницы активируют линию **en [3]**, т. е. выбирают экземпляр памяти **M3**. Эти же старшие биты будут мультиплексировать данные из **M3** на выход **dout3**, а затем и на **data_out**.

Дополнительное задание для самостоятельной работы

Модифицируйте пример из листинга 7.16 так, чтобы данный модуль был параметризируемый и можно было задать память любого размера, кратного 16 словам по 6 бит. Опишите полученный модуль и его работу.

7.12 Упражнения

7.12.1 Основное задание

Выполните дополнительные задания для разделов 7.6, 7.7, 7.8, 7.9.

7.12.2 Задания для самостоятельной работы

Используя примеры кода из данного пособия, реализуйте следующие модули памяти:

1. Разработайте регистровый файл, содержащий 32 регистра разрядностью 32 бита.
2. Разработайте схему **ОЗУ**, у которого входы и выходы данных будут содержать регистры. Разрядность данных – **8**, адреса – **16**.
3. Разработайте схему **ОЗУ** размером **256×8** с битом контроля четности.
4. Разработайте схему **ОЗУ** размером **256×8** с возможностью записи как **8**-, так и **16**-битных данных. Для выбора разрядности используйте сигнал **byteena[1:0]** и руководствуйтесь таблицей ниже.

byteena[1:0]	data_in
[0] = 1	[7:0]
[1] = 1	[15:0]

5. Разработайте схему простой двухпортовой памяти, которая будет хранить данные о цветном изображении с глубиной цвета 8 бит и разрешением **212×104**.
6. Разработайте схему умножителя **8×8** с использованием **ПЗУ**.
7. Разработайте **ПЗУ**, которое будет преобразовывать **8**-разрядный двоичный код в двоично-десятичный.
8. Разработайте **ПЗУ**, которое будет реализовывать дешифратор семисегментного индикатора с точкой. **ПЗУ** должно содержать дополнительный вход для гашения индикатора.
9. Разработайте **ПЗУ**, которое будет содержать **1024 восьми**битных отсчета синуса.

10. Разработайте **ПЗУ**, которое будет содержать **4** функции по **256 восьми**битных отсчетов в каждой: **sin(x)**, **sin(x)/x**, функция Гаусса, экспоненциальное нарастание.
11. Разработайте устройство, которое будет на выходе вырабатывать сигнал синуса. Разрядность шины адреса – **10**. Устройство содержит **ПЗУ** с **256 восьми**битными отсчетами для ¼ периода синуса.
12. Разработайте стек, который будет содержать сигналы «Стек заполнен» и «Стек пуст».
13. Разработайте стек, который будет выдавать сигнал ошибки при попытке читать пустой стек или писать в полный стек.
14. Создайте схему полной двухпортовой памяти, которая позволит проводить как чтение, так и запись из двух портов. Проведите симуляцию работы схемы и объясните работу схемы в режимах чтение во время записи и одновременной записи данных в одну ячейку.
15. Создайте схему простой двухпортовой памяти с двумя тактовыми сигналами. То есть порт записи использует один тактовый сигнал, а порт чтения – другой.

7.12.3 Контрольные вопросы

1. Опишите, как строятся регистры на основе D-триггеров.
2. Как организованы массивы ячеек памяти?
3. Приведите примеры типов входов и выходов памяти. Опишите их назначение.
4. Какими бывают режимы работы памяти? Опишите их.
5. Какими бывают виды двухпортовой памяти? Опишите режимы ее работы.
6. Что такое «информационная емкость»? Как ее рассчитать?
7. Какими бывают типы запоминающих устройств?
8. Как можно классифицировать память по функциям, которые она выполняет?
9. Как устроен логический элемент микросхемы **Intel FPGA MAX10**?
10. Опишите варианты конфигурации блока памяти **M9K** в микросхеме **MAX10**.
11. Как реализовать **ПЗУ** с помощью таблицы перекодировки? Приведите пример на **Verilog**.
12. Для чего используется память вида «стек»? Опишите работу стека.
13. Как реализовать стек с помощью сдвигового регистра? Приведите пример на **Verilog**.
14. Как реализовать стек с помощью сдвига указателя? Приведите пример на **Verilog**.

15. Как реализовать **FIFO**? Приведите пример на **Verilog**.

16. Как устроена встроенная память микросхем **ASIC**?

7.12.4 Темы для индивидуальной работы

Стековый калькулятор

Стековый калькулятор – это калькулятор, который хранит в стеке операнды и коды операций, а при выполнении операций извлекает их из стека. Необходимо реализовать работу калькулятора с **8**-битными беззнаковыми числами. Калькулятор должен выполнять такие операции: сложение, вычитание, умножение. Операнды вводятся с помощью микропереключателя. Операнды и результат выводятся на семисегментный индикатор. Например, для выполнения операции сложения **A + B** необходимо выполнить такие операции (рис. 7.36):

- запись числа «**A**»;
- запись числа «**B**»;
- запись кода операции «+»;
- зачисление суммы «**A+B**».

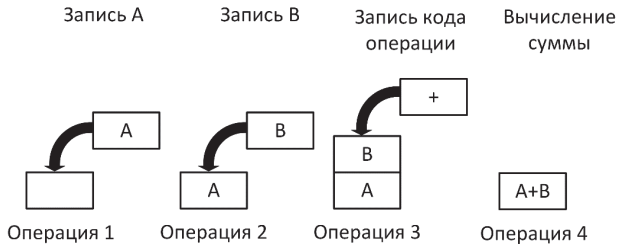


Рис. 7.36 Операция сложения в стековом калькуляторе

Для выполнения операции **A + B - C** необходимо выполнить такие операции (рис. 7.37):

- запись числа «C»;
- запись числа «B»;
- запись числа «A»;
- запись числа «B»;
- запись кода операции «-»;
- запись кода операции «+»;
- вычисление «A+B-C».

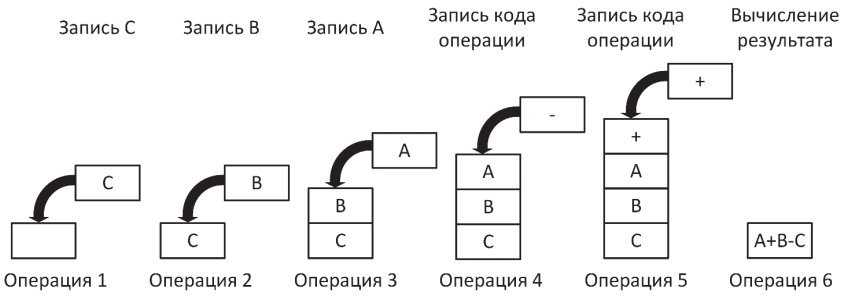


Рис. 7.37 Операция над тремя операндами в стековом калькуляторе

Сергей Иванец, Александр Романов

Цифровой синтез: практический курс

Глава 8. Конечные автоматы: основы

Содержание

В главе рассмотрены описание и работа ***конечных автоматов*** *(**Finite State Machine**, **FSM**). Приведены примеры реализации конечных автоматов двух классических архитектур: **Мили** и **Мура**. Также при прочтении данной главы предлагается выполнить индивидуальное задание, согласно которому нужно сделать автомат распознавания нажатых клавиш и автомат контроллера светофора.*

*Конечные автоматы нашли распространение в большом количестве устройств и программ. Они применяются при разработке пользовательских интерфейсов приложений и систем управления, описании поведения юнитов в играх и арбитров системной шины компьютера. Одним из наиболее известных автоматов является машина Тьюринга. Она является обобщением понятия «**конечный автомат**», и ее работа описывается с помощью конечных автоматов. Наиболее явным примером использования конечных автоматов в повседневной жизни является система управления светофором.*

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет **Quartus Prime** (есть студенческая версия);*
- *пакет **ModelSim Altera Edition** или программы **Icarus Verilog** и **GTKWave**;*
- *отладочная плата компании **Terasic DE10-Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах[1] к данной книге).*

8.1 Конечные автоматы

Конечный автомат представляет собой устройство, которое осуществляет прием, хранение и преобразование дискретной информации по некоторому алгоритму и может находиться в одном из нескольких устойчивых состояний[2]. Слово «конечный» говорит о том, что количество состояний автомата конечно и может быть посчитано. Если выходной сигнал конечного автомата зависит лишь от текущего состояния, то такой автомат называется **автоматом Мура**[3] (рис. 8.1).

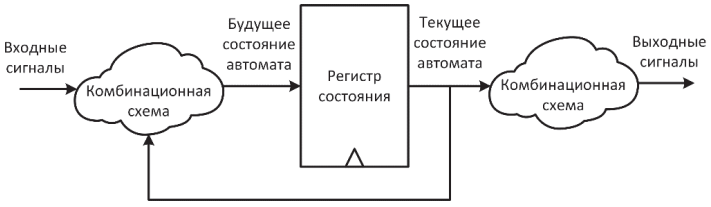


Рис. 8.1 Конечный автомат Мура

[1] https://github.com/RomeoMe5/DDLM.

[2] *Самофалов К. Г.* Прикладная теория цифровых автоматов. Киев: Высшая школа, 1987. 375 с.

[3] *Дэвид М. Харрис, Сара Л. Харрис*. Цифровая схемотехника и архитектура компьютера. Elsevier, 2013. (Рус. перевод – 2015.) 1662 с.

Если выходной сигнал зависит от текущего состояния и входных сигналов, то такой конечный автомат называют **автоматом Мили** (рис. 8.2).

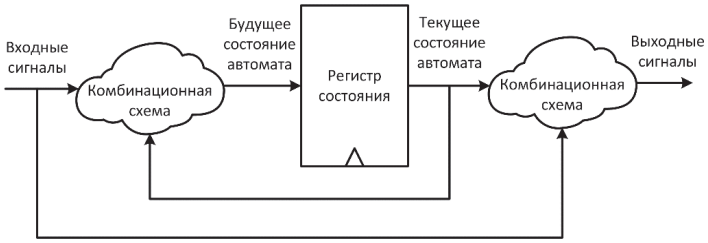


Рис. 8.2 Конечный автомат Мили

Согласно приведенным рисункам, **конечный автомат** содержит две комбинационные схемы: одна – для вычисления будущего состояния автомата, вторая – для вычисления значений выходных сигналов. Также в **конечный автомат** входит регистр для хранения состояний автомата.

Конечный автомат может быть описан с помощью таких представлений:

- в виде ориентированного графа;
- с помощью переходов и выходов.

Представление цифрового автомата **Мура** в виде **ориентированного графа** приведено на рис. 8.3. Здесь в кругах – вершинах графа – показаны состояния цифрового автомата и значения выходного сигнала. Например, **S1/1** – это состояние S1 и значение выхода равно «**1**». Переходы между состояниями показаны дугами между вершинами, а переход в то же самое состояние – **петлей**. Возле дуг и петель приведены значения входных сигналов, при которых происходит этот переход. Например, (**~a**) обозначает, что данный переход произойдет при **a = 0**.

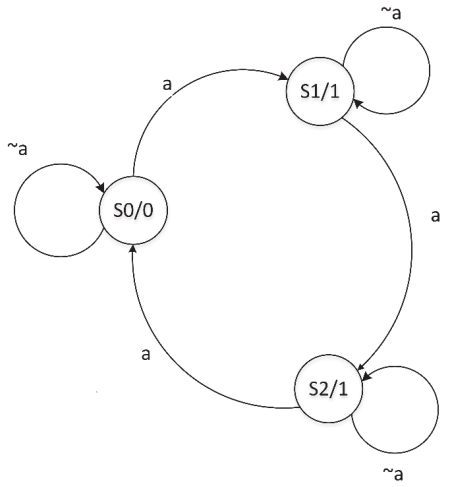


Рис. 8.3 Граф конечного автомата Мура

Представление цифрового автомата с помощью таблиц предполагает наличие таблицы переходов и таблицы выходов. **Таблица переходов** связывает между собой текущее состояние, входные сигналы и будущее состояние конечного автомата. Таблица переходов конечного автомата, описанного графом на рис. 8.3, представлена в виде табл. 8.1.

Таблица 8.1 Таблица переходов конечного автомата Мура

Текущее состояние	Будущее состояние	Условие перехода
S0	S1	a=1
S0	S0	a=0
S1	S1	a=0
S1	S2	a=1
S2	S2	a=0
S2	S0	a=1

Таблица выходов служит для описания соответствия текущего состояния конечного автомата его выходным сигналам:

Таблица 8.2 Таблица выходов конечного автомата Мура

Текущее состояние	Выходной сигнал
S0	0
S1	1
S2	1

При реализации конечного автомата рекомендуется придерживаться принципа разделения на **комбинационную** и **последовательностную** части схемы. При такой интерпретации конечный автомат будет представлен тремя блоками[1] (рис. 8.4):

- комбинационный блок логики переходов;
- регистр для хранения состояний цифрового автомата;
- комбинационный блок для формирования выходных сигналов (отличается для **конечных автоматов Мили** и **Мура**).

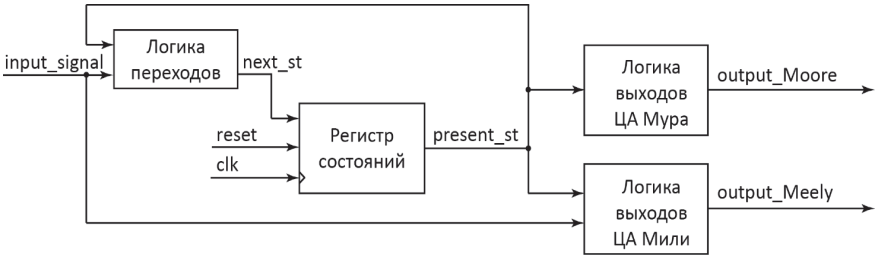


Рис. 8.4 Структурная схема конечного автомата

[1] *Clifford E. Cummings*. Synthesizable Finite State Machine Design Techniques Using the New SystemVerilog 3.0 Enhancements // SNUG'03 (Synopsys Users Group San Jose, CA, 2003) Proceedings, March 2003.

На вход схемы логики переходов поступают код текущего состояния конечного автомата (**present_st**) и внешние сигналы (**input_signal**). Выходом этого блока является код следующего состояния (**next_st**).

В регистр состояний входит три сигнала: тактовый (**clk**), сброса (**reset**) и код следующего состояния (**next_st**). Тактовый сигнал и сигнал сброса предназначены для управления триггерами, которые хранят состояние автомата. По переднему фронту тактового сигнала производится запись следующего состояния (**next_st**) в регистр состояний. Результат записи появляется на выходе регистра состояния в виде сигнала текущего состояния конечного автомата (**present_st**).

Блок формирования выходных сигналов в зависимости от состояния конечного автомата (и входных сигналов для автомата **Мили**) формирует асинхронные выходные сигналы. Для получения синхронных выходных сигналов в этот блок дополнительно встраивают регистр.

Кодирование состояний конечных автоматов

При описании конечного автомата требуется закодировать его состояния, т. е. поставить в соответствие буквенному обозначению состояния какое-то число. Например, три состояния на рис. 8.3 могут быть закодированы как **0**, **1** и **2**. В зависимости от метода кодирования состояний автомата можно получить либо уменьшение схемы, либо увеличение быстродействия, либо устойчивость автомата к помехам.

Для кодирования можно воспользоваться ручным методом, напрямую описав коды состояний при описании конечного автомата на языке **Verilog**, или поручить этот процесс компилятору. Такие инструменты, как **Quartus Prime**, могут автоматически выбирать оптимальный метод кодирования состояния. Компилятор определяет числовые значения для состояний в соответствии с методом кодирования. В этом случае важен не только метод кодирования состояний, но и порядок перечисления значений типа при его определении в описании. Первое значение этого типа должно быть значением по умолчанию.

В **Verilog** существует возможность использовать параметры для кодирования состояний. Приведенный ниже пример объясняет, как определить три параметра, кодирующие состояния автомата на рис. 8.3:

```
parameter [1:0]
   S0 = 0, S1 = 1, S2 = 2;
```

Приведенный пример означает, что **S0** будет иметь значение «**0**», **S1** – значение «**1**», **S2** – значение «**2**».

Существует несколько методов кодирования состояний:

- **auto** – автоматический выбор метода кодирования в зависимости от количества состояний конечного автомата. Если состояний меньше 5, то используется метод кодирования **sequential**. Если состояний больше 5, но меньше 50, то используется метод **one-hot**. Во всех других случаях используется метод с использованием кода Грея (**Gray code**);

- **one-hot** – этот метод потребует **N** бит для кодирования **N** состояний конечного автомата. При кодировании любого состояния только один бит имеет значение, равное **1**, все остальные биты равны **0**. Этот стиль используется в **Quartus Prime** по умолчанию:

```
parameter [2:0]
   S0 = 3'b001,
   S1 = 3'b010,
   S2 = 3'b100;
```

- **Gray** – кодирование состояний в соответствии с кодом Грея. В этом случае значения соседних состояний отличаются только одним битом. **N**-битный код Грея может закодировать 2^N состояний конечного автомата:

```
parameter [1:0]
   S0 = 2'b00,
   S1 = 2'b01,
   S2 = 2'b11;
```

- **Johnson** – кодирование состояний в соответствии с кодом Джонсона. Этот код подобен коду Грея, но требует меньшего количества логики для реализации кодирования;
- **minimal bits** – компилятор будет минимизировать количество состояний конечного автомата, а соответственно, и количество битов, необходимых для их кодирования;
- **sequential** – используется обычный двоичный код для кодирования состояний конечного автомата;
- **user-encoded** – используется тот метод кодирования, который применял разработчик при описании конечного автомата. В этом случае необходимо самостоятельно записать коды для каждого состояния конечного автомата.

В табл. 8.3 приведены значения, используемые для кодирования состояний конечного автомата в различных системах кодирования для чисел от **0** до **7**.

Таблица 8.3 Кодирование состояний в различных системах

Двоичный код	Код Грея	Код Джонсона	Код one-hot
000	000	00000	0000 0001
001	001	00001	0000 0010
010	011	00011	0000 0100
011	010	00111	0000 1000
100	110	01111	0001 0000
101	111	11111	0010 0000
110	101	11110	0100 0000
111	100	11100	1000 0000

Для конечных автоматов одного типа возможен переход к эквивалентному автомату другого типа, и наоборот. Эквивалентные автоматы на одинаковую последовательность входных сигналов будут давать одинаковую последовательность выходных сигналов, но они могут иметь различное количество состояний. В такой ситуации необходимо выбирать тот автомат, который имеет меньшее количество состояний. Обычно при переходе от автомата **Мура** к эквивалентному автомату **Мили** количество состояний конечного автомата не меняется, а вот при переходе от конечного автомата **Мили** к автомату **Мура** количество состояний конечного автомата увеличивается. То есть эквивалентный автомат **Мура** будет иметь больше состояний, а значит, требовать большего количества регистров.

Основным достоинством автомата **Мили** является возможность генерации выхода в том же такте, в котором на вход автомата поступает новый сигнал. То есть задержка между поступлением входного сигнала и реакцией на него будет минимальной. Недостатком такого подхода является то, что он требует асинхронной схемы реализации и приводит к возникновению гонок сигналов. Конечный автомат **Мура** требует для реализации большего количества регистров, но за счет отсутствия логики формирования выходных сигналов в зависимости от входа может работать на более высоких частотах.

Примечание: цифровой автомат следует разрабатывать так, чтобы он был описан максимально понятно и просто. Минимизация обычно осуществляется компилятором на этапе синтеза, в соответствии с параметрами синтеза. Тем не менее следует проверить результат, чтобы убедиться, что он верен, не возникли ли регистры-защелки или другие нежелательные конструкции.

8.2 Описание конечного автомата на языке Verilog

Наиболее общие рекомендации при описании конечных автоматов можно сформулировать следующим образом.

1. Описание конечного автомата должно быть легко модифицируемым для изменения метода кодирования состояний и количества состояний автомата.
2. Описание конечного автомата должно легко поддаваться отладке.
3. Каждый конечный автомат должен быть описан в отдельном модуле. Это дает возможность легко читать код и упрощает работу компилятора.
4. Все последовательные **always**-блоки должны быть описаны с использованием неблокирующего присваивания. Все комбинационные **always**-блоки должны быть описаны с помощью блокирующего присваивания. Эти два условия позволят избежать гонок сигналов при реализации конечного автомата[1].

Для описания на языке **Verilog** конечного автомата, приведенного на рис. 8.4, можно использовать один или несколько блоков **always**.

---

[1] *Clifford E. Cummings.* State Machine Coding Styles for Synthesis // SNUG'98 (Synopsys Users Group San Jose, CA, 1998) Proceedings, section-TB1 (3rd paper), March 1998.

Наиболее просто описывать небольшой конечный автомат с количеством состояний не более 8 с использованием одного блока **always**. В этом случае в одном блоке объединяют и логику переходов, и регистр состояний, и определение выходного сигнала. В данной ситуации очень часто используют оператор **case** для определения следующих состояний. Но этот метод описания хорош для небольших автоматов и становится очень сложно реализуемым при описании автоматов с большим количеством состояний. Кроме того, при подобном подходе становится очень сложно выполнять отладку и поддержку кода. Также такой метод не позволяет эффективно использовать уже существующий код повторно.

Более эффективно использовать два блока **always**. Один из этих блоков является комбинационным и вычисляет код следующего состояния. Другой является последовательностным и описывает регистр состояний. Логика вывода зависит от реализации конечного автомата и может быть описана как отдельный блок **always** или с помощью непрерывного присваивания.

Рассмотрим комбинационный блок **always**. Он **вычисляет следующее состояние** конечного автомата, основанное на текущем состоянии и значении входов, т. е. описывает таблицу переходов. Как правило, он кодируется с использованием оператора **case**, как показано ниже:

```
always @*
   case (state)
   S0:
      if (input_signal)
         next_state = S1;
      else
         next_state = S0;

      //...

      default: next_state = S0;
endcase
```

Листинг 8.1 Блок always для описания таблицы переходов

Так как эта часть кода задает комбинационную часть конечного автомата, то список чувствительности **always** должен содержать все сигналы, которые используются в качестве аргументов в этом блоке. Возможно использование подстановочного оператора (*) для обозначения всех сигналов, включая состояние конечного автомата:

```
always @*
```

Внутри оператора **case** анализируется значение состояний **state**. Затем, в зависимости от значения входного сигнала (**input_signal**), будет определено следующее состояние конечного автомата. Например, если входной сигнал равен «**1**», а текущее состояние – **S0**, то следующее состояние – **S1**. Если входной сигнал равен нулю, следующее состояние – **S0**.

```
S0:
   if (input_signal)
      next_state = S1;
   else
      next_state = S0;
```

Необходимо задать такие конструкции для всех состояний конечного автомата, записанных при объявлении параметра (**parameter**).

Поскольку для кодирования состояния конечного автомата используется определенное количество битов, то нужно указать поведение конечного автомата для всех возможных комбинаций битов. Например, необходимо описать конечный автомат с **4** состояниями, где будет использоваться метод кодирования **one-hot**. Для кодирования состояний используется **4** бита, поэтому количество возможных комбинаций будет составлять **16**. И при описании оператора **case** следует задать все эти комбинации. В конечном автомате используется **4** состояния, а это значит, что **12** значений не используется. Если их не описывать, то компилятор установит переходы между этими состояниями самостоятельно в соответствии со стратегией синтеза конечного автомата или проекта в целом. При нормальной работе конечного автомата это не будет иметь никаких последствий, но это возможно только в идеальных условиях. В реальной жизни на систему воздействуют различные помехи – как по входу, так и по питанию, – поэтому возможна ситуация, когда автомат перейдет в состояние, которое в явном виде не задано. В такой ситуации его поведение непредсказуемо, например автомат может зациклиться в состоянии, из которого нет выхода. Единственный способ выхода из этой ситуации – сброс автомата. Поэтому необходимо описывать все возможные комбинации.

Конечный автомат на рис. 8.3 использует только три из четырех возможных состояний. Одно состояние (**3**) не используется. Чтобы обрабатывать такие состояния, необходимо добавить в оператор **case** строку по умолчанию:

```
default:
   next_state = S0;
```

Когда состояние не соответствует **S0**, **S1** или **S2**, конечный автомат вернется в состояние **S0**.

При описании регистра состояния в список чувствительности должны быть включены тактовые сигналы, а также сигнал асинхронного сброса.

```
always @ (posedge clock or negedge reset_n)
```

При переходе сигнала сброса `reset_n` в **0** конечный автомат переходит в начальное состояние (**S0**).

```
if (! reset_n)
   state <= S0;
```

Когда сброс неактивен, нарастающий фронт тактового сигнала перепишет следующее состояние **next_state** со входа на выход **state** регистра состояния.

8.2.1 Описание выходной логики конечного автомата

Описание выходной логики в наиболее простом случае – это операция непрерывного присваивания. Для **автомата Мура**, выход которого определяется текущим состоянием, это приводит к очень компактному коду. Например:

```
assign y = (state == S2 || state == S1);
```

Для **автомата Мили** в описанную строку необходимо добавить состояние автомата так, чтобы выход зависел от входа и текущего состояния:

```
assign y = (input_signal & state == S1);
```

Использование комбинационного блока для описания выходной логики может приводить к таким проблемам[1]:

- возникновение **гонок сигналов** на выходе во время переходов автомата между состояниями;
- комбинационная логика на выходе может приводить к нарушению требований к проекту по быстродействию.

Рассмотрим последнее утверждение более подробно. Традиционно цифровой дизайн использует разделение всего проекта на части с помощью регистров. При этом комбинационные части между регистрами стараются делать с одинаковой задержкой, так как максимальная задержка определяет период тактовой частоты, с которой тактируются регистры. При соединении различных блоков выходы делают синхронными, т. е. на выходе каждого блока стоят регистры.

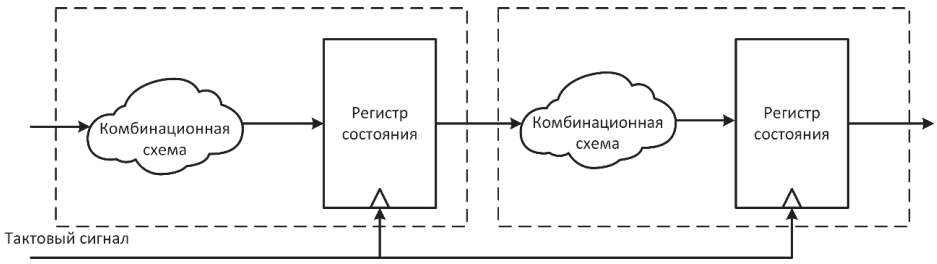


Рис. 8.5 Схема синхронного устройства

Если на выходе блока оставить комбинационную логику, то ее задержка будет суммироваться с задержкой комбинационной логики следующего блока и увеличивать требования к скорости проекта. Например, на рис. 8.6 приведена схема устройства, в котором **блок А** на выходе содержит комбинационную часть, и этот выход управляет несколькими другими блоками. Задержка комбинационной части на выходе **блока А** составляет 3,5 нс.

[1] *Clifford E. Cummings.* Coding And Scripting Techniques for FSM Designs With Synthesis-Optimized, Glitch-Free Outputs // SNUG (Synopsys Users Group Boston, MA 2000). Proceedings, September 2000.

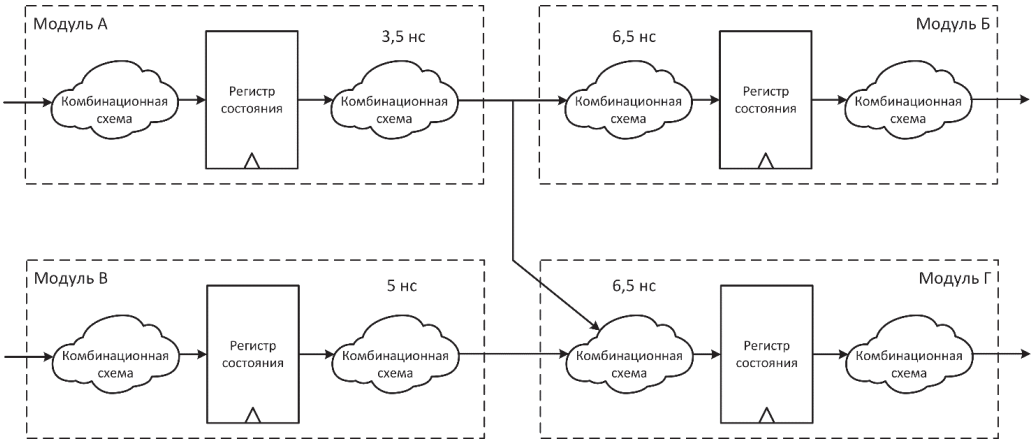


Рис. 8.6 Ограничения по тактовой частоте

При осуществлении разработки к проекту выдвигаются определенные требования по тактовой частоте работы и проводится проверка проекта на соответствие этим требованиям. Допустим, что к разрабатываемому проекту установлено ограничение на максимальное значение периода тактовой частоты 10 нс. Комбинационная часть **блока А** уже забрала себе 3,5 нс, т. е. всем остальным блокам для реализации комбинационной части остается 6,5 нс.

Комбинационная часть **блока Б** имеет задержку 6,5 нс, значит, требование по максимальной длительности периода удовлетворяется.

Комбинационная часть **блока Г** на вход получает сигналы с выхода **блока А** и выхода **блока В**. Задержка в комбинационной части блока Г составляет 6,5 нс. Значит, для связки «**блок А – блок Г**» требования по максимальному периоду удовлетворяются. В блоке В задержка в выходной комбинационной части составляет 5 нс. Значит, для связки «**блок В – блок Г**» требования по максимальному периоду не удовлетворяются, т. к.

$$5\ нс + 6{,}5\ нс = 11{,}5\ нс > 10\ нс.$$

Если бы все блоки на рис. 8.6 заканчивались регистрами, т. е. имели синхронные выходы, такая ситуация не возникла бы, и проект удовлетворял бы требованиям по частоте тактового сигнала.

Для добавления регистров на выход конечного автомата необходимо при описании выходной логики воспользоваться оператором **always**. Тогда с помощью оператора **case** описывают значения выходов в зависимости от состояния автомата для *автомата Мура* или состояния автомата и входа для **автомата Мили** (рис. 8.7).

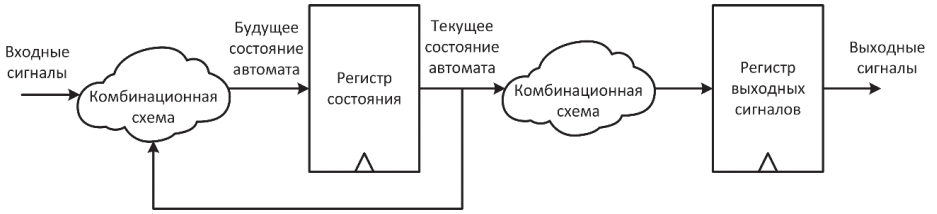


Рис. 8.7 Конечный автомат Мура с синхронными выходами

Использование регистра приводит к задержке выходного сигнала на один такт. Это происходит потому, что выходной сигнал определяется **текущим состоянием конечного автомата** и только в следующем такте будет переписан на выход. Если же для определения выходного сигнала воспользоваться **будущим состоянием конечного автомата**, то такой задержки не произойдет (рис. 8.8).

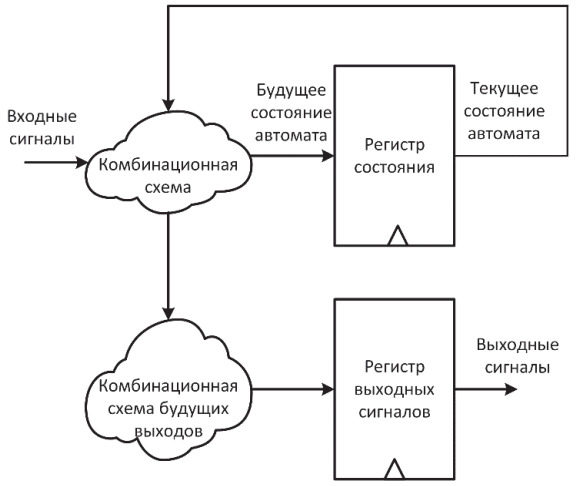


Рис. 8.8 Конечный автомат с тактированием будущего выхода КА

8.3 Описание конечного автомата Мура на языке Verilog

Рассмотрим **HDL**-описание автомата **Мура**, диаграмма состояния которого, таблица переходов и выходная таблица автомата приведены на рис. 8.3, в табл. 8.1 и табл. 8.2.

У автомата есть внешние порты:

```
input clock, // clock for FSM
input reset_n, // reset with low active level
input enable, // enable to write next state
input a, // one-bit input signal
output y // one-bit output of FSM
```

Листинг 8.2 Внешние порты автомата Мура

Этот автомат имеет три состояния **S0**, **S1**, **S2** с двоичным кодированием:

```
parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
```

Объявим два сигнала для текущего состояния (**state**) и будущего состояния (**next_state**).

```
reg [1: 0] state, next_state;
```

Описание регистра состояния:

```
always @ (posedge clock or negedge reset_n)
   if (! reset_n)
      state <= S0;
   else if (enable)
      state <= next_state;
```

Комбинационный блок для определения будущего состояния реализуется с помощью оператора **case**. В операторе case явно прописано поведение для всех трех используемых состояний **S0**, **S1**, **S2**. Кодирование неиспользуемого состояния обрабатывается строкой по умолчанию, которая возвращает конечный автомат в состояние S0.

```
always @*
   case (state)
   S0:
      if (a)
         next_state = S1;
      else
         next_state = S0;
   S1:
      if (a)
         next_state = S2;
      else
         next_state = S1;
   S2:
      if (a)
         next_state = S0;
      else
         next_state = S2;
      default: next_state = S0;
endcase
```

Листинг 8.3 Блок определения будущего состояния для автомата Мура

Выход описывается непрерывным присваиванием:

```
assign y = (state == S2 || state == S1);
```

Полный код, описывающий **автомат Мура**, приведен в листинге 8.4.

```
module lab8_1
(
   input clock,
   input reset_n,
   input enable,
   input a,
   output y
);
   parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
   reg [1:0] state, next_state;

   // State register
   always @ (posedge clock or negedge reset_n)
      if (! reset_n)
         state <= S0;
      else if (enable)
         state <= next_state;

   // Next state logic
   always @*
      case (state)
      S0:
         if (a)
            next_state = S1;
         else
            next_state = S0;
      S1:
         if (a)
            next_state = S2;
         else
            next_state = S1;
      S2:
         if (a)
            next_state = S0;
         else
            next_state = S2;
         default:
            next_state = S0;
      endcase

   // Output logic based on current state
   assign y = (state == S2 || state == S1);
endmodule
```

Листинг 8.4 Конечный автомат Мура

8.3.1 Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 8.4 в файле **lab8_1.v**. Это файл верхнего уровня разрабатываемого проекта. Откомпилируйте проект.

В результате компиляции **автомат Мура** будет синтезирован в схему на рис. 8.9. Данный рисунок был получен с использованием **RTL Veiwer**. В конечном автомате синтезирована схема сброса из логического элемента **ИЛИ**, системы под названием state и выходного логического элемента **ИЛИ**. Если открыть блок **state**, то в **State Machine Viewer** отобразится граф конечного автомата (рис. 8.10). Чтобы открыть этот инструмент **Quartus Prime**, нужно использовать меню **Tools → Netlist Viewer → State Machine Viewer**.

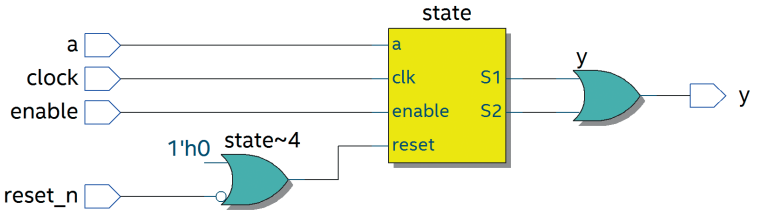


Рис. 8.9 Конечный автомат Мура в RTL Viewer

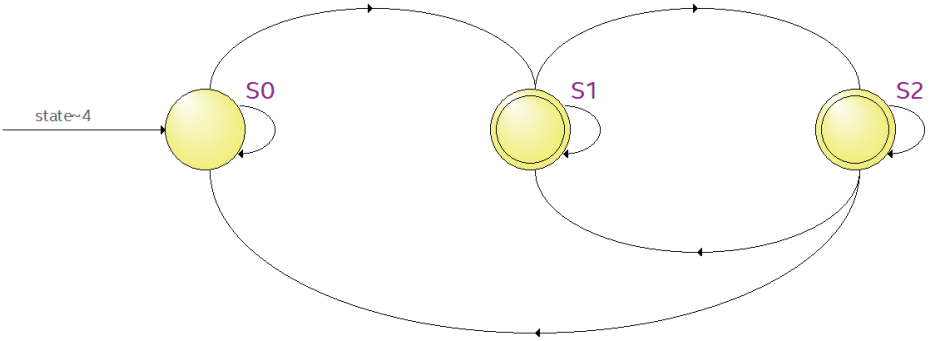
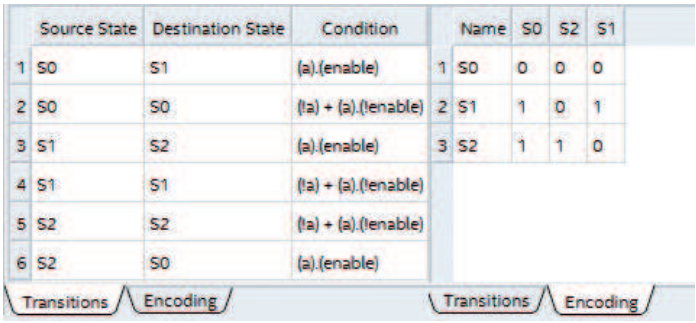


Рис. 8.10 Конечный автомат Мура в State Machine Viewer

Рассмотрим синтезированный конечный автомат. Он несколько отличается от приведенного на рис. 8.3. Прежде всего конечный автомат отличается наличием входа в первое состояние **S0** – вход в него происходит из состояния **state~4**, которое в коде нигде явно не описано. Это результат синтеза строки:

```
default: next_state = S0;
```

Quartus Prime синтезировал уравнения для логики переходов (**Transition Table**) и формирования выходного сигнала (**Encoding Table**), которые открываются в окне **State Machine Viewer** (рис. 8.11). В условиях перехода использованы не только значения сигнала (как это было приведено на исходном графе конечного автомата), но и значения сигнала **enable**.



	Source State	Destination State	Condition
1	S0	S1	(a).(enable)
2	S0	S0	(!a) + (a).(!enable)
3	S1	S2	(a).(enable)
4	S1	S1	(!a) + (a).(!enable)
5	S2	S2	(!a) + (a).(!enable)
6	S2	S0	(a).(enable)

Transitions | Encoding

	Name	S0	S2	S1
1	S0	0	0	0
2	S1	1	0	1
3	S2	1	1	0

Transitions | Encoding

Рис. 8.11 Таблицы переходов и выходов в State Machine Viewer

Назначьте выводы в соответствии с таблицей:

Таблица 8.4 Соответствие выводов для проектов lab8_1–lab8_4

Имя сигнала	Направление передачи данных	Номер вывода в ПЛИС	Стандарт ввода-вывода
a	Input	PIN_C10	3.3-V LVTTL
enable	Input	PIN_C11	3.3-V LVTTL
reset_n	Input	PIN_D12	3.3-V LVTTL
clock	Input	PIN_B8	3.3 V Schmitt Trigger
y	Output	PIN_A8	3.3-V LVTTL

Скомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате. Проверьте работоспособность проекта. Для этого необходимо выполнить следующие действия:

1) используя кнопки и микропереключатели для подачи сигналов на входы, проверьте их значения с помощью светодиодов;

2) проверьте таблицы переходов и выходов конечного автомата.

8.3.2 Моделирование конечного автомата Мура в ModelSim

Для проведения моделирования конечного автомата необходимо проверить логику его переходов в каждом состоянии. Для этого прежде всего нужно выполнить начальную установку сигналов и сброс автомата:

```
clock = 1;
reset_n = 0;
enable = 1;
a = 1;
```

Проведите несколько циклов моделирования в каждом состоянии с такой последовательностью действий:

1. Сброс сигнала разрешения работы **enable** – проверка на останов работы автомата.

2. Установка сигнала разрешения работы **enable** и проверка работы автомата при всех комбинациях входных сигналов. В рассматриваемом случае это два состояния сигнала **a**.

```
repeat (4)
begin
   enable = 0;
   #20; a = 0;
   enable = 1;
   #20; a = 1;
   #20;
end
```

Для того чтобы увидеть в результате моделирования значения состояний автомата, выведем этот сигнал наружу:

```
assign state = dut.state;
```

Полный код тестбенча для разработанного конечного **автомата Мура** приведен в листинге 8.5.

```
`timescale 1ns/1ns
module testbench;
   reg clock;
   reg reset_n;
   reg enable;
   reg a;
   wire y;
   wire [1:0] state;
   lab8_1 dut (clock, reset_n, enable, a, y);
   assign state = dut.state;

initial
   begin
      // set inital values of signal
      clock = 1;
      reset_n = 0;
      enable = 1;
      a = 1;
      #10; reset_n = 1;
      repeat (4)
      begin
         enable = 0;
         #20; a = 0;
         enable = 1;
         #20; a = 1;
         #20;
      end
   end
```

```
   always #10 clock = ~clock;
   initial
      #250 $finish;
   initial
      $monitor("clock=%b, reset_n=%b, enable=%b, a=%b, y=%b, state=%b",
         clock, reset_n, enable, a, y, state);
   initial
      $dumpvars; //iverilog dump init
endmodule
```

Листинг 8.5 Тестбенч для конечного автомата Мура

Результат моделирования конечного автомата **Мура** в пакете **ModelSim** приведен на рис. 8.12.

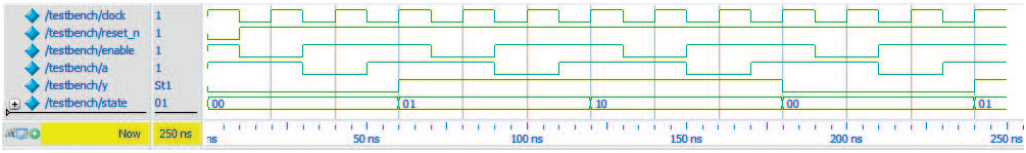


Рис. 8.12 Моделирование конечного автомата Мура в ModelSim

8.3.3 Автомат Мура с регистровыми выходами

Опишем конечный автомат **Мура** с регистровыми выходами. Для этого в код листинга 8.4 необходимо добавить описание регистровых выходов:

```
always @ (posedge clock)
   case (state)
      S0: y <= 0;
      S1: y <= 1;
      S2: y <= 1;
      default: y <= 0;
   endcase
```

Листинг 8.6 Регистровые выходы в автомате Мура

Также необходимо изменить описание выхода:

```
output reg y
```

Результат компиляции конечного автомата в **RTL Viewer** приведен на рис. 8.13. Из этого рисунка следует, что в проект добавился регистр, в котором сохраняется значение выходного сигнала. Этот регистр дает задержку выходного сигнала на один такт (рис. 8.14). Для сравнения можно посмотреть результат симуляции конечного автомата с комбинационной логикой выходов (рис. 8.12).

Тестбенч для симуляции конечного автомата не изменился и совпадает с прошлым примером (листинг 8.5).

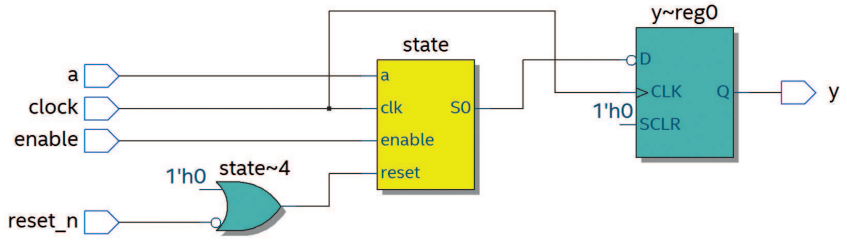


Рис. 8.13 Конечный автомат в RTL Viewer

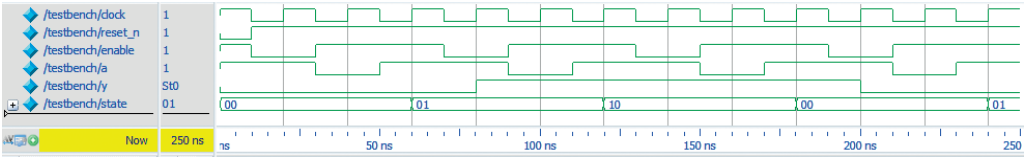


Рис. 8.14 Моделирование конечного автомата Мура в ModelSim

Для того чтобы убрать задержку в один такт, необходимо изменить логику формирования выходных сигналов с учетом будущего состояния конечного автомата. Полное описание автомата **Мура** приведено в листинге 8.7.

```
module lab8_3
(
   input clock,
   input reset_n,
   input enable,
   input a,
   output reg y
);
   parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
   reg [1:0] state, next_state;

   // State register
   always @ (posedge clock or negedge reset_n)
      if (! reset_n)
         state <= S0;
      else if (enable)
         state <= next_state;

   // Next state logic
   always @*
      case (state)
         S0:
            if (a)
               next_state = S1;
            else
               next_state = S0;
         S1:
```

```
            if (a)
               next_state = S2;
            else
               next_state = S1;
         S2:
            if (a)
               next_state = S0;
            else
               next_state = S2;
            default:
               next_state = S0;
      endcase

   // Output logic based on current state
   always @ (posedge clock or negedge reset_n)
      begin
         if (! reset_n)
            y <= 0;
         else if (enable)
            begin
               y <= 1;
               case (state)
                  S0:
                     if (!a)
                        y <= 0;
                  S2:
                     if (a)
                        y <= 0;
               endcase
            end
      end
endmodule
```

Листинг 8.7 Автомат Мура с синхронными выходами с будущим значением выхода

Результат компиляции конечного автомата в **RTL Viewer** приведен на рис. 8.15. В схему, кроме регистра, добавилась логика формирования будущего значения выходного сигнала. Результат симуляции такой версии конечного автомата приведен на рис. 8.16. При сравнении результатов симуляции конечного автомата с комбинационной логикой выходов (рис. 8.12) и полученного результата видно, что они одинаковы.

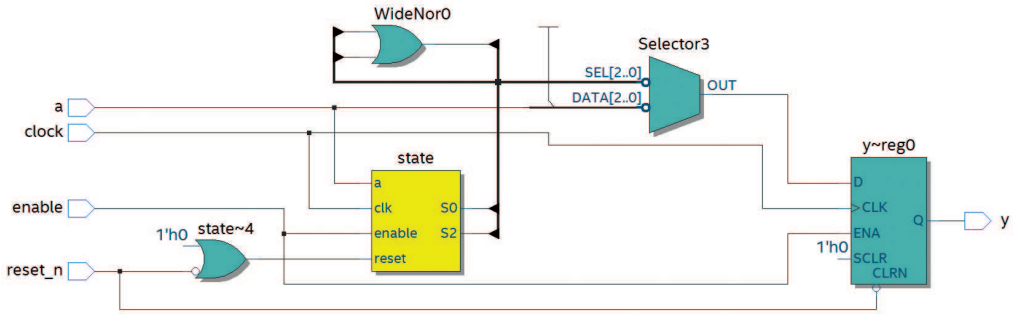


Рис. 8.15 Конечный автомат Мура с регистровым выходом в RTL Viewer

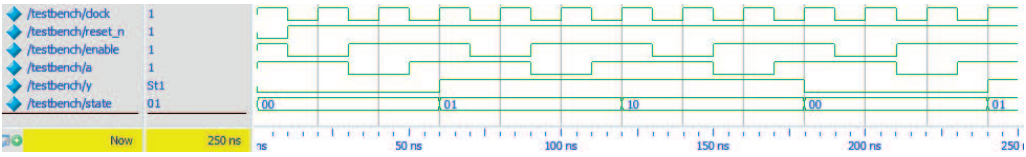


Рис. 8.16 Результат симуляции конечного автомата Мура с регистровым выходом в ModelSim

8.3.4 Создание и компиляция проекта

Создайте новый проект. Сохраните листинг 8.7 в файле **lab8_3.v**. Это файл верхнего уровня для разрабатываемого проекта. Откомпилируйте проект.

Назначение выводов для проекта выполните в соответствии с табл. 8.4.

Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.

Проверьте работоспособность проекта. Для этого выполните следующие действия:

1. Используя кнопки и микропереключатели для подачи сигналов на входы, проверьте их значения с помощью светодиодов.
2. Проверьте таблицы переходов и выходов конечного автомата.

В файле **lab8_3.v** измените блок **always** в соответствии с листингом 8.6. Сохраните листинг в файле **lab8_2.v**. Это файл верхнего уровня для нового проекта. Откомпилируйте проект.

Назначение выводов для проекта выполните в соответствии с табл. 8.4.

Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.

Проверьте работоспособность проекта. Для этого необходимо выполнить следующие действия:

1. Используя кнопки и микропереключатели для подачи сигналов на входы, проверить их значения с помощью светодиодов.
2. Проверить таблицы переходов и выходов конечного автомата.

Обратите внимание на задержку выходного сигнала на один такт.

Дополнительное задание для самостоятельной работы

В приведенной выше реализации **автомата Мура** задайте различные способы кодирования состояний и сравните результаты компиляции.

8.4 Описание конечного автомата Мили на языке Verilog

Разработаем описание **конечного автомата Мили** на языке **Verilog**. Граф и таблица переходов и выходов автомата **Мили** приведены на рис. 8.17, в табл. 8.5 и в 8.6.

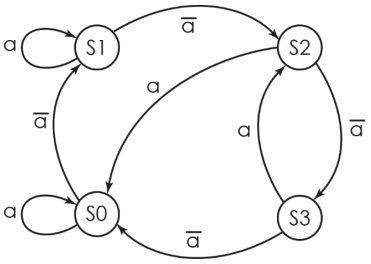


Рис. 8.17 Граф конечного автомата Мили

Таблица 8.5 Таблица состояний конечного автомата Мили

Текущее состояние	Будущее состояние	Условие перехода
S0	S0	a=1
S0	S1	a=0
S1	S1	a=1
S1	S2	a=0
S2	S0	a=1
S2	S3	a=0
S3	S2	a=1
S3	S0	a=0

Таблица 8.6 Таблица выходов конечного автомата Мили

Текущее состояние	a = 0	a = 1
S0	0	0
S1	0	1
S2	0	0
S3	0	0

Внешние порты **автомата Мили** аналогичны портам **автомата Мура**. В разрабатываемом автомате **Мили** использовано четыре состояния **S0**, **S1**, **S2**, **S3**, представленных в коде Грея:

```
parameter [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b11, S3 = 2'b10;
```

Объявите переменные для хранения текущего состояния (**state**) и будущего состояния (**next_state**).

```
reg [1:0] state, next_state;
```

Описание регистра состояний и комбинационной схемы для логики переходов аналогично **автомату Мура** (листинг 8.4). Для генерации выходного сигнала использован оператор непрерывного присваивания. Выход зависит не только от текущего состояния, но и от входного сигнала **a**:

```
assign y = (a & state == S1);
```

Полный код для **автомата Мили** приведен в листинге 8.8.

```
module lab8_4
(
   input clock,
   input reset_n,
   input enable,
   input a,
   output y
);
   parameter [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b11, S3 = 2'b10;
   reg [1:0] state, next_state;

   // State register
always @ (posedge clock or negedge reset_n)
   if (! reset_n)
      state <= S0;
   else if (enable)
      state <= next_state;

// Next state logic
always @*
   case (state)
      S0:
         if (a)
            next_state = S0;
         else
            next_state = S1;
      S1:
         if (a)
            next_state = S1;
         else
            next_state = S2;
      S2:
         if (a)
            next_state = S0;
```

```
            else
               next_state = S3;
         S3:
            if (a)
               next_state = S2;
            else
               next_state = S0;
            default:
               next_state = S0;
      endcase
      // Output logic based on current state
      assign y = (a & state == S1);
endmodule
```

Листинг 8.8 Код конечного автомата Мили

8.4.1 Создание и компиляция проекта

Создайте новый проект. Сохраните код листинга 8.8 в файле **lab8_5.v**. Это файл верхнего уровня для разрабатываемого проекта. Откомпилируйте проект.

Назначение выводов для проекта выполните в соответствии с табл. 8.1.

Откомпилируйте проект и загрузите конфигурацию в **ПЛИС** на отладочной плате.

Проверьте работоспособность проекта. Для этого выполните следующие действия:

1) используя кнопки и микропереключатели для подачи сигналов на входы, проверьте их значения с помощью светодиодов;

2) проверьте таблицы переходов и выходов конечного автомата.

В результате компиляции **конечный автомат Мили** будет синтезирован в схему, приведенную на рис. 8.18. Конечный автомат получил дополнительную логику для формирования выходного сигнала с учетом входного сигнала.

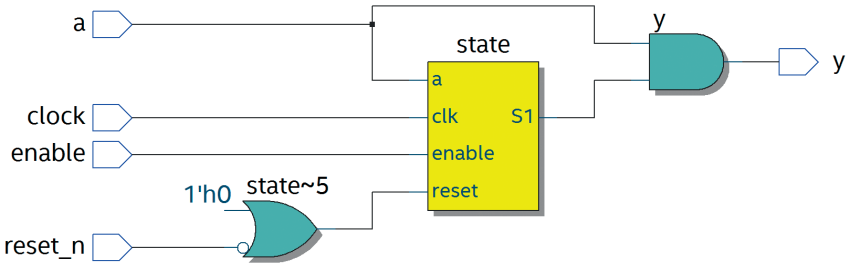


Рис. 8.18 Конечный автомат Мили в RTL Viewer

Граф автомата в **State Machine Viewer** приведен на рис. 8.19. Он несколько отличается от приведенного на рис. 8.17: прежде всего тем, что снова вводится начальное состояние, в которое автомат входит при сбросе. Также добавилась

дополнительная петля для состояния **S2**, которая возникает, когда нет разрешающего сигнала.

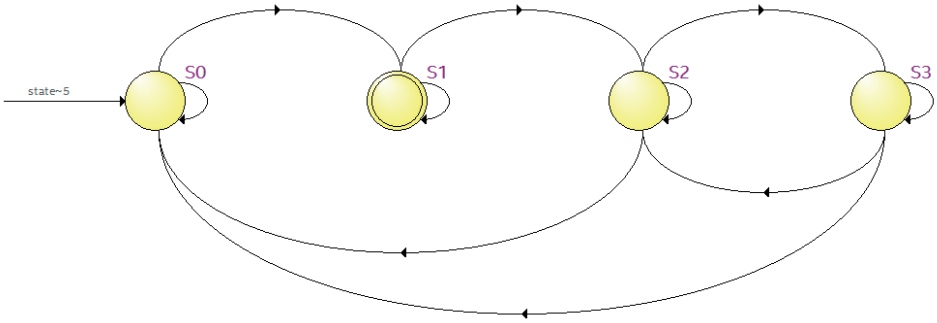


Рис. 8.19 Граф автомата Мили в State Machine Viewer

Таблицы переходов и выходов конечного автомата **Мили** приведены на рис. 8.20.

	Source State	Destination State	Condition
1	S0	S1	(!a).(enable)
2	S0	S0	(!a).(!enable) + (a)
3	S1	S2	(!a).(enable)
4	S1	S1	(!a).(!enable) + (a)
5	S2	S3	(!a).(enable)
6	S2	S2	(!enable)
7	S2	S0	(a).(enable)
8	S3	S3	(!enable)
9	S3	S2	(a).(enable)
10	S3	S0	(!a).(enable)

Transitions | Encoding

State Table

	Name	S2	S3	S1	S0
1	S0	0	0	0	0
2	S1	0	0	1	1
3	S3	0	1	0	1
4	S2	1	0	0	1

Transitions | Encoding

Рис. 8.20 Таблицы переходов и выходов конечного автомата Мили

8.4.2 Моделирование конечного автомата Мили в ModelSim

Тестбенч для **автомата Мили** в точности повторяет тестбенч для **автомата Мура** из листинга 8.5. Результат симуляции **автомата Мили** приведен на рис. 8.21.

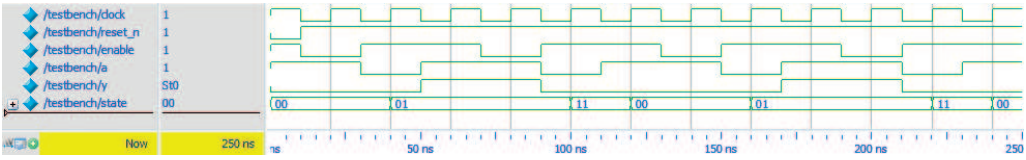


Рис. 8.21 Результат симуляции автомата Мили

Дополнительное задание для самостоятельной работы

В приведенной выше реализации **автомата Мили** задайте различные способы кодирования состояний и сравните результаты компиляции.

8.5 Упражнения

8.5.1 Основное задание

Выполните все дополнительные задания данной главы.

В соответствии с таблицей своего варианта постройте граф конечного автомата. Разработайте код конечного автомата с комбинационными выходами и синхронными выходами. Выполните компиляцию проекта и синтезируйте **RTL**-представление проекта. Проведите моделирование конечного автомата.

Вариант 1

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S1/Y1	S4/Y0	S0/Y0	S0/Y0
A1	S2/Y1	S2/Y3	S1/Y2	S1/Y1
A2	S3/Y1	S4/Y2	S1/Y4	S2/Y2
A3	S4/Y1	S2/Y3	S2/Y4	S3/Y3

Вариант 2

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S1/Y1	S4/Y3	S0/Y2	S0/Y0
A1	S1/Y1	S2/Y3	S2/Y2	S1/Y4
A2	S2/Y1	S4/Y1	S1/Y3	S3/Y2
A3	S2/Y1	S2/Y3	S2/Y4	S3/Y3

Вариант 3

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S1/Y1	S2/Y3	S0/Y2	S0/Y2
A1	S1/Y1	S3/Y1	S2/Y0	S1/Y3
A2	S2/Y1	S4/Y1	S1/Y1	S3/Y3
A3	S2/Y1	S2/Y3	S2/Y4	S3/Y3

Вариант 4

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S1/Y1	S2/Y1	S0/Y0	S0/Y2
A1	S1/Y3	S0/Y0	S1/Y0	S1/Y1
A2	S2/Y3	S4/Y1	S2/Y2	S2/Y0
A3	S2/Y2	S2/Y3	S2/Y4	S3/Y3

Вариант 5

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S3/Y3	S2/Y2	S1/Y1	S1/Y2
A1	S2/Y2	S3/Y2	S0/Y1	S1/Y1
A2	S2/Y2	S2/Y3	S2/Y3	S2/Y2
A3	S3/Y3	S1/Y0	S1/Y0	S2/Y2

Вариант 6

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S0/Y0	S1/Y1	S0/Y1	S2/Y1
A1	S1/Y0	S1/Y2	S2/Y2	S0/Y2
A2	S0/Y1	S1/Y3	S0/Y0	S0/Y3
A3	S2/Y3	S2/Y1	S3/Y0	S3/Y1

Вариант 7

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S1/Y2	S1/Y1	S1/Y3	S3/Y3
A1	S2/Y0	S2/Y2	S3/Y2	S2/Y1
A2	S3/Y2	S2/Y0	S1/Y0	S3/Y0
A3	S1/Y0	S0/Y0	S2/Y2	S2/Y3

Вариант 8

Входной сигнал	Состояние			
	S0	**S1**	**S2**	**S3**
A0	S1/Y0	S3/Y3	S0/Y2	S2/Y1
A1	S3/Y0	S2/Y1	S3/Y0	S2/Y2
A2	S1/Y1	S3/Y1	S3/Y2	S3/Y0
A3	S2/Y0	S2/Y1	S1/Y0	S3/Y0

Вариант 9

Входной сигнал	Состояние			
	S0	S1	S2	S3
A0	S3/Y2	S2/Y3	S3/Y3	S2/Y3
A1	S1/Y2	S3/Y2	S3/Y1	S3/Y0
A2	S2/Y3	S2/Y0	S1/Y3	S3/Y1
A3	S0/Y3	S0/Y3	S0/Y1	S0/Y3

Вариант 10

Входной сигнал	Состояние			
	S0	S1	S2	S3
A0	S0/Y1	S1/Y1	S1/Y3	S1/Y0
A1	S0/Y2	S2/Y3	S2/Y2	S2/Y3
A2	S1/Y3	S2/Y0	S3/Y0	S2/Y1
A3	S1/Y0	S2/Y1	S0/Y2	S2/Y2

Вариант 11

Входной сигнал	Состояние			
	S0	S1	S2	S3
A0	S3/Y1	S2/Y1	S3/Y3	S0/Y0
A1	S0/Y2	S2/Y3	S0/Y2	S1/Y3
A2	S0/Y3	S3/Y0	S0/Y0	S2/Y1
A3	S3/Y0	S1/Y1	S0/Y2	S1/Y2

Вариант 12

Входной сигнал	Состояние			
	S0	S1	S2	S3
A0	S0/Y2	S2/Y3	S1/Y3	S2/Y3
A1	S0/Y2	S1/Y2	S0/Y1	S2/Y0
A2	S3/Y2	S1/Y3	S0/Y3	S3/Y1
A3	S1/Y1	S1/Y2	S3/Y2	S0/Y3

Вариант 13

Входной сигнал	Состояние			
	S0	S1	S2	S3
A0	S0/Y0	S0/Y0	S1/Y1	S3/Y1
A1	S2/Y1	S3/Y2	S3/Y2	S1/Y2
A2	S0/Y1	S3/Y2	S0/Y3	S1/Y0
A3	S3/Y1	S1/Y2	S2/Y2	S2/Y2

Вариант 14

Входной сигнал	Состояние/Выход			
	S0/Y1	S1/Y0	S2/Y2	S3/Y0
A0	S0	S0	S1	S1
A1	S1	S2	S2	S2
A2	S1	S2	S3	S0
A3	S1	S2	S2	S2

Вариант 15

Входной сигнал	Состояние/Выход			
	S0/Y1	S1/Y1	S2/Y1	S3/Y0
A0	S1	S2	S2	S2
A1	S1	S2	S3	S0
A2	S1	S2	S2	S2
A3	S1	S2	S3	S0

Вариант 16

Входной сигнал	Состояние/Выход			
	S0/Y2	S1/Y1	S2/Y0	S3/Y2
A0	S3	S3	S0	S2
A1	S0	S0	S1	S2
A2	S0	S0	S1	S2
A3	S1	S2	S2	S1

Вариант 17

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y1	S2/Y2	S3/Y2
A0	S3	S0	S0	S1
A1	S0	S1	S2	S2
A2	S0	S2	S1	S2
A3	S0	S0	S1	S2

Вариант 18

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y1	S2/Y0	S3/Y2
A0	S1	S2	S1	S1
A1	S2	S1	S0	S0
A2	S2	S2	S2	S3
A3	S2	S1	S2	S3

Вариант 19

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y0	S2/Y1	S3/Y2
A0	S3	S0	S0	S1
A1	S0	S1	S2	S1
A2	S2	S2	S1	S1
A3	S1	S2	S2	S1

Вариант 20

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y1	S2/Y1	S3/Y0
A0	S0	S2	S1	S2
A1	S0	S0	S1	S2
A2	S1	S2	S1	S2
A3	S2	S1	S1	S2

Вариант 21

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y2	S2/Y1	S3/Y2
A0	S1	S2	S2	S2
A1	S1	S2	S3	S0
A2	S1	S3	S2	S3
A3	S3	S0	S2	S0

Вариант 22

Входной сигнал	Состояние/Выход			
	S0/Y2	S1/Y1	S2/Y0	S3/Y0
A0	S1	S2	S2	S2
A1	S1	S2	S3	S0
A2	S1	S3	S2	S3
A3	S3	S0	S2	S0

Вариант 23

Входной сигнал	Состояние/Выход			
	S0/Y2	S1/Y2	S2/Y1	S3/Y0
A0	S0	S1	S2	S1
A1	S2	S1	S2	S1
A2	S1	S1	S2	S1
A3	S2	S1	S2	S1

Вариант 24

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y2	S2/Y1	S3/Y1
A0	S0	S2	S0	S3
A1	S0	S3	S3	S1
A2	S1	S3	S0	S2
A3	S3	S1	S1	S2

Вариант 25

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y1	S2/Y1	S3/Y1
A0	S0	S3	S0	S2
A1	S3	S1	S0	S1
A2	S0	S2	S3	S1
A3	S1	S2	S1	S1

Вариант 26

Входной сигнал	Состояние/Выход			
	S0/Y1	S1/Y2	S2/Y0	S3/Y1
A0	S0	S2	S1	S2
A1	S0	S1	S0	S2
A2	S3	S1	S0	S3
A3	S1	S1	S3	S0

Вариант 27

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y1	S2/Y2	S3/Y0
A0	S1	S2	S2	S1
A1	S0	S2	S0	S1
A2	S0	S3	S0	S2
A3	S3	S1	S0	S1

Вариант 28

Входной сигнал	Состояние/Выход			
	S0/Y0	S1/Y1	S2/Y0	S3/Y1
A0	S0	S2	S0	S1
A1	S0	S3	S0	S2
A2	S2	S3	S0	S0
A3	S0	S0	S1	S2

8.5.2 Задания для самостоятельной работы

1. Напишите возле каждой диаграммы состояния (рис. 8.22) имя соответствующего модуля из листинга 8.9:

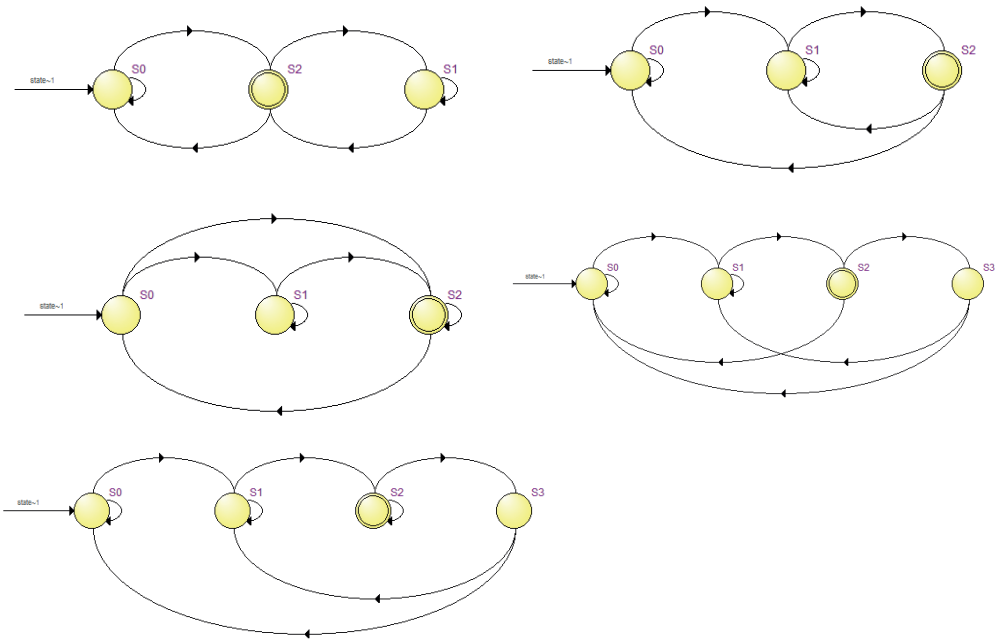


Рис. 8.22 Задание для самостоятельного изучения 1

```
module top1
(
   input clock,
   input resetn,
   input a,
   output y
);
   parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
   reg [1:0] state, next_state;

   // state register
   always @ (posedge clock or negedge resetn)
      if (! resetn)
         state <= S0;
      else
         state <= next_state;

   // next state logic
   always @*
      begin
```

```
        next_state = state;
        case (state)
           S0:
              if (! a)
                 next_state = S1;
           S1:
              if (a)
                 next_state = S2;
           S2:
              if (a)
                 next_state = S0;
              else
                 next_state = S1;
        endcase
     end

  // output logic
  assign y = (state == S2);
endmodule

module top2
(
  input clock,
  input resetn,
  input a,
  output y
);
  parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
  reg [1:0] state, next_state;

  // state register
  always @ (posedge clock or negedge resetn)
     if (! resetn)
        state <= S0;
     else
        state <= next_state;

  // next state logic
  always @*
     begin
        next_state = state;
        case (state)
           S0:
              if (! a)
                 next_state = S2;
           S1:
              if (a)
```

```
                next_state = S2;
            S2:
               if (a)
                  next_state = S0;
               else
                  next_state = S1;
         endcase
      end

   // output logic
   assign y = (state == S2);
endmodule

module top3
(
   input clock,
   input resetn,
   input a,
   output y
);
   parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
   reg [1:0] state, next_state;

   // state register
   always @ (posedge clock or negedge resetn)
      if (! resetn)
         state <= S0;
      else
         state <= next_state;

   // next state logic
   always @*
      begin
         next_state = state;
         case (state)
            S0:
               if (! a)
                  next_state = S1;
               else
                  next_state = S2;
            S1:
               if (a)
                  next_state = S2;
            S2:
               if (a)
               next_state = S0;
         endcase
```

```
      end
   // output logic
   assign y = (state == S2);
endmodule

module top4
(
   input clock,
   input resetn,
   input a,
   output y
);
   parameter [2:0] S0 = 0, S1 = 1, S2 = 2, S3 = 3;
   reg [1:0] state, next_state;

   // state register
   always @ (posedge clock or negedge resetn)
      if (! resetn)
         state <= S0;
      else
         state <= next_state;

   // next state logic
   always @*
      begin
         next_state = state;
         case (state)
            S0:
               if (! a)
                  next_state = S1;
            S1:
               if (a)
                  next_state = S2;
            S2:
               if (a)
                  next_state = S0;
               else
                  next_state = S3;
            S3:
               if (a)
                  next_state = S0;
               else
                  next_state = S1;
         endcase
      end
   // output logic
   assign y = (state == S2);
```

```
endmodule

module top5
(
   input clock,
   input resetn,
   input a,
   output y
);
   parameter [2:0] S0 = 0, S1 = 1, S2 = 2, S3 = 3;
   reg [1:0] state, next_state;

   // state register
   always @ (posedge clock or negedge resetn)
      if (! resetn)
         state <= S0;
      else
         state <= next_state;

   // next state logic
   always @*
      begin
         next_state = state;
         case (state)
            S0:
               if (! a)
                  next_state = S1;
            S1:
               if (a)
                  next_state = S2;
            S2:
               if (a)
                  next_state = S3;
            S3:
               if (a)
                  next_state = S0;
               else
                  next_state = S1;
         endcase
      end
   // output logic
   assign y = (state == S2);
endmodule
```

Листинг 8.9 Задание для самостоятельного изучения 1

2. Напишите возле каждого рисунка (рис. 8.23) имя соответствующего модуля из листинга 8.10:

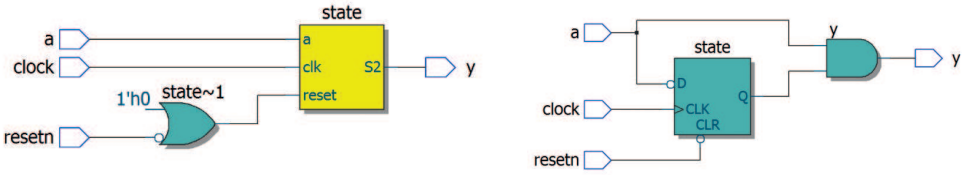


Рис. 8.23 Задание для самостоятельного изучения 2

```
module top1
(
   input clock,
   input resetn,
   input a,
   output y
);
   parameter S0 = 0, S1 = 1;
   reg state, next_state;

   // state register
   always @ (posedge clock or negedge resetn)
      if (! resetn)
         state <= S0;
      else
         state <= next_state;

   // next state logic
   always @*
      case (state)
         S0:
            if (a)
               next_state = S0;
            else
               next_state = S1;
         S1:
            if (a)
               next_state = S0;
            else
               next_state = S1;
         default:
            next_state = S0;
      endcase

   // output logic
   assign y = (a & state == S1);
endmodule

module top15
(
   input clock,
```

```
   input resetn,
   input a,
   output y
);
   parameter [1:0] S0 = 0, S1 = 1, S2 = 2;
   reg [1:0] state, next_state;

   // state register
   always @ (posedge clock or negedge resetn)
      if (! resetn)
         state <= S0;
      else
         state <= next_state;

   // next state logic
   always @*
      case (state)
         S0:
            if (a)
               next_state = S0;
            else
               next_state = S1;
         S1:
            if (a)
               next_state = S2;
            else
               next_state = S1;
         S2:
            if (a)
               next_state = S0;
            else
               next_state = S1;
         default:
            next_state = S0;
      endcase
   // output logic
   assign y = (state == S2);
endmodule
```

Листинг 8.10 Задание для самостоятельного изучения 2

8.5.3 Контрольные вопросы

1. Опишите понятие конечного автомата. Приведите отличия между автоматами **Мура** и **Мили**.
2. Приведите схему конечного автомата **Мура**.
3. Приведите схему конечного автомата **Мили**.

4. Приведите примеры методов кодирования конечных автоматов. Охарактеризуйте их.
5. Приведите пример описания конечного автомата **Мура** на **Verilog**.
6. Приведите пример описания конечного автомата **Мили** на **Verilog**.
7. Как происходит кодирование конечных автоматов в **State Machine Viewer Quartus Prime**?

8.5.4 Темы для индивидуальной работы

Распознавание последовательности нажатий клавиш

Необходимо разработать автомат управления цифровым замком, который распознает последовательность нажатий клавиш. Если последовательность нажатий была правильная, то замок открывается, если последовательность не верна, то замок остается закрытым. Код включает в себя **6** цифр.

При включении питания замок закрывается, и начинается ожидание нажатия клавиш – поступления кодов клавиш на вход автомата. На отладочной плате в качестве клавиш можно использовать переключатели или подключить к плате дополнительную периферию (кнопки, переключатели и т. п.).

При любом нажатии на клавишу более **1** секунды происходит индикация нажатия клавиши, и код клавиши считывается. Если клавиша остается нажатой еще **1** секунду, то индикатор нажатой клавиши гаснет. Если клавиша остается и далее нажатой еще **1** секунду, то опять загорается индикатор, и происходит считывание клавиши. Так продолжается до тех пор, пока не будет считано **6** кодов клавиш.

После считывания **6** кодов клавиш происходит либо индикация открытия замка, либо ошибочно введенного кода.

Система управления светофором с датчиками

Существует светофор, который регулирует движение через улицу. У светофора с каждой стороны есть датчик наличия пешехода.

После включения светофор включает сначала красный сигнал, который горит **3** минуты, затем на **1** секунду включается желтый, и потом все время горит зеленый. Когда пешеход подходит к светофору с любой стороны, зеленый сигнал сначала мигает **5** секунд, затем на **1** секунду загорается желтый, потом на **3** минуты включается красный. После этого **1** секунду горит желтый, и опять зеленый.

Если после включения зеленого сигнала прошло менее **3** минут, то даже при наличии пешехода светофор не переключается на красный цвет. Только после того, как с момента включения зеленого сигнала прошло более **3** минут, светофор начинает реагировать на сигнал с датчика присутствия пешехода.

Необходимо разработать автомат управления светофором. В качестве датчиков присутствия пешеходов используйте переключатели на отладочной плате.

Станислав Жельнио, Александр Романов

Цифровой синтез: практический курс

Глава 9. Использование конечных автоматов для связи с периферийными устройствами

Содержание

Глава рассматривает применение конечных автоматов для решения практических задач по взаимодействию с периферийными устройствами. В качестве такого устройства используется ***цифровой датчик освещенности*** *(модуль* ***Ambient Light Sensor*** *от* ***Digilent****). Также в данной главе представлены различные стили кодирования конечных автоматов на языке* ***Verilog****.*

Требования к аппаратному и программному обеспечению

Для выполнения практических заданий к данной главе необходимо следующее оборудование и программное обеспечение:

- *рабочая станция с ОС Windows*[1]*, x64, 8Гб ОЗУ, порт USB;*
- *ПО* ***Quartus Prime Lite Edition*** *17.0;*
- *ПО* ***Icarus Verilog*** *и* ***GTKWave****;*
- *отладочная плата* ***Terasic DE10-Lite*** *(или аналогичная);*
- *модуль Digilent* ***PMOD ALS***[2] *(датчик освещенности);*
- *соединительный кабель.*

Документация на отладочную плату и датчик освещенности расположена в материалах[3] *к данной главе в подкаталоге* ***doc****. Инструкции к практической части этой главы могут быть адаптированы для использования с любой отладочной платой на основе* ***ПЛИС Intel FPGA/Altera****.*

9.1 Конечные автоматы для связи с периферийными устройствами

9.1.1 Конечный автомат[4]

Конечный автомат (автомат, **FSM – Finite State Machine**, **FSA – Finite State Automaton**) – это мощная абстракция, использование которой помогает управлять сложностью разрабатываемых систем. Основными типами конечных автоматов являются **автомат Мура** (**Moore machine**, рис. 9.1) и **автомат Мили** (**Mealy machine**, рис. 9.2).

[1] Использование виртуальной машины не рекомендуется: возможны проблемы с конфигурированием ПЛИС.

[2] https://store.digilentinc.com/pmod-als-ambient-light-sensor/.

[3] https://github.com/RomeoMe5/DDLM.

[4] Основной материал по конечным автоматам изложен в главе 8. Данный раздел нужен для понимания материала главы 9, если вы по каким-либо причинам ее пропустили.

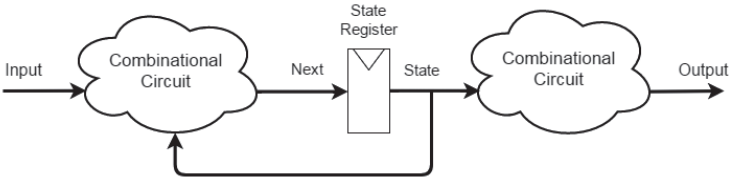


Рис. 9.1 Автомат Мура

В общем случае **автомат Мура** включает:

- входные (**Input**) и выходные (**Output**) сигналы;
- регистр или набор регистров для хранения текущего состояния автомата (**State**);
- комбинационную схему, которая формирует значение следующего состояния (**Next**) на основе входных сигналов (**Input**) и текущего состояния (**State**);
- комбинационную схему, которая формирует значения выходных сигналов (**Output**) на основе текущего состояния (**State**).

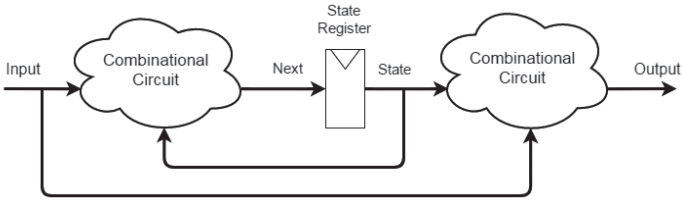


Рис. 9.2 Автомат Мили

Отметим, что в **автомате Мура** (рис. 9.1) значения выходных сигналов полностью определяются текущим состоянием автомата. **Автомат Мили** (рис. 9.2) в этом плане является более гибким: он состоит из тех же компонентов, что и автомат **Мура**, но значения выходных сигналов при этом также зависят еще и от сигналов на входе автомата. Таким образом, в автомате **Мили** выходные сигналы могут изменяться в зависимости от входных, без изменения состояния самого автомата.

9.1.2 Последовательный периферийный интерфейс SPI

Рассматриваемый в данной главе датчик освещенности взаимодействует с **ПЛИС** с помощью **последовательного периферийного интерфейса** (**SPI**, **Serial Peripheral Interface**). **SPI** – это простой синхронный последовательный протокол, используемый для обмена данными на небольших расстояниях, в основном во встраиваемых системах. Разработанный в начале 80-х, **SPI** широко используется и в настоящее время.

Пример подключения ведущего (**master**) устройства к ведомому (**slave**) с использованием **SPI** приведен на рис. 9.3. Каждое из устройств содержит сдвиговый регистр. Эти регистры при подключении устройств друг к другу формируют кольцевой буфер.

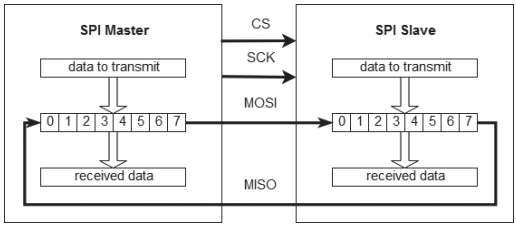


Рис. 9.3 Подключение по интерфейсу SPI

Для подключения устройств используются следующие сигналы:

CS – выбор устройства (**chip select**); чаще всего является инвертированным и устанавливается в низкий уровень во время передачи данных. Данный сигнал также может называться **SS** – **slave select**;

SCK – тактовый сигнал (**serial clock**); по фронту или спаду данного сигнала (в зависимости от настроек) выполняется операция сдвига в регистрах кольцевого буфера. Другой вариант названия – **SCLK**;

MOSI – текущий бит, передаваемый от ведущего устройства к ведомому (**master out slave in**). Данный сигнал также может называться **SDI** (**Slave Data In**) или **SI** (**Slave In**);

MISO – текущий бит, передаваемый от ведомого устройства к ведущему (**master in slave out**). Другие варианты названий – **SDO** (**Slave Data Out**) или **SO** (**Slave Out**).

На рис. 9.4 приведен один из возможных вариантов передачи 8 бит данных по **SPI**: биты **M7**–**M0** передаются от ведущего устройства к ведомому; биты **S7**–**S0** – от ведомого к ведущему. В приведенном примере передающая сторона устанавливает значение на своем выходе по спаду сигнала **SCK**, при этом принимающая считывает значение на своем входе по фронту этого же сигнала.

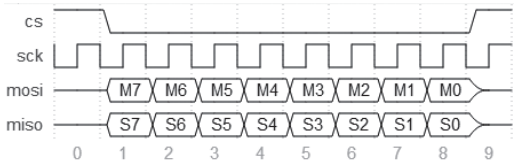


Рис. 9.4 Пример передачи данных по протоколу SPI

9.2 Оборудование и исходный код

9.2.1 Датчик освещенности

Используемый здесь датчик освещенности **Digilent PMOD Ambient Light Sensor**[1] (также встречаются названия **PMOD ALS module** или **PmodALS**) построен на базе

[1] При подготовке данной главы использована документация на датчик освещенности Digilent PMOD Sensor и примененную в нем микросхему АЦП ADC081S021 производства Texas Instruments. Перечисленные документы доступны в материалах данной практической работы в подкаталоге doc.

аналогового датчика, подключенного к 8-битному аналого-цифровому преобразователю (**АЦП**). Ниже приведены его изображение (рис. 9.5) и принципиальная схема (рис. 9.6).

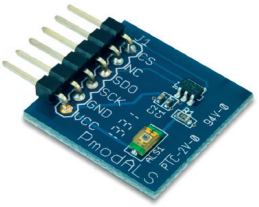


Рис. 9.5 Датчик освещенности Digilent PMOD Ambient Light Sensor

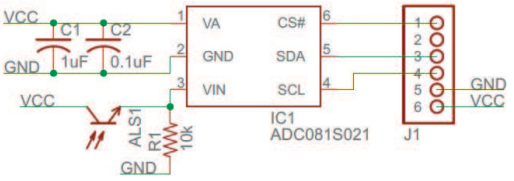


Рис. 9.6 Принципиальная схема модуля

С помощью датчика **ALS1**, в зависимости от уровня освещенности, изменяется напряжение на входе **VIN** микросхемы **АЦП** (**ADC081S021**). **АЦП** преобразует значение аналогового сигнала на своем входе в 8 бит данных. Так, **255** означает, что датчик ярко освещен, а **0** – что свет на него не попадает. Доступ к **АЦП** для чтения данных осуществляется по протоколу **SPI**. Временная диаграмма получения данных от **АЦП** приведена на рис. 9.7.

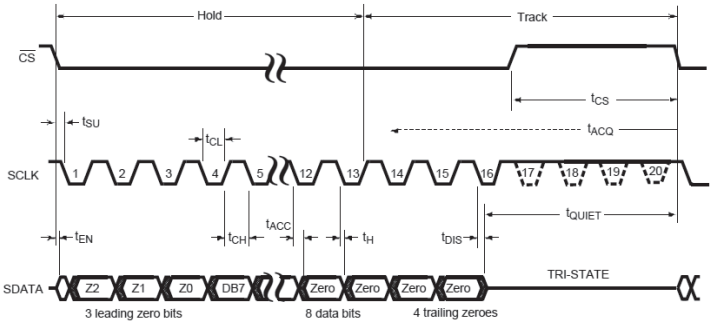


Рис. 9.7 ADC081S021 Временная диаграмма обмена данными[1]

АЦП ADC081S021 выступает в роли ведомого устройства и начинает передачу, после того как ведущее устройство устанавливает уровень сигнала **CS** в **0**. Передача данных выполняется за 16 импульсов тактового сигнала **SCLK**, частота которого должна быть от 1 до 4 МГц. **АЦП** устанавливает значение текущего бита на спаде тактового сигнала **SCLK** (соответственно, его чтение должно производиться по фронту **SCLK**). Первые три передаваемых бита равны **0**, после них следуют

[1] Сигналы **АЦП SCLK** и **SDATA** на плате модуля обозначены как **SCK** и **SDO** соответственно.

8 бит данных (результат **АЦП** преобразования), затем – **4** нуля. На 16-м такте выход **АЦП** устанавливается в **Z**-состояние (высокий импеданс).

9.2.2 Подключение периферийного устройства

Под «периферийным устройством» может пониматься как любой внешний блок, подключенный к компьютеру или системе на кристалле (в данном случае это датчик освещенности **PMOD Ambient Light Sensor**, рис. 9.8), так и любой блок, который подключен к системной шине. В «мире системного программирования» доступный **CPU** (центральный процессорный узел, центральное обрабатывающее устройство) набор устройств называется деревом устройств.

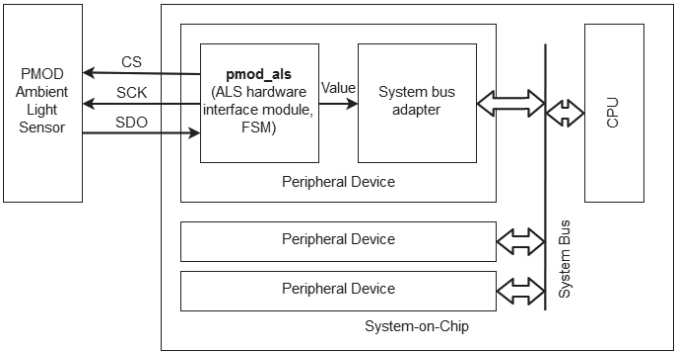


Рис. 9.8 Подключение периферийного устройства

Рассматриваемая система не содержит **CPU** или системной шины – аппаратная конфигурация в данном случае упрощена. Датчик освещенности подключен к отладочной плате. При этом находящаяся на плате **ПЛИС** прошивка сконфигурирована на периодический опрос датчика и вывод полученного от него значения на 7-сегментные индикаторы. Эта работа выполняется модулем **pmod_als**, который спроектирован как конечный автомат.

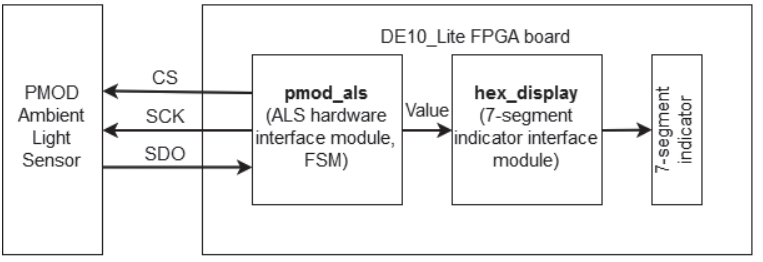


Рис. 9.9 Подключение датчика к отладочной плате

9.2.3 Подключение датчика

Так как датчик освещенности не принимает данные от ведущего устройства, для его подключения по **SPI** используются только три линии (не считая питания и заземления): **CS** (**Chip Select**), **SDA** (**Slave Data**) и **SCK** (**Serial Clock**). Расположение

этих линий на разъеме подключения модуля и его соединение с отладочной платой **DE10-Lite** приведены в табл. 9.1.

Таблица 9.1 Соответствие физических пинов ALS-модуля и платы DE10-Lite

PMOD ALS			DE10-Lite FPGA board				Примечание
PIN	Имя	Описание	PIN	GPIO	Направление	Сигнал	
1	CS	выбор ведомого	39	GPIO[34]	output	cs	отметка на кабеле
2	NC	не подключен	37	GPIO[32]	output	1'bz	
3	SDO	данные	35	GPIO[30]	input	sdo	
4	SCK	тактовый сигнал	33	GPIO[28]	output	sck	
5	GND	заземление	31	GPIO[26]	output	1'b0	
6	VCC	+3.3V	29	VCC			

Пример подключения модуля к отладочной плате показан на рис. 9.10. Для соединения можно использовать как отдельные провода, так и специальный кабель. При использовании последнего следует обратить внимание на размещение белой отметки.

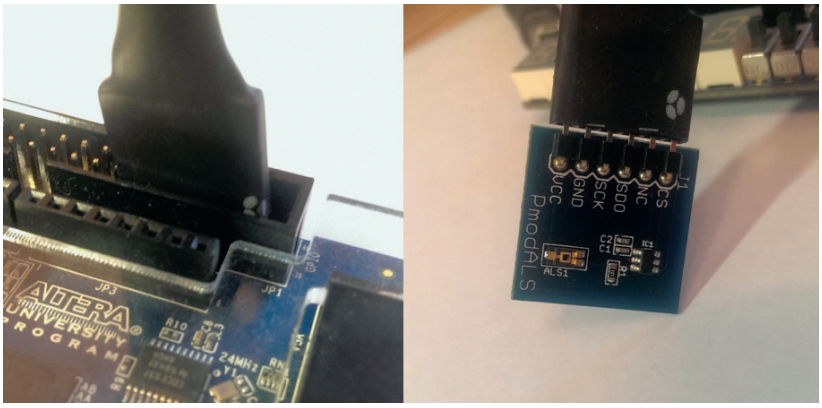


Рис. 9.10 Кабель подключения датчика к отладочной плате

9.2.4 Симуляционная модель периферийного устройства

Перед тем как выполнять синтез и конфигурировать **ПЛИС**, производится тестирование разработанного **Verilog** кода с использованием симулятора. В случае если работа выполняется с реальным устройством, можно подключить датчик освещенности непосредственно к портам **ПЛИС** и проверить корректность функционирования созданного модуля. Но это невозможно сделать в режиме симуляции, так как среда тестирования (**testbench**) – это всего лишь программа, которая выполняется симулятором в памяти компьютера. Решением данной проблемы является **симуляционная модель устройства** – модуль на языке **Verilog** или **SystemVerilog**, имитирующий работу реального устройства.

Для сложных устройств, таких как модули памяти **SDRAM** или **DDR**, **симуляционные модели** часто публикуются изготовителем микросхем (например, хорошие модели можно скачать с сайта компании **Micron**[1]). Для таких простых устройств, как используемый в данном примере датчик освещенности, **симуляционная модель** может быть разработана самостоятельно. Пример подобной модели приведен ниже[2] (листинг 9.1); он основан на документации для микросхемы **ADC081S021**.

Задание: сравните временную диаграмму симуляционной модели (рис. 9.11) с диаграммой сигналов из документации на микросхему (рис. 9.7).

```
module pmod_als_stub
#(
   parameter value = 8'hAB
)(
   input cs,
   input sck,
   output sdo
);
   wire [7:0] tvalue = value;
   wire [15:0] tpacket = { 3'b0, tvalue, 4'b0, 1'bz };

   reg [15:0] buffer;
   reg sdata;

   assign sdo = cs ? 1'bz : sdata;

   always @(negedge sck or posedge cs) begin
      if(!cs)
         { sdata, buffer } <= { buffer, 1'b0 };
      else
         { sdata, buffer } <= { 1'b0, tpacket };
   end
endmodule
```

Листинг 9.1 Симуляционная модель датчика освещенности

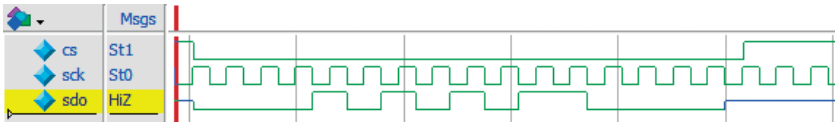


Рис. 9.11 Временная диаграмма симуляционной модели микросхемы ADC081S021

[1] https://www.micron.com.

[2] Данный пример является простейшим, так как представленная модель не содержит эмуляции задержек сигналов на выходах модуля, а также верификации того, что сигналы, поступающие на входы модуля, по своим временным характеристикам (время установки и удержания) соответствуют требованиям документации на микросхему.

9.2.5 Порядок проведения симуляции, синтеза и запуска проекта

Исходный код, необходимый при выполнении данной работы, размещен в каталоге **lab_09**.

Для того чтобы запустить пример в режиме симуляции, необходимо:

- перейти в каталог **lab_09/src/<имя_примера>/simulation**;
- для симуляции с использованием **ModelSim** выполнить скрипт **01_simulate_with_modelsim.bat**;
- для симуляции с использованием **Icarus Verilog** выполнить скрипт **02_simulate_with_icarus.bat**.

Для запуска на отладочной плате необходимо:

- подключить датчик к отладочной плате;
- подключить отладочную плату к рабочей станции;
- перейти в каталог **lab_09/src/<имя_примера>/synthesis**;
- выполнить скрипт **make_project.bat**;
- запустить среду **Quartus Prime**;
- в **Quartus Prime** открыть проект **lab_09/src/<имя_примера>/synthesis/project/de10_lite.qpf**;
- выполнить сборку проекта;
- выполнить программирование отладочной платы;
- выполнить проверку работоспособности устройства: при изменении количества света, падающего на датчик, должны изменяться показания на 7-сегментных индикаторах.

9.3 Проектирование конечного автомата

В данной главе рассматриваются два подхода к проектированию конечных автоматов. При использовании традиционного «академического» подхода разработчик анализирует временные диаграммы сигналов, выделяет на них временные промежутки (могут быть классифицированы как относящиеся к различным состояниям конечного автомата), правила перехода между этими состояниями, а затем кодирует автомат с использованием языка описания схем. Второй подход к проектированию предполагает использование (вместо явно выделенных состояний автомата) значений, получаемых на выходе счетчика (регистр, значение которого на каждом такте увеличивается на единицу).

9.3.1 Академический подход

Данный подход (при его корректном использовании) гарантированно приводит к получению работающей конфигурации автомата. Он включает следующие шаги:

1. Анализ документации и декомпозицию решения. Анализ выполняется на основе документации на периферийное устройство, спецификации используемых протоколов и, в особенности, на приведенных в этих документах временных диаграммах. По результатам анализа инженер принимает решение о необходимости декомпозиции разрабатываемого модуля на подмодули.
2. Определение состояний разрабатываемого автомата, условий перехода из состояния в состояние, количества и типа регистров состояния.
3. Реализация автомата. Тестирование автомата с использованием симулятора и на отладочной плате (или в составе разрабатываемого устройства).

В данном разделе будут последовательно пройдены все описанные выше шаги.

Декомпозиция конечного автомата

Проанализируем временную диаграмму (рис. 9.12) с использованием отдельных фактов, известных о датчике освещенности и об отладочной плате.

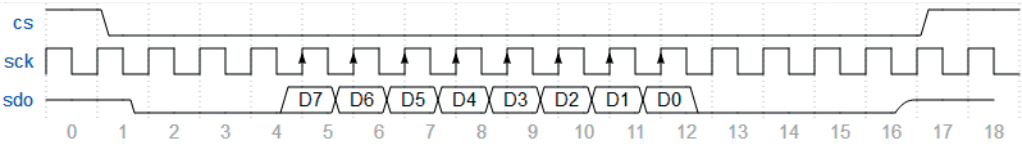


Рис. 9.12 Временная диаграмма обмена данными с датчиком освещенности

Согласно документации, микросхема **АЦП** работает на частоте от 1 до 4 МГц. Частота тактового сигнала на отладочной плате **DE10-Lite** – 50 МГц. Таким образом, периферийное устройство медленнее, чем разрабатываемая система, а значит, в проекте необходимо использовать делитель частоты.

Есть два варианта решения данной задачи:

1. **Использовать несколько тактовых сигналов**. Каждый из тактовых сигналов образует свой временной домен (**clock domain**) – участок цифровой схемы, синхронизированный данным тактовым сигналом. Обмен информацией между различными доменами тактового сигнала может привести к возникновению **метастабильности**. Для того чтобы избежать этих проблем, инженер должен использовать набор приемов, известный как методики **Clock Domain Crossing** (**CDC**)[1].

   Поскольку при этом разрабатываемая схема значительно усложняется, для этого нужны веские основания. Например, необходимость достижения максимальной производительности как вычислительного ядра, так и подсистемы памяти.

[1] Методики **Clock Domain Crossing** (**CDC**) являются популярным вопросом на собеседованиях. Данная тема выходит за рамки текущей лабораторной работы. Очень хорошая презентация на эту тему была подготовлена **Ryan Donohue**; ссылка: https://web.stanford.edu/class/ee183/handouts_spr2003/synchronization_pres.pdf.
Данный документ также доступен в каталоге с дополнительными материалами.

2. Использовать один тактовый сигнал. Данный подход проще, чем описанный выше. В этом случае в проекте необходимо предусмотреть сигнал (единичный импульс, строб), являющийся синхронным основной частоте и указывающий на необходимость выполнить чтение/запись/смену состояния автомата или иное действие. Именно этот подход и будет использован в данной работе.

Из документации на датчик известно, что данные должны быть считаны по фронту тактового сигнала 1–4 МГц. Таким образом, необходимо формировать периодический единичный импульс, частота возникновения которого находится в указанных пределах. Для выполнения этой задачи выделим отдельный модуль **делителя частоты** (**clock divider**), который удовлетворяет следующим требованиям:

- работает на частоте отладочной платы (50 МГц);
- построен на основе счетчика;
- формирует тактовую частоту, необходимую для работы датчика освещенности (1–4 МГц);
- формирует строб, используемый остальной схемой для чтения данных с датчика.

Получение данных с датчика должно выполняться периодически, при этом сессия обмена данными, как правило, происходит редко (в масштабах системы, работающей на частоте 50 МГц). Между этими сессиями необходимо предусмотреть задержку. Также известно (рис. 9.7), что не все биты, получаемые от **АЦП**, содержат полезную информацию. В системе необходим модуль, который будет управлять периодичностью запроса данных с датчика, а также игнорировать приходящие от него нулевые биты. Предварительно назовем данный модуль **менеджером сессий** (**session manager**). И поскольку он управляет задержками, внутри его также предполагается наличие счетчика.

Помимо делителя частоты и менеджера сессий, необходимо предусмотреть **буфер для хранения принимаемых данных**, а также регистр, в котором будет храниться последнее принятое значение.

Проектирование модуля управления 7-сегментными индикаторами в данной главе не раскрывается, так как эта задача решалась в одной из предыдущих практических работ.

Результат декомпозиции задачи приведен на рис. 9.13.

Рис. 9.13 Результат декомпозиции задачи

Следует отметить, что данный результат получен в ходе первичного анализа решаемой задачи. Поэтому архитектура решения, наименования подмодулей, сигналов, их количество и состав в дальнейшем могут быть уточнены или кардинально изменены.

Определение состояний и правил перехода

После того как выполнена декомпозиция задачи и проектируемое устройство поделено на подмодули, перейдем к детальной проработке каждого из этих модулей.

Делитель частоты. Так как выходные сигналы автомата зависят от его текущего состояния, будет логичным разбить временную диаграмму на однородные области (когда состояния выходных сигналов не меняются) и для каждой из этих областей назначить свое имя состояния (рис. 9.14). В рассматриваемом случае можно выделить три состояния автомата:

DOWN — выходной тактовый сигнал находится на низком уровне (**sck = 0**, **sck_edge = 0**);

EDGE — выходной тактовый сигнал и строб – на верхнем уровне (**sck = 1**, **sck_edge = 1**);

UP — выходной тактовый сигнал – на верхнем уровне (**sck = 1**, **sck_edge = 0**).

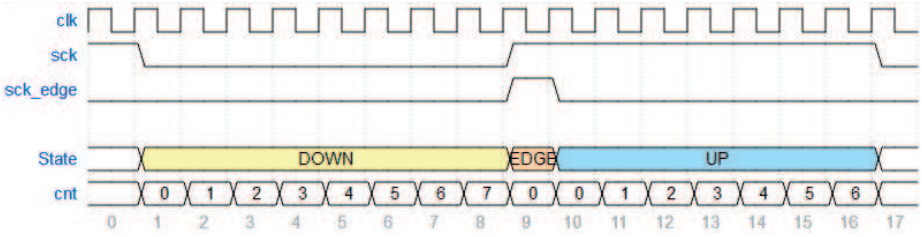


Рис. 9.14 Определение состояний автомата делителя частоты

Так как автомат находится в состояниях **DOWN** и **UP** более одного такта, в проект модуля следует добавить счетчик (**cnt**), значение которого будет использоваться для определения необходимости перехода в следующее состояние. Таким образом, текущее состояние проектируемого автомата определяется двумя регистрами (**state** и **cnt**).

На основе анализа, выполненного по временной диаграмме (рис. 9.14), построим **диаграмму состояний** (граф переходов) конечного автомата (рис. 9.15). После **сброса** (**RESET**) автомат переходит в состояние **DOWN**. При этом значение счетчика **cnt** начинает инкрементироваться (увеличиваться на единицу) с каждым тактом сигнала **clk**. Как только значение счетчика достигнет величины **DOWN_SIZE**, автомат переходит в состояние **EDGE** (для рассматриваемого на рис. 9.14 случая значение параметра **DOWN_SIZE = 7**). Из состояния **EDGE** на следующем такте автомат переходит в состояние **UP**. В нем он находится до тех пор, пока счетчик не станет равен **UP_SIZE** (в рассматриваемом случае **UP_SIZE = 6**).

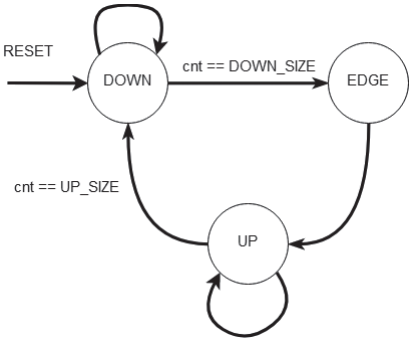


Рис. 9.15 Диаграмма состояний автомата делителя частоты

По аналогии с делителем частоты спроектируем конечный автомат для модуля менеджера сессий (рис. 9.16). Руководствуясь тем же самым подходом, выделим четыре состояния: **IDLE**, **PREFIX**, **DATA** и **POSTFIX**. Сигнал **CS** находится нв высоком уровне только в состоянии **IDLE**. Чтение текущего бита с входа **sdo** выполняется в состоянии **DATA**.

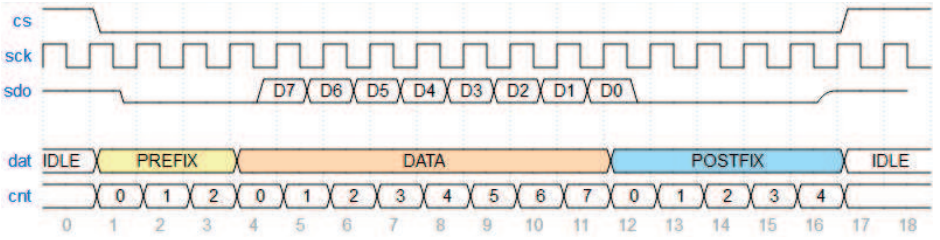


Рис. 9.16 Определение состояний автомата менеджера сессий

Счетчик (**cnt**) изменяет свое значение (инкрементируется) только при получении стробирующего импульса (**sck_edge**) от модуля делителя частоты. Диаграмма состояний менеджера сессий приведена на рис. 9.17.

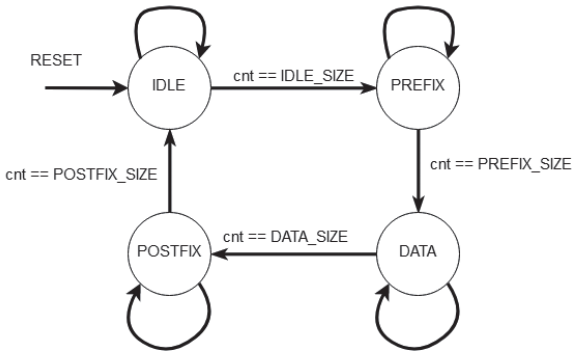


Рис. 9.17 Диаграмма состояний менеджера сессий

9.3.2 Реализация конечного автомата

В данном разделе представлены различные подходы к реализации (стили кодирования) конечного автомата, который был разработан ранее. В первую очередь рассмотрим самый неудачный из способов и разберем различные шаги и подходы, позволяющие улучшить полученный результат.

Неудачный стиль кодирования

Задание:

- используя шаги, описанные в разделе 9.2.5, запустите симуляцию примера **01_badstyle**. Не закрывайте окно симулятора;
- откройте документацию на **АЦП ADC081S021** и сравните временную диаграмму в окне симулятора с временной диаграммой из документации;
- подключите датчик освещенности к отладочной плате так, как это показано в разделе 9.2.3;
- в соответствии с описанием из раздела 9.2.5 выполните синтез проекта, конфигурирование отладочной платы и проверку работоспособности проекта;
- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → RTL Viewer**;
- найдите экземпляр модуля делителя частоты и сравните его с представленным на рис. 9.18;
- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → State Machine Viewer**. Этот же результат можно получить двойным щелчком мыши на желтом прямоугольнике конечного автомата внутри окна **RTL Viewer**;
- сравните диаграмму конечного автомата с вариантом, разработанным на этапе проектирования (рис. 9.15, 9.17);
- откройте файл **01_badstyle/pmod_als.v** в текстовом редакторе;
- сравните этот код с блок-схемами конечных автоматов (рис. 9.1, 9.2).

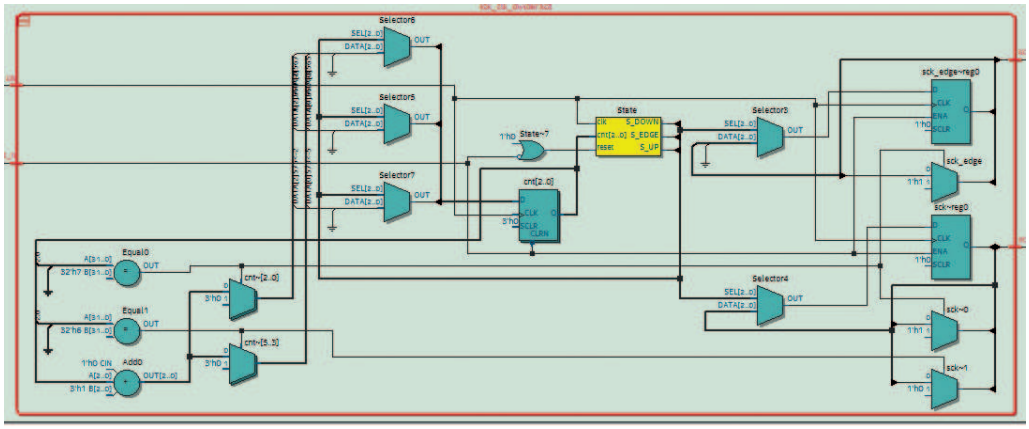


Рис. 9.18 Делитель частоты в представлении RTL View

Выполните анализ исходного кода **01_badstyle/pmod_als.v** (листинг 9.2):

```
// Next state determining
always @ (posedge clk or negedge rst_n)
   if(~rst_n) begin
      State <= S_DOWN;
      cnt <= 3'b0;
      end
   else begin
      case(State)
         S_DOWN : begin
            if(cnt == DOWN_SIZE) begin
               State <= S_EDGE;
               sck_edge <= 1'b1;
               sck <= 1'b1;
            end
            cnt <= (cnt == DOWN_SIZE) ? 0 : cnt + 1;
         end
         S_EDGE : begin
            State <= S_UP;
            cnt <= 0;
            sck_edge <= 1'b0;
         end
         S_UP : begin
            if(cnt == UP_SIZE) begin
               State <= S_DOWN;
               sck <= 1'b0;
            end
            cnt <= (cnt == UP_SIZE) ? 0 : cnt + 1;
         end
```

```
        endcase
end
```

Листинг 9.2 01_badstyle/pmod_als.v (фрагмент)

Результаты анализа:

- этот код работает корректно, что является его несомненным положительным качеством;
- программное обеспечение **Quartus Prime** распознало описание конечного автомата в этом исходном коде – это означает, что при синтезе оно может выполнять необходимые оптимизации, при этом в интерфейсе **State Machine Viewer** доступно представление автомата в виде графа переходов (рис. 9.19);

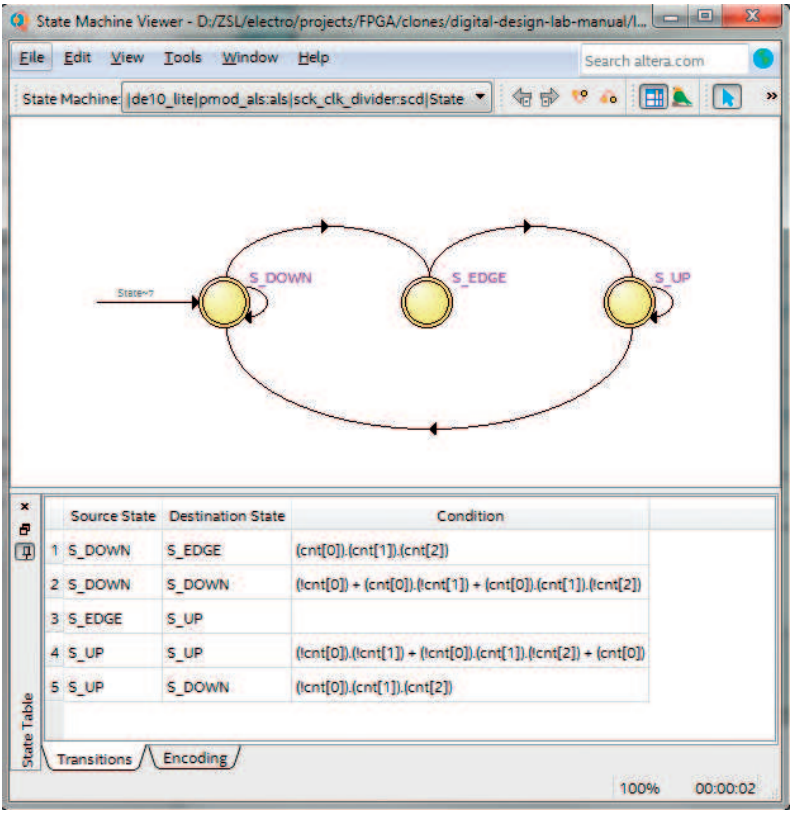


Рис. 9.19 Делитель частоты в представлении State Machine View

- данный стиль описания автомата ближе к разработке программного обеспечения, чем к цифровой схемотехнике: программист описывает конечный автомат, предполагая его работу в одном потоке (выполнение потока программы на одном процессорном ядре); вся логика описывается в одном блоке. Если проанализировать исходный код открытых проектов на **GitHub**, то в них можно встретить много кода, подобного этому;

- в рассматриваемом коде используется только последовательностная логика (листинг 9.3): отсутствуют отдельные блоки комбинационной логики для определения следующего состояния автомата или значений сигналов на его выходах:

```
// Next state determining
always @ (posedge clk or negedge rst_n)
//...
```

Листинг 9.3 Блок последовательностной логики (фрагмент 1)

  Описание только последовательностной логики допустимо, но может привести к синтезу регистров там, где в них нет необходимости;

- значения выходных сигналов вычисляются до смены состояния (на предыдущем такте):

```
S_DOWN : begin
   if(cnt == DOWN_SIZE) begin
      State <= S_EDGE;
      sck_edge <= 1'b1;
      sck <= 1'b1;
   end
...
end
```

Листинг 9.4 Блок последовательностной логики (фрагмент 2)

  В ряде случаев это может быть полезным (рассмотрено в разделе 9.3.3), но может быть и избыточным;

- основной недостаток данного стиля кодирования: **такой код очень тяжело поддерживать** для конечных автоматов, когда количество состояний превышает **10**. В следующем разделе демонстрируется важная особенность языка **Verilog** (если сравнивать его с классическими языками программирования): все его блоки синтезируются в схемотехнические конструкции, работающие параллельно. Поэтому можно разделить единый блок с описанием всего автомата на три отдельных, что снизит сложность поддержки этого кода без потери его производительности.

Компромиссный стиль кодирования

Исходный код из предыдущего раздела работает корректно, однако тот стиль, в котором он описан, не рекомендован для описания сложных конечных автоматов. В данном разделе предлагается ряд улучшений, которые сделают исходный код более читаемым и легче поддерживаемым.

Задание:

- используя шаги, описанные в разделе 9.2.5, запустите симуляцию примера **02_compromise**. Не закрывайте окно симулятора;

- в соответствии с описанием из раздела 9.2.5 выполните синтез проекта, конфигурирование отладочной платы и проверку работоспособности проекта;

- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → RTL Viewer**;
- сравните отображаемую схему со схемой из **RTL View** для примера **01_badstyle**;
- в программе **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → State Machine Viewer**;
- сравните диаграмму конечного автомата с вариантом, разработанным на этапе проектирования (рис. 9.15, 9.17);
- откройте файл **02_compromise/pmod_als.v** в текстовом редакторе;
- сравните этот код с блок-схемами конечных автоматов (рис. 9.1, 9.2).

Результаты анализа исходного кода **02_compromise/pmod_als.v**:

- этот код также распознается средой **Quartus Prime** как конечный автомат, и его диаграмма состояний доступна для просмотра в представлении **State Machine Viewer**;
- вся последовательностная логика выделена в отдельный блок (листинг 9.5):

```
// Next value to state on every clock
always @ (posedge clk or negedge rst_n)
   if(~rst_n) begin
      State <= S_IDLE;
      cnt <= 24'b0;
      buffer <= 8'b0;
      value <= 8'b0;
      end
   else begin
      State <= Next;
      cnt <= cntNext;
      buffer <= bufferNext;
      value <= valueNext;
   end
```

Листинг 9.5 02_compromise/pmod_als.v (фрагмент 1)

- добавлен отдельный блок комбинационной логики для вычисления следующего состояния автомата (листинг 9.6):

```
// Next state determining
always @(*) begin
   Next = State;
   if(sck_edge)
      case(State)
         S_IDLE : if(cnt == IDLE_SIZE) Next = S_PREFIX;
         S_PREFIX : if(cnt == PREFIX_SIZE) Next = S_DATA;
         S_DATA : if(cnt == DATA_SIZE) Next = S_POSTFIX;
         S_POSTFIX : if(cnt == POSTFIX_SIZE) Next = S_IDLE;
```

```
      endcase
end
```

Листинг 9.6 02_compromise/pmod_als.v (фрагмент 2)

- добавлен отдельный блок комбинационной логики для вычисления состояния выходов конечного автомата (листинг 9.7):

```
// output
assign cs = (State == S_IDLE);
```

Листинг 9.7 02_compromise/pmod_als.v (фрагмент 3)

- этот код существенно легче читать и поддерживать;
- по этому коду можно без особых затруднений восстановить диаграмму состояний автомата.

Основанный на примитивах стиль кодирования

Задание:

- используя шаги, описание которых дано в разделе 9.2.5, запустите симуляцию примера **03_industrial**. Не закрывайте окно симулятора;
- выполните синтез проекта, конфигурирование отладочной платы и проверку работоспособности проекта;
- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → RTL Viewer**;
- сравните отображаемую схему со схемой из **RTL View** для примера **02_compromise**;
- в программе **Quartus Prime** запустите **Tools → Netlist Viewers → State Machine Viewer**;
- откройте в текстовом редакторе файлы **03_industrial/register.v**, **03_industrial/register_we.v** и **03_industrial/pmod_als.v**.

Стиль кодирования конечного автомата, который приведен в данном разделе, можно описать как построенный на использовании примитивов (подобный стиль кодирования, в частности, используется внутри софт-процессора **MIPSfpga**, доступ к необфусцированным (открытым) исходным кодам которого можно получить в рамках **MIPS Academic Community**[1]. **MIPSfpga** является вариантом процессорного ядра **MIPS microAptiv UP**, используемого, в частности, в микро-

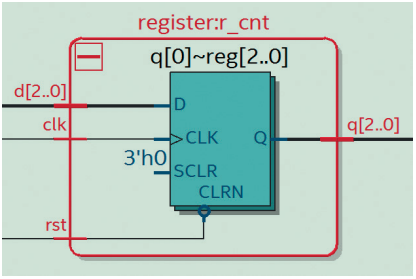
[1] Исходный код процессорного ядра доступен как часть пакета **MIPSfpga** 2.0, который можно скачать на ресурсе https://www.mips.com/mac/. Подробное описание **MIPSfpga** 2.0 приведено в: *Chaver D., Panchul Y., Sedano E., Harris D. M., Owen R., Kakakhel Z. L., Harris S. L.* Practical experiences based on MIPSfpga // Proceedings of the 19th Workshop on Computer Architecture Education. 2017, June. P. 1–9. ACM (перевод: https://habr.com/post/335848/). Кроме того, существует открытый проект MIPSfpga-plus – основанная на данном ядре система на кристалле, не содержащая исходных кодов самого процессора (их предлагается взять из пакета **MIPSfpga**): https://github.com/MIPSfpga/mipsfpga-plus.

контроллерах **Microchip PIC32MZ**). При этом подходе вся последовательностная логика изолируется внутри.

К особенностям такого подхода можно отнести следующее:

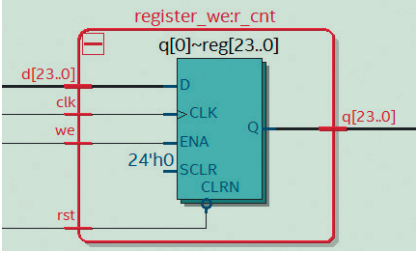
- модуль больше не распознается средствами **Quartus Prime** как конечный автомат и не отображается в **State Machine Viewer**. Работоспособность кода при этом не нарушается;
- все остальные модули, за исключением модулей-примитивов, описываются с использованием комбинационной логики и экземпляров модулей-примитивов[1].

```
module register #(
   parameter SIZE = 32
)(
   input clk,
   input rst,
   input [ SIZE - 1 : 0 ] d,
   output reg [ SIZE - 1 : 0 ] q
);
always @ (posedge clk or negedge rst)
   if(~rst)
      q <= { SIZE { 1'b0}};
   else
      q <= d;
endmodule
```



Листинг 9.8 Модуль register (03_industrial/register.v) и результаты его синтеза

```
module register_we #(
   parameter SIZE = 32
)(
   input clk,
   input rst,
   input we,
   input [ SIZE - 1 : 0 ] d,
   output reg [ SIZE - 1 : 0 ] q
);
always @ (posedge clk or negedge rst)
   if(~rst)
      q <= { SIZE { 1'b0}};
```



[1] При использовании этого подхода и имея дело в основном только с комбинационной логикой, очень просто сделать ошибку и увлечься созданием кода минимального размера. Может наступить момент, когда результат будет казаться вам элегантным и эффективным, но для остальных разработчиков он будет напоминать **netlist**, который формируется средствами синтеза. Если провести аналогию с программированием на языках высокого уровня, то это можно сравнить с использованием ассемблерных вставок без особой необходимости. Лучше не усердствовать с преждевременной оптимизацией комбинационных схем – зачастую средства синтеза делают это эффективнее людей. Задача инженера состоит не только в том, чтобы описать эффективно работающую схему, но и в том, чтобы сделать ее код максимально очевидным и простым в поддержке другими людьми.

```
   else
      if(we) q <= d;
endmodule
```

Листинг 9.9 Модуль register_we (03_industrial/register_we.v) и результаты его синтеза

Пример использования модуля-примитива приведен ниже (листинг 9.10).

```
// State hold registers
wire [1:0] State;
reg [1:0] Next;
register_we #(.SIZE(2)) r_state(clk, rst_n, sck_edge, Next, State);
...
// Next state determining
always @(*) begin
   Next = State;
   case(State)
      S_IDLE : if(cnt == IDLE_SIZE) Next = S_PREFIX;
      S_PREFIX : if(cnt == PREFIX_SIZE) Next = S_DATA;
      S_DATA : if(cnt == DATA_SIZE) Next = S_POSTFIX;
      S_POSTFIX : if(cnt == POSTFIX_SIZE) Next = S_IDLE;
   endcase
end
```

Листинг 9.10 Пример использования модуля-примитива

9.3.3 Задержка выходных сигналов автомата

Задание:

- используя шаги, описание которых дано в разделе 9.2.5, запустите симуляцию примера **04_outdelay**. Не закрывайте окно симулятора;
- выполните синтез проекта, конфигурирование отладочной платы и проверку работоспособности проекта;
- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → RTL Viewer**;
- сравните отображаемую схему со схемой из **RTL View** для примера **03_industrial**;
- откройте в текстовом редакторе файл **04_outdelay/pmod_als.v**.

В классическом описании **автоматов Мура** (рис. 9.1) и **Мили** (рис. 9.2) значения выходных сигналов задаются комбинационной схемой, входом которой являются регистры состояния и (для автомата **Мили**) входные сигналы. В случае если выход с конечного автомата подключен напрямую к выводам **ПЛИС**, эта комбинационная схема вносит дополнительную задержку в распространение сигналов. Это может привести к снижению частоты, на которой осуществляется взаимодействие между устройствами, или к неработоспособности подключения, если будут нарушены временны́е требования (время установки и время удержания)

одного из устройств[1]. Для решения данной проблемы на выход конечного автомата следует добавить дополнительный регистр, выход которого, в свою очередь, будет подключен к контактам чипа (такие автоматы называют **Out-registered FSM**, рис. 9.20).

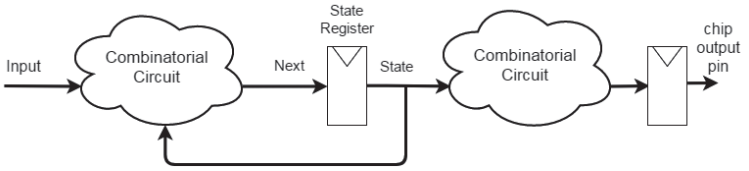


Рис. 9.20 Использование буферного регистра на выходе автомата (вариант 1)

Недостатком данного решения является увеличение на один такт задержки между входными и выходными сигналами конечного автомата. В ряде случаев такая дополнительная задержка может быть неприемлема. В этом случае ее можно «переместить» внутрь логики самого автомата (рис. 9.21), как это уже делалось в главе 8. Это может уберечь от необходимости перепроектирования модуля «с нуля» или его конвейеризации. Ценой такого решения является увеличение критического пути внутри автомата.

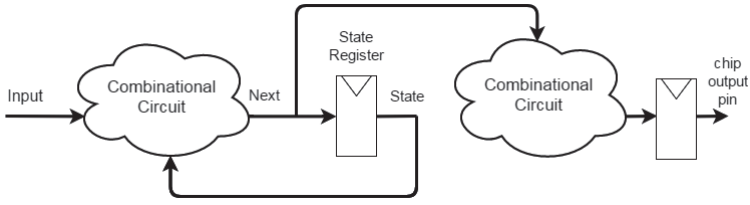


Рис. 9.21 Использование буферного регистра на выходе автомата (вариант 2)

Ниже показан пример такого решения. Все, что необходимо, – это заменить комбинационную схему

```
// output
assign cs = (State == S_IDLE);
```

следующим кодом:

```
// buffered output
wire csNext = (Next == S_IDLE);
register_we #(.SIZE(1)) r_cs(clk, rst_n, sck_edge, csNext, cs);
```

В этом случае вычисление значения выходного сигнала производится на предыдущем такте (до переключения состояния автомата) – полностью аналогично тому, как это было реализовано в примере **01_badstyle**, – что позволяет устранить дополнительную задержку на выводе чипа.

[1] Описание временных ограничений (Timing Constraints) является обширной темой, которая не вошла в данную практическую работу. Более подробно с ней можно познакомиться, например, в **Intel FPGA Wiki**: https://fpgawiki.intel.com/wiki/Timing_Constraints.

9.3.4 Автомат на основе счетчика

Академический подход к проектированию конечного автомата является надежным методом, однако в ряде случаев он может привести к получению излишне усложненной схемы. Альтернативой является построение автомата на основе счетчика (также известного как **Counter-based FSM** или **Pointer-Based FSM**[1], рис. 9.22). В ряде случаев этот подход позволяет получить более компактное и эффективное решение.

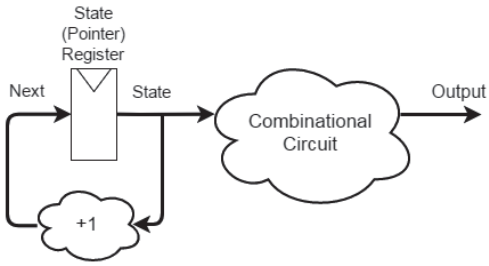


Рис. 9.22 Автомат на основе счетчика

Основой такого автомата является счетчик (регистр, значение которого на каждом такте увеличивается на единицу). К выходу счетчика подключена комбинационная схема, формирующая необходимые выходные сигналы в зависимости от его текущего значения. Вместо комбинационной схемы также может использоваться память, которая служит в качестве таблицы поиска (**Lookup Table**, **LUT**); значение счетчика в этом случае выступает в роли указателя на текущую ячейку этой памяти.

Счетчик как источник периодических сигналов

Задание:

- используя шаги, описанные в разделе 9.2.5, запустите симуляцию примера **05_optimized**. Не закрывайте окно симулятора;
- выполните синтез проекта, конфигурирование отладочной платы и проверку работоспособности проекта;
- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → RTL Viewer**;
- сравните отображаемую схему со схемой из **RTL View** для примера **03_industrial**, отдельно сравните модули делителей частоты;
- откройте в текстовом редакторе файл **05_optimized/pmod_als.v**.

Четырехразрядный двоичный счетчик за время работы непрерывно проходит все множество своих возможных значений от **4'b0000** до **4'b1111**. При этом отдельные биты счетчика могут рассматриваться как источники периодического сиг-

[1] *Pedroni V. A.* Finite state machines in hardware: theory and design (with VHDL and SystemVerilog). MIT press, 2013.

нала, частота которого кратна частоте, на которой работает счетчик (рис. 9.23). В случае если необходимый нам сигнал можно получить делением тактовой частоты (на **2**, **4**, **8** и т. д.), то нет необходимости проектировать отдельный конечный автомат – достаточно напрямую подключить вывод конечного автомата к соответствующему биту регистра-счетчика.

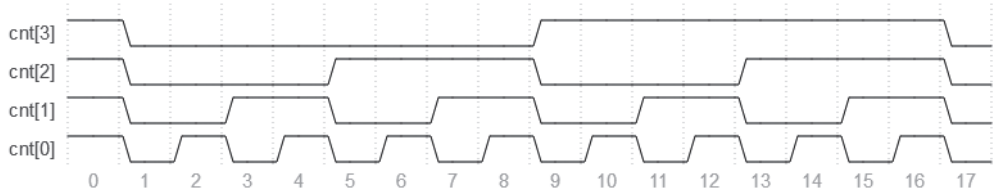


Рис. 9.23 4-битный счетчик и значения его битов

Оптимизация автомата с использованием счетчика

Для того чтобы перепроектировать блок делителя частоты с использованием предлагаемого подхода, необходимо сравнить желаемую временную диаграмму с временной диаграммой двоичного счетчика (рис. 9.24).

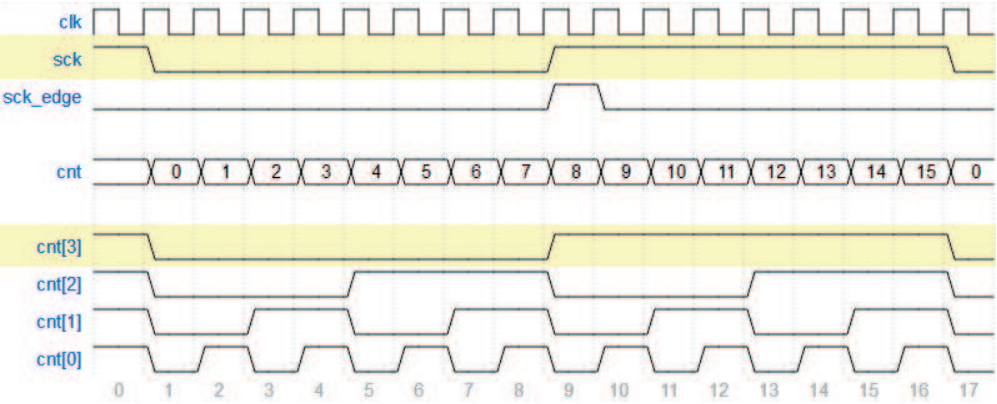


Рис. 9.24 Формирование сигнала sck (вверху) с помощью 4-битного двоичного счетчика (внизу)

На диаграмме (рис. 9.24) видно, что сигнал **cnt[3]** полностью совпадает с тем сигналом **sck**, который должен быть получен на выходе блока делителя частоты. Что касается единичного импульса **sck_edge**, который также должен формироваться модулем, то ему соответствует значение счетчика **4'b1000** (рис. 9.25).

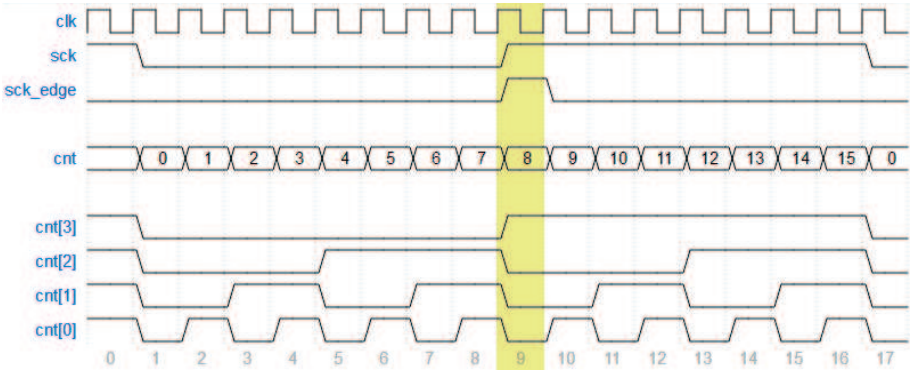


Рис. 9.25 Формирование сигнала sck_edge (вверху) с помощью 4-битного двоичного счетчика (внизу)

Исходный код итогового делителя тактового сигнала представлен в листинге 9.11. В результате его синтеза получается схема (рис. 9.26), которая компактнее, чем результаты, полученные при академическом подходе к проектированию. Единственное ограничение данного подхода: частота выходных сигналов должна быть кратна частоте тактового сигнала.

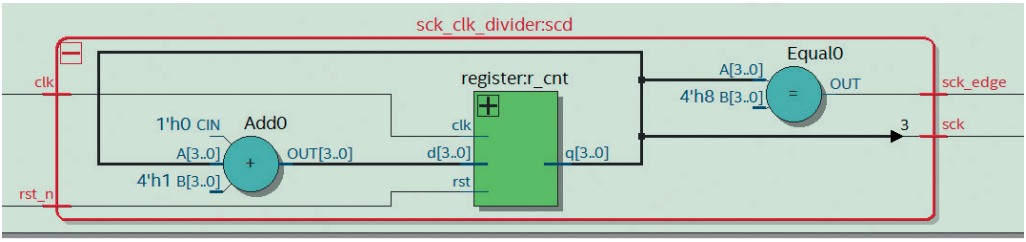


Рис. 9.26 Результат синтеза делителя тактового сигнала на основе счетчика

```
module sck_clk_divider
(
   input clk,
   input rst_n,
   output sck,
   output sck_edge
);
   wire [3:0] cnt;
   wire [3:0] cntNext = cnt + 1;
   register #(.SIZE(4)) r_cnt(clk, rst_n, cntNext, cnt);

   assign sck = cnt[3];
   assign sck_edge = (cnt == 4'b1000);
endmodule
```

Листинг 9.11 Делитель тактового сигнала на основе счетчика

Задание: сравните результаты синтеза делителей частоты при различных подходах к проектированию (рис. 9.18, 9.26).

Данная схема является точно таким же конечным автоматом, как и рассмотренные ранее. Его диаграмма состояний приведена на рис. 9.27.

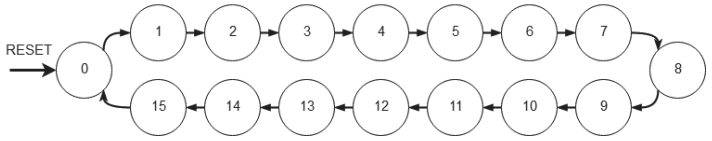


Рис. 9.27 Диаграмма состояний автомата, построенного на базе счетчика

Перепроектирование конечного автомата с использованием счетчика

В текущем разделе подход к проектированию конечного автомата на основе счетчика применяется ко всему модулю взаимодействия с датчиком освещенности.

Задание:

- используя шаги, описание которых приведено в разделе 9.2.5, запустите симуляцию примера **06_counter**. Не закрывайте окно симулятора;
- выполните синтез проекта, конфигурирование отладочной платы и проверку работоспособности проекта;
- в среде **Quartus Prime** откройте пункт меню **Tools → Netlist Viewers → RTL Viewer**;
- сравните отображаемую схему со схемой из **RTL View** для примера **03_industrial**;
- откройте в текстовом редакторе файл **06_counter/pmod_als.v**.

После перепроектирования весь код модуля **pmod_als** умещается на половине страницы (листинг 9.12). Ниже приведена временная диаграмма его работы (рис. 9.28) и схема – результат синтеза (рис. 9.29).

Сложно не согласиться с тем, что это самое эффективное решение задачи за всю практическую работу.

Задание: проанализируйте листинг 9.12, отследите сигналы **sck**, **cs**, **sampleBit** и **valueDone** в окне симулятора и на временной диаграмме (рис. 9.28). Особое внимание уделите тому, как они формируются. Обоснуйте, как оптимизация автомата сказалась на затратах ресурсов чипа, и проиллюстрируйте это отчетами компилятора после сборки проекта.

```
module pmod_als
#(
   parameter CNTSIZE = 11
)
(
   input clk,
```

```
   input rst_n,
   output cs,
   output sck,
   input sdo,
   output [7:0] value
);
   wire [ CNTSIZE - 1:0] cnt;
   wire [ CNTSIZE - 1:0] cntNext = cnt + 1;
   register #(.SIZE(CNTSIZE)) r_counter(clk, rst_n, cntNext, cnt);

   assign sck = cnt [3];
   assign cs = ~(cnt[CNTSIZE - 1:8] == { (CNTSIZE - 8) { 1'b1 }});

   wire sampleBit = ( cs == 1'b0 && cnt [3:0] == 4'b1000 );
   wire valueDone = ( cs == 1'b0 && cnt [7:0] == 8'b10101001 );
   wire [7:0] shift;
   wire [7:0] shiftNext = (shift << 1) | sdo;
   register_we #(.SIZE(8)) r_shift(clk, rst_n, sampleBit, shiftNext,
      shift);

   wire [7:0] valueNext = shift;
   register_we #(.SIZE(8)) r_value(clk, rst_n, valueDone, valueNext,
      value);
endmodule
```

Листинг 9.12 Модуль обмена данными с датчиком освещенности на основе счетчика

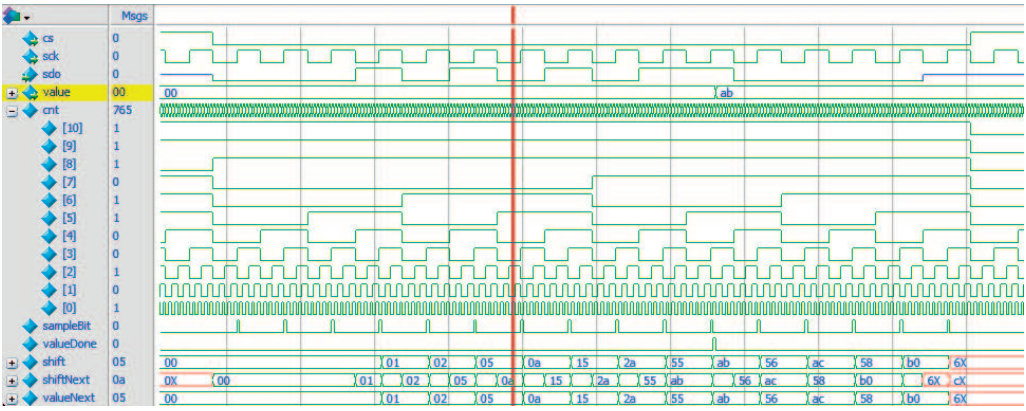


Рис. 9.28 Временная диаграмма модуля на основе счетчика

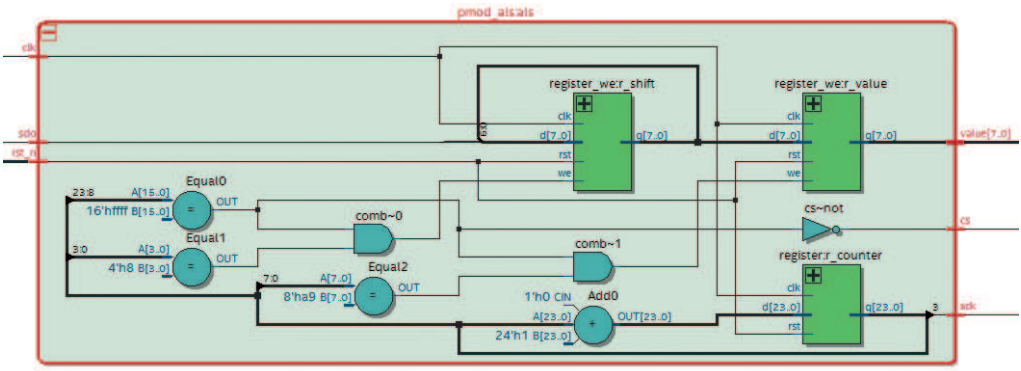


Рис. 9.29 RTL View модуля на основе счетчика

9.4 Упражнения

9.4.1 Основное задание

Выполните все задания данной главы.

9.4.2 Задания для самостоятельной работы

Низкий уровень сложности

1. Добавьте выходной буферный регистр в проект **02_compromise**, проверьте его работу в режиме симуляции и на отладочной плате.
2. Добавьте выходной буферный регистр в проект **06_counter**, проверьте его работу в режиме симуляции и на отладочной плате.
3. Соберите данные из **Fitter report** об использованных ресурсах **ПЛИС**, а также информацию о максимальной частоте, на которой могут работать все созданные в данной практической работе автоматы. Проанализируйте эту информацию (то, что сам датчик работает на пониженной частоте, можно проигнорировать).
4. Добавьте конфигурационный регистр в состав модуля обмена данными с датчиком освещенности. Значение этого регистра задается переключателями на плате и должно определять периодичность, с которой опрашивается датчик. Подготовьте реализацию в одном из вариантов: с помощью академического подхода либо с помощью конечного автомата на основе счетчика.
5. Реализуйте модуль, который будет эмулировать функцию датчика освещения. Передаваемые данные должны устанавливаться с помощью переключателей, установленных на отладочной плате. Соедините две отладочные платы, чтобы подтвердить работоспособность проекта: одна плата – ведущее устройство, несколько вариантов которого представлены в данной работе; вторая плата – ведомое устройство, разработанное в ходе выполнения задания.

Средний уровень сложности

1. Реализуйте приемник **UART**, проверьте его работу на отладочной плате с подключенным **USB-UART** конвертером.
2. Реализуйте передатчик **UART**, проверьте его работу на отладочной плате с подключенным **USB-UART** конвертером.
3. Реализуйте интерфейсный модуль протокола **I2C**, проверьте его работу на отладочной плате с подключенным датчиком температуры и влажности (**DHT11**).
4. Реализуйте интерфейсный модуль для датчика атмосферного давления (**BMP085** или похожий).
5. Реализуйте интерфейсный модуль для **1602 LCD** модуля (**HD44780** или похожий).
6. Реализуйте интерфейсный модуль для 16-кнопочной клавиатуры (**Pmod KYPD** или похожий).
7. Реализуйте интерфейсный модуль для ультразвукового датчика расстояния (**SR04** или похожий).
8. Реализуйте интерфейсный модуль для валкодера (**KY-040** или похожий).
9. Реализуйте интерфейсный модуль для **I2C** акселерометра (**MPU-6050** или похожий).
10. Реализуйте интерфейсный модуль для передатчика **RF 433MHz**.
11. Реализуйте интерфейсный модуль для приемника **RF 433MHz**.
12. Реализуйте интерфейсный модуль для **I2C TinyRTC**.
13. Реализуйте интерфейсный модуль для датчика жестов (**APDS-9960** или похожий).

Задания для работы в группе

1. Реализуйте передатчик сигналов Морзе. В качестве входного устройства должен выступать клавиатурный модуль, в качестве выходного – пьезоизлучатель.
2. Реализуйте приемник сигналов Морзе. В качестве входного устройства должен выступать датчик звука (**Arduino microphone sensor**), в качестве выходного – **LCD**-модуль.
3. Реализуйте передатчик сигналов Морзе. В качестве входного устройства должен выступать клавиатурный модуль, в качестве выходного – передатчик **RF 433MHz**.
4. Реализуйте приемник сигналов Морзе. В качестве входного устройства должен выступать приемник **RF 433MHz**, в качестве выходного – **LCD**-модуль.
5. Реализуйте часы, получающие информацию с **I2C TinyRTC** модуля и выводящие ее на **1602 LCD**-модуль.

6. Реализуйте погодную станцию, получающую информацию о температуре с **I2C**-датчика температуры и влажности и выводящую ее на **1602 LCD**-модуль.
7. Реализуйте погодную станцию, получающую информацию об атмосферном давлении с **I2C**-датчика и выводящую ее на **1602 LCD**-модуль.
8. Реализуйте автоматическую погодную станцию, которая сообщает о результатах измерений (влажность и температура) с помощью кода Морзе (звуком).
9. Реализуйте автоматическую погодную станцию, которая сообщает о результатах измерений (давление и температура) с помощью кода Морзе (звуком).
10. Реализуйте приемник, который будет принимать сведения с автоматической погодной станции, получая информацию с помощью **Arduino microphone sensor** и выводя ее на экран **LCD**-модуля.
11. Реализуйте автоматическую погодную станцию, которая сообщает о результатах измерений (влажность и температура) с помощью **RF 433MHz Transmitter**.
12. Реализуйте автоматическую погодную станцию, которая сообщает о результатах измерений (давление и температура) с помощью **RF 433MHz Transmitter**.
13. Реализуйте приемник, который будет принимать сведения с автоматической погодной станции, получая информацию с помощью **RF 433MHz Receiver** и выводя ее на экран **LCD**-модуля.

9.4.3 Контрольные вопросы

1. Какая диаграмма состояний (рис. 9.30) и какая временная диаграмма (рис. 9.31) соответствуют данному коду (листинг 9.13)?

```
always @(*) begin
   Next = State;
   case(State)
      S_A : Next = turnOff ? S_E : S_B;
      S_B : Next = S_C;
      S_C : if(cnt == C_SIZE) Next = S_A;
      S_D : ;
      S_E : Next = S_D;
   endcase
end
```

Листинг 9.13 Контрольный вопрос 1 (листинг кода)

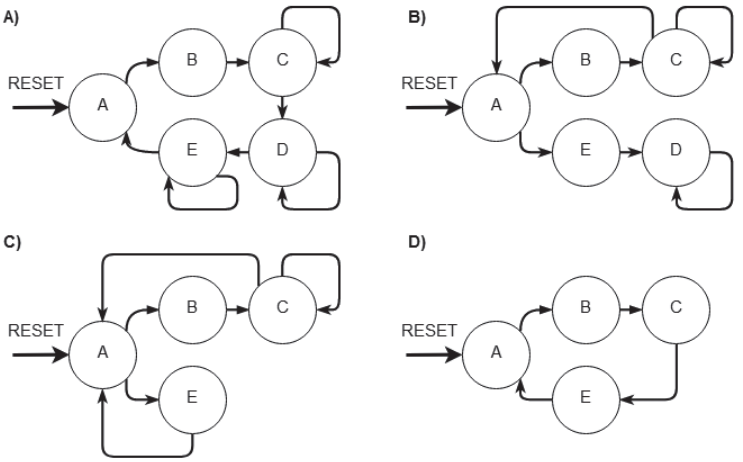


Рис. 9.30 Контрольный вопрос 1 (диаграммы состояний)

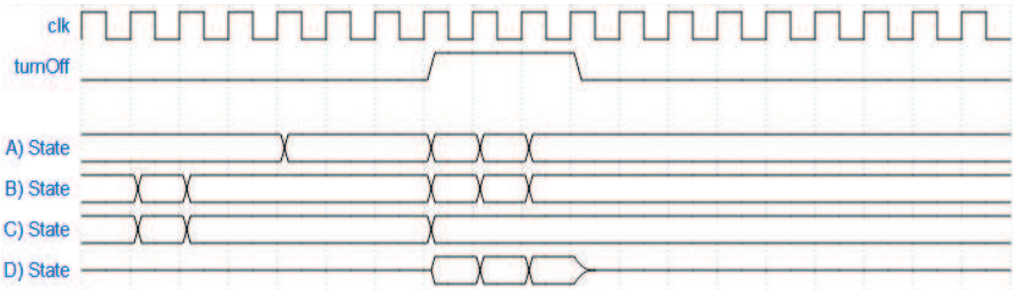


Рис. 9.31 Контрольный вопрос 1 (временная диаграмма)

2. Изобразите блок-схему **автомата Мура** с использованием символов комбинационной логики («облако»), регистра и соединительных линий.
3. Изобразите блок-схему **автомата Мили** с использованием символов комбинационной логики («облако»), регистра и соединительных линий.
4. Изобразите блок-схему автомата **Мура** с выходным буфером с использованием символов комбинационной логики («облако»), регистра и соединительных линий.
5. Изобразите блок-схему автомата **Мили** с выходным буфером с использованием символов комбинационной логики («облако»), регистра и соединительных линий.
6. Изобразите блок-схему автомата, построенного на базе счетчика с использованием символов комбинационной логики («облако»), регистра и соединительных линий. К какому типу он принадлежит: **Мура** или **Мили**? Можно ли изобразить два автомата, построенных на счетчиках и относящихся к различным типам?

7. Изобразите блок-схему автомата, построенного на базе счетчика с выходным буфером с использованием символов комбинационной логики («облако»), регистра и соединительных линий. К какому типу он принадлежит: **Мура** или **Мили**? Можно ли изобразить два автомата, построенных на счетчиках и относящихся к различным типам?

8. Заполните таблицу, данную ниже, значением задержки распространения сигнала от входа автомата к его выходу (задержку измерять в тактах):

Таблица 9.2 Таблица задержек распространения сигнала от входа автомата к его выходу

№ п/п	Тип автомата	Задержка распространения сигнала с входа на выход (такты)
1	Автомат Мура	
2	Автомат Мура с выходным буфером	
3	Автомат Мили	
4	Автомат Мили с выходным буфером	

9. Нарисуйте диаграмму состояний для детектора последовательности **1011**. Текущее значение бита – вход автомата; выход автомата должен принять значение 1, если на его входе будет обнаружена искомая последовательность.

10. Нарисуйте диаграмму состояний для кодового замка, ключевая последовательность которого – **три** символа. Входом могут быть: «не нажата ни одна из кнопок» либо код нажатой кнопки. Выход – «открыто» или «закрыто». Предусмотрите наличие таймера, определяющего длительность нахождения в состоянии «открыто».

Юрий Панчул, Александр Антонов, Александр Романов

Цифровой синтез: практический курс

Глава 10. Конвейерная обработка данных

Содержание

Глава знакомит с концепцией конвейеризации в проектировании цифровых схем. Основной целью практической работы является знакомство с основами конвейеризации для улучшения характеристик цифровых схем по сравнению с комбинационной и неконвейеризированной последовательностной реализациями. Дается описание свойств вычислительной нагрузки и деталей организации цифровых схем, которые следует принимать во внимание для достижения максимальной пользы от внедрения конвейеризации. Также рассматривается вопрос обеспечения эффективности кодирования конвейерных структур, для чего демонстрируются: разница между поведенческим и структурным стилями; использование циклов и выражений ***generate*** *для реализации регулярных схем; использование сигнала* ***enable*** *для управления запуском вычислений и уменьшения энергопотребления; синхронизация конвейера с операциями с переменной задержкой.*

Требования к аппаратным и программным средствам

Для выполнения практических работ вам понадобится следующее программное и аппаратное обеспечение:

- *персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;*
- *пакет* ***Quartus Prime*** *(есть студенческая версия);*
- *пакет* ***ModelSim Altera Edition*** *или программы* ***Icarus Verilog*** *и* ***GTKWave****;*
- *отладочная плата компании* ***Terasic DE10-Lite*** *или другая отладочная плата на основе* ***ПЛИС Intel FPGA*** *или* ***Xilinx*** *(может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах*[1] *к данной книге).*

Для запуска функциональных моделей требуется установленная среда программирования на языке ***Python*** *с подключенными библиотеками* ***NumPy****,* ***SciPy****,* ***Matplotlib****.*

10.1 Концепция конвейерной обработки как основы проектирования высокопроизводительных цифровых схем

Конвейеризация (**pipelining**) – это прием, который широко используется в проектировании цифровых схем с 1960-х гг. и позволяет в разы повысить пропускную способность цифровой схемы. При этом конвейеризация позволяет избежать кратного увеличения количества элементов цифровой логики. Данный подход полезен для проектирования вычислительных блоков различного назначения, включая процессоры, коммуникационные блоки и блоки подсистем памяти.

Основная идея конвейеризации заключается в применении следующих решений (рис. 10.1):

[1] https://github.com/RomeoMe5/DDLM.

1) тракт данных разбивается на последовательность стадий (ступеней конвейера) таким образом, что задержка прохождения данных в каждой из стадий меньше задержки всего исходного тракта данных;

2) запуск обработки новой порции данных (запроса), поступивших на вход конвейера, производится по мере освобождения первой стадии. Таким образом, обработка нового запроса начинается параллельно с продолжением обработки предыдущих запросов на более поздних стадиях конвейера.

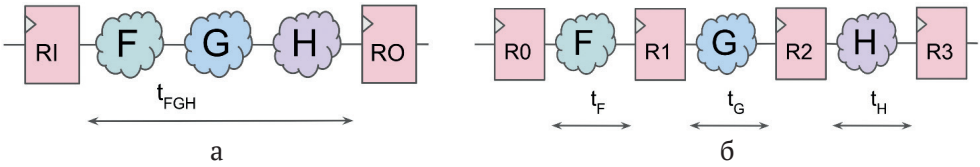


Рис. 10.1 Сравнение комбинационной (а) и конвейерной (б) реализаций цифровой схемы

Основой повышения производительности является распараллеливание обработки данных во времени: каждая стадия обрабатывает свой запрос параллельно с тем, как другие стадии обрабатывают другие запросы. Обе схемы могут принять к выполнению и завершить по одному запросу в течение одного цикла синхросигнала. Поскольку каждая из задержек t_F, t_G и t_H меньше общей задержки t_{FGH} (до трех раз), конвейерная схема может работать на более высокой тактовой частоте (до трех раз соответственно), что, в свою очередь, обеспечивает большую пропускную способность (тоже до трех раз). При этом с точки зрения объема цифровой логики конвейеризированная схема добавляет только буферные регистры для промежуточных результатов вычисления. Данные закономерности определяются следующими формулами:

$$BW_{peak(а)} = Fmax_а = \frac{1}{t_{FGH}},$$

$$BW_{peak(б)} = Fmax_б = \frac{1}{\max(t_F, t_G, t_H)},$$

$$\max(t_F, t_G, t_H) < t_{FGH} \Rightarrow BW_{peak(б)} > BW_{peak(а)},$$

где $BW_{peak(а)}$ и $BW_{peak(б)}$ – пиковая пропускная способность (в запросах за с); $Fmax_а$ и $Fmax_б$ – максимальная тактовая частота для схем (а) и (б) соответственно.

Конвейеризация широко применяется в современных цифровых аппаратных блоках. Примером такого блока является процессорное ядро **MIPS M5150** с конвейерной организацией (рис. 10.2). Данный конвейер обрабатывает следующие машинные инструкции: арифметические операции, операции с памятью, управление ветвлением и т. д. Подобные процессоры в настоящее время широко используются для встроенных приложений и приложений реального времени.

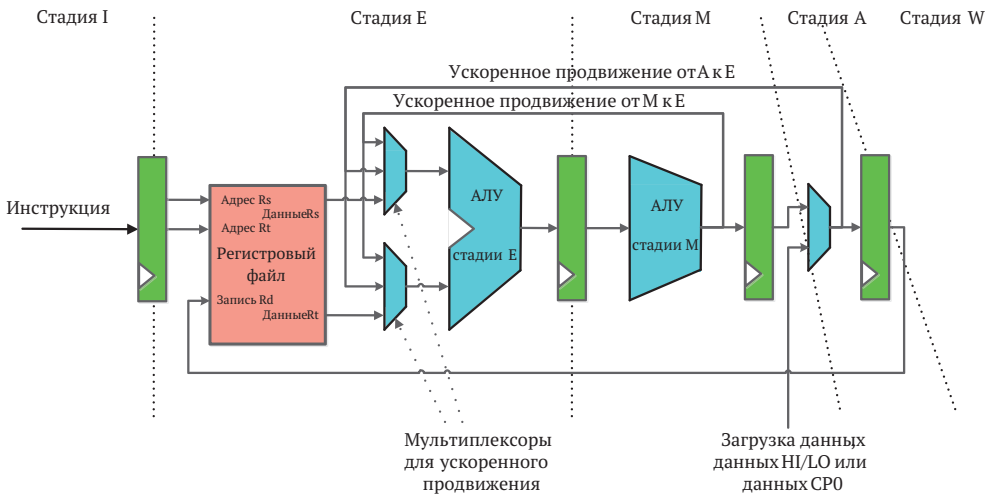


Рис. 10.2 Пример процессорного ядра с конвейерной организацией (MIPS M5150)

10.1.1 Конвейерные идеализмы

Эффективность применения конвейеризации ограничена определенными внутренними свойствами реализованных вычислений. К сожалению, не каждый вычислительный процесс может быть эффективно реализован в конвейерном стиле. Для максимизации пользы от применения конвейеризации разработчик должен адаптировать свою схему под эти свойства. В литературе[1] эти свойства носят название «**конвейерные идеализмы**». Выделяют три таких свойства:

- одинаковая задержка стадий;
- унифицированные типы запросов;
- независимость запросов.

Рассмотрим эти концепции более подробно.

10.1.2 Одинаковая задержка стадий

В идеальном случае задержка конвейера должна быть распределена по стадиям конвейера в равных пропорциях. В этом случае введение конвейера с **k** стадиями позволяет пропорционально повысить тактовую частоту и, соответственно, пропускную способность, в **k** раз (без учета временных затрат на предустановку и удержание сигналов на регистрах). Если задержка одной из стадий больше задержек остальных стадий, она становится **критической** для всего конвейера, т. е. ограничивает возможность повышения тактовой частоты конвейера в целом.

Неравенство задержек на стадиях конвейера называется **внутренней фрагментацией**. Устранение этого неравенства за счет перемещения вычислений между

[1] *Shen J. P., Lipasti M. H.* Modern processor design: fundamentals of superscalar processors. Waveland Press, 2013.

стадиями называется **балансировкой конвейера**. В том случае, если перемещение не имеет видимых побочных эффектов, инструменты синтеза могут выполнять его автоматически (этот процесс называется **retiming**). При этом отметим, что в общем случае правильное распределение вычислений по стадиям является обязанностью разработчика.

10.1.3 Унифицированные типы запросов

В простых конвейерных структурах тракт данных одинаков для всех типов запросов. В том случае, если определенный вычислительный ресурс какой-либо стадии не востребован для обработки какого-либо типа запросов (либо, в наихудшем случае, стадия конвейера для этих запросов сводится исключительно к копированию промежуточных результатов вычислений на другую стадию), поступление такого запроса приведет к простаиванию вычислительного ресурса и снижению потенциальной эффективности конвейера. Это простаивание называется **внешней фрагментацией**, и оно минимизируется разработчиками за счет унификации типов запросов и их обработки конвейерной структурой.

10.1.4 Независимость запросов

Поскольку конвейеризация подразумевает распараллеливание процесса обработки запросов, достижение пиковой эффективности возможно только в случае взаимной независимости запросов. Такая вычислительная нагрузка носит название **потоковой** – отдельные запросы в ней не имеют общих данных. В случае невозможности обеспечения независимости запросов может потребоваться введение дополнительных блокировок, приостанавливающих вычисления до разрешения зависимостей. Например, для выполнения одного запроса потребуется ожидание результатов выполнения другого.

В рамках данной практической работы все примеры основаны на потоковых арифметических вычислениях, поэтому такая проблема не возникает. При этом следует отметить, что более сложные блоки (например, современные высокопроизводительные процессоры) реализуют агрессивное распараллеливание обработки инструкций, сохраняя для программиста видимость последовательного выполнения инструкций. Это ведет к появлению отдельного класса проблем (таких как, например, **конвейерные конфликты**, вызванные конвейерной обработкой последовательных инструкций). Более подробно данные вопросы освещаются при изучении примера реализации микроархитектуры простого **MIPS**-процессора в главе 11.

10.1.5 Планирование вычислительной нагрузки

Потенциальная эффективность конвейеризированной цифровой схемы ограничена степенью присутствия конвейерных идеализмов. В случае если достичь заданных параметров не удается, решением может стать изменение самой вычислительной нагрузки с целью большего ее согласования с конвейерными идеализмами и ее лучшей адаптации к конвейерной реализации. Например, при проектировании системы команд процессора следует иметь в виду, что миними-

зация взаимосвязей между ними и унификация их обработки сделает аппаратуру эффективнее, а ее проектирование – менее трудоемким.

10.2 Практическое сравнение комбинационной, многотактной и конвейерной реализаций арифметического блока

В реальных цифровых схемах комбинационный, многотактный и конвейерный подходы применяются совместно. Выбор конкретного решения зависит от множества факторов: свойств вычислительной нагрузки, требований к тактовой частоте, характеристик компонентов технологической библиотеки и стратегии оптимизации. В данной главе будет проводиться сравнение следующих характеристик:

1) максимальной тактовой частоты;
2) максимальной пропускной способности в форме интервала инициации (минимального количества тактов синхросигнала между запросами) и количества запросов, обрабатываемых за секунду;
3) минимальной задержки обработки запроса (в тактах синхросигнала и наносекундах);
4) потребления ресурсов (логических ресурсов, регистров, арифметических блоков);
5) потребления энергии.

В данной главе проектирование конвейерных структур рассматривается на примере арифметических блоков. Эти блоки имеют единственный тип запроса, поскольку один и тот же алгоритм применяется к каждому входному набору данных. Кроме того, запросы обрабатываются независимо. Такие блоки могут быть особенно эффективно реализованы в конвейерном стиле: второй и третий идеализмы удовлетворяются автоматически (раздел 10.1.1). Первый же идеализм все еще остается актуальным.

В качестве демонстрационного примера рассмотрим **8**-разрядный блок возведения числа в **пятую** степень. Поскольку алгоритм предполагает повторение одной и той же операции (умножения), он может быть естественным путем разбит на стадии с одинаковой задержкой. Таким образом, первый идеализм также легко удовлетворяется, если количество операций является кратным количеству стадий конвейера. Как результат данный пример в целом демонстрирует идеализированную ситуацию, при которой конвейеризация дает наибольший прирост эффективности схемы. Проектирование реальной аппаратуры, как правило, существенно сложнее и может потребовать от разработчика дополнительного анализа и принятия компромиссных решений.

10.2.1 Функциональное моделирование

Перед проектированием непосредственно RTL-модели цифрового блока полезно создать его **эталонную модель** (**Golden model**). **Эталонная модель** предоставляет набор выходных реакций на тестовые векторы для верификации функциональности проектируемой аппаратуры. Данная модель будет общей для различных реализаций (комбинационной, многотактной, конвейеризированной) аппаратного блока. Такие модели также называют функциональными. В этой практической работе для описания функциональной модели используется язык программирования **Python**. Исходный код функциональной модели представлен ниже (листинг 10.1). Следует обратить внимание, что модель аппаратуры должна включать в себя исключительно «синтезируемые» операции (те, которые могут быть реализованы аппаратно). Корректность самой модели верифицируется с помощью встроенной математики Python.

```
# alg model - python math can be used here
def pow5_alg(x):
   return pow(x, 5) & 0xff

# hw model - synthesizable operations only
def pow5_hw(x):
   y = 1;
   for mul in range(0, 5):
      y = (y * x) & 0xff
   return y

# generating stimulus
for x in range(0, 10):
   alg_val = pow5_alg(x)
   hw_val = pow5_hw(x)
   if (alg_val == hw_val):
      print("Correct! x: ", hex(x).ljust(6), "; y: ", hex(hw_val).ljust(6))
   else:
      print("ERROR! x: ", hex(x).ljust(6), "; y(model): ",hex(alg_val).
         ljust(6), "; y(hw): ", hex(hw_val).ljust(6))
```

Листинг 10.1 Функциональная модель модуля возведения числа в степень

Тестовый вывод модели:

```
Correct! x: 0x0 ; y: 0x0
Correct! x: 0x1 ; y: 0x1
Correct! x: 0x2 ; y: 0x20
Correct! x: 0x3 ; y: 0xf3
Correct! x: 0x4 ; y: 0x0
Correct! x: 0x5 ; y: 0x35
Correct! x: 0x6 ; y: 0x60
Correct! x: 0x7 ; y: 0xa7
Correct! x: 0x8 ; y: 0x0
```

```
Correct! x: 0x9 ; y: 0xa9
```

Листинг 10.2 Тестовый вывод функциональной модели модуля возведения числа в степень

10.2.2 Комбинационная реализация

Комбинационная реализация – это простейшая реализация аппаратного блока обработки. С одной стороны, по сравнению с конвейерной реализацией, комбинационная реализация имеет меньшую задержку обработки запроса благодаря отсутствию внутренней фрагментации и необходимости буферизировать промежуточные результаты вычислений. Но в случае поступления нескольких различных типов запросов может возникнуть проблема внешней фрагментации. Тогда критическая задержка будет оставаться постоянной для всех типов запросов. Аналогично конвейерной реализации, комбинационная реализация требует отдельных вычислительных ресурсов для каждой операции и функционирует на относительно низкой тактовой частоте. Технически такие схемы могут работать в высокочастотном домене в виде так называемых **многотактных путей** (когда итерация вычислений производится раз в несколько тактов), но из-за необходимости ожидать эти несколько тактов пропускная способность всего тракта данных остается низкой.

Ниже приведен пример комбинационного модуля, вычисляющего пятую степень числа (файл **lab_10_1_pow_5\02_syn_pow_5_single_cycle_struct**):

```
module pow_5_single_cycle_struct
# (
   parameter w = 8
)
(
   input clk,
   input rst_n,
   input arg_vld,
   input [w - 1:0] arg,
   output res_vld,
   output [w - 1:0] res
);

   wire arg_vld_q;
   wire [w - 1:0] arg_q;
   reg_rst_n i_arg_vld (clk, rst_n, arg_vld, arg_vld_q);
   reg_no_rst # (w) i_arg (clk, arg, arg_q);
   wire res_vld_d = arg_vld_q;
   wire [w - 1:0] res_d = arg_q * arg_q * arg_q * arg_q * arg_q;
   reg_rst_n i_res_vld (clk, rst_n, res_vld_d, res_vld);
   reg_no_rst # (w) i_res (clk, res_d, res);
endmodule
```

Листинг 10.3 Исходный код комбинационной реализации модуля возведения числа в степень

Для моделирования схемы в среде **Modelsim** и получения временных диаграмм необходимо выполнить следующие действия:

1) запустить скрипт моделирования, который произведет моделирование всех модулей в проекте: **lab_10\src\lab_10_1_pow_5\01_sim_pow_5\ 01_simulate_with_modelsim.bat**;

2) открыть вкладку **Wave**. Результат должен быть таким же, как и на временных диаграммах:

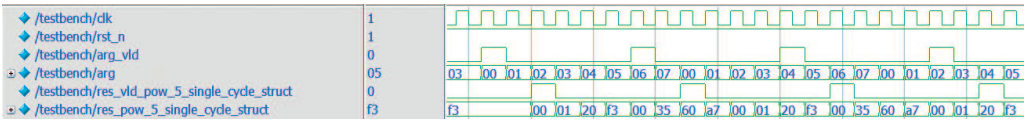


Рис. 10.3 Результаты моделирования комбинационной реализации модуля возведения числа в степень

Следует обратить внимание, что в результате моделирования получен вектор входных и выходных пар значений (**0x00/0x00**, **0x06/0x60**, **0x04/0x00**, **0x02/0x20** и т. д.), совпадающих с выходами функциональной модели (раздел 10.2.1). Это подтверждает корректность работы спроектированного блока.

Для сборки проекта и его прототипирования на **ПЛИС** необходимо выполнить следующие действия:

1) запустить скрипт **make_project.bat** для создания папки для синтеза и для хранения файлов проекта: **lab_10\src\lab_10_1_pow_5\02_syn_pow_5_single_cycle_struct\ de10_lite\make_project.bat**;

2) открыть файл проекта в среде **Quartus Prime**: **lab_10\src\lab_10_1_pow_5\ 02_syn_pow_5_single_cycle_struct\ de10_lite\project\de10_lite.qpf**;

3) запустить сборку проекта (**Processing → Start Compilation**);

4) после завершения компиляции можно открыть просмотр **RTL** (**Tools → Netlist Viewers → RTL Viewer**) и оценить количество функциональных блоков и их схему их соединения. **RTL**-представление комбинационной реализации показано на рис. 10.4);

5) сгенерировать файл прошивки (**bitstream**) и загрузить его в плату с **ПЛИС**.

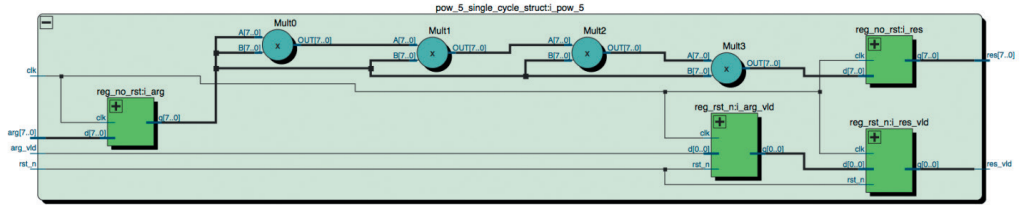


Рис. 10.4 RTL-представление комбинационной реализации модуля возведения числа в степень

Для выполнения прототипирования на плате с **ПЛИС** необходимо перевести переключатель **SW[9]** в активное положение, а затем вернуть в выключенное положение. Это перезапустит работу системы. Затем требуется задать входное значение переключателями **SW[7:0]** и нажать **KEY[0]**, для того чтобы отправить его на обработку. Индикация на светодиодах обозначит момент готовности выходных данных. Выходной байт данных будет выведен на два последних семисегментных индикатора.

Для оценки максимальной тактовой частоты **Fmax** (рис. 10.5) требуется запустить **TimeQuest Timing Analyzer → Slow 1200mV 85C Model → Fmax Summary**.

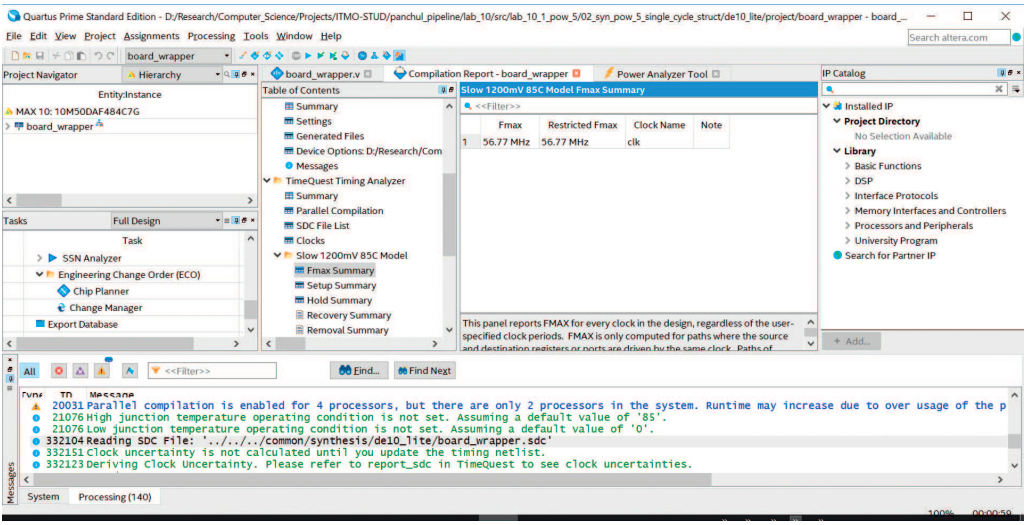


Рис. 10.5 Максимальная тактовая частота для комбинационной реализации модуля возведения числа в степень

В получившемся примере максимальная тактовая частота составляет **56,77 МГц**. Интервал инициации комбинационной реализации занимает один такт. Соответственно, пропускная способность для вычисленной частоты составляет **56,77** млн запросов в секунду.

Задержка комбинационной реализации также равна одному такту. Соответственно, задержка для заданной тактовой частоты составляет **1/Fmax** (**17,61 нс**).

Для оценки потребления ресурсов кристалла (рис. 10.6) необходимо открыть раздел **Flow Summary**:

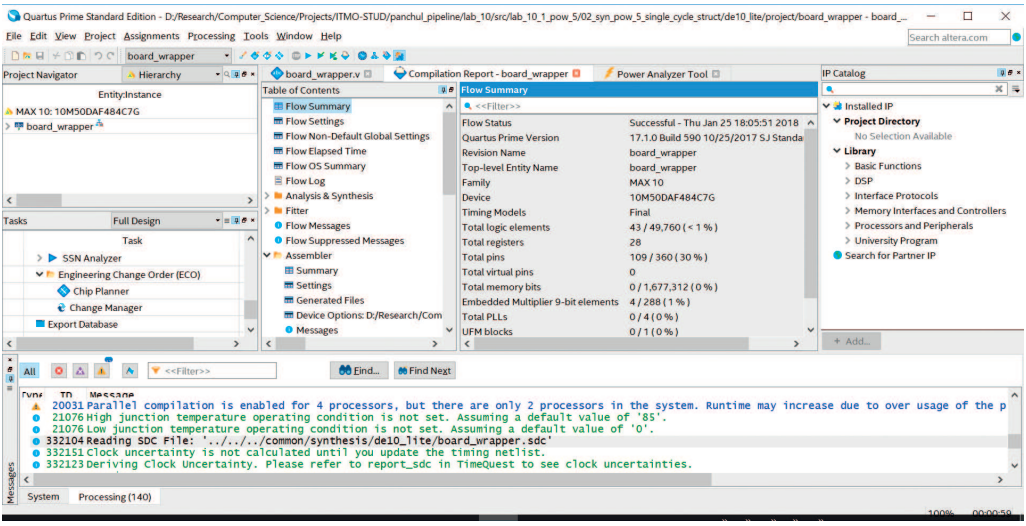


Рис. 10.6 Оценка потребления ресурсов ПЛИС

Для приведенного примера получены следующие оценки потребления ресурсов: **43** логических элемента, **28** регистров, **4** умножителя. Зафиксируем данные значения для последующего сравнения.

Для оценки энергопотребления (рис. 10.7) требуется запустить меню **Processing → Power Analyzer Tool**. Здесь можно выставить произвольную частоту переключения **ввода-вывода** (**I/O toggle rate**). Оставив значение по умолчанию – **12,5 %**, – нужно нажать **Start**. По завершении анализа нажать **Report**.

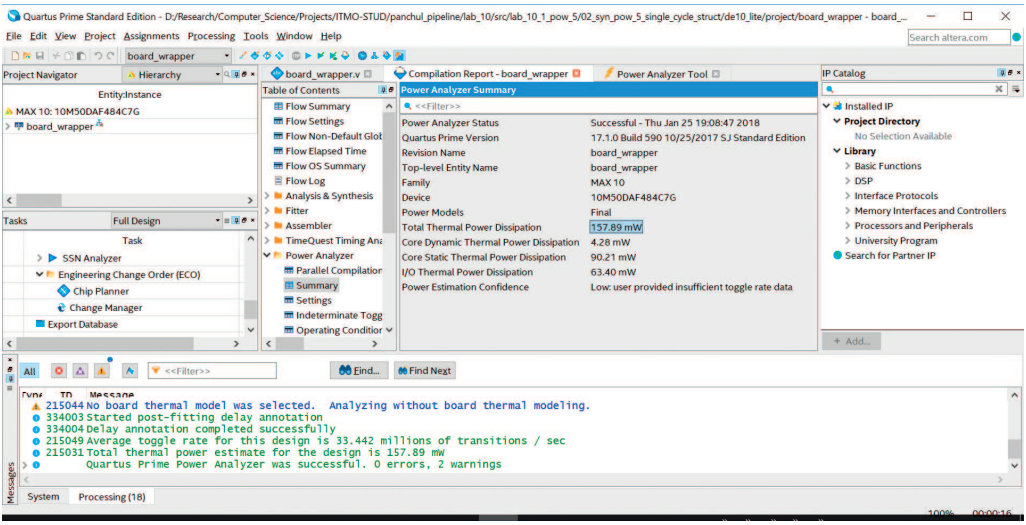


Рис. 10.7 Оценка энергопотребления

Для приведенного примера получены следующие оценки энергопотребления: **90,21 мВт** (статическое) и **4,28 мВт** (динамическое). Зафиксируем данные значения для последующего сравнения.

Сбор данных о разработанном модуле завершен.

10.2.3 Многотактная реализация

Многотактная неконвейеризированная реализация, по сравнению с комбинационной и конвейерной реализациями, позволяет сэкономить аппаратные ресурсы за счет их повторного использования на различных этапах обработки запроса. Здесь, как и в конвейерной реализации, требуется буферизация для хранения результатов промежуточных вычислений. Кроме того, координация передачи данных между вычислительными устройствами требует добавления дополнительной логики управления.

По сравнению с конвейерной реализацией, в многотактной внешняя фрагментация приводит к меньшим потерям эффективности (неиспользуемые стадии вычислений могут быть пропущены), при этом проблема внутренней фрагментации все равно сохраняется. В случае нескольких различных типов запросов пропускная способность многотактной реализации, как правило, выше, чем у комбинационной, но хуже, чем у конвейерной. Это объясняется тем, что запуск обработки нового запроса не начинается, пока не завершится предыдущий.

Исходный код примера многотактной реализации модуля возведения числа в пятую степень приведен ниже:

```
module pow_5_multi_cycle_struct
# ( parameter w = 8 )
(
   input clk,
   input rst_n,
   input arg_vld,
   input [w - 1:0] arg,
   output res_vld,
   output [w - 1:0] res
);
   wire arg_vld_q;
   wire [w - 1:0] arg_q;

   reg_rst_n i_arg_vld (clk, rst_n, arg_vld, arg_vld_q);
   reg_no_rst_en # (w) i_arg (clk, arg_vld, arg, arg_q);

   wire [3:0] shift_q;
   wire [3:0] shift_d = arg_vld_q ? 4'b1000 : shift_q >> 1;

   reg_rst_n # (4) i_shift (clk, rst_n, shift_d, shift_q);

   assign res_vld = shift_q [0];

   wire [w - 1:0] mul_q;
```

```
    wire [w - 1:0] mul_d = (arg_vld_q ? arg_q : mul_q) * arg_q;
    wire mul_en = arg_vld_q || shift_q [3:1] != 3'b0;
    reg_no_rst_en # (w) i_mul (clk, mul_en, mul_d, mul_q);
    assign res = mul_q;
endmodule
```

Листинг 10.4 Исходный код многотактной реализации модуля возведения числа в степень

Результаты моделирования многотактной реализации приведены ниже:

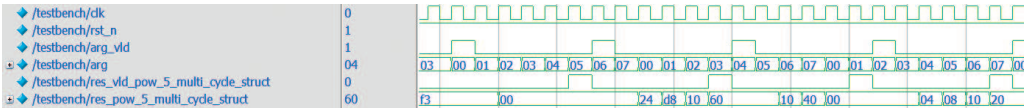


Рис. 10.8 Результаты моделирования многотактной реализации модуля возведения числа в степень

Следует обратить внимание на то, что пары входных и выходных значений идентичны для функциональной модели и комбинационной реализации, но поскольку цикл вычисления теперь занимает четыре такта, задержка формирования результата также составляет четыре такта.

RTL-представление многотактной реализации следующее:

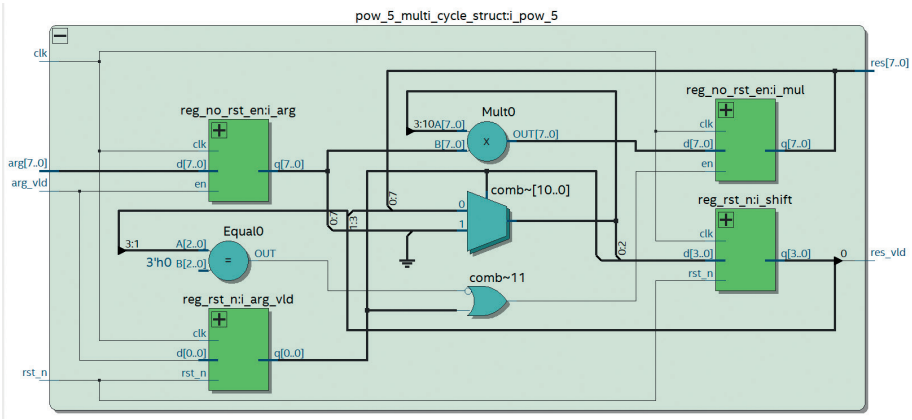


Рис. 10.9 RTL-представление многотактной реализации

Ход сбора данных аналогичен комбинационной реализации. Для этого необходимо повторить шаги, описанные в предыдущем разделе, для многотактной реализации. Проект расположен в дополнительных материалах[1] к данной главе в каталоге **lab_10\src\ lab_10_1_pow_5\ 05_syn_pow_5_multi_cycle_struct**.

10.2.4 Конвейерная реализация

В данном разделе описан пример конвейерной реализации проекта. Конвейерная реализация обеспечивает более высокую пропускную способность, требуя

[1] https://github.com/RomeoMe5/DDLM.

при этом больше аппаратных ресурсов (отдельные вычислительные ресурсы для каждой операции аналогично комбинационной реализации и аналогично многотактной реализации, дополнительные ресурсы на буферизацию). Также конвейерная реализация может иметь увеличенную задержку обработки запроса из-за задержек на буферизацию и (при наличии) внутренней фрагментации.

В соответствии с первым конвейерным идеализмом задержку всей обработки целесообразно разбивать на одинаковые части. Это снижает внутреннюю фрагментацию и повышает максимальную тактовую частоту работы блока. Для рассматриваемого примера применен простой подход, который состоит в том, что алгоритм четырехкратного умножения разбит на четыре равные части. Таким образом, конвейер состоит из четырех стадий (по одному умножению на стадию).

Исходный код конвейерной реализации модуля возведения числа в **пятую** степень приведен ниже:

```
module pow_5_pipe_struct
# (
   parameter w = 8
)
(
   input clk,
   input rst_n,
   input arg_vld,
   input [ w - 1:0] arg,
   output [ 4:0] res_vld,
   output [5 * w - 1:0] res
);
   wire arg_vld_q_1;
   wire [w - 1:0] arg_q_1;

   reg_rst_n i0_arg_vld (clk, rst_n, arg_vld, arg_vld_q_1);
   reg_no_rst # (w) i0_arg (clk, arg, arg_q_1);

   assign res_vld [4 ] = arg_vld_q_1;
   assign res [4 * w +: w] = arg_q_1;

//--------------------------------------------------------------------
   wire [w - 1:0] mul_d_1 = arg_q_1 * arg_q_1;
   wire arg_vld_q_2;
   wire [w - 1:0] arg_q_2;
   wire [w - 1:0] mul_q_2;

   reg_rst_n i1_arg_vld ( clk , rst_n , arg_vld_q_1 , arg_vld_q_2 );
   reg_no_rst # (w) i1_arg ( clk , arg_q_1 , arg_q_2 );
   reg_no_rst # (w) i1_mul ( clk , mul_d_1 , mul_q_2 );

   assign res_vld [3 ] = arg_vld_q_2;
   assign res [3 * w +: w] = mul_q_2;
```

```
//------------------------------------------------------------------------
   wire [w - 1:0] mul_d_2 = mul_q_2 * arg_q_2;
   wire arg_vld_q_3;
   wire [w - 1:0] arg_q_3;
   wire [w - 1:0] mul_q_3;

   reg_rst_n i2_arg_vld ( clk , rst_n , arg_vld_q_2 , arg_vld_q_3 );
   reg_no_rst # (w) i2_arg ( clk , arg_q_2 , arg_q_3 );
   reg_no_rst # (w) i2_mul ( clk , mul_d_2 , mul_q_3 );

   assign res_vld [2 ] = arg_vld_q_3;
   assign res [2 * w +: w] = mul_q_3;

//------------------------------------------------------------------------
   wire [w - 1:0] mul_d_3 = mul_q_3 * arg_q_3;
   wire arg_vld_q_4;
   wire [w - 1:0] arg_q_4;
   wire [w - 1:0] mul_q_4;

   reg_rst_n i3_arg_vld ( clk , rst_n , arg_vld_q_3 , arg_vld_q_4 );
   reg_no_rst # (w) i3_arg ( clk , arg_q_3 , arg_q_4 );
   reg_no_rst # (w) i3_mul ( clk , mul_d_3 , mul_q_4 );

   assign res_vld [1 ] = arg_vld_q_4;
   assign res [1 * w +: w] = mul_q_4;

//------------------------------------------------------------------------
   wire [w - 1:0] mul_d_4 = mul_q_4 * arg_q_4;
   wire arg_vld_q_5;
   wire [w - 1:0] arg_q_5;
   wire [w - 1:0] mul_q_5;

   reg_rst_n i4_arg_vld ( clk , rst_n , arg_vld_q_4 , arg_vld_q_5 );
   reg_no_rst # (w) i4_arg ( clk , arg_q_4 , arg_q_5 );
   reg_no_rst # (w) i4_mul ( clk , mul_d_4 , mul_q_5 );

   assign res_vld [0 ] = arg_vld_q_5;
   assign res [0 * w +: w] = mul_q_5;
endmodule
```

Листинг 10.5 Исходный код конвейерной реализации модуля возведения числа в степень

Результаты моделирования конвейерной реализации следующие:

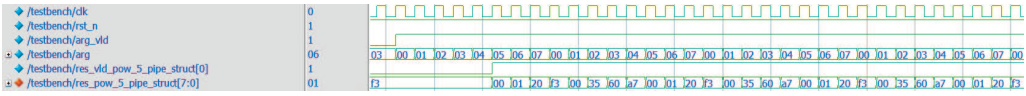


Рис. 10.10 Результаты моделирования конвейерной реализации

Обратите внимание, что пары входных и выходных значений аналогичны функциональной модели и предыдущим реализациям. Задержка формирования результата составляет четыре такта, как и в многотактной реализации. При этом аппаратный блок может принимать новую порцию данных каждый такт.

RTL-представление конвейерной реализации следующее:

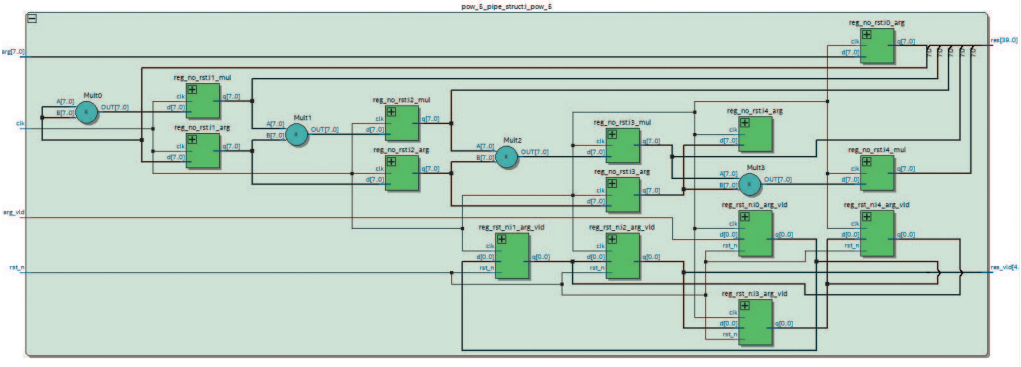


Рис. 10.11 RTL-представление конвейерной реализации

Анализ характеристик получившейся реализации аналогичен анализу, проведенному для комбинационной и многотактной реализаций. Для этого необходимо повторить шаги, описанные в разделе 10.2.2, для конвейерной реализации. На плате с **ПЛИС** результаты работы трех последних стадий отображаются на шести семисегментных индикаторах (по два индикатора на значение).

Проект расположен в каталоге l**ab_10\src\lab_10_1_pow_5\07_syn_pow_5_pipe_struct**.

10.2.5 Сравнение результатов

Полученные характеристики разработанных реализаций модуля возведения числа в пятую степень представлены в следующих сводных таблицах:

Таблица 10.1 Сравнение производительности различных реализаций разработанного модуля возведения числа в степень

Вариант реализации	Fmax, МГц	Пропускная способность		Задержка	
		Интервал инициации, такт	Моп/с	такт	нс
Комбинационная	56,77	1	56,77	1	17,61
Многотактная	165.07	4	41,27	4	24,23
Конвейерная	186.25	1	186.25	4	21,48

Таблица 10.2 Сравнение потребления ресурсов

Реализация	Аппаратные затраты			Энергопотребление, мВт	
	Комб. логика, LE	Регистры, ед.	Умножители, ед.	Статическое	Динамическое
Комбинационная	43	28	4	90,21	4,28
Многотактная	52	37	1	90,16	3,25
Конвейерная	137	54	4	90,25	5,09

В результате сравнения разработанных реализаций можно сделать следующие выводы:

1) **максимальная тактовая частота**. Наиболее эффективной является конвейерная реализация благодаря низкой задержке на каждой из стадий. Максимальная частота многотактной реализации немного ниже из-за дополнительной логики управления. Комбинационная реализация является самой медленной, поскольку все вычисления производятся за один такт, и межрегистровая передача в этом случае самая длинная;

2) **пропускная способность**. Конвейерная реализация является наилучшей благодаря высокой тактовой частоте и параллелизму обработки. Комбинационная и многотактная реализации существенно хуже. Многотактная реализация немного хуже комбинационной из-за задержек, добавляемых буферизацией, а также потому, что в данном примере внешняя фрагментация не играет роли (все запросы предусматривают одни и те же вычисления);

3) **задержка обработки запроса**. Комбинационная реализация является наилучшей. Многотактная реализация хуже из-за задержек на буферизацию. Конвейерная реализация лучше многотактной благодаря более высокой тактовой частоте;

4) **аппаратные затраты**. Многотактная реализация является наилучшей благодаря повторному использованию вычислительных ресурсов. Комбинационная реализация хуже в части потребления умножителей, но лучше по регистрам (поскольку не подразумевает буферизации промежуточных результатов вычислений). Наихудшей является конвейерная реализация: она требует затрат ресурсов как на буферизацию, так и отдельных ресурсов для каждой стадии конвейера;

5) **энергопотребление**. Энергопотребление в КМОП-схемах может быть вычислено по следующим формулам:

$$P_{\text{total}} = P_{\text{static}} + P_{\text{dynamic}}; P_{\text{static}} = V \times I_{\text{leakage}}; P_{\text{dynamic}} = Q \times f \times C \times V^2,$$

где P_{total} – общее энергопотребление, P_{static} – статическое энергопотребление, P_{dynamic} – динамическое энергопотребление, V – напряжение питания, I_{leakage} – ток утечки, Q – количество переключающихся транзисторов, f – частота синхросигнала, C – емкость.

Как правило, разработчик **RTL** не может напрямую управлять физическими параметрами чипа (такими как напряжение питания и физические параметры транзисторов). Тем не менее дизайн спроектированного блока существенно влияет на характеристики энергопотребления микросхемы. Во-первых, большее количество логики означает большее количество устройств, подключенных к цепи питания. Это увеличивает токи утечки и статическое энергопотребление. Во-вторых, динамическая составляющая мощности линейно зависит от интенсивности переключения транзисторов за единицу времени.

Поскольку в рассматриваемом примере микросхема **ПЛИС** не позволяет управлять подключением к питанию индивидуальных блоков на кристалле, все спроектированные блоки имеют практически идентичный объем статического энергопотребления. При одинаковой интенсивности переключения транзисторов (**12,5 %**) динамическое энергопотребление коррелирует с аппаратными затратами. Таким образом, можно утверждать следующее: многотактная реализация является наилучшей, в то время как комбинационная и конвейерная реализации хуже, поскольку они потребляют больше ресурсов кристалла.

10.3 Дополнительные приемы эффективного проектирования конвейерных схем

До сих пор были рассмотрены преимущества применения конвейеризации в цифровых схемах. Существует также несколько полезных приемов, которые позволяют еще больше улучшить конвейеризованные схемы как по характеристикам, так и в сторону их большей гибкости и удобства для применения на практике.

10.3.1 Поведенческий и структурный стили кодирования

Язык описания аппаратуры **Verilog**, аналогично ряду других языков описания аппаратуры, изначально был разработан для моделирования цифровых схем и лишь затем стал использоваться для синтеза. Данный язык предусматривает различные стили моделирования аппаратуры. Для синтезируемого подмножества (т. е. подмножества языка, служащего для создания описания цифровой системы, из которого можно синтезировать дизайн микросхемы и прошивку для **ПЛИС**) выделяются два подхода к проектированию: поведенческий и структурный. Рассмотренный выше пример демонстрирует структурный подход, так как содержит присвоения к типу **wire** и инстанциирование модулей. Все присвоения функционируют одновременно, что прямо соответствует структурной природе аппаратуры как набора взаимосвязанных компонентов.

В свою очередь, поведенческий подход зачастую более удобен и широко применяется на практике (глава 6). Описание фрагмента аппаратной логики в виде небольшой последовательной программы внутри блока **always** может существенно упростить кодирование поведения аппаратуры. При этом следует иметь в виду, что знание семантики языка **Verilog**, включая списки чувствительности и использование блокирующих и неблокирующих присвоений, и понимание того, ка-

ким образом управляющие конструкции языка (**if**, **case**, **for** и др.) транслируются в цифровые схемы, необходимы для создания работоспособных и эффективных аппаратных блоков.

Рассмотрим конвейерную реализацию, выполненную в поведенческом стиле:

```
module pow_5_pipe_always

# (
parameter w = 8
)
(
   input clk,
   input rst_n,
   input arg_vld,
   input [ w - 1:0] arg,
   output [ 4:0] res_vld,
   output [5 * w - 1:0] res
);
   reg [w - 1:0] arg1, arg2, arg3, arg4;
   reg [w - 1:0] pow2, pow3, pow4, pow5;
   reg arg_vld_1, arg_vld_2, arg_vld_3, arg_vld_4, arg_vld_5;

   always @ (posedge clk or negedge rst_n)
      if (! rst_n)
      begin
         arg_vld_1 <= 1'b0;
         arg_vld_2 <= 1'b0;
         arg_vld_3 <= 1'b0;
         arg_vld_4 <= 1'b0;
         arg_vld_5 <= 1'b0;
      end
      else
      begin
         arg_vld_1 <= arg_vld;
         arg_vld_2 <= arg_vld_1;
         arg_vld_3 <= arg_vld_2;
         arg_vld_4 <= arg_vld_3;
         arg_vld_5 <= arg_vld_4;
      end

   always @ (posedge clk)
   begin
      arg1 <= arg;
      arg2 <= arg1;
      arg3 <= arg2;
      arg4 <= arg3;
      pow2 <= arg1 * arg1;
```

```
      pow3 <= pow2 * arg2;
      pow4 <= pow3 * arg3;
      pow5 <= pow4 * arg4;
   end
   assign res_vld = { arg_vld_1, arg_vld_2, arg_vld_3, arg_vld_4,
      arg_vld_5 };
   assign res = { arg1 , pow2 , pow3 , pow4 , pow5};
endmodule
```

Листинг 10.6 Конвейерная реализация модуля возведения числа в степень, выполненная в поведенческом стиле

Следует обратить внимание на то, что по сравнению со структурным вариантом кодирования объем исходного кода существенно сократился с **83** до **52** строк. Также упростился анализ функциональности, т. к. множество соединенных элементов заместилось компактным описанием поведения блока.

Выполнив имплементацию модуля и сравнив характеристики с предыдущими реализациями, можно убедиться, что характеристики синтезированного модуля не изменились. Проект с исходным кодом располагается в следующем каталоге дополнительных материалов[1] к данной главе: **lab_10\src\lab_10_1_pow_5\ 09_syn_pow_5_pipe_always**.

Дополнительное задание для самостоятельной работы

Реализуйте однотактный и многотактный модули в поведенческом стиле. В качестве примера используйте примеры реализаций данных стилей проектирования в следующих каталогах: **lab_10\src\lab_10_1_pow_5\03_syn_pow_5_single_cycle_always** и **lab_10\src\ lab_10_1_pow_5\06_syn_pow_5_multi_cycle_always** (комбинационная и многотактная реализации соответственно).

10.3.2 Реализация регулярных структур

Типичные цифровые схемы, особенно если речь идет о высокопроизводительных схемах, могут содержать множество обрабатывающих узлов, работающих параллельно, для повышения производительности. Кодирование таких структур на языке **Verilog** в виде множества повторяющихся объявлений операторов, конструкций **always** и модулей, особенно когда их количество больше десяти, приводит к нарушению принципа «**недублирования кода**». Нарушение этого принципа приводит к возникновению ошибок в именовании или индексации экземпляров модулей и соединений, а также необходимости модификации исходного кода каждый раз, когда нужен новый модуль с аналогичной организацией, но иным количеством обрабатывающих узлов.

Для решения этой проблемы повторяющуюся логику удобно описывать с использованием параметризированного шаблона. Такой шаблон содержит логику, которая описывает аппаратуру в соответствии с заданными параметрами. При этом нужно иметь в виду, что анализ этих параметров выполняется до того, как модуль

[1] https://github.com/RomeoMe5/DDLM.

будет превращен в список соединений (**netlist**), и их значения не могут быть изменены без повторной компиляции модуля.

В рамках поведенческого стиля повторяющиеся структуры описываются с помощью циклов внутри блоков **always** и многомерных переменных. Ниже приведен пример модуля, который выполняет возведение **w**-разрядного числа в степень **n**:

```
module pow_n_pipe_always
# (
   parameter w = 8, n = 5
)
(
   input clk,
   input rst_n,
   input arg_vld,
   input [ w - 1:0 ] arg,
   output reg [ n - 1:0 ] res_vld,
   output reg [ n * w - 1:0 ] res
);
   reg [ w - 1 : 0 ] arg_reg [ 1 : n - 1 ];
   reg [ w - 1 : 0 ] pow [ 2 : n ];
   reg [ 1 : n ] arg_vld_reg;

   integer i;
   always @ (posedge clk or negedge rst_n)
      if (! rst_n)
      begin
         for (i = 1; i <= n; i = i + 1)
            arg_vld_reg [i] <= 1'b0;
      end
      else
      begin
         arg_vld_reg [1] <= arg_vld;
         for (i = 1; i <= n - 1; i = i + 1)
            arg_vld_reg [i + 1] <= arg_vld_reg [i];
      end

   always @ (posedge clk)
      begin
         arg_reg [1] <= arg;
         for (i = 1; i <= n - 2; i = i + 1)
            arg_reg [i + 1] <= arg_reg [i];
         pow [2] <= arg_reg [1] * arg_reg [1];
         for (i = 2; i <= n - 1; i = i + 1)
            pow [i + 1] <= pow [i] * arg_reg [i];
      end

   always @*
```

```
    begin
      for (i = 1; i <= n; i = i + 1)
        res_vld [n - i] = arg_vld_reg [i];
      res [(n - 1) * w +: w] = arg_reg [1];
      for (i = 2; i <= n; i = i + 1)
        res [(n - i) * w +: w] = pow [i];
    end
endmodule
```

Листинг 10.7 Генерация повторяющейся аппаратуры в рамках поведенческого стиля кодирования

Экземпляр данного модуля может быть объявлен с любым значением параметра. Кроме того, можно использовать в одном проекте несколько таких модулей с различными значениями параметра. Таким образом, различные экземпляры будут реализовывать различные алгоритмы, в то время как исходный код модуля будет оставаться неизменным.

Дополнительное задание для самостоятельной работы

Реализуйте данный модуль с параметром **w**, равным значению **5**, проанализируйте его характеристики и убедитесь, что они не изменились по сравнению с предыдущими реализациями. Проект находится в соответствующей папке дополнительных материалов к данной главе: **lab_10\src\lab_10_1_pow_5\ 11_syn_pow_n_pipe_always**.

Задайте различные значения параметров модуля **pow_n_pipe_always**. Убедитесь в работоспособности получившихся реализаций, выполните имплементацию и сравните их характеристики с характеристиками исходной реализации.

10.3.3 Конструкция generate в рамках поведенческого стиля описания аппаратуры

В предыдущем примере для реализации повторяющейся аппаратуры внутри блока **always** использовалась процедурная логика. Тем не менее для присвоений в рамках структурного стиля, а также для объявления экземпляров модулей данный подход неприменим. Для этих случаев в **Verilog** предусмотрена конструкция **generate**. Эта конструкция реализует метапрограммирование в **Verilog**, выполняется на начальном этапе компиляции модуля (фаза **elaboration**) и позволяет программировать объявление экземпляров произвольных аппаратных структур (присвоения к типу **wire**, блоки **always**, подмодули и т. д.). Как и в любой программе на языке процедурного программирования, в ней можно использовать условные операторы и циклы. Для хранения промежуточных значений **generate** использует специальные переменные типа **genvar**, которые также существуют только на стадии **elaboration**.

Ниже приведен пример реализации разрабатываемого модуля с использованием структурного стиля, где степень числа задается с помощью конструкции generate:

```
module pow_n_pipe_struct
# (
   parameter w = 8, n = 5
)
(
   input clk,
   input rst_n,
   input arg_vld,
   input [ w - 1:0 ] arg,
   output [ n - 1:0 ] res_vld,
   output [ n * w - 1:0 ] res
);
   wire [w - 1:0] mul_d [ 1 : n - 1 ];
   wire arg_vld_q [ 0 : n ];
   wire [w - 1:0] arg_q [ 0 : n ];
   wire [w - 1:0] mul_q [ 2 : n ];

   assign arg_vld_q [0] = arg_vld;
   assign arg_q [0] = arg;
   assign mul_d [1] = arg_q [1] * arg_q [1];

   generate
      genvar i;

      for (i = 2; i <= n - 1; i = i + 1)
      begin : b_mul
         assign mul_d [i] = mul_q [i] * arg_q [i];
      end

      for (i = 1; i <= n - 1; i = i + 1)
      begin : b_mul_reg
         reg_no_rst # (w) i_mul (clk, mul_d [i], mul_q [i + 1]);
      end

      for (i = 0; i <= n - 1; i = i + 1)
      begin : b_regs
         reg_rst_n i_arg_vld
            (clk, rst_n, arg_vld_q [i], arg_vld_q [i + 1]);
         reg_no_rst # (w) i_arg (clk, arg_q [i], arg_q [i + 1]);
      end

      for (i = 2; i <= n; i = i + 1)
      begin : b_res
         assign res_vld [ n - i ] = arg_vld_q [i];
         assign res [ ( n - i ) * w +: w ] = mul_q [i];
      end
   endgenerate

   assign res_vld [ n - 1 ] = arg_vld_q [1];
```

```
   assign res [ ( n - 1 ) * w +: w ] = arg_q [1];
endmodule
```

Листинг 10.8 Генерация повторяющейся аппаратуры с помощью конструкции generate

Дополнительное задание для самостоятельной работы

Выполните имплементацию модуля со значением параметра **n = 5**, проанализируйте его характеристики и убедитесь, что они не изменились. Проект находится в папке **lab_10\src\lab_10_1_pow_5\08_syn_pow_n_pipe_struct**.

Попробуйте задать различные значения параметров модуля **pow_n_pipe_struct**. Убедитесь в работоспособности получившихся реализаций, выполните имплементацию и сравните их характеристики с характеристиками исходной реализации.

10.3.4 Использование сигнала enable для разрешения работы модуля

До сих пор для искусственного понижения скорости функционирования и наблюдаемости работы устройства с помощью светодиодных индикаторов в данной главе использовался делитель тактовой частоты. Такой подход не самый лучший в использовании по ряду причин:

1) **пропускная способность**. Поскольку пиковая пропускная способность устройства в этом случае линейно зависит от тактовой частоты, низкая тактовая частота ограничивает производительность блока;

2) **энергопотребление**. В соответствии с разделом 10.2.5 динамическое энергопотребление линейно зависит от интенсивности переключения транзисторов. Транзисторы переключаются при изменении значения входных сигналов, и эти изменения распространяются по схеме даже тогда, когда вычисление само по себе не требуется. Поэтому при увеличении тактовой частоты динамическая составляющая энергопотребления также увеличивается;

3) **согласование тактовых доменов**. Использование множества тактовых сигналов может быть полезно для индивидуального управления соотношением производительности и энергопотребления укрупненных узлов в составе системы. Но сопряжение доменов синхронизации требует специальных решений для предотвращения гонок и метастабильных состояний[1].

Таким образом, небольшие аппаратные блоки рекомендуется тактировать одним синхросигналом (или, другими словами, размещать их в рамках одного домена синхронизации). Для домена с высокопроизводительными блоками, скорее всего, потребуется синхросигнал с высокой частотой. Управляемость вычислений внутри этих блоков можно обеспечить за счет дополнительного сигнала, стробирующего поступление запросов (сигнал **enable**). Этот сигнал разрешает запуск вычислений, позволяя оставить модуль в высокочастотном домене. В то же время этот сигнал используется для управления записью значений в регистры, пре-

[1] *Cummings C. E.* Clock domain crossing (cdc) design & verification techniques using SystemVerilog. SNUG2008, Boston.

дотвращая переключение транзисторов в том случае, если вычисление не требуется. Недостатком подхода является наличие дополнительного сигнала и логики управления. Тем не менее положительный эффект может компенсировать обозначенный выше недостаток.

Реализация модуля возведения числа в степень с сигналом **enable** представлена ниже:

```
module pow_5_en_pipe_always
# (
   parameter w = 8
)
(
   input clk,
   input rst_n,
   input clk_en,
   input arg_vld,
   input [ w - 1:0] arg,
   output [ 4:0] res_vld,
   output [5 * w - 1:0] res
);
   reg [w - 1:0] arg1, arg2, arg3, arg4;
   reg [w - 1:0] pow2, pow3, pow4, pow5;
   reg arg_vld_1, arg_vld_2, arg_vld_3, arg_vld_4, arg_vld_5;

   always @ (posedge clk or negedge rst_n)
      if (! rst_n)
      begin
         arg_vld_1 <= 1'b0;
         arg_vld_2 <= 1'b0;
         arg_vld_3 <= 1'b0;
         arg_vld_4 <= 1'b0;
         arg_vld_5 <= 1'b0;
      end
      else if (clk_en)
      begin
         arg_vld_1 <= arg_vld;
         arg_vld_2 <= arg_vld_1;
         arg_vld_3 <= arg_vld_2;
         arg_vld_4 <= arg_vld_3;
         arg_vld_5 <= arg_vld_4;
      end

   always @ (posedge clk)
      if (clk_en)
      begin
         arg1 <= arg;
         arg2 <= arg1;
```

```
        arg3 <= arg2;
        arg4 <= arg3;

        pow2 <= arg1 * arg1;
        pow3 <= pow2 * arg2;
        pow4 <= pow3 * arg3;
        pow5 <= pow4 * arg4;
    end

  assign res_vld = { arg_vld_1 , arg_vld_2 , arg_vld_3 , arg_vld_4 ,
    arg_vld_5 };
  assign res = { arg1 , pow2 , pow3 , pow4 , pow5};
endmodule
```

Листинг 10.9 Пример использования сигнала enable

После выполнения имплементации модуля с сигналом **enable** (**lab_10\src\lab_10_1_pow_5\ 18_syn_pow_5_en_pipe_always**) его следует сравнить с оригинальным конвейерным модулем без сигнала **enable** (**lab_10\src\lab_10_1_pow_5\09_syn_pow_5_pipe_always**). Результаты сравнения сведены в таблицу:

Таблица 10.3 Сравнение потребления ресурсов реализациями с/без сигнала enable

Модуль	Потребление аппаратных ресурсов			Энергопотребление	
	Комбинационная логика, LE	Регистры, ед.	Умножители, ед.	Статическое	Динамическое
Без сигнала enable (начальная реализация)	137	54	4	90,22	4,86
С сигналом enable (новая реализация)	147	55	4	90,18	1,85

Получившийся результат демонстрирует то, что модуль с сигналом **enable** потребляет немного больше аппаратных ресурсов по сравнению с исходной реализацией. При этом динамическое энергопотребление уменьшилось почти в три раза.

Дополнительное задание для самостоятельной работы

Спроектируйте модуль с сигналом **enable** в структурном стиле (с настраиваемой степенью). В качестве примера используйте проект **lab_10\src\lab_10_1_pow_5\ 17_syn_pow_n_en_pipe_struct**. Убедитесь в корректности его работы, проведите имплементацию и сравните его характеристики с характеристиками исходной реализации. Ответьте на вопрос: при использовании какого стиля кодирования реализация сигнала **enable** проще?

10.3.5 Синхронизация стадий конвейера с переменными задержками

Промежуточные вычисления в арифметических конвейерах, представленных в предыдущих разделах, имеют фиксированную, предсказуемую задержку. Тем не менее в некоторых конвейерных схемах определенные стадии могут иметь

непредсказуемую задержку. Такая ситуация возможна, например, если конвейер работает с вводом-выводом или с разделяемыми ресурсами. Соответственно, полезно знать приемы эффективной синхронизации продвижения запроса по конвейерной структуре с операциями такого типа.

Одним из возможных решений проблемы синхронизации стадий конвейера с переменными задержками является приведенная ниже схема синхронизирующего буфера для конвейерных схем (**pipeline_stomach**, листинг 10.10). Помимо обязательных сигналов синхронизации и инвертированного сброса (**clock** и **reset_n** соответственно), модуль имеет входной и выходной интерфейсы (порты **up_*** и **down_***) с сигналами взаимной синхронизации источника и приемника данных (так называемые «**рукопожатия**», **handshacking**). Запись в буфер осуществляется установкой данных на шине **up_data** и переводом в активное состояние сигнала **up_valid**. В некотором смысле сигнал **up_valid** является аналогом сигнала **enable** из предыдущего раздела. Активный уровень сигнала **up_ready** означает отсутствие в данный момент данных в буфере и его готовность принять новую порцию данных. Запись производится в момент перевода **up_ready** в активное состояние. На другой стороне буфера модуль устанавливает выходные данные на шине **down_data**, а сигнал **down_valid** стробирует эти данные. Когда внешняя логика устанавливает в активное состояние сигнал down_ready, данные стираются из буфера.

```
module pipeline_stomach
# (
   parameter width = 8
)
(
   input clock,
   input reset_n,

   input [width - 1:0] up_data,
   input up_valid,
   output up_ready,

   output [width - 1:0] down_data,
   output down_valid,
   input down_ready
);
   wire stomach_load, dataout_load, dataout_unload;
   reg mux_select;

   wire [width - 1:0] data_in;
   reg [width - 1:0] stomach_out, mux_out, data_out;

   reg data_in_dataout, data_in_stomach;

   assign data_in = up_data;
   assign down_valid = data_in_dataout;
   assign dataout_unload = down_valid & down_ready;
```

```
   assign dataout_load =
      (up_valid & (! data_in_dataout | dataout_unload)) |
      (data_in_stomach & dataout_unload);
   assign stomach_load =
      up_valid
      & ! data_in_stomach
      & (data_in_dataout & ! dataout_unload);
   assign up_ready = ! data_in_stomach;

   always @ (posedge clock or negedge reset_n)
      if (! reset_n)
      begin
         data_in_stomach <= 1'b0;
         data_in_dataout <= 1'b0;
      end
      else
      begin
         data_in_stomach <= stomach_load | (data_in_stomach & !
            dataout_unload);
         data_in_dataout <= dataout_load | data_in_stomach |
            (data_in_dataout & ! dataout_unload);
      end

   always @ (posedge clock or negedge reset_n)
      if (! reset_n)
         mux_select <= 1'b0;
      else
         mux_select <= stomach_load | (mux_select & ! dataout_load);

   always @ (posedge clock or negedge reset_n)
      if (! reset_n)
         stomach_out <= { width { 1'b0 } };
      else if (stomach_load)
         stomach_out <= data_in;

   always @*
      mux_out = mux_select ? stomach_out : data_in;

   always @ (posedge clock or negedge reset_n)
      if (! reset_n)
         data_out <= { width { 1'b0 } };
      else if (dataout_load)
         data_out <= mux_out;

   assign down_data = data_out;
endmodule
```

Листинг 10.10 Реализация модуля pipeline stomach для синхронизации конвейера

Дополнительное задание для самостоятельной работы

Выполните имплементацию двух арифметических конвейеров, у которых одна из стадий является разделяемым ресурсом. Для синхронизации продвижения запросов используйте модуль **pipeline_stomach**. Убедитесь в работоспособности схемы путем ее моделирования.

10.4 Упражнения

10.4.1 Основное задание

Выполните все дополнительные задания данной главы.

10.4.2 Задания для самостоятельной работы

1. Повторите упражнения из разделов 10.2 и 10.3 для алгоритма вычисления квадратного корня числа. Используйте структурный или поведенческий стиль проектирования. Функциональная модель и ее вывод представлены в листингах 10.11 и 10.12.

```
import math
# alg model - python math can be used here
def sqrt_alg(x):
   return math.floor(math.sqrt(x))
# hw model - synthesizable operations only
def sqrt_hw(x):
   m = 0x40000000;
   y = 0;
   while (m != 0): # Do 16 times
      b = y | m;
      y >>= 1;
      if (x >= b):
         x -= b;
         y |= m;
         m >>= 2;
   return y;
   # generating stimulus
   for x in range(0, 100, 10):
      alg_val = sqrt_alg(x)
      hw_val = sqrt_hw(x)
      if (alg_val == hw_val):
         print("Correct! x: ", hex(x).ljust(6), "; y: ",
            hex(hw_val).ljust(6))
      else:
         print("ERROR! x: ", hex(x).ljust(6), "; y(model): ",
            hex(alg_val).ljust(6), "; y(hw): ", hex(hw_val).ljust(6))
```

Листинг 10.11 Функциональная модель для алгоритма вычисления квадратного корня числа

```
Correct! x: 0x0 ; y: 0x0
Correct! x: 0xa ; y: 0x3
Correct! x: 0x14 ; y: 0x4
Correct! x: 0x1e ; y: 0x5
Correct! x: 0x28 ; y: 0x6
Correct! x: 0x32 ; y: 0x7
Correct! x: 0x3c ; y: 0x7
Correct! x: 0x46 ; y: 0x8
Correct! x: 0x50 ; y: 0x8
Correct! x: 0x5a ; y: 0x9
```

Листинг 10.12 Тестовый вывод функциональной модели вычисления квадратного корня числа

2. Повторите упражнения из разделов 10.2 и 10.3 для алгоритма вычисления кубического корня числа. Используйте структурный или поведенческий стиль проектирования. Функциональная модель и ее вывод:

```
import math
import numpy

# alg model - python math can be used here
def cube_alg(x):
   return math.floor(numpy.cbrt(x))

# hw model - synthesizable operations only
def cube_hw(x):
   y = 0;
   for s in range(30, -3, -3):
      y = 2 * y;
      b = (3 * y * (y + 1) + 1) << s;
      if (x >= b):
         x = x - b;
         y = y + 1;
   return y;

# generating stimulus
for x in range(0, 100, 10):
   alg_val = cube_alg(x)
   hw_val = cube_hw(x)
   if (alg_val == hw_val):
      print("Correct! x: ", hex(x).ljust(6), "; y: ",
         hex(hw_val).ljust(6))
   else:
      print("ERROR! x: ", hex(x).ljust(6), "; y(model): ",
         hex(alg_val).ljust(6), "; y(hw): ", hex(hw_val).ljust(6))
```

Листинг 10.13 Функциональная модель для алгоритма вычисления кубического корня числа

```
Correct! x: 0x0 ; y: 0x0
Correct! x: 0xa ; y: 0x2
Correct! x: 0x14 ; y: 0x2
Correct! x: 0x1e ; y: 0x3
Correct! x: 0x28 ; y: 0x3
Correct! x: 0x32 ; y: 0x3
Correct! x: 0x3c ; y: 0x3
Correct! x: 0x46 ; y: 0x4
Correct! x: 0x50 ; y: 0x4
Correct! x: 0x5a ; y: 0x4
```

Листинг 10.14 Тестовый вывод функциональной модели алгоритма вычисления кубического корня числа

3. Повторите упражнения из разделов 10.2 и 10.3 для алгоритма Битонной сортировки для 8 элементов. Используйте структурный или поведенческий стиль проектирования. Функциональная модель и ее вывод:

```
import math
import numpy

# alg model - python math can be used here
def sort_alg(x):
   y = x[:];
   y.sort();
   return y;

# hw model - synthesizable operations only
def sort_asc(x, index0, index1):
   if (x[index0] > x[index1]):
      x[index0], x[index1] = x[index1], x[index0]

def bitonic_sort_8_hw(x):
   y = x[:]

   # step 1
   sort_asc(y, 0, 1)
   sort_asc(y, 3, 2)
   sort_asc(y, 4, 5)
   sort_asc(y, 7, 6)

   # step 2
   sort_asc(y, 0, 2)
   sort_asc(y, 1, 3)
   sort_asc(y, 6, 4)
   sort_asc(y, 7, 5)
   sort_asc(y, 0, 1)
   sort_asc(y, 2, 3)
   sort_asc(y, 5, 4)
   sort_asc(y, 7, 6)
```

```
   # step 3
   sort_asc(y, 0, 4)
   sort_asc(y, 1, 5)
   sort_asc(y, 2, 6)
   sort_asc(y, 3, 7)
   sort_asc(y, 0, 2)
   sort_asc(y, 1, 3)
   sort_asc(y, 4, 6)
   sort_asc(y, 5, 7)
   sort_asc(y, 0, 1)
   sort_asc(y, 2, 3)
   sort_asc(y, 4, 5)
   sort_asc(y, 6, 7)
   return y;

# generating stimulus
   x = [51, 160, 4, 77, 194, 223, 13, 84]
   y_alg = sort_alg(x)
   y_hw = bitonic_sort_8_hw(x)
   for index in range(0, 8):
      if (y_alg[index] == y_hw[index]):
         print("Correct! index: ", hex(index).ljust(6), "; y: ",
            hex(y_hw[index]).ljust(6))
      else:
         print("ERROR! index: ", hex(index).ljust(6), "; y(model): ",
            hex(y_alg[index]).ljust(6), "; y(hw): ",
            hex(y_hw[index]).ljust(6))
```

Листинг 10.15 Функциональная модель для алгоритма Битонной сортировки

```
Correct! index: 0x0 ; y: 0x4
Correct! index: 0x1 ; y: 0xd
Correct! index: 0x2 ; y: 0x33
Correct! index: 0x3 ; y: 0x4d
Correct! index: 0x4 ; y: 0x54
Correct! index: 0x5 ; y: 0xa0
Correct! index: 0x6 ; y: 0xc2
Correct! index: 0x7 ; y: 0xdf
```

Листинг 10.16 Тестовый вывод функциональной модели алгоритма Битонной сортировки

4. Повторите упражнения из разделов 10.2 и 10.3 для алгоритма вычисления синуса числа (первый квадрант) с помощью ряда Тейлора (первые три терма):

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots.$$

Используйте 16-битные переменные с фиксированной запятой и коэффициентом масштабирования **1/256**. Примените структурный или поведенческий стиль проектирования.

Функциональная модель и ее вывод:

```
import matplotlib.pyplot as plt

# hw model - synthesizable operations only
def sin_hw(x):
   div6 = 0x2a
   div120 = 0x02

   term0 = x

   pow2 = (x * x) >> 8
   pow3 = (pow2 * x) >> 8
   term1 = (pow3 * div6) >> 8

   pow5 = (pow2 * pow3) >> 8
   term2 = (pow5 * div120) >> 8
   y = term0 - term1 + term2
   return y;

# generating stimulus
x = []
y = []
for index in range(0, 384, 1):
   x.append(index)
   y.append(sin_hw(index))

plt.plot(x, y)
plt.show()
```

Листинг 10.17 Функциональная модель для алгоритма вычисления синуса числа с использованием ряда Тейлора

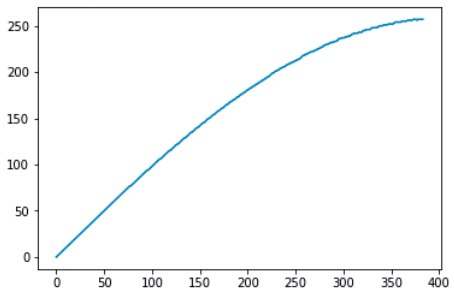


Рис. 10.12 Тестовый вывод функциональной модели вычисления синуса числа с использованием ряда Тейлора

Повторите упражнения из разделов 10.2 и 10.3 для алгоритма вычисления косинуса (первый квадрант) по методу **CORDIC**. Используйте 32-битные переменные

с фиксированной запятой и коэффициентом масштабирования **1/65 536**. Используйте структурный или поведенческий стиль проектирования.

Функциональная модель и ее вывод:

```
import matplotlib.pyplot as plt

# hw model - synthesizable operations only
def shift_right_arith(val, n):
   return val * 2.0**(-n)

def cordic_iteration(angle):
   atan_table = [0xc910, 0x76b2, 0x3eb7, 0x1fd6,
      0x0ffb, 0x07ff, 0x0400, 0x0200,
      0x0100, 0x0080, 0x0040, 0x0020,
      0x0010, 0x0008, 0x0004, 0x0002]

   x = 65536
   y = 0
   z = angle
   for i in range(0, 16):
      if z <= 0:
         d = -1
      else:
         d = 1

      nextx = x - shift_right_arith((y * d), i)
      nexty = y + shift_right_arith((x * d), i)

      x = nextx
      y = nexty
      z = z - (d * atan_table[i])
   return x

# generating stimulus
x = []
y = []
for index in range(0, 98304, 128):
   x.append(index)
   y.append(cordic_iteration(index))

plt.plot(x, y)
plt.show()
```

Листинг 10.18 Функциональная модель для алгоритма вычисления косинуса числа с использованием метода CORDIC

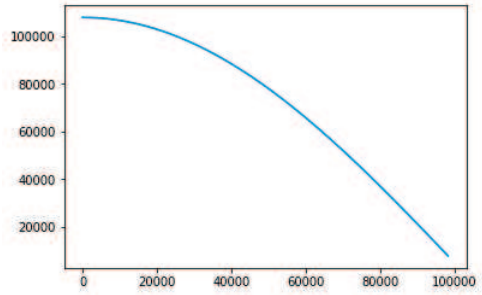


Рис. 10.13 Тестовый вывод функциональной модели вычисления косинуса числа с использованием метода CORDIC

10.4.3 Контрольные вопросы

1. Можно ли за счет конвейеризации неограниченно повышать пропускную способность блоков арифметической обработки, аналогичных описанным в данной главе? Чем могут быть вызваны ограничения?

2. В чем разница между внутренней и внешней фрагментацией в конвейерных схемах? Каким образом эти типы фрагментации влияют на эффективность конвейерной схемы?

3. При каких свойствах вычислительной нагрузки и стратегии оптимизации применение конвейеризованной реализации является предпочтительным по сравнению с комбинационной, и наоборот?

4. При каких свойствах вычислительной нагрузки и стратегии оптимизации применение конвейерной реализации является предпочтительным по сравнению с многотактной, и наоборот?

5. Необходимо спроектировать аппаратный блок. Существует возможность выбрать один из двух алгоритмов. Первый из них состоит из трех последовательно зависимых операций со следующими задержками: **L1 = 20 нс**, **L2 = 15 нс**, **L3 = 15 нс**. Второй алгоритм состоит из двух последовательно зависимых операций со следующими задержками: **L1 = 25 нс**, **L2 = 15 нс**. Задержка на буферизацию близка к нулю (ею можно пренебречь). Какой именно из двух алгоритмов и какую именно реализацию (комбинационную или конвейерную) следует выбрать для аппаратного блока? Какую стратегию оптимизации (по пропускной способности или по задержке) следует выбрать? Какие характеристики (пропускная способность и задержка) будут у этих блоков?

6. Алгоритм состоит из четырех операций, две первые из которых являются независимыми, третья использует результат первой операции, а последняя – результат второй и третьей операций. Задержки выполнения операций составляют: **L1 = 2 нс**, **L2 = 10 нс**, **L3 = 5 нс**, **L4 = 5 нс**. Задержка на буферизацию близка к нулю (ею можно пренебречь). Какую именно реализацию (комбинационную или конвейерную) следует выбрать для аппаратного блока? Какую стратегию оптимизации (по пропускной способности или по задержке)

следует выбрать? Какие характеристики (пропускная способность и задержка) будут у этих блоков?

7. Алгоритм состоит из восьми одинаковых последовательно зависимых операций. Кроме того, результат третьей операции используется в первой операции у следующего запроса. Какова будет структура конвейера, оптимизированного по пропускной способности?

8. В каких случаях для реализации регулярных схем можно использовать и циклы, и конструкцию **generate**? Когда можно использовать только конструкцию **generate**?

9. В каких случаях введение сигнала **enable** в конвейерную схему может отрицательно сказаться на энергопотреблении?

10. Разработчик аппаратного блока получил критическую задержку в **20 нс** на одной из стадий конвейера (на каждой стадии запрос обрабатывается один такт). После некоторых размышлений он сделал эту стадию многотактной, что снизило критическую задержку в конвейере до **10 нс. 20 %** запросов на этой стадии стали занимать по четыре такта, а **80 %** запросов занимают один такт. Насколько изменилась максимальная тактовая частота конвейера и его средняя пропускная способность?

Станислав Жельнио, Александр Романов

Цифровой синтез: практический курс

Глава 11. Софт-процессор: основы микроархитектуры

Содержание

Глава посвящена пошаговому проектированию простейшего микропроцессора с последующим запуском на нем нескольких несложных программ, написанных на ассемблере. Для этого используется ***schoolMIPS*** *– ядро, представляющее собой упрощенный вариант процессора* ***Сары Харрис***[1]*. Идея разработки* ***schoolMIPS*** *принадлежит Юрию Панчулу, которому в 2017 году понадобился процессор, реализованный на* ***Verilog*** *и подходящий для обучения старшеклассников основам цифровой схемотехники и архитектуры компьютера. С момента своего создания* ***schoolMIPS*** *эволюционировал от самой простой версии (на которой построена данная практическая работа) до конвейерного процессора с блоком обработки прерываний и поддержкой шины* ***AHB-Lite****.*

Требования к аппаратному и программному обеспечению

Для выполнения работы необходимы:

- *рабочая станция с ОС* ***Windows***[2]*, x64, 8Гб ОЗУ, порт USB;*
- *ПО* ***Quartus Prime Lite Edition*** *17.0;*
- *ПО* ***Icarus Verilog*** *и* ***GTKWave****;*
- *ПО* ***MARS*** *(****MIPS Assembler and Runtime Simulator****);*
- *ПО* ***Visual Studio Code*** *с плагинами* ***eirikpre.systemverilog, kdarkhan.mips****;*
- *отладочная плата* ***Terasic DE10-Lite****.*

Документация на отладочную плату расположена среди дополнительных материалов[3] *к данной главе в подкаталоге doc. Инструкции к данной работе могут быть адаптированы для практически любой отладочной платы на основе* ***ПЛИС Intel FPGA / Altera****.*

Исходные коды процессора ***schoolMIPS*** *расположены в каталоге с файлами данной работы. Последняя версия исходных кодов* ***schoolMIPS*** *также доступна на сайте проекта*[4]*. В данной главе рассматривается простейшая версия процессора, опубликованная в* ***git****-ветке* ***00_simple***[5]*.*

Работа с процессором ***schoolMIPS*** *представляет собой один из примеров «смешанной» разработки – когда одновременно используются различные языки программирования (****Verilog, C, assembler****). Для упрощения работы предполагается использование универсального текстового редактора с открытым исходным кодом* ***Visual Studio Code***[6]*.*

---

1 *Харрис Д. М., Харрис С. Л.* Цифровая схемотехника и архитектура компьютера. 2-е изд. 2013.

2 Использование виртуальной машины не рекомендуется: возможны проблемы с конфигурированием ПЛИС.

3 https://github.com/RomeoMe5/DDLM.

4 https://github.com/MIPSfpga/schoolMIPS.

5 https://github.com/MIPSfpga/schoolMIPS/tree/00_simple.

6 В качестве альтернативы данному редактору можно упомянуть **Sublime** или **Notepad++**. Функциональность универсальных редакторов легко адаптируется под конкретные задачи с помощью дополнительных модулей (например, для подсветки синтаксиса **Verilog** или **MIPS assembler**), а встроенный терминал позволяет не тратить время на переключение между окнами.

11.1 Теоретические основы процессорного ядра schoolMIPS

11.1.1 Микроархитектура

Под **архитектурой** вычислительной системы (**ISA**, **Instruction Set Architecture**) понимается совокупность системы команд и доступных разработчику регистров внутри процессора. Наличие стандартизованной архитектуры позволяет писать программы, которые с одинаковыми результатами выполняются на процессорах разных производителей, а также обеспечивает обратную совместимость с более старым оборудованием.

Микроархитектура – это особенности реализации конкретного процессора, то, как он построен внутри и как взаимодействуют его блоки. Современные процессоры **Intel** и **AMD** относятся к архитектуре **x86-64**, что позволяет выполнять на них одни и те же программы без перекомпиляции, однако отличаются по своему внутреннему строению (микроархитектуре).

В процессоре **schoolMIPS** реализовано подмножество архитектуры **MIPS**. Краткое описание инструкций и регистров, предусмотренных данной архитектурой, приведено в документе **MIPS32 Instruction Set Quick Reference**[1]. Детальное описание инструкций приводится в **MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**[2]. Оба файла доступны в подкаталоге doc материалов данной работы.

С формальной точки зрения любая микропроцессорная система относится к **конечным автоматам**. У нее есть элементы, которые определяют текущее состояние автомата (рис. 11.1): **счетчик команд** (**PC**, **Program Counter**), **память инструкций** (**Instruction Memory**), **регистры общего назначения** (**Register File**) и **память данных** (**Data Memory**).

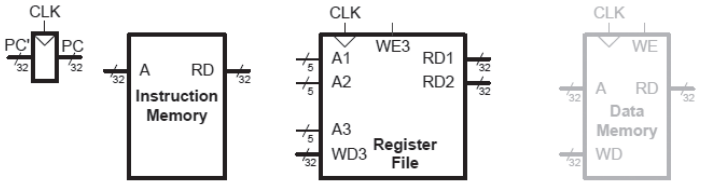


Рис. 11.1 Элементы микропроцессорной системы

Счетчик команд – это регистр, который содержит адрес выполняемой процессором в текущий момент команды в **памяти инструкций**. В памяти инструкций хранится программа, представляющая собой последовательность инструкций, выполняемых процессором.

Регистры общего назначения хранят обрабатываемые в текущий момент данные, а также промежуточные результаты вычислений. 32 таких регистра (на 32 би-

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set. 2013.
[2] MIPS32 ® Instruction Set Quick Reference. 2008.

та каждый) объединены в блок с двумя портами чтения и одним портом записи (регистровый файл). Для чтения номер (адрес) регистра подается на порт **A1** или **A2**, после чего значение, хранящееся в регистре, можно считать с порта **RD1** или **RD2** соответственно. Для записи необходимо выставить в **1** флаг **WE3** (разрешение записи), подать записываемые данные на порт **WD3**, а номер (адрес) регистра для записи – на порт **A3**.

Память данных используется для хранения всех обрабатываемых данных и отсутствует в простейшей реализации процессора **schoolMIPS**. По этой причине на рис. 11.1 она изображена серым цветом.

Перечисленные блоки являются единственными **последовательностными** схемами в процессоре; все остальные блоки, которые мы будем рассматривать, относятся к комбинационной логике, определяющей следующее состояние конечного автомата под названием «процессор».

В следующих разделах рассматриваются пошаговое проектирование и реализация процессора: **тракта данных** (**datapath**), который представляет собой набор цифровых блоков внутри процессора, непосредственно задействованный в обработке пользовательских данных, и **устройства управления** (**Control Unit**), обеспечивающего выдачу управляющих сигналов на блоки тракта данных.

11.1.2 Проектирование. Тракт данных

При проектировании процессора мы будем придерживаться следующего подхода: выборочно рассмотрим несколько инструкций (команд) архитектуры **MIPS** и добавим к элементам состояния процессора такие блоки и соединения между ними, которые необходимы для реализации этих команд. Полное описание всех инструкций архитектуры **MIPS** доступно в **MIPS Architecture For Programmers. Volume II-A: The MIPS32 Instruction Set**.

Инструкция ADDIU

Первой из рассматриваемых инструкций будет **ADDIU** – целочисленное беззнаковое сложение операнда с константой. Без этой команды не может обойтись практически никакая программа. Ниже представлен формат данной команды (рис. 11.2).

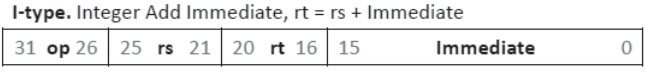


Рис. 11.2 Инструкция ADDIU

После выполнения данной инструкции в регистр, номер которого указан в поле rt, должен быть сохранен результат сложения константы **Immediate** и данных из регистра под номером **rs**. То, что операция сложения называется беззнаковой, означает, что возможная ситуация переполнения будет проигнорирована. Поле op содержит код команды (константа), который позволяет идентифицировать ее именно как **ADDIU**.

Данная инструкция относится к группе команд **I-типа** (**I-type instruction**), что означает, что один из операндов операции содержится непосредственно в коде инструкции (поле **Immediate**), а второй находится в регистре общего назначения.

Задание: откройте документ **MIPS Architecture For Programmers. Volume II-A: The MIPS32 Instruction Set** и найдите в нем описание инструкции **ADDIU**. Сравните ее с описанием инструкции **ADDI**.

Процессор **schoolMIPS** является однотактным. Это означает, что каждая из обрабатываемых инструкций выполняется не дольше **одного** периода тактового сигнала (при этом частота тактового сигнала должна быть достаточно низкой, чтобы успела выполниться самая медленная инструкция). Для упрощения процесса проектирования рассмотрим каждую из инструкций как набор шагов, последовательно выполняющихся на одном такте работы процессора.

Шаг № 1. Выборка команды из памяти инструкций. Адрес выполняемой команды хранится в счетчике команд, поэтому необходимо соединить выход регистра-счетчика команд со входом памяти инструкций (рис. 11.3). После этого шага команда **ADDIU** появится на выходе памяти инструкций.

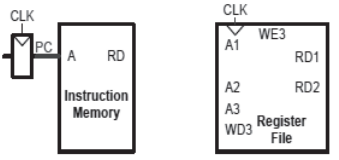


Рис. 11.3 ADDIU. Шаг 1: выборка команды из памяти инструкций

Шаг № 2. Получение операнда ADDIU из блока регистров общего назначения. Для этого номер регистра, который хранится в поле **rs** (биты **25:21** инструкции), необходимо подать на вход одного из портов разрешения чтения (**A1**), после чего эти данные станут доступны на выходе **RD1** (рис. 11.4).

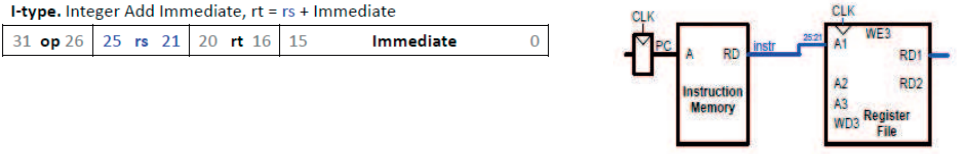


Рис. 11.4 ADDIU. Шаг 2: чтение операнда из регистрового файла

Шаг № 3. Расширение знака. Так как обрабатываемые данные в регистре имеют разрядность 32 бита, а константа, с которой они складываются, – только 16 бит (поле **Immediate**, биты инструкции 15:0), необходимо увеличить разрядность константы до 32 бит и правильно учесть знак этой константы, выполнив расширение знака. Константа хранится в дополнительном коде (например, десятичное число –1 хранится в виде двоичного кода 16'hFF). Поэтому для увеличения разрядности константы до 32 бит достаточно в биты **31:16** поместить значение знака этой константы – 15-й бит (рис. 11.5).

Рис. 11.5 ADDIU. Шаг 3: расширение знака для поля Immediate

Шаг № 4. Выполнение арифметической операции. За арифметические операции в процессоре отвечает **арифметико-логическое устройство** (**АЛУ**, **ALU**)[1]. Реализация АЛУ подробно описана в главе 5. Так как **АЛУ** может выполнять различные операции (сложение, умножение, деление и т. д.), то на его вход, помимо операндов, необходимо подать код выполняемой в данный момент операции. Этот код (сигнал **aluControl**, рис. 11.7) выставляется устройством управления на основании кода выполняемой операции (биты **31:26** кода инструкции, рис. 11.6).

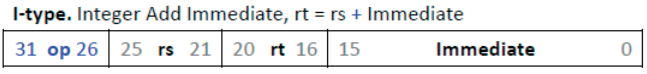


Рис. 11.6 Код операции инструкции ADDIU

После того как будет вычислен результат арифметической операции (в данном случае – сложения), он появится на выходе **АЛУ** (сигнал **result**, рис. 11.7).

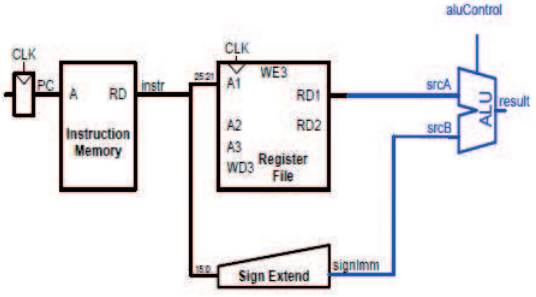


Рис. 11.7 ADDIU. Шаг 4: вычисление результата арифметической операции

Шаг № 5. Определение регистра, в который будут записаны результаты выполнения операции, и запись результатов в блок регистров общего назначения. Номер регистра, в который необходимо записать результат сложения, задается полем **rt** инструкции (биты **20:16**, рис. 11.8).

[1] Для выполнения операций с плавающей точкой в современных процессорах предусмотрен специальный блок операций с плавающей точкой; в процессоре **schoolMIPS** он отсутствует и в данной главе не рассматривается.

I-type. Integer Add Immediate, rt = rs + Immediate

31 **op** 26	25 **rs** 21	20 **rt** 16	15 **Immediate** 0

Рис. 11.8 Регистр назначения инструкции ADDIU

Для того чтобы выполнить сохранение результата в этот регистр, его номер устанавливается на входе порта **A3** блока регистров общего назначения. Сохраняемое значение (сигнал **result**) – на порт **WD3**; при этом устройство управления обеспечивает подачу сигнала **WE** на вход регистрового файла (рис. 11.9). Непосредственная запись данных в регистр производится по фронту тактового сигнала **CLK**.

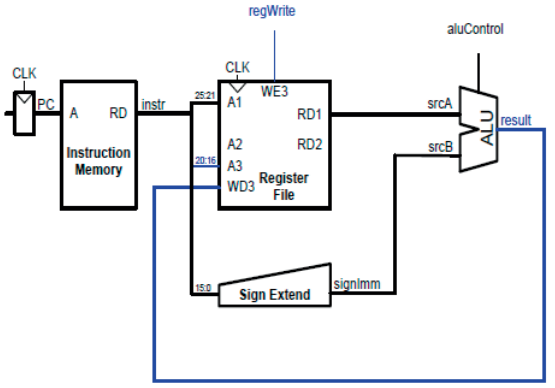


Рис. 11.9 ADDIU. Шаг 5: запись результата в регистровый файл

Шаг № 6. Переход к следующей команде. Для этого необходимо увеличить значение счетчика команд на единицу[1] (рис. 11.10). Запись нового значения адреса инструкции в счетчик команд (**PC**) также выполняется по переднему фронту тактового сигнала.

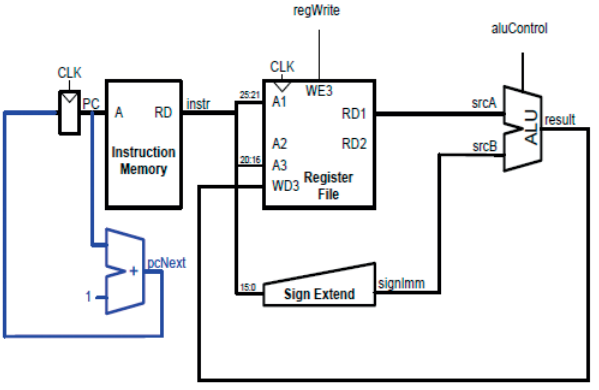


Рис. 11.10 ADDIU. Шаг 6: определение адреса следующей инструкции

[1] В данной версии процессора **schoolMIPS** память инструкций адресуется не байтами, а 32-битными словами. При адресации байтами нам бы пришлось при переходе к следующей инструкции прибавлять 4.

Таким образом, в результате выполнения всех описанных действий проектирование тракта данных, достаточного для выполнения инструкции **ADDIU**, завершено. Еще раз подчеркнем, что все шаги прохождения данных, рассмотренные выше, выполняются в течение одного такта **CLK**.

Инструкция ADDU

Спроектированный в предыдущем разделе тракт данных позволяет выполнять одну из инструкций **I-типа**: **ADDIU**. Вместе с тем только одной инструкции недостаточно для написания даже простейшей программы на ассемблере, выполняющей что-то полезное. По этой причине мы расширим набор команд инструкцией **ADDU**, которая позволяет выполнять целочисленное беззнаковое сложение операндов. Данная команда уже относится к группе инструкций **R-типа**. В отличие от рассмотренного ранее **I-типа**, где один из операндов является константой, команды **R-типа** позволяют выполнять действия с операндами, хранящимися в регистрах общего назначения.

Формат инструкции **ADDU** показан на рис. 11.11. Действие, которое выполняется над операндами (в данном случае это сложение), определяется полями **op** (биты **31:26**) и **funct** (биты **5:0**). Исходные данные для арифметической операции поступают из регистров, номера которых заданы в полях **rs** (биты **25:21**) и **rt** (биты **20:16**). Результат выполнения операции записывается в регистр, определяемый полем **rd** (биты **15:11**).

R-type. Integer Add Unsigned, **rd = rs + rt**

31 **op** 26	25 **rs** 21	20 **rt** 16	15 **rd** 11	10 **sa** 6	5 **funct** 0

Рис. 11.11 Инструкция ADDU

Задание: откройте документ **MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set** и найдите в нем описание инструкции **ADDU**.

Шаги выборки инструкции и получения первого операнда совпадают с такими же шагами для инструкции **ADDIU**. Шаг получения операнда, определяемого полем **rt**, приведен на рис. 11.12: номер регистра подается на порт **A2**, а хранящиеся в нем данные считываются с порта **RD2**.

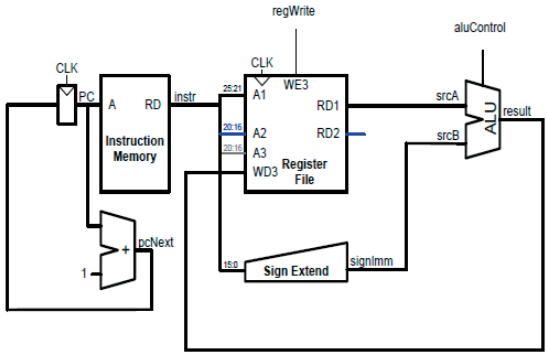


Рис. 11.12 Инструкция ADDU. Чтение второго операнда из регистрового файла

Для выполнения операции необходимо, чтобы данные с порта **RD2** блока регистров попали на вход **srcB АЛУ**. Но для реализации **ADDIU** этот вход был подключен к выходу блока расширения знака. Для того чтобы обеспечить работу с обеими инструкциями (как **ADDU**, так и **ADDIU**), в тракт данных добавлен мультиплексор, позволяющий переключать сигнал на входе **srcB** в **АЛУ** (рис. 11.13). Устройство управления обеспечивает подачу корректного сигнала **aluSrc** на вход данного мультиплексора в зависимости от выполняемой в данный момент инструкции.

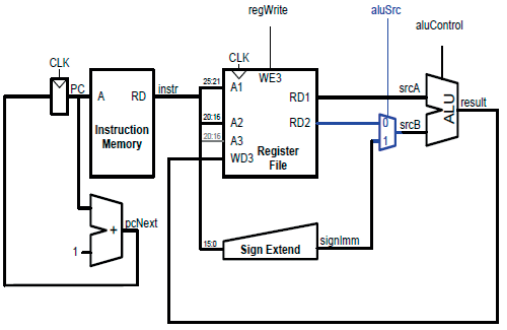


Рис. 11.13 Инструкция ADDU. Передача второго операнда в АЛУ

Следующим шагом, на котором проявляется отличие между инструкциями **ADDIU** и **ADDU**, является запись результата выполняемой операции. В данном случае она производится в регистр, определенный полем **rd** (биты **15:11**), а не **rt**, как в предыдущем случае (рис. 11.14).

R-type. Integer Add Unsigned, **rd = rs + rt**

31 **op** 26	25 **rs** 21	20 **rt** 16	**15 rd 11**	10 **sa** 6	5 **funct** 0

Рис. 11.14 Регистр назначения инструкции ADDU

По этой причине в тракт данных добавлен еще один мультиплексор на адресном входе A3 регистрового файла (рис. 11.15). Сигнал **regDst** также задается устройством управления в зависимости от текущей инструкции. Это изменение тракта данных является последним из необходимых для реализации инструкции **ADDU**.

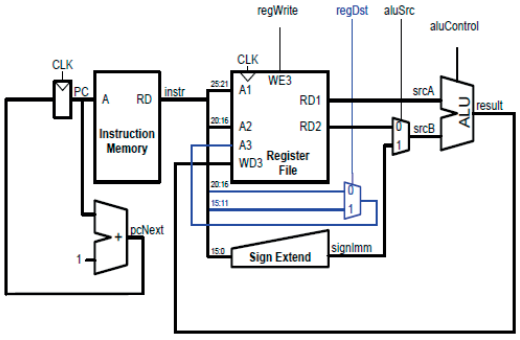


Рис. 11.15 Инструкция ADDU. Запись результата в регистр назначения (rd)

Инструкция SRL

Инструкция **SRL** реализует логический сдвиг вправо (рис. 11.16). Данная инструкция выполняется над значением, которое хранится в регистре с номером **rt** (биты **20:16**). Величина сдвига (количество разрядов, на которое необходимо сдвинуть исходное число) хранится в поле **sa** (биты **10:6**). При этом итоговое значение сохраняется в регистр, определяемый полем **rd** (биты **15:11**).

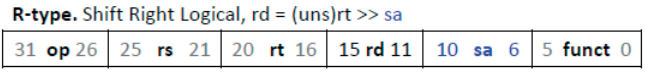


Рис. 11.16 Инструкция SRL

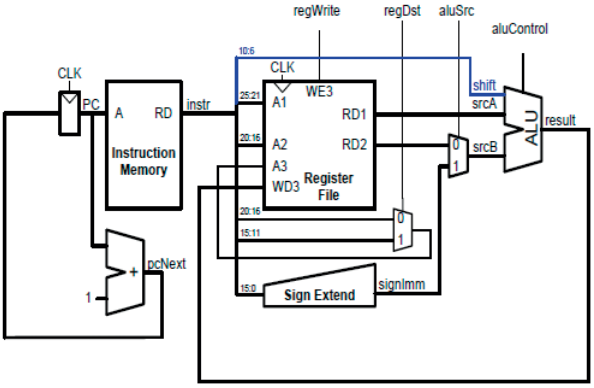


Рис. 11.17 Инструкция SRL. Значение величины сдвига

Для корректного выполнения данной инструкции необходимо передавать в **АЛУ** величину сдвига (рис. 11.17). В остальном тракт данных ничем не отличается от спроектированного в предыдущих разделах.

Инструкция BEQ

Сложно представить алгоритм, не требующий операций условного перехода, которые обеспечивают работу циклов и условных операторов. Одной из инструкций условного перехода является **BEQ** – **branch-on-equal** (рис. 11.18). В случае если значение в регистре с номером rs совпадает со значением в регистре с номером **rt**, то, помимо простого увеличения счетчика команд на единицу, к его новому значению добавляется значение смещения (**offset**) из поля **Immediate** (биты **15:0**) инструкции[1]. Константа интерпретируется как знаковое целое в дополнительном коде, что позволяет выполнять условные переходы как «вперед» по адресам программы, так и «назад».

[1] Следует отметить, что в данном софт-процессоре для упрощения его реализации адресация команд выполнена пословной. То есть счетчик команд (**PC**) считает не байты, а 32-битные слова (4 байта – столько занимает одна команда в памяти). В классической **MIPS**-архитектуре адресация побайтная, поэтому для обычных команд инкремент команд **I**- и **R**-типов осуществляется на **4**, а для команд ветвления – на **(offset << 2) +4**.

I-type. Branch On Equal, if (Rs == Rt) PC += (int)offset

31 **op** 26	25 **rs** 21	20 **rt** 16	15 **Immediate** 0

Рис. 11.18 Инструкция BEQ

Для обработки данной инструкции в **тракт данных** добавлен сумматор, вычисляющий новое значение счетчика команд (сигнал **pcBranch**, рис. 11.19), а также мультиплексор, управляемый сигналом **pcSrc**, который определяет, каким будет следующее значение счетчика команд. Если **pcSrc** равен **0**, то после фронта тактового сигнала **CLK** в счетчик команд будет записан адрес **pcNext**. Если **pcSrc** равен **1**, то в счетчик команд будет записано значение **pcBranch**, т. е. будет выполнен переход по новому адресу или, иными словами, операция ветвления.

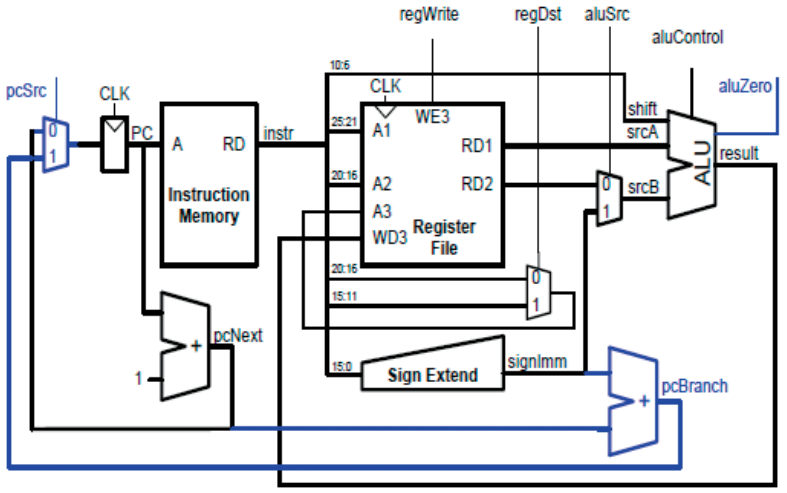


Рис. 11.19 Инструкция BEQ. Расчет адреса следующей инструкции в случае перехода

Необходимость выполнения перехода зависит от результата сравнения операндов инструкции **BEQ**. Это сравнение выполняется в арифметико-логическом устройстве путем вычитания операндов. Если результатом вычитания является **0** (на выходе **АЛУ** сигнал **aluZero = 0**), то операнды равны, а значит, в случае **BEQ** необходимо выполнить условный переход. Окончательное решение о необходимости выполнения перехода принимает устройство управления в зависимости от результатов декодирования команды.

Итоговая цепочка выполняемых сравнений выглядит следующим образом:

- равны ли операнды, поступившие на вход **АЛУ** (сигнал **aluResult**)?
- является ли инструкция командой ветвления (сигнал **branch**)?
- должен ли быть переход, если операнды равны (сигнал **condZero**)?

Схема процессора, полученная в результате добавления инструкции ветвления, представлена на рис. 11.20.

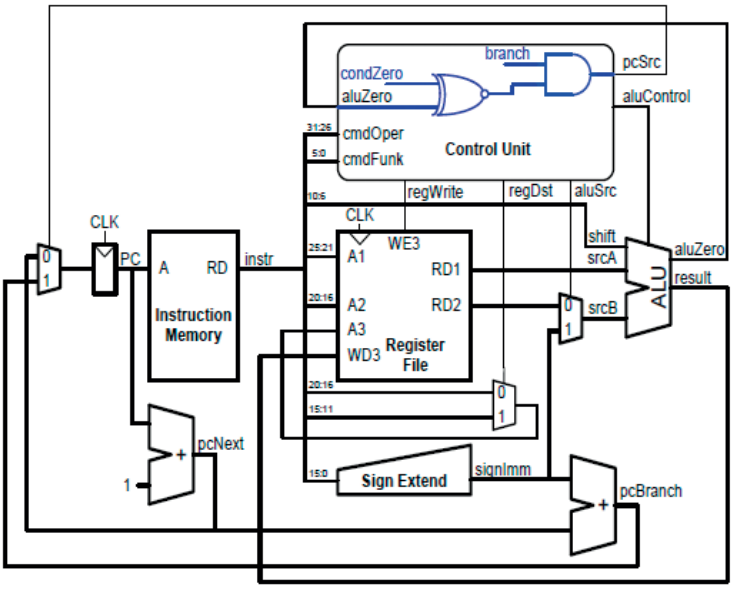


Рис. 11.20 BEQ. Определение необходимости перехода (сигнал aluZero)

Итоговая блок-схема процессора **schoolMIPS** (в его простейшем варианте) представлена на рис. 11.21. В следующем разделе рассмотрен процесс проектирования устройства управления софт-процессором.

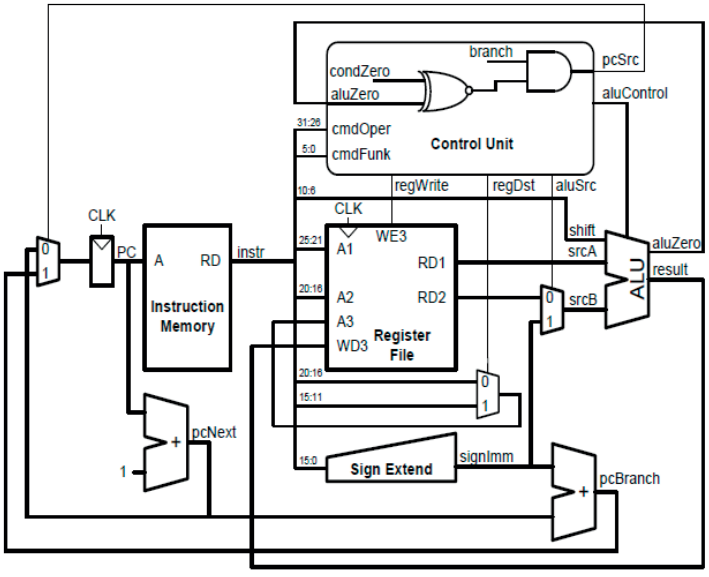


Рис. 11.21 Итоговая схема микроархитектуры schoolMIPS

11.1.3 Проектирование. Устройство управления

Задача устройства управления – подавать управляющие сигналы на блоки тракта данных в зависимости от результатов декодирования текущей инструкции. Процесс проектирования устройства управления заключается в формализации перечня и значений этих сигналов. Для простоты сведем их в таблицу (табл. 11.1), где каждой строке соответствует одна выполняемая инструкция (**Instr**), а в столбцах приведены значения полей, необходимые для ее декодирования (**cmdOper**, **cmdFunc**), и значения управляющих сигналов (**branch**, **condZero** и т. д.).

Таблица 11.1 Управляющие сигналы schoolMIPS

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl

Инструкция ADDIU

Заполним первую строку таблицы для команды **ADDIU** (табл. 11.2). Код операции (**cmdOper**) для данной инструкции: **6'b001001**. Поле cmdFunc для данной инструкции значения не имеет (это инструкция **I**-типа, и данное поле в ней отсутствует). Эта операция не является командой ветвления, поэтому управляющий сигнал **branch = 0**, что приводит к тому, что и сигнал **pcSrc = 0** (табл. 11.2), при этом сигнал **condZero** может иметь любое значение.

Таблица 11.2 Управляющие сигналы. Инструкция ADDIU

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0				000

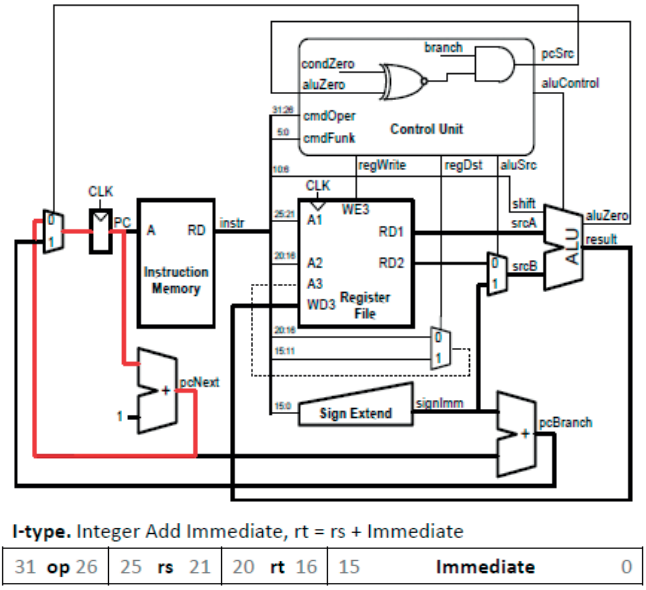
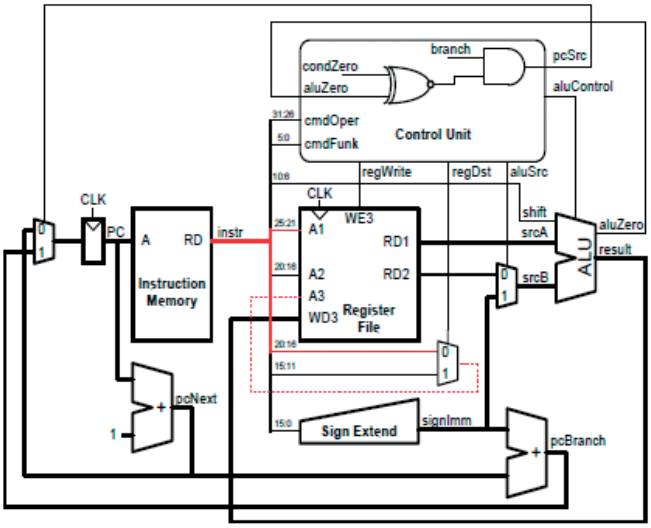


Рис. 11.22 Управляющие сигналы инструкции ADDIU (branch = 0)

Так как результат выполнения инструкции **ADDIU** сохраняется в регистр, определяемый полем **rt** (биты **20:16**), то **regDst = 0**.

Таблица 11.3 Управляющие сигналы. Инструкция ADDIU

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0	0			000



I-type. Integer Add Immediate, rt = rs + Immediate

31 **op** 26	25 **rs** 21	20 **rt** 16	15 **Immediate** 0

Рис. 11.23 Управляющие сигналы инструкции ADDIU (regDst = 0)

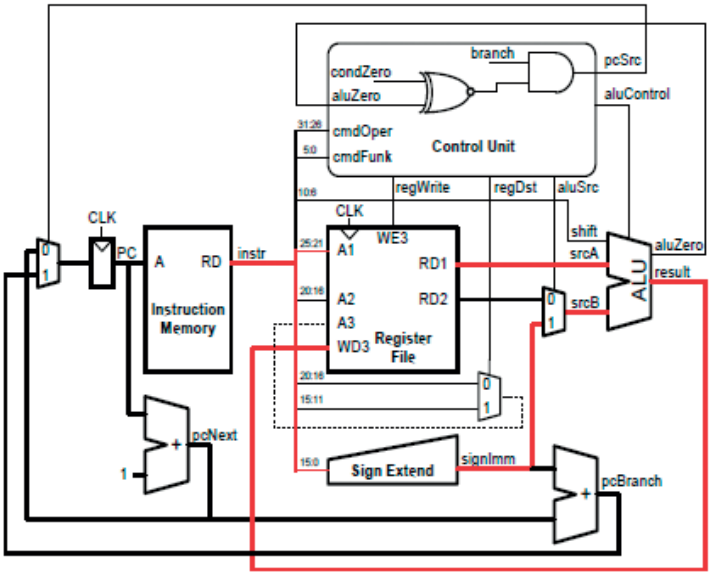
При выполнении инструкции **ADDIU** производится сохранение результата в один из регистров общего назначения, что означает запись в регистровый файл. Для этого сигнал **regWrite** должен быть установлен в **1**.

Одним из аргументов команды **ADDIU** является константа **Immediate** (биты **15:0**), над которой выполняется операция знакового расширения. Для того чтобы эти данные поступили на вход **АЛУ**, сигнал **aluSrc** установлен в **1**.

На этом формирование списка управляющих сигналов для команды **ADDIU** можно считать завершенным.

Таблица 11.4 Управляющие сигналы. Инструкция ADDIU

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0	0	1	1	000



I-type. Integer Add Immediate, rt = rs + Immediate

31 **op** 26	25 **rs** 21	20 **rt** 16	15 **Immediate** 0

Рис. 11.24 Управляющие сигналы инструкции ADDIU (сигналы regWrite, aluSrc)

Инструкция ADDU

Рассмотрим сигналы блока управления, необходимые для работы команды **ADDU** (табл. 11.5). Данная команда является инструкцией **R**-типа. По этой причине для ее идентификации необходимо учитывать не только значение поля **cmdOper** (**6b'000000**), но и **cmdFunc** (**6b'100001**). Поскольку, как и в предыдущем случае, команда **ADDU** не является операцией ветвления, сигнал **branch = 0**, а **condZero** может иметь любое значение.

В отличие от предыдущего случая, сигнал **regDst** установлен в **1**, так как для команды **ADDU** сохранение результата производится в регистр с номером **rd** (биты **15:11**).

Таблица 11.5 Управляющие сигналы. Инструкция ADDU

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0	0	1	1	000
addu	000000	100001	0	0	1			000

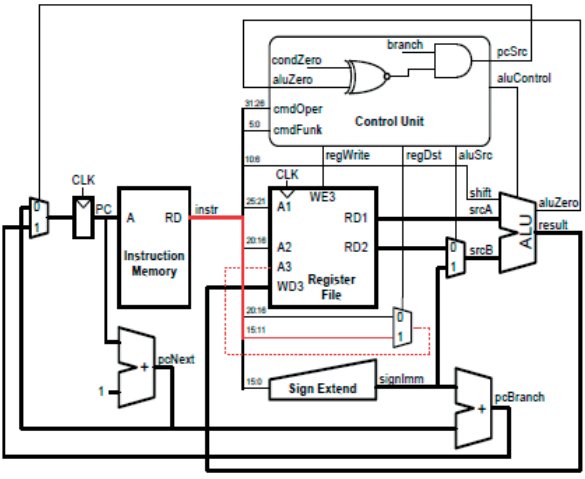


Рис. 11.25 Управляющие сигналы инструкции ADDU (regDst)

Результат выполнения команды **ADDU** сохраняется в регистровый файл, поэтому **regWrite = 1**. Оба операнда данной инструкции поступают из регистрового файла, поэтому **aluSrc** должен быть установлен в **0**. На этом формирование управляющих сигналов для **ADDU** можно считать завершенным (табл. 11.6).

Таблица 11.6 Управляющие сигналы. Инструкция ADDU

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0	0	1	1	000
addu	000000	100001	0	0	1	1	0	000

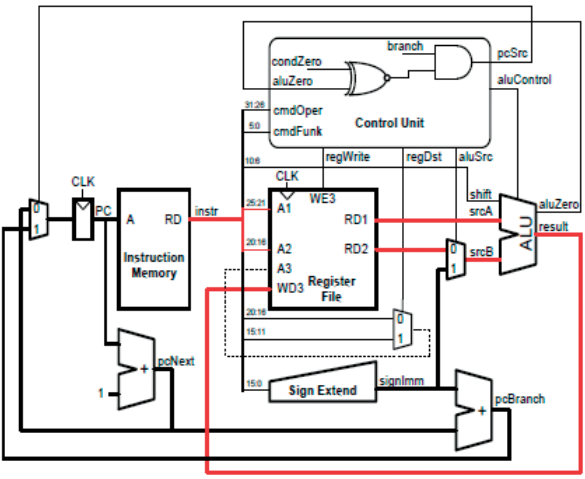


Рис. 11.26 Управляющие сигналы инструкции ADDU (regWrite, aluSrc)

Инструкция SRL

Данная команда по части управляющих сигналов отличается от рассмотренной выше команды **ADDU** только сигналом **aluControl = 3'b011**. Под действием этого сигнала **АЛУ** вместо ранее выполняемой операции сложения выполняет операцию сдвига (табл. 11.7).

Таблица 11.7 Управляющие сигналы. Инструкция SRL

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0	0	1	1	000
addu	000000	100001	0	0	1	1	0	000
srl	000000	000010	0	0	1	1	0	011

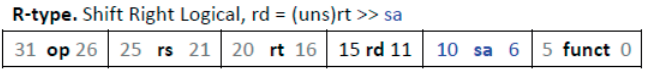


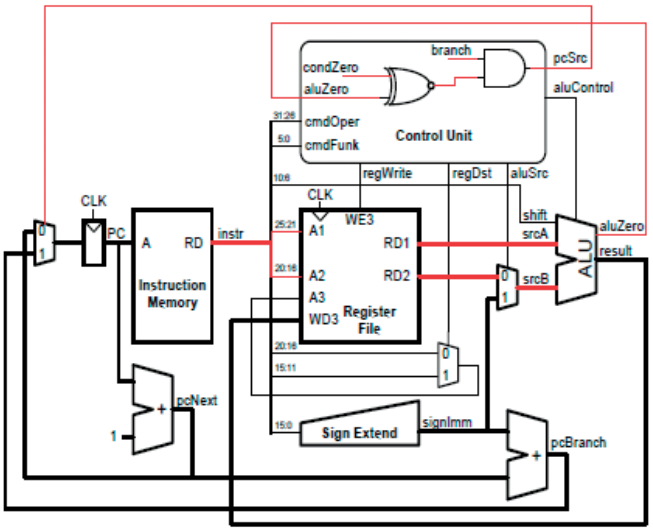
Рис. 11.27 Инструкция SRL

Инструкция BEQ

Операция ветвления **BEQ** выполняет переход по заданному смещению в случае, если операнды операции равны (регистры с номерами **rt** и **rd** содержат одинаковые значения). Так как оба операнда – регистры, то **aluSrc = 0**. Под сравнением двух чисел понимается операция вычитания (**aluControl = 3'b101**): если два числа равны, то на выходе **АЛУ** устанавливается значение **0**, и сигнал **aluZero** принимает значение **1**. Сигнал **branch** устанавливается в **1** только для команд ветвления (таких как **BEQ**). Сигнал **condZero** при этом устанавливает, при каком значении **aluZero** условный переход будет совершен. Для команды **BEQ** сигнал **branch = 1** и сигнал **condZero = 1** (табл. 11.8).

Таблица 11.8 Управляющие сигналы. Инструкция BEQ

Instr	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
addiu	001001	??????	0	0	0	1	1	000
addu	000000	100001	0	0	1	1	0	000
srl	000000	000010	0	0	1	1	0	011
beq	000100	??????	1	1	0	0	0	101



I-type. Branch On Equal, if (Rs == Rt) PC += (int)offset

31 **op** 26	25 **rs** 21	20 **rt** 16	15 **Immediate** 0

Рис. 11.28 Инструкция BEQ. Определение необходимости перехода

При условии что **aluZero = 1**, следующим значением счетчика команд станет **pcBranch** (рис. 11.29). В противном случае, если **aluZero = 0**, счетчик команд будет равен **pcNext** (рис. 11.30).

На этом проектирование устройства управления можно считать завершенным.

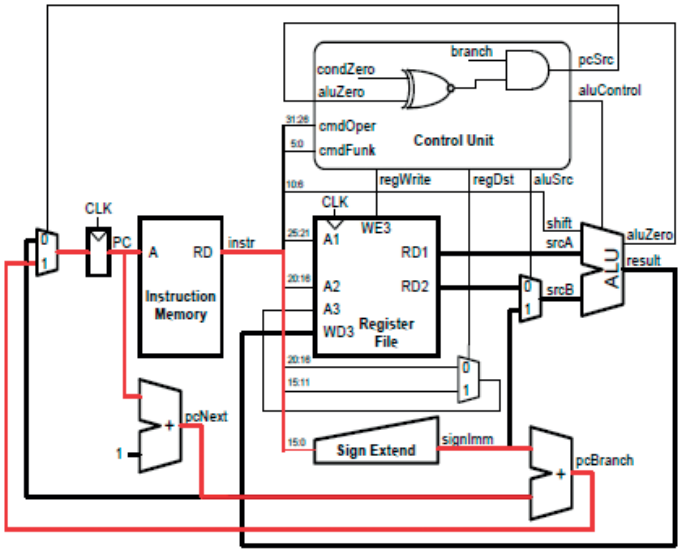


Рис. 11.29 Инструкция BEQ. Выполнение перехода

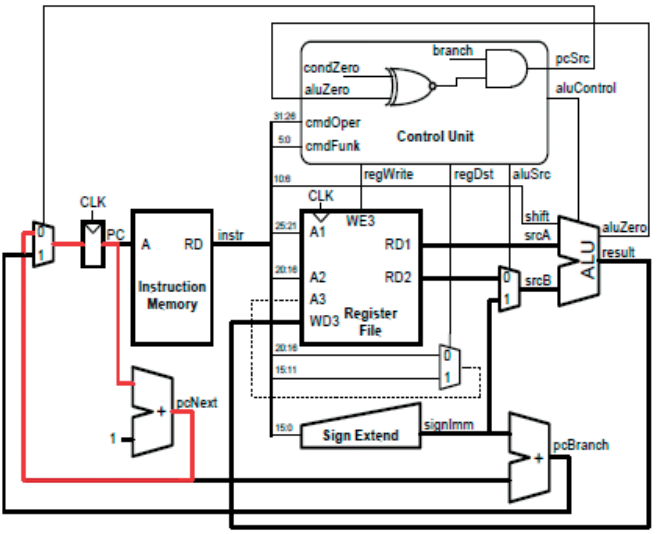


Рис. 11.30 Инструкция BEQ. Отсутствие перехода

11.1.4 Реализация. Тракт данных

В данном разделе рассмотрена реализация спроектированного в предыдущих главах процессорного ядра **schoolMIPS** на языке **Verilog**.

Задание: откройте каталог с исходными кодами практической работы[1] или сайт проекта на ресурсе **GitHub** (ветка **00_simple**[2]). При изучении отдельных фрагментов кода софт-процессора найдите эти строчки в файлах с исходными кодами.

Структура каталогов проекта приведена в табл. 11.9, перечень основных файлов с исходным кодом – в табл. 11.10.

Таблица 11.9 Структура каталогов проекта

№ п/п	Каталог	Описание
1	src	Платформонезависимый исходный код процессорного ядра **schoolMIPS** на языке **Verilog**
2	board	Платформозависимый исходный код. Проекты для средств синтеза (**Quartus**, **Vivado**) и модули верхнего уровня, предназначенные для различных отладочных плат
3	board / program	**Hex**-файлы памяти программ, используемые при синтезе для последующей инициализации блоков памяти **ПЛИС**
4	board / <наименование платы>	Проект **Quartus**, включая модуль верхнего уровня –специфический для конкретной отладочной платы **ПЛИС**

[1] Рекомендуется использовать для этого редактор **Visual Studio Code**.
[2] https://github.com/MIPSfpga/schoolMIPS/tree/00_simple.

Продолжение табл. 11.9

№ п/п	Каталог	Описание
5	program	Примеры программ (тестовые программы), включая исходные коды, скрипты компиляции и моделирования (симуляции). Работа каждого примера проверена в режиме моделирования и на отладочной плате
6	program / 00_ counter	Пример программы простого инкрементального счетчика
7	program / 01_ fibonacci	Пример программы вычисления последовательности чисел Фибоначчи
8	program / 02_sqrt	Пример программы вычисления квадратного корня итеративным способом
9	testbench	**Verilog** – модули для тестирования. Используются только в режиме моделирования
10	scripts	Служебные скрипты и утилиты

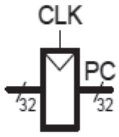
Таблица 11.10 Основные файлы с исходным кодом

№ п/п	Файл	Описание
1	src / sm_cpu.v	Тракт данных и устройство управления
2	src / sm_cpu.vh	Битовые константы (коды операций, функций, управления **АЛУ**)
3	src / sm_hex_display.v	Интерфейс к семисегментному индикатору
4	src / sm_register.v	Регистр. **IP**-блок
5	src / sm_rom.v	Асинхронная однопортовая память **ROM**. **IP**-блок
6	src / sm_top.v	Модуль верхнего уровня (платформонезависимый код)
7	board / <имя_платы> / <имя_платы>.v	Модуль верхнего уровня (для текущей отладочной платы)
8	testbench / testbench.v	Модуль верхнего уровня (для запуска в симуляторе)

Счетчик команд

Исходный код счетчика команд и его графическое изображение приведены ниже. Счетчик команд представляет собой 32-битный регистр. Тип сброса – асинхронный. Модуль **sm_register** описан в файле **sm_register.v** и используется в файле с основным описанием тракта данных: **sm_cpu.v**.

```
// sm_cpu.v (line 33)
sm_register r_pc(clk ,rst_n, pc_new, pc);
...
// sm_register.v (line 3-15)
module sm_register
(
   input clk,
   input rst,
   input [ 31 : 0 ] d,
```



```
   output reg [ 31 : 0 ] q
);
   always @ (posedge clk or negedge rst)
      if(~rst)
         q <= 32'b0;
      else
         q <= d;
endmodule
```

Листинг 11.1 Счетчик команд

Память команд

Память команд представляет собой параметризированный модуль, описание которого содержится в файле **sm_rom.v**. Чтение данных из памяти осуществляется в асинхронном режиме: данные на выходе **RD** доступны в том же такте, когда на вход **A** был установлен адрес данных. Это является обязательным условием функционирования однотактного процессора, которым является простейшая версия **schoolMIPS**. Следует отметить, что память с асинхронным доступом на чтение работает медленнее, чем память с синхронным доступом (когда считываемые данные доступны в следующем такте). В реальных проектах задержка доступа к блочной памяти обычно составляет **1–2** такта[1].

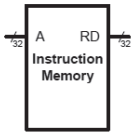
Встроенная функция (листинг 11.2) обеспечивает инициализацию памяти **rom** начальными значениями из файла **program.hex**. Особенность данной функции состоит в том, что она успешно распознается как средствами моделирования, так и средствами синтеза.

```
$readmemh ("program.hex", rom);
```

Листинг 11.2 Инициализация памяти начальными значениями

Таким образом, программирование памяти команд в процессоре **schoolMIPS** выполняется на этапе синтеза проекта. К негативным последствиям такого решения можно отнести необходимость повторной компиляции проекта и конфигурации отладочной платы в случае, если требуется изменить содержимое памяти команд.

```
// sm_cpu.v (line 35-37)
sm_rom reset_rom(pc, instr);
...
// sm_rom.v (line 2-17)
```



[1] Описание примитива памяти в таком случае содержит один или два последовательно соединенных буферных регистра на выходе памяти. Для того чтобы подобный модуль на **Verilog** был распознан такими средствами синтеза, как **RAM**- или **ROM**-память (эта функциональность называется **memory inference**), исходный код, описывающий память, должен быть написан определенным образом. Данный вопрос подробно рассмотрен в документации на программное обеспечение средств синтеза. Для **Quartus Lite** примеры описания подобных примитивов доступны из интерфейса программы в меню **Edit -> Insert Template -> Verilog HDL -> Full Designs -> RAMs and ROMs** (перед этим в редакторе должен быть открыт файл с исходным кодом). Для **ПЛИС Xilinx** примитивы описаны в документе **Vivado Design Suite User Guide. Synthesis** (**UG901**).

```
module sm_rom
#(
   parameter SIZE = 64
)
(
   input [31:0] a,
   output [31:0] rd
);
   reg [31:0] rom [SIZE - 1:0];
   assign rd = rom [a];

   initial begin
      $readmemh («program.hex», rom);
   end
endmodule
```

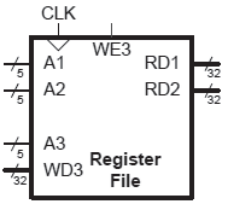
Листинг 11.3 Память команд

Регистры общего назначения

Исходный код регистрового файла (блока регистров общего назначения) приведен ниже (листинг 11.4). Реализация предусматривает **три** порта чтения (**A0–A2**, **RD0–RD2**) и один порт записи (**A3**, **WD3**, **WE3**). На графическом изображении не обозначен порт чтения (**A0**, **RD0**) – этот порт используется для отладочного доступа к регистрам (его описание дано в разделе 11.1.4). В соответствии с архитектурой **MIPS** при чтении из нулевого регистра (**zero**) всегда возвращается ноль.

В реальных процессорах (**ASIC**-реализациях) для повышения производительности данный модуль используется в виде топологического блока.

```
// sm_cpu.v (line 161-182)
module sm_register_file
(
   input clk,
   input [ 4:0] a0,
   input [ 4:0] a1,
   input [ 4:0] a2,
   input [ 4:0] a3,
   output [31:0] rd0,
   output [31:0] rd1,
   output [31:0] rd2,
   input [31:0] wd3,
   input we3
);
   reg [31:0] rf [31:0];
   assign rd0 = (a0 != 0) ? rf [a0] : 32'b0; //for debug
   assign rd1 = (a1 != 0) ? rf [a1] : 32'b0;
   assign rd2 = (a2 != 0) ? rf [a2] : 32'b0;

   always @ (posedge clk)
```



```
      if(we3) rf [a3] <= wd3;
endmodule
```

Листинг 11.4 Регистровый файл

Арифметико-логическое устройство

АЛУ представляет собой блок, который в зависимости от поступающего на него управляющего сигнала – кода операции **oper** (табл. 11.11) – возвращает результат выполнения одной из математических/логических операций.

Таблица 11.11 Коды операций АЛУ

oper2:0	Функция	Описание
000	ADD	A + B
001	OR	A | B
010	LUI	B << 16
011	SRL	B >> shift
100	SLTU	(A < B) ? 1 : 0
101	SUBU	A - B
110	reserved	
111	reserved	

Задание: в табл. 11.11 приведены операции логического **ИЛИ** (**OR**), загрузки старших двух байт (**LUI**) и сравнения двух регистров (**SLTU**), которые ранее не были рассмотрены. Проанализируйте их, определите, к какому типу инструкций они относятся. Объясните, почему им назначен именно такой код **oper**.

Графическое изображение и исходный код **АЛУ** приведены в листинге 11.5. Основная сложность **АЛУ** заключается в реализации арифметических операций (особенно таких, как деление, умножение и т. п.; в представленном софт-процессоре не реализованы). В случае **ПЛИС** с их реализацией справляются средства синтеза, которые при выполнении ряда условий могут задействовать имеющиеся в **ПЛИС DSP**-блоки. При проектировании заказных интегральных схем (**ASIC**) разработчики самостоятельно подбирают необходимую реализацию, как правило, учитывая ограничения по задействованной площади кристалла и требования к производительности.

```
// sm_cpu.vh (line 11-17)
`define ALU_ADD 3'b000
`define ALU_OR 3'b001
`define ALU_LUI 3'b010
`define ALU_SRL 3'b011
`define ALU_SLTU 3'b100
`define ALU_SUBU 3'b101
// sm_cpu.v (line 137-159)
module sm_alu (
   input [31:0] srcA,
```

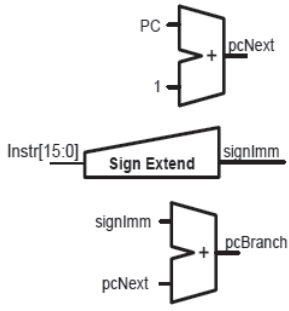
```
   input [31:0] srcB,
   input [ 2:0] oper, // aluControl
   input [ 4:0] shift,
   output zero, // aluZero
   output reg [31:0] result
);
   always @ (*) begin
      case (oper)
         default : result = srcA + srcB;
         `ALU_ADD : result = srcA + srcB;
         `ALU_OR : result = srcA | srcB;
         `ALU_LUI : result = (srcB << 16);
         `ALU_SRL : result = srcB >> shift;
         `ALU_SLTU : result = (srcA < srcB) ? 1 : 0;
         `ALU_SUBU : result = srcA - srcB;
      endcase
   end
   assign zero = (result == 0);
endmodule
```

Листинг 11.5 АЛУ

Остальные элементы тракта данных

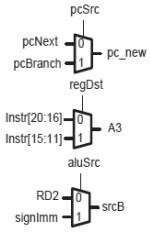
Остальные элементы тракта данных, к которым относятся **сумматоры** и **блок расширения знака** (листинг 11.6), а также мультиплексоры (рис. 11.7), не вынесены в отдельные модули Verilog и реализованы в виде комбинационной логики внутри модуля **sm_cpu**.

```
//program counter sm_cpu.v (line 28-31)
wire [31:0] pc;
wire [31:0] pcBranch;
wire [31:0] pcNext = pc + 1;

//sign extension sm_cpu.v (line 64)
wire [31:0] signImm
   = { {16 { instr[15] }}, instr[15:0] };

//branch address calculation sm_cpu.v (line 65)
assign pcBranch = pcNext + signImm;
```



Листинг 11.6 Сумматоры тракта данных и блок расширения знака

```
// next PC mux: branch or +1 (line 32)
wire [31:0] pc_new = ~pcSrc ? pcNext : pcBranch;
// register file address A3 (line 44)
wire [ 4:0] a3 = regDst ? instr[15:11] : instr[20:16];
// alu source B (line 68)
wire [31:0] srcB = aluSrc ? signImm : rd2;
```



Листинг 11.7 Мультиплексоры тракта данных

11.1.5 Реализация. Устройство управления

Устройство управления реализовано в виде комбинационной схемы. На ее вход поступают:

- поле инструкции «код операции» (**cmdOper** – соответствует полю **op**, рис. 11.31);
- поле инструкции «код функции» (**cmdFunk** – соответствует полю **funct**, рис. 11.31);
- признак равенства поступивших на вход **АЛУ** операндов (**aluZero**).

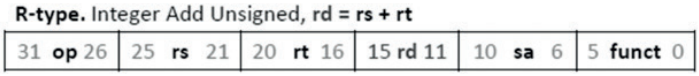


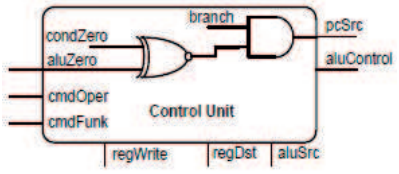
Рис. 11.31 Инструкция ADDU

На выходе данного блока формируются:

- управляющие сигналы для мультиплексоров тракта данных:
  - **pcSrc** – определяет следующее значение счетчика команд в зависимости от необходимости выполнения условного перехода;
  - **regDst** – определяет регистр, в который будет записан результат выполнения текущей инструкции;
  - **aluSrc** – определяет, откуда будет получен операнд **АЛУ**: из регистрового файла или из блока расширения адреса;
- сигнал разрешения записи в регистровый файл (regWrite);
- управляющие сигналы для **АЛУ** (**aluControl**), коды которых рассмотрены в табл. 11.11 и определены в файле **sm_cpu.vh**.

Фрагмент описания данного блока на языке **Verilog**, включая интерфейс модуля и логику формирования сигнала **pcSrc**, приведен в листинге 11.8. Пример формирования остальных управляющих сигналов на примере инструкции **ADDU** дан в листинге 11.9.

```
// control unit (line 95-134)
module sm_control
(
   input [5:0] cmdOper,
   input [5:0] cmdFunk,
   input aluZero,
   output pcSrc,
   output reg regDst,
   output reg regWrite,
   output reg aluSrc,
   output reg [2:0] aluControl
);
   reg branch;
   reg condZero;
```



```
   assign pcSrc = branch & (aluZero == condZero);
   always @ (*) begin
      ...
   end
endmodule
```

Листинг 11.8 Устройство управления (фрагмент 1)

```
// control unit (line 95-134)
module sm_control
...
   always @ (*) begin
      //control signals default values
      branch = 1'b0;
      condZero = 1'b0;
      regDst = 1'b0;
      regWrite = 1'b0;
      aluSrc = 1'b0;
      aluControl = `ALU_ADD;

      casez( {cmdOper,cmdFunk} )
         default : ;
         { `C_SPEC, `F_ADDU } : begin
            regDst = 1'b1;
            regWrite = 1'b1;
            aluControl = `ALU_ADD;
            end

         { `C_SPEC, `F_OR } : begin
            regDst = 1'b1;
            regWrite = 1'b1;
            aluControl = `ALU_OR;
            end
         ...
      endcase
   end
endmodule
```

Листинг 11.9 Устройство управления (фрагмент 2)

11.2 Использование процессорного ядра schoolMIPS на практике

В данном разделе описывается порядок выполнения практической работы с процессорным ядром **schoolMIPS**, его программирования и запуска.

Задание: при ознакомлении с данным разделом выполните все описываемые шаги на примере программы **00_counter**. Для понимания логики работы программы используйте в качестве справки по командам ассемблера **MIPS** документ

MIPS32 Instruction Set Quick Reference. Самостоятельно повторите эти действия для программ **01_fibonacci и 02_sqrt**.

11.2.1 Элементы управления и индикации

Обобщенная блок-схема развертывания процессора **schoolMIPS** на отладочной плате приведена на рис. 11.32[1]. Контроль исполнения программы (последовательности инструкций) процессором осуществляется следующим образом:

- набор переключателей **SW[8:5]**, формирующих сигнал **clkDivide**, задает режим работы делителя частоты **sm_clk_divider** и позволяет снизить тактовую частоту (сигнал **clk**), на которой работает процессорное ядро, до 0,5–1 Гц;
- переключатель **SW[9]** и кнопка **KEY[1]**, управляющие сигналом **clkEnable**, позволяют временно (кнопка) или на более длительный срок (переключатель) приостановить выполнение инструкций, запрещая или разрешая передачу тактового сигнала **clk** на вход процессорного ядра;
- с помощью переключателей **SW[4:0]**, задающих сигнал **regAddr**, устанавливается номер регистра, значение которого в реальном времени выводится на семисегментные индикаторы (сигнал **regData**). В соответствии с архитектурой **MIPS** значение, считываемое с регистра **zero** (нулевой регистр), всегда равно **0**. По этой причине при **regData = 0** на индикаторы выводится значение счетчика команд **PC**;
- с помощью кнопки **KEY[0]**, управляющей сигналом **rst_n**, реализован аппаратный сброс процессорного ядра. После того как эта кнопка будет отпущена, процессор начнет выполнение программы с нулевого (**0x00000000**) адреса.

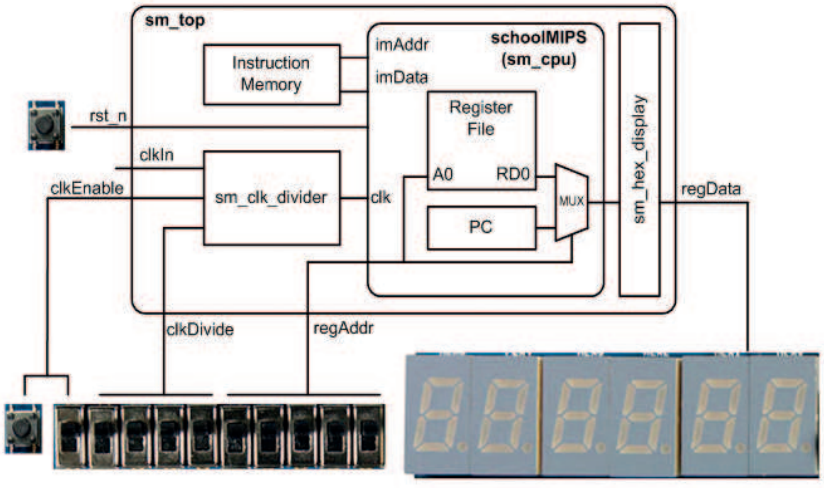


Рис. 11.32 Элементы управления и индикации

[1] На примере отладочной платы **Terasic DE10-Lite**. Для других плат реализация может незначительно отличаться.

Приведенная выше логика работы реализована на **Verilog** в файлах:

- **src/sm_top.v** – платформонезависмый код (общий для всех плат);
- **board/de10_lite/de10_lite.v** – специфический для отладочной платы код.

11.2.2 Сборка и запуск проекта

Работа **schoolMIPS** на отладочной плате определяется двумя составляющими:

- **RTL**-кодом, который определяет микроархитектуру ядра и работу периферийных модулей;
- программным кодом, который определяет состав инструкций, выполняемых процессором.

В общем случае, для того чтобы получить из исходных кодов работающую на отладочной плате конфигурацию, необходимо выполнить ряд шагов, включающих:

1) синтез файла конфигурации **ПЛИС** из **RTL**-кода;
2) конфигурирование **ПЛИС** («прошивку» платы);
3) компиляцию программного кода (**C**, **assembler** и т. д.) в исполняемый (объектный) файл (**ELF**);
4) загрузку скомпилированной программы в память синтезированной системы;
5) передачу управления программе.

Этапы 1 и 2 выполняются с помощью инструментов, которые поставляются производителями **ПЛИС**: программное обеспечение **Quartus Prime** (**Intel FPGA**) или **Vivado** (**Xilinx**). На шаге 3 используются компилятор и компоновщик, входящие в состав пакета программ разработчика (**toolchain**). В зависимости от процессорного ядра и его архитектуры данное программное обеспечение может быть как с открытым исходным кодом (к примеру, компилятор **MIPS**, доступный в репозитории **Ubuntu**), так и поставляться коммерческими компаниями (компиляторы **ARM** и **Intel**). Этапы 4 и 5 могут выполняться либо с помощью аппаратного отладчика, управляющего системой (рис. 11.33-A), либо так называемым загрузчиком при штатной самостоятельной загрузке системы (рис. 11.33-B)[1].

Ядро **schoolMIPS** является предельно простым, поэтому не все описанные выше возможности в нем доступны, и маршрут сборки и запуска проекта отличается.

Для старших (конвейерных) версий ядра **schoolMIPS** используется **MIPS toolchain**[2] (рис. 11.33-C). Полученный в результате компиляции **ELF** файл применяется для генерации **HEX**-дампа памяти программ. Инициализация памяти программ начальными значениями, полученными из **HEX**-дампа, выполняется на этапе синтеза конфигурации **ПЛИС** (листинг 11.1).

[1] Работа загрузчика – это сложный многоступенчатый процесс; на схеме он представлен условно.
[2] С тем же результатом можно использовать инструментарий из репозитория Ubuntu.

В младших версиях процессорного ядра **schoolMIPS** отсутствует необходимость выполнять компиляцию исходных кодов на языке **C**. Поэтому вместо **MIPS toolchain** используется симулятор **MARS**, который позволяет получить **HEX**-дамп напрямую из исходного кода на ассемблере (рис. 11.33-D). Именно этот маршрут используется в данной главе.

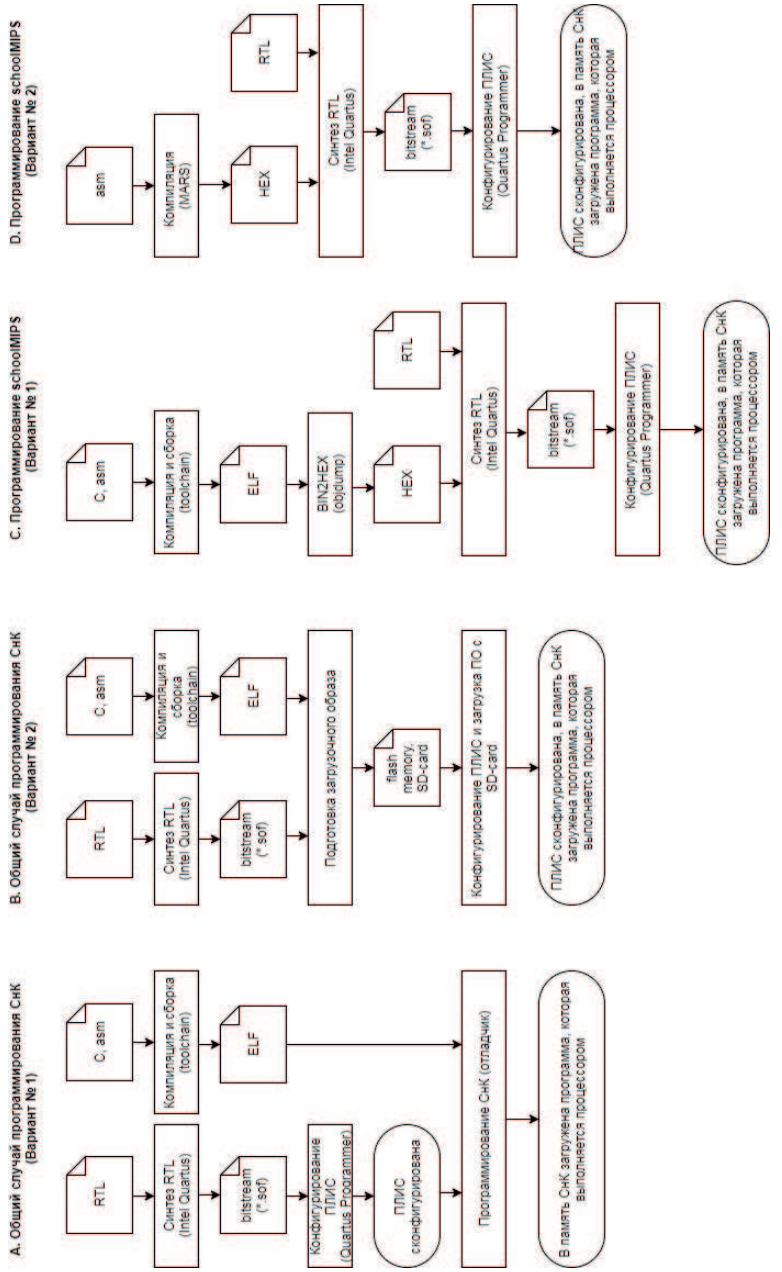


Рис. 11.33 Маршруты программирования системы

11.2.3 Структура каталога программы

В состав рассматриваемой простейшей версии **schoolMIPS** входит ряд программ-примеров: простейший инкрементальный счетчик (**00_counter**), вычисление последовательности чисел Фибоначчи (**01_fibonacci**) и вычисление квадратного корня итеративным способом (**02_sqrt**). Они расположены в каталоге **lab_11/src/program**. Структура каталога программы-примера и описание находящихся там файлов приведены ниже.

Таблица 11.12 Структура каталога программы

№ п/п	Файл	Описание
1	00_clean_all.bat	Скрипт. Выполняет очистку каталога от временных файлов, оставшихся от предыдущей сборки программы
2	01_run_mars.bat	Скрипт. Выполняет запуск симулятора MARS[1]
3	02_compile_to_hex_with_mars.bat	Скрипт. Выполняет компиляцию программы из исходного кода в **HEX**-файл
4	03_simulate_with_modelsim.bat	Скрипт. Выполняет запуск моделирования с использованием симулятора Modelsim
5	04_simulate_with_icarus.bat	Скрипт. Выполняет запуск моделирования с использованием симулятора **Icarus Verilog**
6	05_copy_program_to_board.bat	Скрипт. Выполняет копирование **HEX**-файла в каталог **board/program**, где он используется на этапе синтеза конфигурации **ПЛИС**
7	main.S	Исходный код программы на языке **MIPS assembler**
8	modelsim_script.tcl	Скрипт с командами симулятора Modelsim
9	program.hex	**HEX**-файл. Результат компиляции. Создается скриптом **02_compile_to_hex_with_mars.bat** и удаляется скриптом **00_clean_all.bat**

Для моделирования также используется файл **lab_11/src/board/testbench/testbench.v** – верхний уровень тестового окружения, в том числе обеспечивающий вывод отладочных сообщений в консоль симулятора.

Служебные скрипты (***.bat**) рекомендуется запускать в уже открытом терминале – на случай, если их вывод будет содержать информацию об ошибках.

Для синтеза проекта и его запуска на отладочной плате используются файлы, расположенные в каталоге **lab_11/src/board/<имя отладочной платы>** (табл. 11.13), а также описанный выше файл **lab_11/src/program/project/program.hex**.

[1] MARS – MIPS Assembler and Runtime Simulator.

Таблица 11.13 Файлы проекта для синтеза

№ п/п	Файл	Описание
1	de10_lite.qpf	Файл проекта для среды **Quartus Prime**. Его присутствие необходимо, но сам файл может быть пустым
2	de10_lite.qsf	Файл настроек проекта в среде **Quartus Prime**. Содержит перечень файлов, используемых при сборке, настройке выводов **ПЛИС** и их связи с **RTL**-кодом и т. д. Данный файл может быть отредактирован как средствами графического интерфейса среды Q**uartus Prime**, так и в текстовом редакторе
3	de10_lite.sdc	Настройки временных ограничений на выводах **ПЛИС** в формате **SDC** (**Synopsys Design Constraint**)
4	de10_lite.v	Специфический для используемой отладочной платы **RTL**-код **schoolMIPS**
5	make_project.bat	Скрипт. Выполняет удаление результатов сборки, оставшейся от предыдущего запуска, и создает каталог с проектом **Quartus Prime**

11.2.4 Порядок запуска программы

В общем случае работа с программой-примером включает следующие шаги:

- запуск и отладка программы с использованием симулятора **MARS** – программного симулятора процессора архитектуры **MIPS**;
- запуск и отладка **RTL**-кода **schoolMIPS** в симуляторе **Modelsim** и/или **Icarus Verilog** (с памятью команд, инициализированной образом программы);
- запуск и отладка на отладочной плате.

Данные шаги следует выполнять последовательно. Их подробное описание приведено ниже.

Симулятор MARS

Для выполнения программы в симуляторе **MARS** необходимо выполнить следующие шаги:

1. Запустить терминал и перейти в каталог с примером программы. В редакторе **Visual Studio Code** это можно сделать с помощью контекстного меню **Open in Command Prompt** (рис. 11.34). Для переключения запущенной консоли из открытого в скрытое состояние (и наоборот) можно использовать сочетание клавиш **Ctrl+`**.
2. В консоли выполнить запуск скрипта **01_run_mars.bat**. Результатом этого станет запуск симулятора **MARS**.
3. В окне симулятора открыть файл с исходным кодом программы (**main.S**). Для этого следует использовать пункт меню **File → Open**. Интерфейс программы примет вид как на рис. 11.35.

4. Выполнить компиляцию программы: **Run → Assemble**. Интерфейс программы примет вид как на рис. 11.36.

5. Выполнить проверку работы программы в пошаговом режиме (**Run → Step**) либо путем установки точки останова (рис. 11.36) с последующим запуском моделирования (**Run → Go**). При пошаговом выполнении программы поле регистра, который был изменен в результате работы предыдущей инструкции, выделяется цветом (рис. 11.38).

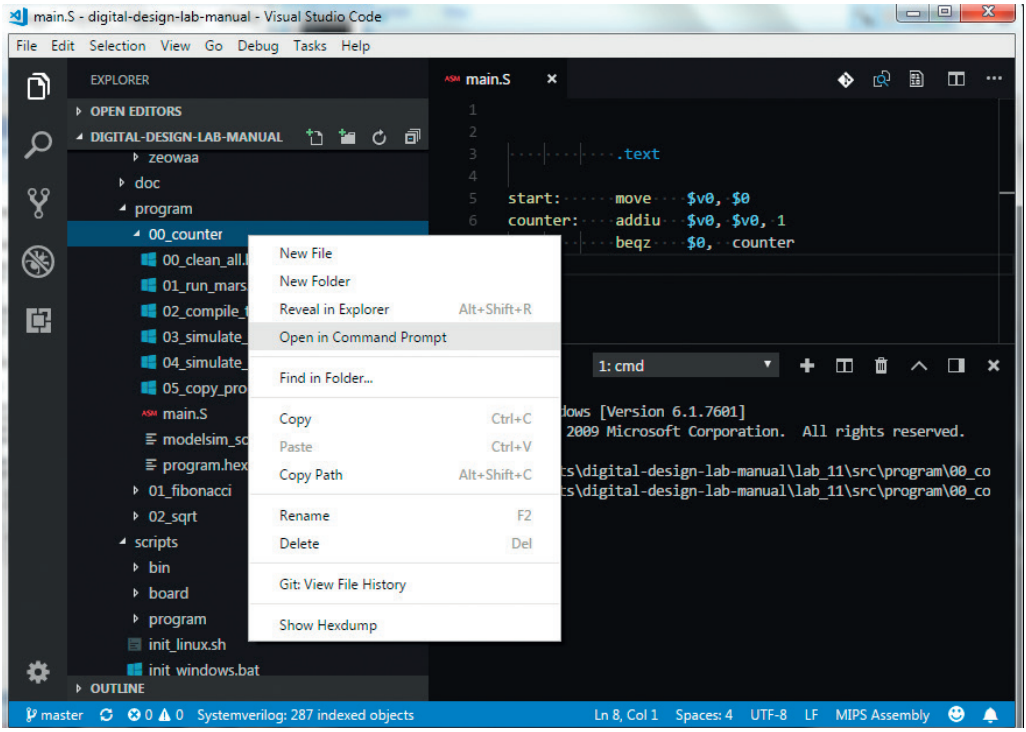


Рис. 11.34 Открытие встроенной консоли в редакторе Visual Studio Code

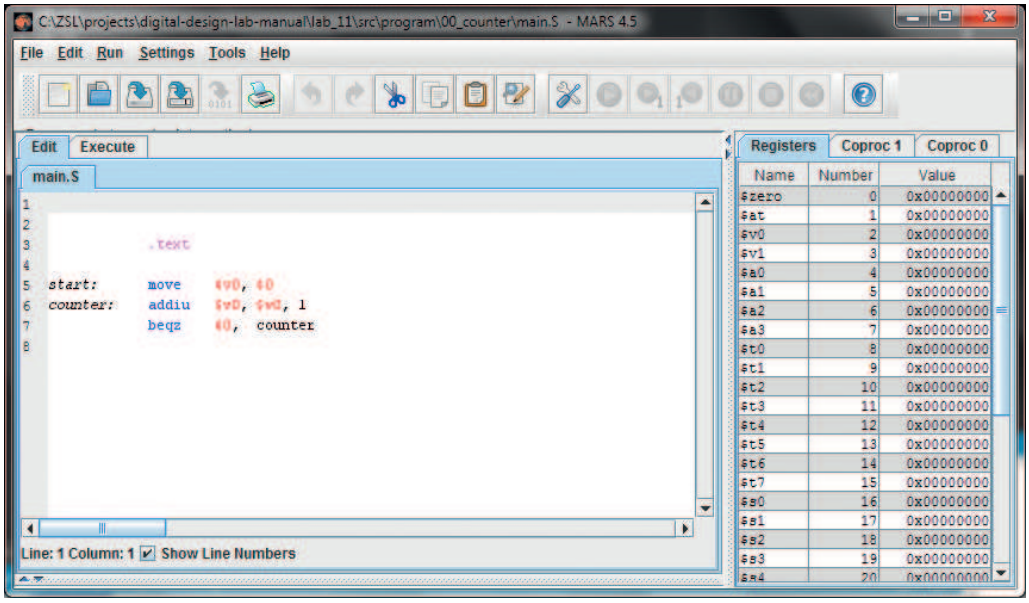


Рис. 11.35 Интерфейс симулятора MARS

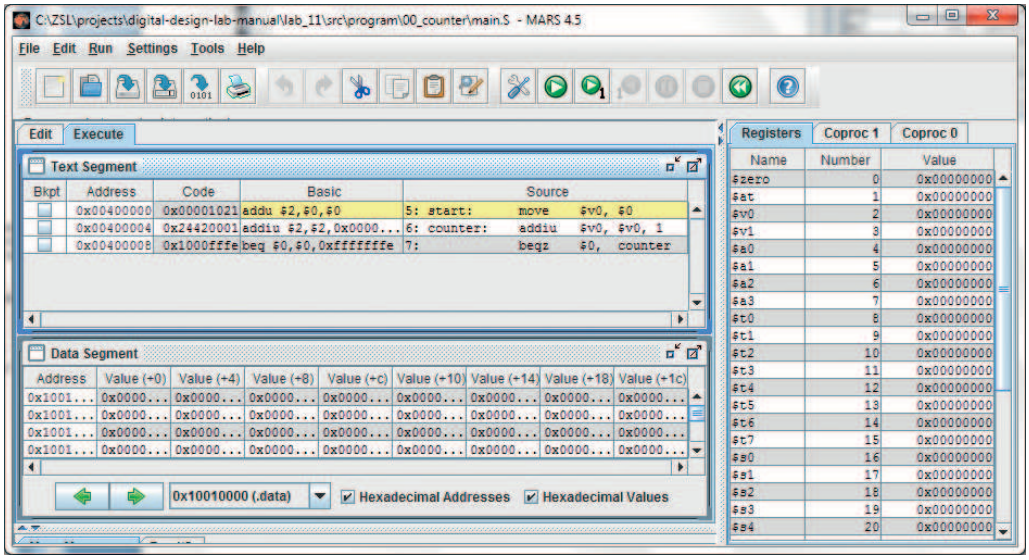


Рис. 11.36 Симулятор MARS в режиме отладки

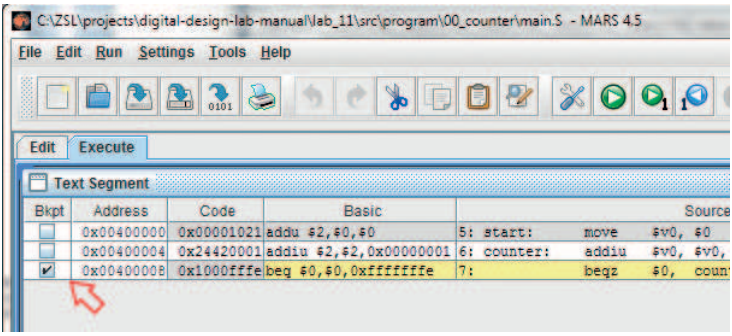


Рис. 11.37 Установка точки останова в интерфейсе MARS

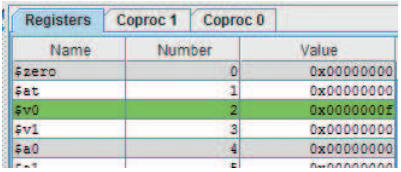


Рис. 11.38 Окно регистров в интерфейсе MARS

Симулятор Modelsim

Для моделирования работы процессора с помощью Modelsim необходимо выполнить следующие шаги:

1. Запустить терминал и перейти в каталог с примером программы.
2. Удалить временные файлы, оставшиеся от предыдущего запуска (скрипт **00_clean_all.bat**).
3. Выполнить компиляцию программы и сформировать **HEX**-файл с ее образом (скрипт **02_compile_to_hex_with_mars.bat**). Результатом работы скрипта будет текстовый **HEX**-файл, который содержит коды инструкций процессора в шестнадцатеричном виде.
4. Запустить выполнение в симуляторе **Modelsim** (скрипт **03_simulate_with_modelsim.bat**). В результате запуска этого скрипта на экране отобразится окно **Modelsim** с запущенным моделированием и выводом отладочных сообщений.
5. Отладочный вывод (рис. 11.39) содержит:
   - номер текущего такта (**cycle**);
   - значение счетчика команд (**pc**);
   - адрес текущей команды в памяти (**pcaddr**, без учета словной адресации);
   - код выполняемой инструкции (**instr**);
   - значение регистра **v0**;
   - дизассемблированное представление текущей инструкции.

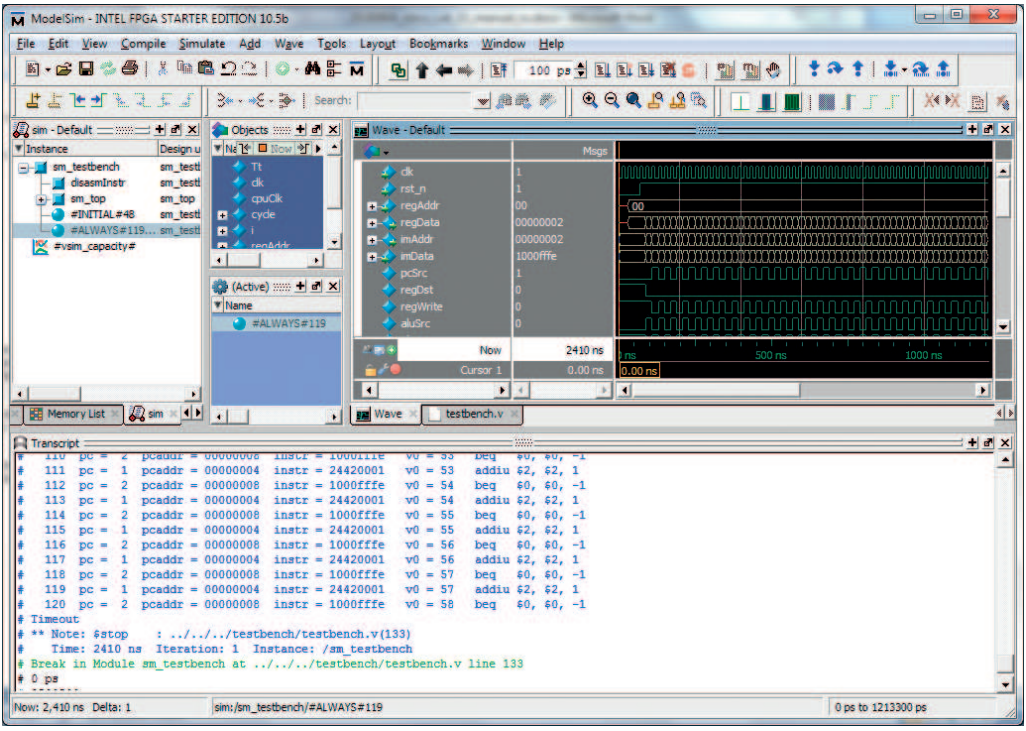


Рис. 11.39 Запуск моделирования в Modelsim

Симулятор Icarus Verilog

Для моделирования с помощью **Icarus Verilog** необходимо выполнить следующие шаги:

1. Запустить терминал и перейти в каталог с примером программы.
2. Удалить временные файлы, оставшиеся от предыдущего запуска (скрипт **00_clean_all.bat**).
3. Выполнить компиляцию программы и сформировать **HEX**-файл с ее образом (скрипт **02_compile_to_hex_with_mars.bat**).
4. Запустить выполнение в симуляторе **Icarus Verilog** (скрипт **04_simulate_with_icarus.bat**). В результате запуска этого скрипта в консоль будут выведены отладочные сообщения. На экране отобразится окно **GtkWave** для просмотра временной диаграммы. В открывшемся окне в иерархии объектов моделирования необходимо выбрать все сигналы внутри **sm_testbench.sm_top.sm_cpu** и добавить их на временную диаграмму (рис. 11.40).

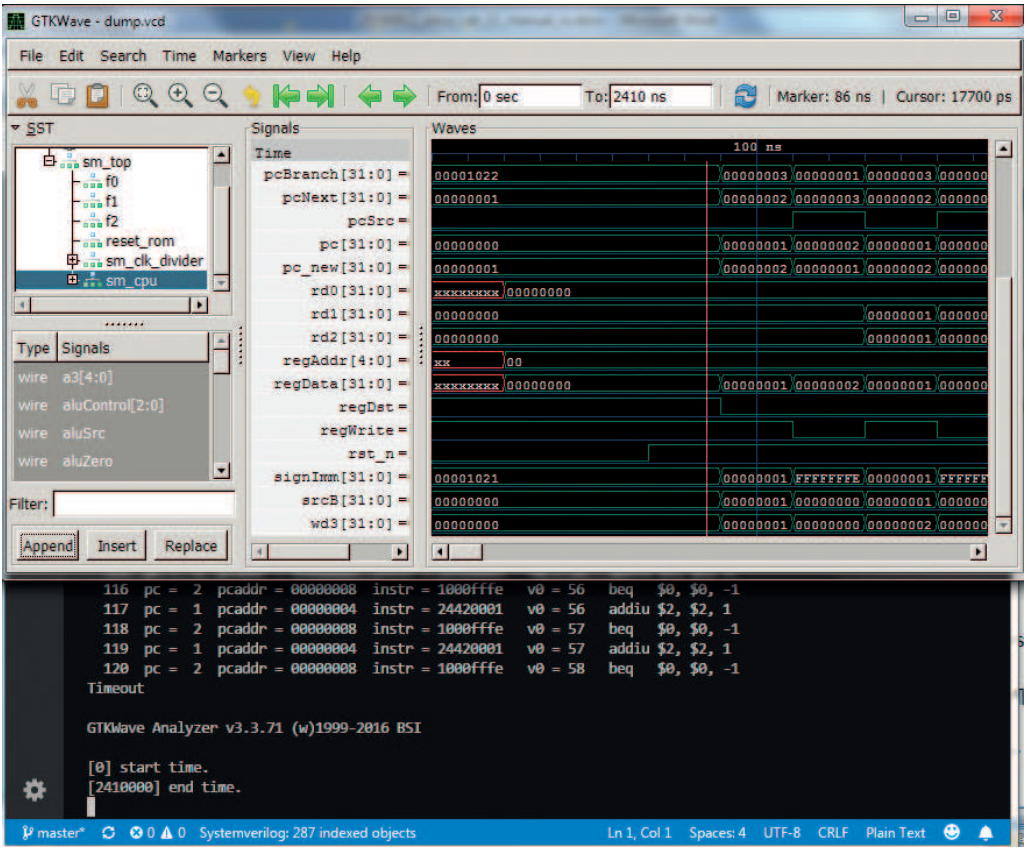


Рис. 11.40 Отладочный вывод Icarus Verilog и интерфейс GTKWave

Запуск на отладочной плате

Для запуска проекта на отладочной плате необходимо выполнить следующие шаги:

1. Запустить терминал и перейти в каталог с примером программы.
2. Удалить временные файлы, оставшиеся от предыдущего запуска (скрипт **00_clean_all.bat**).
3. Выполнить компиляцию программы и сформировать **HEX**-файл с ее образом (скрипт **02_compile_to_hex_with_mars.bat**).
4. Скопировать **HEX**-файл в каталог, доступный для средств синтеза (скрипт **05_copy_program_to_board.bat**).
5. Перейти в каталог с файлами, специфическими для используемой отладочной платы, или открыть новый терминал в данном каталоге: **lab_11/src/board/<имя отладочной платы>**.
6. Сформировать проект для **Quartus Prime** (скрипт **make_project.bat**).

7. Запустить программный пакет для синтеза **Quartus Prime** (версия не ниже **16.1**).
8. Открыть в запущенном пакете проект **lab_11/src/board/<имя отладочной платы>/project/<имя отладочной платы>.qpf**. Для этого можно использовать пункт меню **File → Open Project**.
9. Выполнить компиляцию проекта: **Processing → Start Compilation**.
10. Подключить отладочную плату к рабочей станции.
11. Запустить интерфейс программатора: **Tools → Programmer**.
12. В интерфейсе программатора (при необходимости) выполнить настройку подключения платы (кнопка **Hardware Setup**), после чего осуществить программирование **ПЛИС** (нажать кнопку **Start**).
13. Используя для управления выполнением программы переключатели и кнопки, расположенные на отладочной плате (описание приведено в разделе 11.2.1), произвести проверку работы программы.

11.3 Упражнения

11.3.1 Основное задание

Базовые задачи

1. Используя информацию из документов **MIPS32 Instruction Set Quick Reference, MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**, а также результаты моделирования с применением программного обеспечения **MARS**, опишите в свободной форме пошаговую работу программ **00_counter** и **01_fibonacci**.
2. Используя информацию из документов **MIPS32 Instruction Set Quick Reference, MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**, а также результаты моделирования с применением программного обеспечения **MARS**, опишите в свободной форме пошаговую работу программы **02_sqrt**. Общее понимание алгоритма вычисления целочисленного квадратного корня итеративным способом при этом не требуется[1].

11.3.2 Задания для самостоятельной работы

Задачи среднего уровня сложности

1. Добавьте в процессор реализацию инструкций: **SLL**, **SUBU**, **BGEZ**. Напишите программу (или несколько программ), которая подтверждает их работоспособность. Продемонстрируйте работу программы в симуляторе и на отладочной плате.

[1] Данный алгоритм реализован как иллюстрация того, что даже на таком простом процессоре, как **schoolMIPS**, можно вычислить квадратный корень. Если вас заинтересовала математическая сторона вопроса, то можете найти описание этого алгоритма в книге: *Warren H. S.* Hacker's delight. Pearson Education. 2013 (*Уоррен Г.* Алгоритмические трюки для программистов. 2-е изд. М.: Вильямс, 2014).

2. Добавьте в процессор реализацию инструкций: **ORI**, **SEB**, **BGTZ**. Напишите программу (или несколько программ), которая подтверждает их работоспособность. Продемонстрируйте работу программы в симуляторе и на отладочной плате.
3. Добавьте в процессор реализацию инструкций: **XORI**, **AND**, **BLTZ**. Напишите программу (или несколько программ), которая подтверждает их работоспособность. Продемонстрируйте работу программы в симуляторе и на отладочной плате.
4. Добавьте в процессор реализацию инструкций: **ANDI**, **SEH**, **J**. Напишите программу (или несколько программ), которая подтверждает их работоспособность. Продемонстрируйте работу программы в симуляторе и на отладочной плате.
5. Добавьте в процессор реализацию инструкций: **SLTI**, **MOVN**, **BLEZ**. Напишите программу (или несколько программ), которая подтверждает их работоспособность. Продемонстрируйте работу программы в симуляторе и на отладочной плате.

Задачи уровня выше среднего[1]

1. Добавьте в процессор реализацию инструкции **CLO**. Напишите программу, которая подтверждает ее работоспособность. Продемонстрируйте ее работу в симуляторе и на отладочной плате.
2. Добавьте в процессор реализацию инструкции **CLZ**. Напишите программу, которая подтверждает ее работоспособность. Продемонстрируйте ее работу в симуляторе и на отладочной плате.
3. Добавьте в процессор реализацию инструкции **MUL**. Напишите программу, которая подтверждает ее работоспособность. Продемонстрируйте ее работу в симуляторе и на отладочной плате.

11.3.3 Контрольные вопросы

1. Используя информацию из документов **MIPS32 Instruction Set Quick Reference, MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**, обозначьте на схеме маршрут прохождения сигналов (активные и значимые сигналы) при выполнении инструкции **SLL**.

[1] Описание алгоритмов можно найти в книге: *Warren H. S.* Hacker's delight. Pearson Education. 2013 (*Уоррен Г.* Алгоритмические трюки для программистов. 2-е изд. М.: Вильямс, 2014).

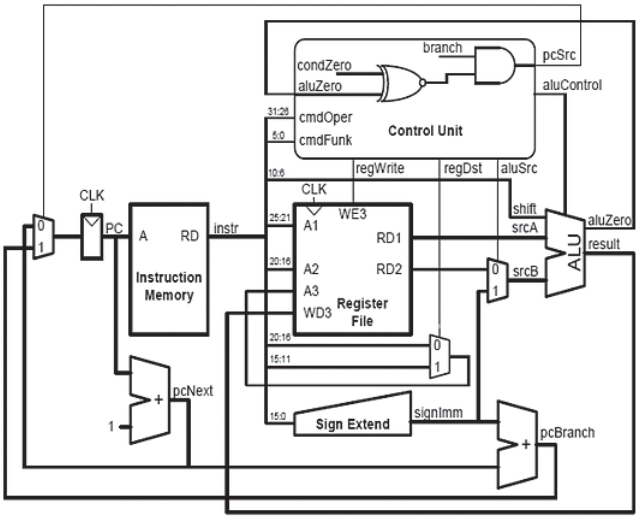


Рис. 11.41 Контрольный вопрос 1

2. Используя информацию из документов **MIPS32 Instruction Set Quick Reference, MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**, обозначьте на схеме маршрут прохождения сигналов (активные и значимые сигналы) при выполнении инструкции **SUBU**.

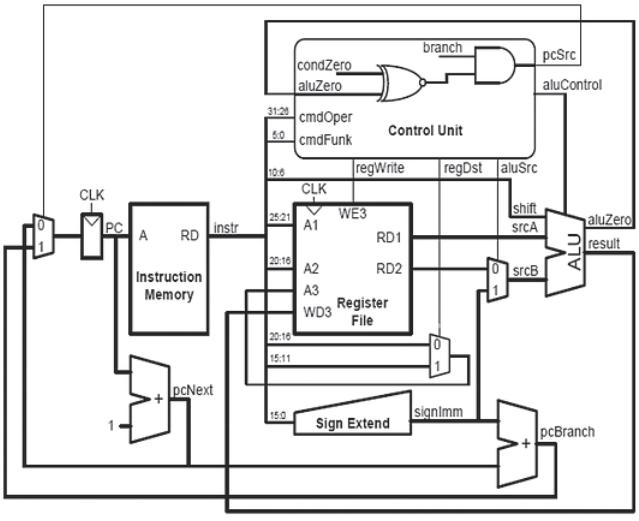


Рис. 11.42 Контрольный вопрос 2

3. Используя информацию из документов **MIPS32 Instruction Set Quick Reference, MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**, обозначьте на схеме маршрут прохождения сигналов (активные и значимые сигналы) при выполнении инструкции **BNE** для случая, когда условный переход должен быть выполнен.

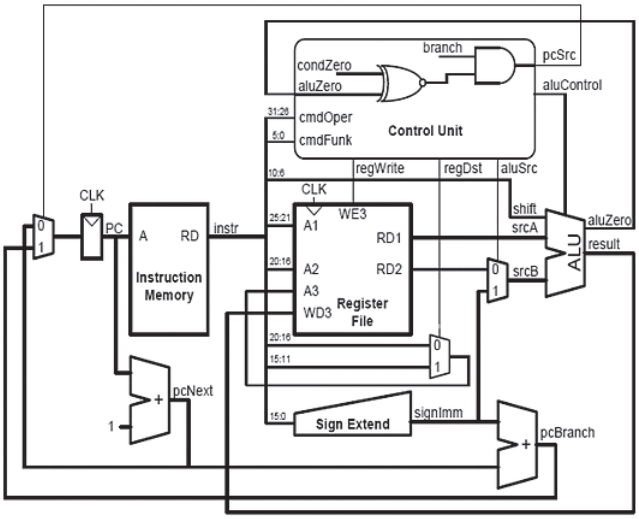


Рис. 11.43 Контрольный вопрос 3

4. Используя информацию из документов **MIPS32 Instruction Set Quick Reference, MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set**, обозначьте на схеме маршрут прохождения сигналов (активные и значимые сигналы) при выполнении инструкции **BNE** для случая, когда условный переход не должен быть выполнен.

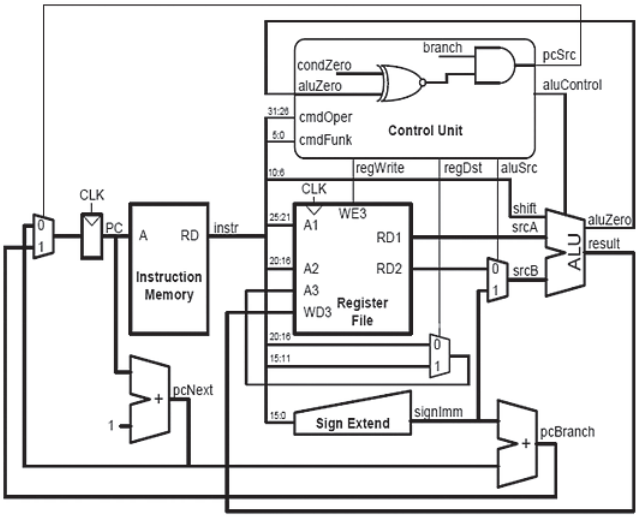


Рис. 11.44 Контрольный вопрос 4

5. На основании ответов на вопросы 1–4 заполните таблицу:

Таблица 11.14 Таблица ответов на контрольные вопросы 1–4

Инструкция	cmdOper	cmdFunc	branch	condZero	regDst	regWrite	aluSrc	aluControl
SLL								
SUBU								
BNE								

Чарльз Данчек, Александр Романов

Цифровой синтез: практический курс

Приложение А. Путь вперед: от устройств на базе FPGA к массовому рынку ASIC для популярных гаджетов

Содержание

*Данное приложение является дорожной картой, в которой поэтапно описывается процесс разработки чипов **ASIC**. Его цель – дать инженерам, прошедшим обучение проектированию на **FPGA**, возможность с успехом применять и развивать свои навыки по разработке чипов **ASIC** для конкретных применений.*

А.1 Прототипирование на FPGA как один из этапов проектирования ASIC

Чипы **ASIC** адаптированы к конкретному приложению – например, как элемент цифровой камеры, мобильного телефона или **GPS**-трекера. В отличие от **FPGA**, чип ASIC не имеет накладных логических или проводных ресурсов. Проекты **ASIC** начинаются с голой кремниевой матрицы, на которой размещают только логику и память, необходимые для приложения. Такие микросхемы для конкретных приложений работают быстрее, потребляют меньше энергии и имеют меньшую площадь, чем соответствующие устройства на **FPGA**. Этим обеспечиваются главные преимущества при производстве гаджетов массового спроса для потребительского рынка, а также других конкурентоспособных по стоимости коммерческих продуктов.

Замечание 1: согласно обширному компаративному исследованию университета Торонто, при сравнении **90 нм FPGA** с **ASIC** оказалось, что при миграции с устройства **FPGA** на чип **ASIC** в среднем обеспечивается уменьшение площади в **21** раз, уменьшение задержки в критических путях до **2,1** раза и динамическое снижение мощности в **9** раз. Точные результаты варьируются в зависимости от технологии проектирования, скорости, а также смешивания логических элементов, памяти и блоков **DSP**. Тем не менее можно сделать вывод, что в целом реализация **ASIC** имеет значительные преимущества в скорости, по занимаемой площади и потреблению энергии. Следует также учитывать и экономические факторы. **FPGA** характеризуются относительно высокой стоимостью каждого устройства, но нулевыми затратами на проектирование. **ASIC**, как объясняется в следующем разделе, имеют значительную одноразовую стоимость проектирования, но более низкую цену отдельного устройства. Для больших объемов производства привлекательна низкая себестоимость на единицу продукции с более высокой скоростью и меньшим энергопотреблением.

А.1.1 Стратегия прототипирования

Успешные компании-разработчики чипов часто создают новый продукт, начиная с создания устройства на базе **FPGA**. Этот ранний прототип служит удобной платформой для проверки функционирования оборудования, а также корректности работы соответствующей программы и прикладного программного обеспечения в целевой системе. Проверка реального аппаратного прототипа устройства более реалистична, чем использование программного обеспечения для имитационного моделирования. Хотя прототип на базе **FPGA** работает не так быстро, как целевой **ASIC**, он все же намного быстрее, чем симуляция. Вместо того чтобы ожидать завершения прогона симуляции (он может занять не одну неделю), команда

разработчиков может запустить прототип и начать исправлять ошибки уже через несколько часов.

Применение раннего прототипирования также уменьшает время выхода на рынок нового гаджета. Компания, которая своевременно выпускает новый продукт для критической ниши рынка (даже если это всего лишь ранний прототип), может занять на нем значительную долю. Затем компания может продолжить длительный цикл разработки **ASIC**, превращая проверенное устройство **FPGA** в высокопроизводительный чип **ASIC**. Прототип является также удобным средством для кодирования и тонкой настройки встроенного программного обеспечения или кода, которые тесно взаимодействуют с оборудованием.

Ранние прототипы также снижают затраты в долгосрочной перспективе. Миниатюризация технологий приводит к тому, что затраты на проектирование для изготовления **ASIC** быстро растут. В стоимости проектирования львиную долю затрат составляют единовременные издержки на создание набора от **30** до **50 CMOS**-масок. Для **90 нм ASIC** только этот набор масок может стоить до **1** млн долларов США. И эта стоимость продолжает расти с каждым технологическим шагом. При **32 нм** набор масок может уже стоить **3** млн долларов США.

При отладке прототипа **FPGA** функциональные ошибки обнаруживаются на ранней стадии. В этом случае **ASIC** с большей вероятностью будет работать уже после первого цикла изготовления. Это позволяет избежать дорогостоящего повторного цикла, связанного с изменением одной или нескольких **CMOS**-масок.

Эта стратегия прототипирования **FPGA** настолько распространена, что упоминается даже в объявлениях о вакансиях для инженеров-конструкторов. Типичное описание должностных требований, недавно опубликованное для большой компании в Кремниевой долине, выглядело примерно так:

Job Requirments:

- Write RTL code for on chip logic modules, based on microarchitecture specs.
- Perform early chip *prototyping,* using an Altera or Xilinx FPGA device.
- Synthesize all modules, meet speed and power goals, and reach timing closure.
- Support the Design Verification (DV) team for functional verification of modules.
- Hand off and integrate all logic modules into the target ASIC environment.

Рис. А.1 Типичное объявление о работе в Кремниевой долине

Должностные требования:

1. Разработка **RTL**-кода для логических модулей на чипе на основе спецификаций микроархитектуры.
2. Выполнение раннего прототипирования микросхем с использованием **FPGA Altera** или **Xilinx**.

3. Синтез модулей с соблюдением требований по скорости, энергопотреблению, а также временным ограничениям.
4. Поддержка команды верификации для функциональной проверки модулей.
5. Трансфер и интеграция логических модулей в целевую среду **ASIC**.

Хотя прототипирование **FPGA** является проверенной стратегией, в нем есть свои риски. Как бы то ни было, следует избегать наихудшего сценария, при котором прототипирование **FPGA** все еще находится на стадии устранения проблем проектирования, в то время как первые корпусированные чипы **ASIC** уже поступают с завода. Чтобы избежать проскальзывания графика, необходимо понимать риски.

A.1.2 Предостережения относительно прототипирования

Потенциальные пользователи прототипа **FPGA** стремятся получить его как можно скорее. В частности, команда разработчиков программного обеспечения может захотеть запустить тяжелое программное обеспечение (например, встроенный **Linux**) на прототипе, чтобы как можно раньше идентифицировать любые функциональные ошибки. Но прототипирование, начатое слишком рано (или включающее слишком много аппаратных средств), может привести к ряду проблем, на которых акцентировано внимание ниже.

Дождитесь зрелого RTL-кода

Прототипирование занимает значительное время, требует наличия умений и разработки отладочной платы. Эти усилия не должны быть потрачены впустую в попытках отладить незрелый, нестабильный или неполный **RTL**-код. Эффективнее выполнять отладку раннего **RTL**-кода, используя традиционные методы моделирования. **RTL**-код готов к прототипированию только тогда, когда отдельные логические модули проверены на уровне блоков, объединены вместе и способны передавать все тестовые векторы в базовом наборе тестбенчей. Следует дождаться зрелого кода, даже если разработчики программного обеспечения хотят получить прототип пораньше.

Не размещайте все модули

Некоторые проекты **ASIC** легче размещаются на устройствах **FPGA**. В качестве простого примера можно указать на то, что проект с высоким соотношением триггеров и логических элементов может легче уместиться на одном **FPGA**, архитектура которого, как правило, содержит много триггеров.

Не каждый аппаратный модуль на целевой **ASIC** всегда может отображаться на платформе **FPGA**; некоторые из них лучше всего прототипируются с использованием аппаратного обеспечения, внешнего по отношению к **FPGA**. Одним из примеров является встроенный макрос **RF CMOS**, который реализует малошумный усилитель или квадратурный модулятор, используемый в беспроводной связи. Другим примером служат физические (топологические) **IP**-ядра без **RTL**-кода или списка соединений для **FPGA**, такие как, например, блок канального уров-

ня (**PHY**) внутри трансивера **Ethernet**, являющийся схемой самого низкого уровня, которая управляет сериализованными данными на витой паре или оптоволоконном кабеле.

Подобные модули требуют дополнительных аппаратных устройств (коммерческий чип или отладочная плата от поставщика оборудования). Задача сопряжения этого внешнего оборудования с модулями ввода-вывода **FPGA** должна выполняться с осторожностью, поскольку **FPGA** содержат настраиваемые буферы ввода-вывода, позволяющие управлять уровнями напряжения и тока на пинах чипа.

Некоторые **ASIC**-блоки, если у них имеются кольца контактных площадок ввода-вывода, нельзя прототипировать на **FPGA**. Например, **ПО Quartus Prime** от **Intel FPGA** (**Altera**) по умолчанию обнаруживает только встроенные входы/выходы и не воспринимает инструкций по инстанциированию для ячеек ввода-вывода, имеющихся в библиотеке **ASIC**. Это также относится и к ячейкам библиотеки **ASIC**, созданным в **RTL**-коде, например к регистру, предназначенному для синхронизации с асинхронным источником данных.

Логика, предназначенная только для тестирования изготовленной и корпусированной **ASIC** на наличие производственных дефектов, также должна быть исключена из прототипа, поскольку он предназначен лишь для функционального анализа **RTL**-кода и встроенного программного обеспечения. Существуют более эффективные методы для проверки логики тестирования **ASIC** и для изготовления тестовых шаблонов с использованием логического моделирования.

Минимизируйте фрагментацию проекта

Предположим, что платформой для **FPGA**-прототипирования является чип **Intel FPGA MAX10** (**10M50DAF484C7G**). Он содержит **50** тыс. логических элементов (**LE**). **LE** – самый маленький модуль логики в этом семействе устройств. Каждый **LE** имеет четыре основных входа данных, достаточную логику для реализации любой функции этих входов и одного выходного триггера. Таким образом, один **LE** может выполнять работу десяти стандартных ячеек на целевой **ASIC**. Емкость чипа рассматриваемого **FPGA** эквивалентна приблизительно **500 000** логическим вентилям **ASIC**. Если целевой **ASIC** требуется **1** млн логических элементов, то для прототипирования понадобится два устройства **MAX10** (при предполагаемом полном использовании чипов, которое достигается редко). Это означает, что модули, описанные в **RTL**-коде, требуется распределить на два (или более) устройства. Каждое устройство должно иметь достаточное количество ресурсов (выводов, буферов тактовых сигналов, блоков **RAM**) для выполнения возложенных на него функций. Связь между двумя или более устройствами, имеющими много интерфейсов, может быть узким местом; следовательно, для решения задачи разбиения требуются значительные усилия, а полученный прототип может работать медленнее, чем хотелось бы.

Если весь проект осуществляется в рамках одной **FPGA**, задача прототипирования становится значительно проще.

Замечание 2: типичное использование ресурсов **FPGA** при проектировании составляет **75 %**. Для прототипирования лучше не превышать уровень использования ресурсов более **50–60 %**. Это оставляет место для изменений, облегчает задачу размещения и маршрутизации, а также сокращает время запуска проекта.

Поставщики инструментов **EDA** предлагают высококачественные, готовые к запуску прототипы **FPGA**, известные как эмуляторы. Эмулятор **Cadence** – это **Palladium**. Эмулятор для **ПО Synopsys** под названием **ZeBu** использует коммерческие **FPGA Xilinx**. **Mentor Graphics** предлагает применять эмулятор **Veloce**, который позволяет обрабатывать проекты с миллиардами логических элементов без разбиения на части, но он предназначен только для верификации и отладки проектов.

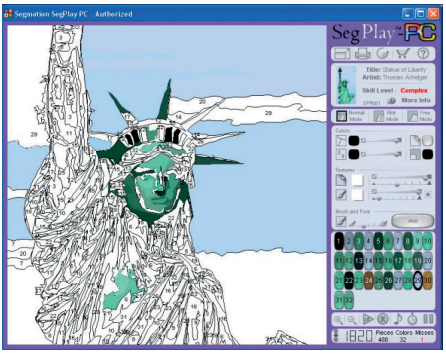
Поскольку эмуляция может сократить срок внедрения **ASIC**, часто нет необходимости создавать ранний прототип на **FPGA**.

A.2 Последовательность этапов проектирования ASIC

Как было указано ранее, устройство **FPGA** (например, **MAX10**) собирают из готовых элементов. В устройстве содержится до 50 тысяч логических элементов (**LE**); каждый – со своим триггером, а также имеется много килобайтов оперативной и флеш-памяти, множителей **18×18** и аналого-цифровых преобразователей (**ADC**). Необходимые ресурсы маршрутизации для соединения этих элементов встроены в устройство. Благодаря этому устройство может быть легко сконфигурировано для использования требуемых ресурсов на чипе.

Типичная **ASIC** на базе стандартных ячеек начинается с пустой матрицы. Она не имеет заранее созданной логики и соединительных линий. Неизвестен даже размер матрицы. Все аппаратные элементы должны быть размещены в чипе, включая **IP**-ядра (например, **MIPS M5150 CPU**). Таким образом, проходя через этапы проектирования **ASIC**, необходимо помнить о многих сопутствующих рисках:

1. Проект на **FPGA** (даже большой и сложный) намного более предсказуем, легче отлаживается и модифицируется.
2. **FPGA** с большей вероятностью будет работать правильно на первом же цикле разработки. В качестве упрощенной описательной аналогии процесс конфигурации **FPGA** может быть представлен в виде нумерованной раскраски на рис. A.2(a). Разработка **ASIC** – это словно рисование на пустом холсте (рис. A.2(a)). Команда разработчиков **ASIC** должна начинать с нуля, с оптимальной компоновки встроенных ресурсов – плана размещения. Этот план может потребовать много подробных итераций, прежде чем будет достигнута оптимальная схема маршрутизации на чипе.



а)

б)

Рис. А.2 Сравнение FPGA и ASIC

3. На кристалле **ASIC** нет встроенной схемы подключения. Разработчики должны решить, какие металлические слои использовать для распределения электропитания, синхронизации, прокладки соединений. После маршрутизации разработчики могут столкнуться с проблемами синхронизации. Исправление одного нарушения синхронизации часто влечет за собой другое. Разработка **ASIC** менее предсказуема, более итеративна, и при этом возникает больше сложностей при отладке. Используемые инструменты электронной автоматизации проектирования (**EDA**) более сложны, чем инструменты **Quartus Prime** или **Icarus**. Несмотря на более длительный цикл проектирования, новая **ASIC** с меньшей вероятностью будет работать после первого цикла разработки.

А.2.1 Схема разработки ASIC

На рисунке ниже приведены основные этапы проектирования **ASIC**, которое начинается с переработки файлов **RTL**-кода, написанных для прототипа FPGA. Для соответствия **FPGA MAX10** на данном рисунке указано, что разработка ведется для **65-нм CMOS**-технологии. Эта технология, как и ранее, применяется для создания многих чипов, таких как микроконтроллеры. Основные принципы логического и физического проектирования, рассмотренные в следующих разделах, за некоторыми исключениями, по-прежнему применяются к **32-**, **14-** и **7-нм** технологиям.

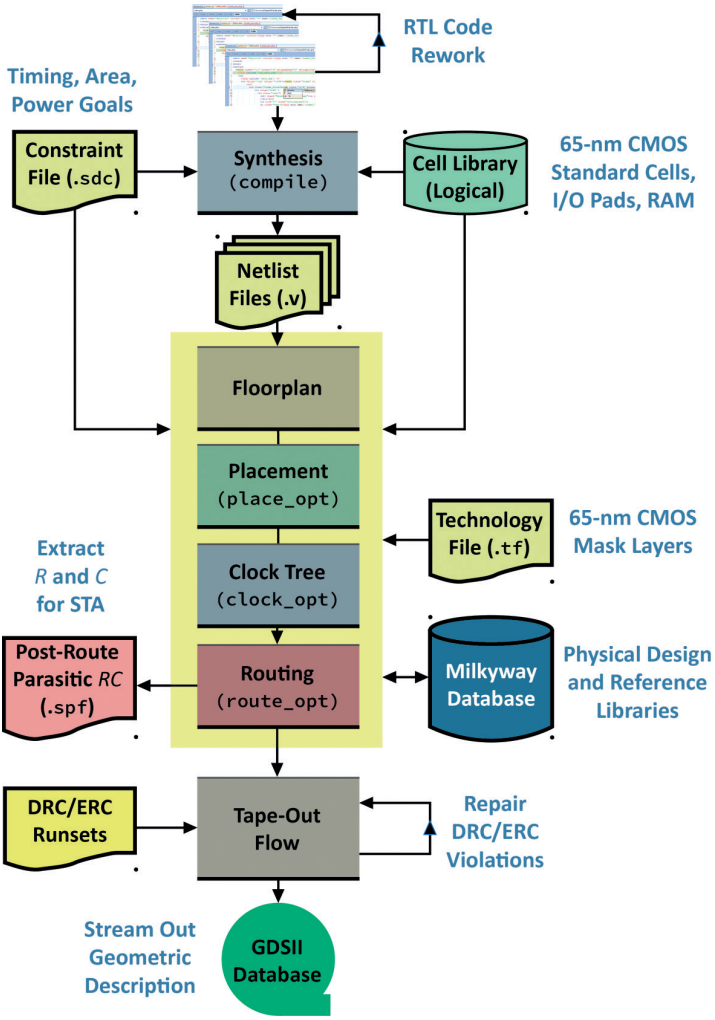


Рис. А.3 Последовательность этапов проектирования ASIC

Прямоугольники на рис. А.3 соответствуют этапам синтеза, размещения или маршрутизации с использованием коммерческих инструментов **EDA**. Файлы, считанные или записанные инструментами синтеза, изображены в виде значков документов. Библиотеки изображены в цилиндрических контейнерах.

Некоторые прямоугольники аннотируются круглыми скобками с командами ключевых инструментов, используемых при выполнении последовательной компиляции для синтеза. Эти **TCL**-команды относятся к инструментам **Synopsys**, таким как **Design Compiler**. Они отличаются в других средствах проектирования: например, в **Genius Cadence** команде **compile** соответствует команда **synthesize**. Далее рассмотрена вся последовательность этапов проектирования **ASIC**. При этом внимание обращено на возможные «подводные камни» и на то, как их можно обойти.

А.2.2 Усовершенствование проекта (design refinement)

Преодолевая в процессе проектирования этапы, изображенные на рис. А.3, полезно учитывать один руководящий принцип: на каждом шаге инструмент сводит описание проектирования более высокого уровня абстракции к описанию более низкого уровня абстракции. Инструменты синтеза, например, компилируют код уровня **RTL** в технологически зависимую схему уровня логических вентилей. Этот принцип актуален на каждом шаге процесса перехода от исходного кода **RTL** к базе данных **GDSII**. Данные **GDSII** – чрезвычайно низкого уровня, что сводит сложность **ASIC** до рисунка компоновки из простых геометрических фигур.

Другая ключевая концепция заключается в том, что многие этапы разработки **ASIC** – итерационные. На рис. А.3 не показаны все петли циклов разработки, чтобы избежать перегруженности рисунка, но в любом графике проектирования **ASIC** необходимо закладывать некоторый запас времени на каждый этап проектирования, чтобы были возможны итерации в течение каждого этапа проектирования. Так, ранее уже упоминалось, что этап размещения проекта на кристалле часто является итеративным.

В качестве другого примера предположим, что первый цикл синтеза не отвечает одному из указанных ограничений по синхронизации. В таком случае происходит повторная итерация этого цикла. Если нарушение невелико, его можно попытаться исправить перекомпиляцией кода с установленной настройкой более высокого уровня усилий компилятора, например путем указания параметра командной строки: **compile -map_effort high**.

Большое нарушение синхронизации, скажем **25 %** от периода тактового сигнала, может потребовать доработки **RTL**-кода для повышения скорости работы модуля. В главе 10 приведены примеры разработки **HDL**-кода для повышения скорости конечного устройства с использованием параллелизма и конвейерной обработки.

В крайнем случае **ASIC** можно перевести на более быстрый **CMOS**-процесс (для **ASIC** нет показателей быстродействия (**speed grade**), как для **FPGA**). Однако на заводе, где будет производиться ваш **ASIC**, помимо предлагаемого **CMOS**-процесса (скажем, **65-нм**), часто производят микросхемы и по более продвинутым технологиям. Например, **Europractice** предлагает широкий спектр технологий **GLOBALFOUNDRIES**, начиная со **180-нм**.

Важно отметить, что все нарушения синхронизации должны быть устранены до передачи в производство.

В следующих разделах (в соответствии с рис. А.3) рассмотрены этапы проектирования **ASIC** и даны описания используемых инструментов, входных и выходных файлов, преобразования проекта во время разработки, сделан акцент на том, как избежать ошибок, которые могут возникать при проектировании.

А.3 Этап логического синтеза

Этап логического синтеза изображен на рис. А.4. Он проходит на высоком уровне абстракции. Логические инструменты понимают только логические функции, а не физические свойства.

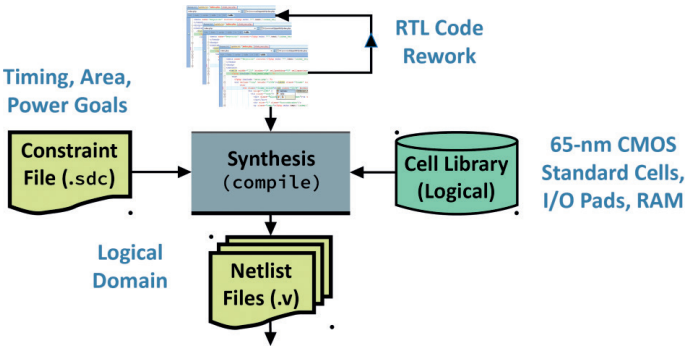


Рис. А.4 Этап логического синтеза

Рассмотрим простой пример, взятый из модуля **b1_mux_2_1_sel** в главе 4:

```
assign Y = sel? d1:d0;
```

Данная строка кода синтезируется как мультиплексор **2в1** из библиотеки стандартных ячеек (или в виде эквивалентной логики). Но инструмент синтеза не учитывает того, что находится на транзисторном уровне внутри этой стандартной ячейки, а также то, расположен ли выходной вывод **Y** на уровне **METAL1** или на правой либо левой стороне ячейки; каким образом обеспечивается питание и где стандартная ячейка будет располагаться на кристалле. Эти свойства лежат за пределами логической области. Они являются строго аппаратными и учитываются только инструментами физического проектирования.

Инструменты, работающие в логической области, служат только для проектирования топологии синтезированной схемы. **Design Compiler** точно знает, какие логические элементы или триггеры подключаются к выходу **Y** мультиплексора **2в1**. Он может легко отобразить схему синтезированной логики, по требованию разработчика размещая ее таким образом, чтобы выделить различные детали, но он не имеет доступа к геометрии топологии ячейки или координатам расположения ее контактов.

Только инструменты физического проектирования поддерживают команды, библиотеки микросхем и огромную базу данных проектирования, необходимую для преобразования логической схемы проекта в физический домен. В следующем разделе представлена верхняя часть рис. А.4. В нем рассмотрено, как адаптировать файлы **RTL**-кода, которые были разработаны для прототипа **FPGA**, после их переработки для **ASIC** с использованием технологии стандартных ячеек. Там же рассмотрена работа с исключениями.

Код уровня регистровых передач (RTL) записывается на относительно высоком уровне абстракции, в значительной степени не зависящем от целевого **FPGA**-у-

стройства или библиотеки стандартных ячеек **ASIC**. В нем описывается функциональность синтезируемой логики, в частности то, как изменяется состояние триггеров; основное достоинство такого подхода состоит в том, что проектирование осуществляется на высоком уровне абстракции.

Замечание 3: как правило, **RTL**-описание функционального логического блока записывается с использованием того же **Verilog**-кода, но без учета того, является ли платформа чипом **FPGA** или **ASIC**. Это относится к широкому спектру технологий, начиная с **65-нм**.

A.3.1 Переработка RTL-кода

Пример **simple_counter**, который приведен в главе 6, описывает двоичный счетчик. Он содержит цепочку из **WIDTH**-триггеров. Сам **RTL**-код описывает только логические функции счетчика. Он не содержит требований к настройке или хранению для отдельных триггеров; не указывает, должны ли триггеры быть реализованы с использованием **LE**, ячеек ввода-вывода или же блока **RAM**. Все эти детали оставлены на усмотрение инструмента синтеза **FPGA**.

Инструмент синтеза **ASIC**, такой как **Design Compiler**, предполагает исходный **RTL**-код того же уровня. Файл кода считывается путем заполнения диалогового окна или ввода команд в консоли. Синтаксис команд основан на **Tool Command Language** (**TCL**) – популярном языке сценариев, используемом **Synopsys** и другими поставщиками **EDA**. На этом базовом синтаксисе пользователю легче писать сценарии для других инструментов, а опытные пользователи предпочитают команды, которые могут быть объединены в сложные сценарии.

Рассмотрим пример команды:

```
dc> read_file -f Verilog *.v
```

Приведенная команда может быть первой строкой в сценарии синтеза **TCL**, поскольку **TCL** имеет встроенную команду чтения для дискового ввода-вывода **Linux**. Таким образом, **Design Compiler** определяет команду **read_file** для ввода файлов исходного кода **RTL**. Опция формата **-f[format]** обязательна, так как этот инструмент поддерживает другие языки, такие как **VHDL**. Также можно указать одно или несколько имен файлов.

Замечание 3 является общим правилом, но с некоторыми исключениями. Ниже приведено два общих исключения, а также практический метод кодирования для обработки особых случаев – без необходимости использовать несколько версий файлов.

Отсутствие блоков initial

Некоторые инструменты синтеза **FPGA**, такие как **Quartus**, могут задавать начальные состояния регистров или памяти, используя конструкцию initial языка **Verilog** (такие инструменты также могут задавать начальные значения из объявлений со встроенной инициализацией). Благодаря этой удобной функции инструмента обеспечивается соответствие состояния включения исходного **RTL**-ко-

да во время симуляции состоянию включения синтезированной конструкции в **FPGA**. Например, в главе 6 обсуждается регистр сдвига с линейной обратной связью (**LFSR**), используемый в разнообразных приложениях, таких как генерация псевдослучайных чисел и коды обнаружения ошибок. Блок кода **Verilog** при включении инициализирует **LFSR** с начальным значением **8'hA7** (приведен в листинге A.1).

```
//Initialize 8-Bit LFSR:
reg [7:0] LFSR;
initial
begin: SEED
   LFSR = 8'b1010_0111;
end
```

Листинг А.1 Регистр сдвига с линейной обратной связью

Это удобная функция, но она приводит к возможной ошибке. Не все инструменты синтеза **FPGA** поддерживают этот структурный компонент. Например, инструмент **Xilinx Vivado** игнорирует блоки **initial** во время синтеза и поддерживает только блоки **always**.

Исключение из замечания 3: инструменты синтеза **ASIC**, такие как **Design Compiler**, будут игнорировать блоки initial; подобные исключения должны быть условно скомпилированы, как показано ниже.

Избегайте RTL-синхронизации тактовых импульсов

Прототип **FPGA** – не лучшее средство для отладки логики, связанной с синхронизацией в **ASIC**. Большинство **FPGA** используют глобальные шины синхронизации, имеющие небольшую задержку. Эти шины распределяют сигналы синхронизации по всем триггерам в тактовом домене. **FPGA** также может иметь блоки управления тактовыми импульсами, которые выбирают или изменяют тактовые сигналы либо для экономии энергии на время отключают тактовый домен.

В **ASIC**, разработанных для низкого потребления энергии, используют элементы снижения электропотребления, распределенные по кристаллу, например логику запрета подачи тактовых импульсов (**clock gating**). Запрет тактовых импульсов уменьшает динамическую мощность – энергию, потребляемую при переключении, – но не влияет на статическую мощность из-за утечки тока даже при отсутствии активности в электрической цепи.

На рис. A.5 изображена ячейка, управляющая запретом тактовых импульсов и служащая для снижения активности в цепочке триггеров. В соответствии с логикой работы устройства триггеры должны сохранять арифметический результат при каждом третьем цикле из трехступенчатой конвейерной передачи данных. Запрет синхросигнала **CLK** предотвращает внутреннее переключение триггеров во время двух безразличных циклов, когда они сохраняют неверные данные. Это уменьшает потребление динамической мощности. Для чипа, в котором используется значительное количество триггеров, общее снижение мощности может быть весьма существенным.

Логический элемент **AND** на рисунке не дает сигналу **CLK** достичь этих триггеров до третьего цикла, когда новый результат будет корректным. Только тогда синхронизированный тактовый сигнал **GCLK** достигает банка триггеров. Такой подход позволяет снизить активность на **67 %**. Однако использование логического элемента **AND** само по себе может приводить к возникновению сбоев. Обратите внимание на интервал времени с красной полосой на рис. А.5, в котором оба сигнала **DAV** и **CLK** имеют (на мгновение) высокий уровень. Данный перехлест приводит к фатальному сбою – возникновению ложного импульса **GCLK**, поступающего на синхровходы триггеров.

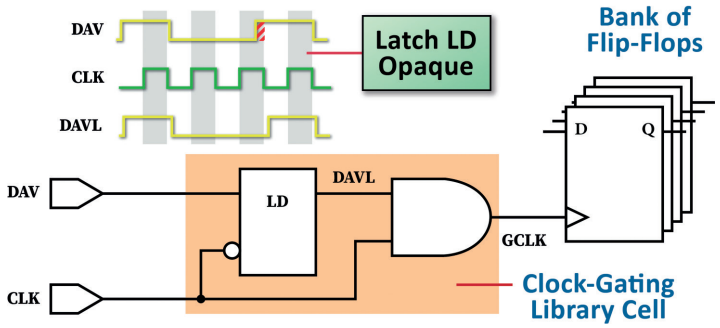


Рис. А.5 Управление запретом тактовых импульсов для ASIC

Для предотвращения сбоев логическому элементу **AND** предшествует прозрачная защелка. Комбинированная ячейка управления запретом тактового сигнала (имеет оранжевый цвет фона) присутствует в любой библиотеке ячеек, поддерживающих снижение энергопотребления. Следует отметить, что защелка остается непрозрачной при высоком уровне сигнала **CLK**. При этом в зонах, выделенных серым цветом, сигнал **DAVL** является стабильным и не может изменяться. Таким образом, защелка предотвращает сбои, в то время как логический элемент **AND** отключает ненужные переключения тактового сигнала.

Управление запретом тактовых импульсов может использоваться и для **FPGA**, однако применение такого подхода не рекомендуется при высокой тактовой частоте. Вставленный логический элемент **AND** увеличивает расфазировку тактовых сигналов, что противоречит назначению цепей тактирования в **FPGA**. Чтобы избежать ненужной переделки проекта, логика синхронизации должна быть исключена из **RTL**-кода. Инструмент синтеза (рис. А.4) может вставить ячейки библиотеки, обеспечивающие управление запретом тактовых сигналов, с помощью использования команды

```
compile -gate_clock
```

Инструмент автоматически помещает специализированные блоки запрета тактового сигнала вместо цепей, использующих вход разрешения работы **D**-триггера. Это позволяет снизить энергопотребление схемы, оставляя ее **RTL**-описание без изменений.

Использование условной компиляции

Verilog поддерживает условную компиляцию сегментов кода. Можно легко изолировать команды **RTL**, которые принадлежат только чипу **ASIC**, а не прототипу **FPGA**. Простой пример использования такого подхода приведен ниже. Этот код включает в себя несколько директив препроцессора **Verilog** с префиксом **backtick (`)**, обозначенных серым цветом:

```
//Conditional code:
`ifdef ASIC
   //ASIC Clock Divider:
   CLK_DIVx10 U_DIV(REF_IN, CLK_OUT, RST, ...);
`else
   //FPGA Clock Manager:
   MMCM U_MMCMx10(REF_IN, CLK_OUT, RST, ...);
`endif
```

Листинг А.2 Условная компиляцию кода Verilog

Код, заключенный в эти условные директивы, будет компилироваться по-разному в зависимости от того, определен ли в настоящее время макрос с именем **ASIC** (макрос не должен иметь текстового содержимого, только имя). Если он определен, на чипе создастся конкретный **ASIC**-блок делителя частоты. Если нет, то на устройстве **FPGA** будет создан специфичный для **Xilinx** экземпляр блока управления тактовым сигналом со смешанным режимом (**MMCM**), настроенный для деления на **10**.

После этапа предварительной обработки кода компилятором (**preprocessing**) осуществляется компиляция только одной команды **Verilog** (выделено синим). Другая часть кода игнорируется. Для облегчения определения макроса с именем **ASIC** для использования на этапе синтеза (рис. А.4) в команде **Design Compiler**, читающей **RTL**-файлы, имеется возможность определить необходимую опцию:

```
dc> read_file … -define ASIC *.v
```

Эта команда сделает активным код, создающий экземпляр модуля **CLK_DIVx10**, который впоследствии будет синтезирован. Используя эти директивы препроцессора, можно обрабатывать различные ситуации, в которых для этапов прототипирования и синтеза нужны разные фрагменты кода, и при этом не поддерживать две версии **RTL**-кода.

А.3.2 Настройка библиотеки

Инструмент синтеза логики, такой как **Design Compiler**, использует библиотеку стандартных ячеек для преобразования команд **Verilog**. Например, приведенная ниже команда, описывающая единичный счетчик, синтезируется как ячейка полного сумматора:

```
cnt <= cnt + 1'b1;
```

На рис. А.4 библиотека стандартных ячеек представлена в виде цилиндра. На практике библиотека ячеек может быть разделена на несколько файлов: для ло-

гических ячеек, для площадок ввода-вывода, а также для блоков двухпортовой **RAM** различных размеров. Библиотека стандартных ячеек создается на заводе группой разработчиков библиотеки специально для конкретного **CMOS**-процесса. Характеристики (атрибуты) ячеек внутри библиотеки тщательно проверяют на точность.

Кратко рассмотрим данные, имеющиеся в простой библиотеке ячеек. **Design Compiler** может (теоретически) синтезировать практически любую цифровую схему, используя минимальную библиотеку логических вентилей **AND**, **OR** и **INV** с **D**-триггером. Код библиотеки, приведенный ниже, описывает простую ячейку инвертора, у которой кратко указаны три атрибута. Логическая функция инвертора – **A'** (не **A**). Его площадь на кремнии (указанная в квадратных микронах или в других единицах библиотеки) равна **1**. Его собственная (или ненагруженная) задержка от вывода **A** до вывода **Z** составляет **0,15 нс** для заднего фронта сигнала. Для библиотеки, учитывающей энергопотребление, атрибуты будут также включать статическое и динамическое энергопотребление.

```
/* Liberty Format
* Cell Attributes
*/

cell(INV) {
   area: 1;
   pin(A) {
      direction: input;
      capacitance: 1.0;
   }
   pin(Z) {
      direction: output;
      function: A';
      timing() {
         intrinsic_rise: 0.38;
         intrinsic_fall: 0.15;
         . . .
         related_pin: A;
      }
   }
}
```

Листинг А.3 Пример описания инвертора в библиотеке стандартных ячеек

Подобные описания стандартных ячеек находятся в исходном текстовом файле с суффиксом **.lib** (данный формат синтаксиса имеет название **Liberty**). Как правило, доступ к файлам **.lib** имеется только на заводе. Команда разработчиков получает скомпилированный двоичный файл библиотеки с суффиксом **.db**. Он является частью набора для проектирования технологического процесса производства **ASIC**, поставляемого заводом (**Process Design Kit**, **PDK**). Любая библиотека также включает общебиблиотечные атрибуты, такие как физические единицы измерения площади, задержки и мощности.

Чтобы настроить двоичный файл библиотеки для использования при синтезе, пользователь инструмента должен ввести команду **TCL** в командной строке **Design Compiler** или включить ее в сценарий настройки **TCL**:

```
dc> set target_library sc_65nm_max.db
```

Как только **Design Compiler** загрузит этот файл библиотеки, он сможет быстро найти атрибуты ячейки. Это то, что позволяет инструменту переводить код **RTL** в логические функции. Операция Verilog, такая как **~EN**, может быть реализована с помощью ячейки **INV**, и инструмент может быстро оптимизировать логику, обеспечив баланс между задержками прохождения сигналов и площадью, занимаемой ячейками.

Хотя **Design Compiler** может обойтись минимальной библиотекой, лучше всего иметь большой набор ячеек. Богатая на различные варианты ячеек библиотека обычно содержит: логические элементы, буферы, сумматоры, вычитатели, мультиплексоры, защелки и триггеры. Каждый тип ячеек имеет разные размеры для обеспечения наилучшей производительности.

Замечание 4: как правило, чем богаче библиотека ячеек, тем лучше качество результатов (**Quality of Results**, **QOR**) при синтезе, а значит – производительность чипа.

QOR отображает отчет о результатах работы **Design Compiler**, который характеризует, насколько достигнуты цели проектирования, связанные с увеличением скорости, уменьшением энергопотребления и занимаемой площади.

Стандартные ячейки также имеют и другие библиотечные атрибуты, которые важны для инструментов синтеза **ASIC**. Каждый входной контакт ячейки представляет нагрузочную емкость для логики, управляющей им. Например, входной контакт **A** ячейки **INV** имеет емкость **1,0 пФ**. Зарядка этой емкости до логической **1** (или разрядка до **0**) в течение разумного времени требует приложения соответствующего уровня тока. Способность логики обеспечить этот уровень тока является его силой тока. Как правило, использование более высокой силы тока требует более широкой компоновки ячеек. При учете этих атрибутов инструменты синтеза могут обеспечить баланс скорости и площади проекта.

Design Compiler может увеличивать размер ячейки на критическом пути – ячейки занимают больше места, но быстрее управляют нисходящей логикой. Это обеспечивает возможность работы **ASIC** на более высокой частоте. С другой стороны, инструмент синтеза может уменьшить размер ячейки, лежащей вне критического пути, и уменьшить площадь, а также, возможно, и электропотребление за счет скорости. Стандартные ячейки в расширенной библиотеке могут иметь несколько вариантов реализаций с силой тока **x1**, **x2**, **x4**, **x8**....

A.3.3 Ограничения по проектированию (design constraints)

Работа инструментов синтеза во многом определяется заданными ограничениями. Синтезирующий **САПР** не только переводит исходный код **RTL** в цифровую

логику, но также и оптимизирует ее в попытках соответствовать детальным проектным ограничениям. Эти ограничения обычно считываются из файла сценария **TCL**. На рис. А.4 приведен сценарий **Synopsys Design Constraint** (**.sdc**), который может включать подробную информацию об ограничениях по частоте работы схемы, площади кристалла и потребляемой мощности. Существует возможность также задавать уровень напряжения внешней логики, управляющей входным портом или емкостью нагрузки на выходном порту.

Приведенная ниже команда **TCL** – простой пример ограничения частоты работы схемы. Она может быть включена в сценарий или введена в командной строке **Design Compiler**. Данная команда задает целевую тактовую частоту **186,25 МГц** входного порта **CLK**. Соответствующий целевой тактовый период равен $1/f$, или **5.37 нс**:

```
dc> create_clock -period 5.37 -port CLK
```

В этом случае эффективный период тактирования составляет **5,37 нс**, что должно быть меньше, чем время установки регистра назначения (в этом простом примере игнорируется время установки). Инструмент синтеза оптимизирует логику между стадиями конвейера, используя сначала независимые от технологии методы оптимизации, чтобы уменьшить количество логических уровней. Оптимизация продолжается также на уровне логических элементов – путем применения методов отображения ячеек. Примером простой методики отображения является применение **законов де Моргана**, превращающих логику **AND-OR** в более быструю **NAND-NAND**. Могут использоваться и другие методы. Если результирующая задержка распространения через оптимизированную эквивалентную логическую схему меньше, чем эффективный период тактовых импульсов, то ограничение по синхронизации будет выполнено. Разница между требуемой задержкой распространения сигнала и реальной задержкой называется слэк (**slack**, запаздывание сигнала). Он может быть отрицательным или положительным. Если задержка распространения вдоль пути прохождения тактового сигнала превышает эффективный тактовый период – это значит, что возникает отрицательный слэк и проект нарушает ограничения по тактовой частоте. Все такие нарушения должны быть устранены до передачи проекта из логического домена в физический (стратегия агрессивного проектирования может нарушать требования по временным параметрам во многих путях прохождения сигналов, но в конечном итоге нарушения все равно должны быть устранены на следующих этапах проектирования).

Таким образом, инструмент синтеза не всегда может удовлетворять заданным ограничениям по тактовой частоте. Единственный способ узнать, будут ли выполняться данные требования, – это синтезировать **ASIC**-проект. В следующем разделе подробно рассматривается этап синтеза. Если в предыдущих разделах было представлено много полезных примеров проектирования – от сумматоров до счетчиков и целых конвейеров данных, – то теперь будут раскрыты методы, используемые для эффективного синтеза больших иерархий модулей и подмодулей.

А.3.4 Синтез иерархии проекта

После того как соответствие требуемым ограничениям достигнуто, необходимо синтезировать описание **RTL** на уровне логических вентилей. Базовая команда **compile** ПО **Design Compiler** приведена в прямоугольном блоке на рис. А.4. Эта команда предназначена для рекурсивного выполнения работы, начиная с текущего проекта и заканчивая всеми его подмодулями более низкого уровня:

```
dc> compile -map_effort medium
```

По умолчанию опция, отвечающая за эффективность выбора логических элементов из библиотеки, установлена в значение **medium** (среднее) и может быть опущена. Точкой входа, с которой начнется процесс компиляции, должен быть выбран модуль верхнего уровня иерархии. Это может быть модуль самого высокого уровня в микросхеме или любой другой подмодуль в иерархии. Хотя эта команда может обрабатывать модуль любого размера, время выполнения синтеза быстро возрастает с увеличением числа логических элементов. Из этого следует следующее замечание.

Замечание 5: как правило, команда **compile** лучше всего работает с модулями размером от 500 до 200 тыс. логических элементов. Если число их слишком мало, то инструмент не всегда может объединить логические элементы. Если их количество слишком велико, то время выполнения становится чрезмерно большим.

Если разрабатываемая **ASIC** занимает более **1** млн логических элементов, в таком случае следует выбрать одну из стратегий, описанных ниже.

Компиляция сверху вниз (top-down)

Эта стратегия намного проще, чем другие, но может занять много времени. Следует начать с установки ограничений к проекту только на самом верхнем уровне, после чего выполнить компиляцию всей иерархии проекта сверху вниз. В итоге, скорее всего, будет достигнуто хорошее **QOR**, так как инструмент может компилировать каждый модуль в контексте его логического окружения, но время компиляции может оказаться неприемлемо большим. В простом примере из раздела А.3.2, где порт **OUT1** модуля управляет одной стандартной ячейкой простого **INV**, на основании своих библиотечных атрибутов команда compile определяет емкость нагрузки, представленной входным контактом **A** инвертора (**1,0 пФ**). На основании этого для порта **OUT1** инструмент выберет логику соответствующей силы тока.

Компиляция снизу вверх (bottom-up)

Эта стратегия более сложная и требует использования сценариев, но зато время выполнения компиляции более контролируемо. При такой стратегии следует начать с анализа больших модулей. К ним применяют предварительные оценочные ограничения. Например, если ожидается, что порт **OUT1** будет управлять простым инвертором, то в файле ограничений должна быть указана емкостная нагрузка:

```
dc> set_load 1.0 OUT1
```

Это только предварительная оценка. Логика управления затем компилируется для обработки расчетной нагрузки, установленной на более реалистичное значение, чем нагрузка по умолчанию (**0,0 пФ**). Поскольку одновременно компилируется только один модуль, время выполнения невелико. Затем к скомпилированному модулю применяется атрибут **dont_touch**, и такая процедура повторяется для всех модулей.

После компиляции всех модулей по принципу снизу вверх компиляция всего чипа завершена. Атрибут **dont_touch** запрещает вносить изменения в уже скомпилированные модули. Таким образом, нисходящая компиляция может быть выполнена за значительно меньшее время.

В распоряжении ПО **Design Compiler** также имеются еще более продвинутые методы, позволяющие улучшить первичные оценки ограничений проекта и уменьшить время компиляции. Их рекомендуется использовать для очень больших чипов.

Постепенная компиляция

В случае если весь чип был скомпилирован с использованием предыдущих стратегий, но один путь или несколько все же нарушают временные требования к проекту, то один из действенных методов – постепенный повторный запуск команды compile с более высоким уровнем оптимизации:

```
dc> compile -inc -map_effort high
```

Этот высокий уровень оптимизации заставляет инструмент работать в полную силу и перестраивать логику внутри и вокруг длинного пути, нарушающего временные требования, которые предъявляются к проекту.

Параметр -inc ограничивает перестройку текущей логики блока без изменения ее общей архитектуры. Это безопасный вариант, поскольку изменения сохраняются только в том случае, если уменьшаются задержки пути. Это переназначение лучше всего подходит для библиотеки с большим количеством вариантов базовых ячеек. Включение режима повышенной оптимизации без опции -inc может привести к тому, что инструмент полностью перекомпилирует проект с высокой загрузкой процессора.

В рассматриваемом методе использовались две последовательные команды компиляции: одна запускалась с настройками по умолчанию, а вторая – в режиме повышенной оптимизации. Распространенная ошибка новичков – запуск последовательных компиляций с настройками по умолчанию в надежде на улучшение **QOR**. Как правило, это непродуктивно, поскольку **QOR** во многом зависит от начальной точки, с которой началась компиляция. Синтезированный проект с нарушением временных требований не является хорошей отправной точкой; обычно лучше начинать проектирование заново, анализируя исходный **RTL**-код, уточняя ограничения и пробуя различные варианты компиляции.

А.3.5 Оценка результатов компиляции

По завершении проектирования следует оценить итоговое качество результатов **QOR**. При этом проверка схематики иерархической микросхемы может не дать достаточных результатов. Поэтому в **Design Compiler** имеется много информативных команд для создания отчетов с детальным описанием временных характеристик, занимаемой площади и энергопотребления синтезированного проекта. Чтобы запросить сводный отчет, достаточно вызвать команду

```
dc> report_QOR
```

В нем приводятся обобщенные результаты: время проектирования, площадь, время выполнения и другая статистика, связанная с компиляцией.

Поскольку оптимизация временных параметров схемы имеет более высокий приоритет, она указана первой. Можно сразу определить, были ли соблюдены целевые ограничения на тактовый сигнал и существуют ли какие-либо нарушения. Типичная выдержка из отчета по **QOR** приведена ниже:

```
Timing Path Group 'default'
-----------------------------------
Levels of Logic:  6.00
Critical Path Length:      5.36
Critical Path Slack:       0.01
Critical Path CLK Period:  5.37
Total Negative Slack:      0.00
No. of Violating Paths:    0.00
--------------------------------

Группа тракта тактирования «по умолчанию»
-----------------------------------
Уровни логики:  6.00
Длина критического пути:   5.36
Слэк критического пути:    0.01
CLK-период критического пути:     5.37
Сумма всех слэков:   0.00
Количество путей, нарушающих временные требования:    0.00
--------------------------------
```

Листинг А.4 Пример отчета по QOR

Данная выдержка сообщает о критическом пути, который ограничивает скорость работы синтезированной логики. Здесь ограничение обусловлено шестью уровнями логики (на пути прохождения сигнала имеется цепочка из шести логических элементов, соединенных последовательно). Общая задержка распространения сигнала составляет **5,36 нс**. К счастью, тактовый период в **5,37 нс** длится больше, т. е. имеется положительное запаздывание, составляющее **0,01 нс**. Таким образом, данный путь удовлетворяет временным ограничениям, но так бывает далеко не всегда.

В большой **ASIC** существуют миллионы путей прохождения сигналов. В качестве небольшого примера можно рассматривать неконвейеризированный проект, приведенный в главе 10. Существует множество путей через логические облака **F**, **G** и **H**, различающиеся величиной слэка. Некоторые из них могут превышать эффективный тактовый период и нарушать временные параметры схемы. Такие инструменты, как **Design Compiler**, позволяют строить гистограмму (рис. А.6), каждый столбец которой содержит количество путей, найденных в заданном диапазоне слэка.

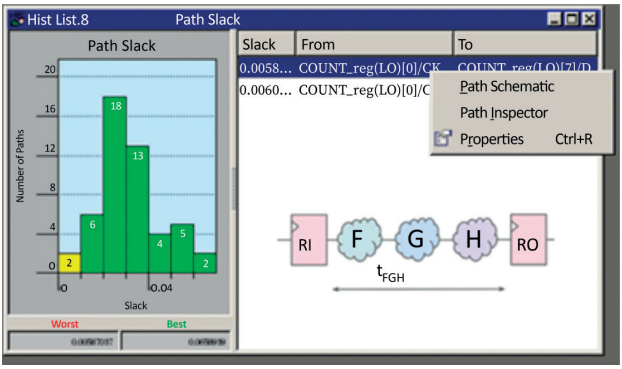


Рис. А.6 Пример гистограммы слэков путей в Design Compiler

На рисунке два пути имеют слэк **0,01 нс** или менее (соответствующий столбец обозначен желтым). Их конечные точки перечислены в правой панели. Шесть путей имеют слэк от **0,01** до **0,02** (зеленый столбец) и т. д. Такая гистограмма полезна для оценки прогресса в улучшении временных параметров крупного проекта **ASIC**.

Удобной метрикой для оценки проекта с большим количеством нарушений временных параметров является также сумма всех отрицательных слэков (**TNS**), которая указана в приведенном выше отчете по **QOR** (листинг А.4). В большой **ASIC** могут содержаться миллионы отдельных путей задержки, однако все временные цели считаются достигнутыми при **TNS**, равной **0**.

Следует понимать, что в данном разделе речь идет только о логическом домене. Логические элементы в этих логических облаках (рис. А.6) еще не размещены и не маршрутизированы. В лучшем случае на данном этапе **Design Compiler** осуществляет лишь статистическую оценку. Интервал $\mathbf{t_{FGH}}$, приведенный на рисунке, включает только задержки в соединениях. При этом оценка этих задержек осуществляется приблизительно.

После размещения компонентов и трассировки соединений будут рассчитаны более точные задержки соединения (рассмотрены в разделе А.4, где идет речь о физическом домене), в результате чего можно столкнуться с непредвиденными тактовыми нарушениями из-за неожиданно длинного маршрута. Исправление этих задержек может потребовать нескольких итераций, пока, наконец, нарушения не будут устранены. Тем не менее рекомендуется, чтобы проект удовлетворял всем временным ограничениям уже на стадии логического синтеза, поскольку инстру-

менты размещения и маршрутизации способны выполнять ограниченный синтез и не предназначены для синтеза всей иерархии проекта.

Ситуация, когда синтезированный дизайн **ASIC** соответствует временным ограничениям, а **TNS** равен нулю, приведена в нижней части рис. А.4. На этом этапе логическое представление проекта передается дальше инструментам физического проектирования. Традиционный формат передачи – это набор файлов списка соединений, выполненный на **Verilog-95**.

А.3.6 Список соединений на Verilog, передаваемый на уровень физического синтеза (hand-off Verilog netlist)

Когда синтезированный модуль соответствует всем ограничениям, пользователь может представить его в виде списка соединений на **Verilog**. Список соединений – это текстовое описание схемы, которое является средством передачи проекта от логического домена к физическому домену.

Пример файла списка соединений для небольшого проекта показан ниже. Он состоит из логических элементов и триггеров, специфичных для библиотеки стандартных ячеек и выполненных по **65-нм КМОП**-технологии. Список соединений традиционно записывается в более старом формате на **Verilog-95**. В **Verilog-95** каждый порт, указанный в круглых скобках, должен быть затем объявлен внутри модуля (выделено синим цветом). Порт **PI1**, например, является однобитным входным портом, имеющим стандартный тип **wire** по умолчанию:

```
/* Typical Netlist
* in Verilog-95 Format
*/
module NETWORK_3FF( . . . );
   output PO1, PO2;
   input PI1, PI2, PI3;
   input CLK, RSTn, test_si, test_se;
   wire Q1, N1, n5, n7;

   SDN_FSDPRBQ_1 Q1_reg(
      .D(PI1),.SI(test_si),.SE(test_se),
      .CK(CLK),.RD( RSTn),.Q(Q1)
   );
   . . . .
   SDN_AN2_1 U10(.A1(Q1), .A2(PI2), .X(N1));
endmodule
```

Листинг А.5 Пример файла списка соединений

Код, выделенный красным цветом, является серией реализаций модулей на **Verilog**. Например, **SDN_AN2_1** представляет собой двухвходовый библиотечный логический элемент **AND** с мощностью вывода **1**, инстанцированный в качестве экземпляра модуля под именем **U10**. Его вход **A1** управляется соединением **Q1** с выходом **Q** триггера. Вход **A2** управляется входным портом **PI2**.

Когда структурное описание на **Verilog**, подобное приведенному в примере, передается инструментам физического проектирования, им известна каждая стандартная ячейка, которую необходимо поместить на кристалл. При этом задача размещения этих ячеек, а затем и маршрутизации их логических взаимосвязей является сложной. В физическом домене единичное соединение, такое как **Q1**, может быть реализовано несколькими металлическими слоями с изгибами или переходами между слоями. Размещение компонентов и трассировка соединений рассмотрены в следующем разделе.

Замечание 6: следует избегать распространенной ошибки, свойственной новичкам, – при сохранении файлов списка соединений на уровне логических элементов на **Verilog** следует использовать расширение файла **.vg**, а не **.v**. Это позволяет избежать случайной перезаписи файла исходного кода **RTL** с тем же базовым именем, а также делает его более узнаваемым для других членов команды разработчиков.

Другой ключевой файл – это сценарий ограничений проекта. В следующем разделе будут рассмотрены физический домен и основные этапы проектирования. Средством физического проектирования у **Synopsys** является **IC Compiler**; у **Cadence** – **Innovus**. Точный синтаксис команд варьируется от одного инструмента к другому, но ключевые шаги в процессе физического проектирования – планирование, размещение и маршрутизация – одинаковы для обоих инструментов.

После того как синхронизированный проект синтезирован и подготовлен список соединений на уровне логических элементов на **Verilog**, необходимо переходить от логической области к физической. Данный этап представляет собой более низкий уровень абстракции. Каждый шаг выполняется с учетом особенностей кристалла. На этом уровне проектирование **ASIC** становится намного более детальным. Инструменты учитывают как физические, так и электрические характеристики – например, расположение стандартной ячейки, размещенной на кристалле; следовательно, и база данных проекта становится намного больше.

А.4 Этап физического синтеза

На рис. А.7 изображен процесс разработки **ASIC** в физическом домене. Данный этап сводится к размещению компонентов и трассировке соединений. При этом, как будет показано ниже, инструменты постоянно усложняются. В частности, необходимо осуществлять дополнительные шаги для решения проблемы наноразмерных технологий.

О сложности используемых на данном этапе инструментов позволяет судить следующее: в то время как **Design Compiler** имеет около **800** команд, **IC Compiler** поддерживает уже **1200**. Последний фактически представляет собой целый набор инструментов для работы с кристаллом (выделены желтым на рис. А.7). При этом большинство пользователей применяют лишь небольшое подмножество из всего набора команд. На рисунке показаны всего три основные команды **IC Compiler**; они будут рассмотрены далее.

Инструменты физического проектирования должны быть глубоко интегрированы. Во времена полупроводников команда разработчиков приобретала один инструмент для поуровневого планирования, другой – для размещения и еще один – для маршрутизации. В настоящее время работа с наноразмерными транзисторами и металлическими межсоединениями требует использования глубоко интегрированных наборов инструментов, которые могут управлять взаимодействиями между всеми компонентами на физическом уровне (рис. А.7).

В качестве примера нежелательных взаимодействий рассмотрим поуровневое планирование аналогового макроса (например, малошумящего усилителя) на кристалле. Его расположение накладывает ограничения на маршрутизацию шин. Шумные цифровые шины не следует прокладывать поверх чувствительных аналоговых макросов, для чего при маршрутизации должна быть выбрана опция уменьшения перекрестных помех. Управление многими такими тонкими взаимодействиями на большом чипе и требует высокоинтегрированного набора инструментов.

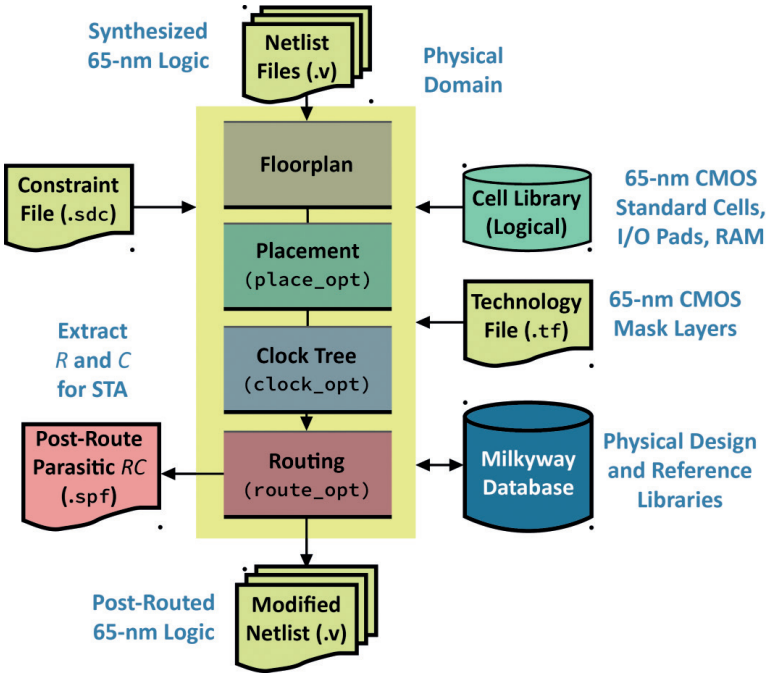


Рис. А.7 Физический синтез

А.4.1 Настройка библиотек

Все инструменты в интегрированном наборе должны взаимодействовать с одной и той же базой данных проектирования. База данных физического проектирования **Synopsys** представляется в формате **Milkyway**. В дополнение к физическим библиотекам **IC Compiler** должен также иметь доступ к той же логической библиотеке стандартных ячеек, которая использовалась при синтезе. Все библиотеки за-

висят от технологии и должны быть предоставлены заводом – например, в виде комплекта **Process Design Kit** (**PDK**) вместе с правилами и другими данными, относящимися к процессу проектирования. Это гарантирует точное соответствие ячеек заявленным характеристикам.

Настройка логической библиотеки

После загрузки библиотеки стандартных ячеек в **IC Compiler** (рис. А.7) инструмент быстро находит ключевые атрибуты, такие как площадь ячейки, задержка между выводами или емкость входных контактов, для любой стандартной ячейки.

Рассмотрим простой пример. Предположим, что инструменту нужно вставить буфер в путь маршрутизации сигнала. Напряжение питания буфера позволяет улучшить синхронизацию длинного пути, страдающего от большой емкостной нагрузки. Минимальный буфер можно получить, вставив две ячейки **INV** последовательно. На основе временных характеристик **INV** инструмент определяет, что общая внутренняя задержка через эту неинвертирующую пару составляет **0,38 + 0,15**, или **0,53 нс**. Также известна их общая площадь ячейки, равная **2**. Это позволяет инструменту быстро и точно обеспечивать баланс между скоростью и занимаемой площадью.

IC Compiler имеет доступ к той же библиотеке ячеек, что и **Design Compiler**, и он также может синтезировать логику. Например, при маршрутизации сигнала инструмент будет вставлять буферы по мере необходимости в длинные соединения или увеличивать ячейки вдоль критического пути. Следует отметить, что хотя **IC** Compiler может выполнять ограниченный логический синтез, он не предназначен для синтеза или повторной оптимизации всей иерархии **ASIC**.

Замечание 7: синтезированный проект **ASIC**, переданный в IC Compiler, уже должен быть синхронизированным – т. е. иметь **TNS = 0,0 нс**. Это не значит, что нарушения синхронизации при размещении или маршрутизации не могут возникать даже в синтезированном проекте. Нарушения могут происходить из-за плохого планирования, недооцененных задержек маршрутизации и других причин. Для больших **TNS** может потребоваться повторная итерация. Маршрутизируемый проект возвращается на уровень логического синтеза. На этот раз – с использованием обратно аннотированных данных маршрутизации, извлеченных **IC Compiler** (этот процесс рассмотрен в разделе А.5.1). Имея доступ к реальным задержкам прохождения сигналов для критических путей, **Design Compiler** сможет лучше оптимизировать результирующую схему.

Настройка физической библиотеки

Физическая библиотека намного сложнее, чем логическая библиотека. Она описывает геометрию каждой стандартной ячейки, включая прямоугольную границу ячейки (ограничивающий прямоугольник, **bounding box**), расположение входных и выходных выводов внутри этого прямоугольника, а также конкретные формы и слои маски каждого геометрического элемента в ячейке. Все эти физические атрибуты необходимы для того, чтобы **IC Compiler** мог осуществлять эф-

фективное планирование, размещение компонентов и трассировку соединений при проектировании большой **ASIC**.

На рис. А.8 изображена типичная стандартная ячейка с полным сумматором **ad01d0** с **28** транзисторами. Обратите внимание на то, что она является широкой и узкой по высоте. Стандартные ячейки в библиотеке стандартизированы по высоте. Это позволяет многим соседним ячейкам, расположенным в длинном ряду, иметь общую шину **VDD** (источник питания), проходящую через верхнюю часть рисунка, а также общий канал **VSS** (земля), проходящий через нижнюю часть. Эти общие шины (**METAL1**) составляют шину питания. Поскольку высота стандартизирована, более сложные ячейки – такие как этот полный сумматор – имеют большую ширину, чем более простые ячейки – такие как инверторы.

В качестве одного из примеров использования геометрических атрибутов **IC Compiler** определяет ограничивающую рамку каждой стандартной ячейки. Она может примыкать к соседним ячейкам (как вагоны в поезде) вдоль ряда. Эти ряды эффективно используют доступную площадь кристалла. В качестве другого примера **IC Compiler** определяет точное местоположение и металлический слой входных и выходных выводов полного сумматора. На рис. А.8 вывод суммы **S** ячейки выделен желтым цветом. При таком низком уровне абстракции вывод является металлической площадкой, к которой во время маршрутизации может быть осуществлено подключение. Вывод **S** находится на самом нижнем металлическом слое **METAL1**, что демонстрирует синяя заштрихованная область на рисунке. При маршрутизации **IC Compiler** соединит этот вывод с последующей логикой. На рисунке показана белая стрелка, выходящая из вывода. Поскольку в этой компоновке ячеек нет других металлических слоев, маршрутизатор может использовать слой **METAL2** для создания соединения. При этом нет риска того, что маршрутизируемое межсоединение окажется слишком близко к другим элементам на слое **METAL2** внутри ячейки. Это возможно потому, что инструмент знает обо всех соединениях, поскольку на предыдущем этапе ему были переданы файлы списка соединений на **Verilog**, как показано в верхней части рис. А.7.

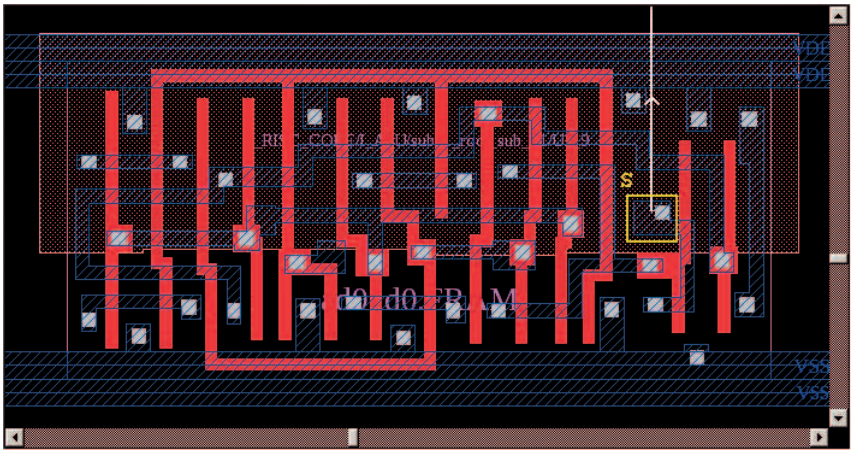


Рис. А.8 Стандартная ячейка ad01d0 (полный сумматор)

При работе с **IC Compiler** требуется указывать больше настроек библиотеки, чем при использовании инструментов логического синтеза. Пользователь может вводить команды в командную строку инструмента или формировать из них **TCL**-сценарии (скрипты):

```
icc> create_mw_lib ASIC_MW -technology «../ref/tech/CMOS_65nm.tf» \
     -mw_reference_library «../ref/mw_lib/sc ../ref/mw_lib/io»
```

Обратная косая черта – это просто символ продолжения строки. Приведенная команда создает файл физической библиотеки проекта с именем **ASIC_MW**, после чего он может быть открыт для редактирования. Файл увеличивается в размере по мере того, как проектирование проходит через последовательные этапы на рис. A.7. Каждый шаг добавляет в проект элементы: стандартные ячейки, ячейки **RAM** и ячейки ввода-вывода. Информация об этих элементах находится в специализированных справочных библиотеках, открытых только для чтения.

Чтобы вместить этот большой и постоянно растущий объем данных физического проектирования, Synopsys хранит файлы как проектных, так и справочных библиотек на жестком диске компьютера в фирменном формате **Milkyway**. Открытая библиотека проектов и используемые ею справочные библиотеки обозначены значком большого цилиндра (рис. A.7). При физическом проектировании данные передаются между **IC Compiler** и базой данных **Milkyway** в обоих направлениях. Основной единицей данных в этой огромной базе данных остается ячейка; даже проект **ASIC** верхнего уровня является ячейкой.

Технологический файл

На рис. A.7 демонстрируется, что **IC Compiler** также использует технологический файл (**.tf**, **technology file**). Этот текстовый файл описывает в удобочитаемом формате ключевые параметры процесса изготовления **CMOS** для конкретного завода. Поскольку изготовление происходит послойно, технологический файл описывает каждый слой по имени, номеру маски и другим атрибутам.

Ниже приведен пример описания самого критического слоя кристалла, поликремния **CMOS** (часто называемого просто **poly**):

```
Layer CP {
   layerNumber = 11
   color = RED
   minWidth = 0.13
...
}
```

Листинг А.6 Пример описания слоя poly кристалла

Атрибут цвета **RED** предписывает **IC Compiler** отображать слой poly в виде фигур, имеющих сплошное заполнение красным цветом (рис. A.8). Атрибут **minWidth** – пример геометрической технологической нормы, зависящей от конкретного завода. Гарантируется, что могут быть созданы линии из поликремния, ширина которых составляет не менее **130 нм** (или **0,13 мкм**). В разделе A.5.2 показано, как выполняется проверка норм проектирования (**DRC**) при физическом проектиро-

вании, чтобы выявлять любые нарушения правил проектирования, определяемых заводом.

Инженеры относят нижние слои **poly** к базовым слоям. Верхние слои (начиная с **METAL1**, **VIA12** и выше) являются стеком соединений. Атрибуты для металлических слоев содержат информацию о паразитном сопротивлении **R** и емкости **C**. В разделе А.5.1 будет показано, как IC Compiler извлекает паразитные данные для металлических слоев, чтобы точно рассчитать задержки.

А.4.2 Разработки плана размещения

Разработка **ASIC** похожа на рисование на чистом холсте. Художник, писавший картину на рис. А.2(а), сначала должен был изобразить очертания тарелки, бокала и солонки перед нанесением краски. Аналогичным образом группа разработчиков должна наметить оптимальное расположение аппаратных **IP**-ядер (**hard IP**), блоков **RAM**, аналоговых макросов и площадок ввода-вывода на кристалле. Такое оптимальное расположение больших блоков на кристалле называется **планом размещения** (**floorplan**). Эти блоки могут быть упорядочены с помощью скриптовых команд или путем ручного редактирования. Пользователи могут перемещать, вращать или зеркально отражать любой блок на чипе. Следует отметить, что план размещения является крупнозернистым и не включает отдельные стандартные ячейки. Они размещаются после этапа планирования.

При синтезе плана размещения особое внимание уделяется расположению каждого макроса. Термин «макрос» относится к «жесткому» (реализованному на транзисторном уровне) цифровому или аналоговому блоку – **ОЗУ** или **ПЗУ**, аппаратному **IP**-ядру или блоку **PHY**. Макрос может использовать несколько металлических слоев. Жесткая структура не позволяет маршрутизатору его оптимизировать. Чтобы избежать коротких замыканий, маршрутизатор не может проложить шину **METAL2** через макрос, который интенсивно использует этот уровень.

На рис. А.9 приведен план размещения небольшого чипа для применения в беспроводной связи. По периметру чипа находятся **48** контактных ячеек, расположенных в виде кольца. Тонкий золотой провод (не показан) присоединяется к каждой контактной площадке во время сборки корпуса, чтобы передавать входные/выходные сигналы в/из чипа. Контактная площадка – это большая библиотечная ячейка, которая примерно в **100** раз больше стандартной ячейки. Выходные контактные площадки содержат большие буферы, способные выдерживать высокую внешнюю емкостную нагрузку. Каждая площадка имеет защиту от электростатического разряда (**electrostatic discharge**, **ESD**), чтобы предотвратить накопление статического электричества, которое может повредить внутренние цепи микросхемы.

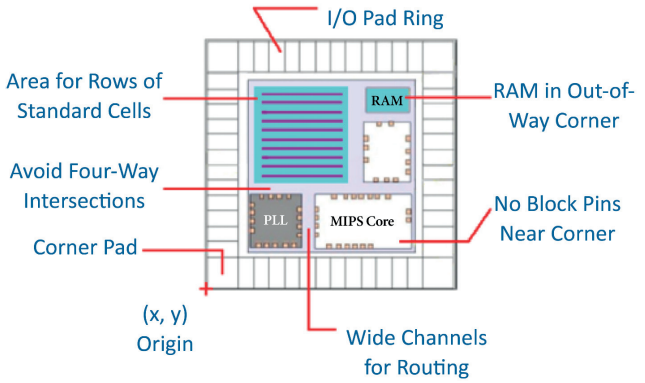


Рис. А.9 План размещения кристалла ASIC

Контактные площадки разработаны так, чтобы обеспечить хорошее соединение. Источники питания внутри каждой ячейки площадки передают напряжение питания контактного кольца по окружности. Для того чтобы обеспечивать питание вокруг кристалла, в каждом углу расположена угловая ячейка. Таким образом, данный пример схемы **ASIC** ограничен контактными площадками (**pad-limited**), и размер кристалла определяется шириной кольца контактных площадок, а не количеством логики ядра.

При проектировании **ASIC** план размещения должен быть оптимальным. Плохое планирование неизбежно приводит к последующим проблемам в проектировании. Примеры типичных проблем: серьезные нарушения временных характеристик, перекрестные помехи и сбои в маршрутизации. Для достижения оптимального плана размещения может потребоваться несколько итераций и ручных настроек, каждая из которых должна сопровождаться повторной маршрутизацией схемы.

После того как завершается этап планирования топологии кристалла, **IC Compiler** определяет местоположение **(x, y)** каждой ячейки контактной площадки и макроса с жесткой структурой. На этом же этапе предварительно определяются размеры кристалла. Хотя если будет необходимость расширить каналы маршрутизации или изменить положение макроса, может потребоваться его увеличение.

Любая область кристалла, оставшаяся внутри кольца контактных площадок и не занятая макросом, доступна для следующего шага: размещения тысяч стандартных ячеек в равномерно расположенных рядах. На это указывает прямоугольная область, заштрихованная горизонтальными линиями на рис. А.9.

Прежде чем перейти к стандартному размещению ячеек, приведем ряд интуитивно понятных рекомендаций по созданию плана размещения.

Замечание 8: для синтеза оптимального плана размещения рекомендуется соблюдать следующие эмпирические правила:

1) выделять одну большую прямоугольную область для размещения равномерно расположенных рядов стандартных ячеек;

2) помещать макросы **RAM** или **ROM** в труднодоступные углы логического ядра, чтобы уменьшить блокировки;
3) группировать блоки вместе, если они имеют общие шины или много общих сигналов;
4) располагать чувствительные аналоговые макросы близко к соответствующим площадкам ввода-вывода для уменьшения шума;
5) избегать размещения вводов-выводов **IP**-ядер и других логических блоков вблизи кольца контактных площадок;
6) избегать перекрещивания путей, которые быстро могут стать перегруженными при маршрутизации;
7) обеспечивать широкие каналы маршрутизации между соседними блоками, чтобы избежать перегрузки соединений в дальнейшем.

После синтеза для **ASIC** плана размещения можно осуществлять первое размещение стандартных ячеек, указанных в файлах списка соединений на **Verilog**. Для того чтобы управлять алгоритмом размещения ячеек, некоторые параметры следует задавать заранее, как это показано на примере команды ниже. Она указывает **IC Compiler** использовать прямоугольную область логического ядра внутри кольца контактных площадок:

```
icc> create_floorplan -pad_limit -io2core 80 \
      -core_utilization 0.75 -keep_macro
```

Первая опция сообщает инструменту, что чип ограничен контактными площадками. Соседние площадки ввода-вывода должны примыкать друг к другу. В свою очередь, инструмент синтеза приложит все усилия, чтобы разместить все стандартные ячейки внутри логического ядра. Следующая опция резервирует **80** микрон для канала маршрутизации между кольцом контактных площадок и логическим ядром. Опция на следующей строке указывает, насколько близко будут расположены соседние ряды стандартных ячеек. Параметр **0,75** означает, что **75 %** доступной площади ядра может быть занято логикой – строками стандартных ячеек и макросами с жесткой структурой. Таким образом, **25 %** площади кристалла резервируется для маршрутизации каналов. Последняя опция запрещает изменять заданное ранее расположение макросов.

Инструменты синтеза позволяют добиться и **100%-го** заполнения ядра логики при расположении строк стандартных ячеек без промежутков между ними для маршрутизации. Они зеркально располагают ряды чередующихся ячеек, объединяя смежные каналы питания и заземления. Это обеспечивает компактность расположения логики на кристалле. В таком случае вся маршрутизация осуществляется на вышележащем слое над стандартными ячейками. Тем не менее такой подход может привести к потенциальному затору: возникновению на кристалле участков, где маршрутизация слишком перегружена, чтобы удовлетворять требованиям **DRC** для расстояний между соединительными линиями.

Замечание 9: коэффициент использования **0,75** – это практический предел для высокопроизводительных чипов, даже если технология предусматривает наличие десяти или более металлических слоев. Размещение всех соединений в слоях над стандартными ячейками может быстро привести к перегрузке. Чтобы устранить нарушения целостности сигнала, следует оставлять некоторое пространство для маневра (раздел А.5.1).

Такой подход позволяет средствам синтеза при маршрутизации выбирать слой **METAL2** для очень коротких локальных соединений, а **METAL3** – для соединений в пределах блока или нескольких блоков. Более высокие слои сохраняются для прокладки линий тактирования, сигнальных каналов и длинных шин, пересекающих весь кристалл.

Ключевая часть синтеза плана размещения для кристалла – это синтез сетей электропитания (**power-supply network synthesis**, **PNS**). Размещение тысяч стандартных ячеек в горизонтальных рядах показано фиолетовыми элементами на рис. А.10. Каждый ряд получает энергию от питающих и заземляющих шин, которые проходят через стандартные ячейки.

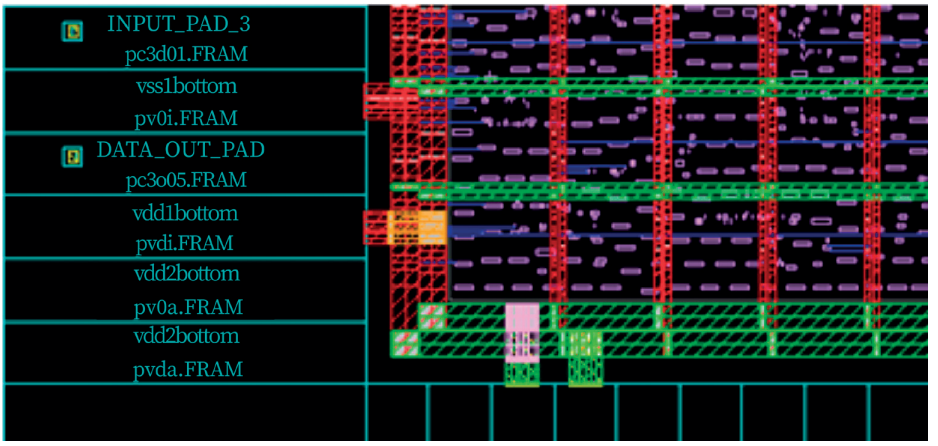


Рис. А.10 Сеть электропитания

На рис. А.10 также показано, как подается питание на все эти крошечные каналы. На рисунке левый нижний угол ограничен контактными площадками кристалла, подобно тому, как показано и на рис. А.9. Контактные площадки ввода-вывода соприкасаются, образуя прямоугольное кольцо по периферии. Некоторые из них являются питающими площадками (**supply pads**), которые предназначены для подключения внешних сигналов **VDD** и **VSS**. Контактная площадка **VSS** питает широкую вертикальную шину **METAL3** (красная поперечная штриховка). В свою очередь, она соединена переходом с широкой горизонтальной шиной **METAL4** (зеленая поперечная штриховка), проходящей снизу.

Эти горизонтальные и вертикальные металлические шины проложены вокруг всего ядра кристалла, питаемого специальными контактными площадками **VSS**, которые образуют кольцо заземления (**ground ring**). Прямо внутри него нахо-

дится концентрическое кольцо электропитания (**power ring**). Эти кольца питания разделены соответствующим проектным нормам интервалом (соответствующим **DRC**), хотя зазор на этом увеличенном рисунке виден нечетко.

В свою очередь, каждое кольцо питания питает более тонкие, равномерно расположенные в виде сетки шины **VDD** и **VSS** (красная или зеленая поперечная штриховка). Шины решетки доставляют **VDD** и **VSS** от кольца питания к местным каналам стандартных ячеек. Сеть электропитания синтезируется с помощью специального скрипта. Параметры решетки – ширина, шаг и размещение шины в слоях – выбираются так, чтобы падение напряжения питания для любой отдельной ячейки находилось в допустимых пределах. Слишком малое количество шин может означать чрезмерное падение напряжения питания, что приводит к замедлению логики **CMOS**.

A.4.3 Обеспечение низкого энергопотребления

Если одной из ключевых целей для чипа **ASIC** является низкое энергопотребление, то почти каждый шаг в цикле разработки должен включать проведение оптимизации с этой целью. Библиотеки ячеек должны учитывать энергопотребление как обязательное условие, о чем было сказано в разделе A.3.2. Хотя методы проектирования с низким энергопотреблением выходят за рамки приложения, в данном разделе кратко рассмотрено, как чип **ASIC** потребляет энергию и какие команды физического проектирования используются для снижения электропотребления.

Мощность, потребляемая **CMOS**-логикой, состоит из двух составляющих: динамической и статической. **Динамическая мощность** (**dynamic power**) – это энергия, потребляемая при переключении элементов в разные логические состояния. При переключении узлов электрической цепи с **0** в **1** динамическая мощность выражается в виде следующей суммы для всех узлов в схеме:

$$P_{dyn} = \tfrac{1}{2} VDD^2 \sum (C_{load} \cdot TR),$$

где C_{load} – емкость нагрузки узла;

TR – скорость перехода между состояниями.

Этот показатель увеличивается с ростом частоты тактирования. При размещении ячеек и синтезе тактового дерева (этапы, описание которых дано в следующих разделах) инструменты синтеза попытаются уменьшить большие нагрузки (особенно в путях, тактируемых с высокой частотой) за счет сокращения длины соединений и других приемов. Например, добавление логики для запрета подачи тактовых импульсов (**clock-gating**), как было показано на рис. A.5, напрямую снижает **TR** для триггеров.

Статическая мощность (**static power**) – это энергия, потребляемая даже тогда, когда ни один узел не изменяет свое состояние, равна сумме мощностей утечки для всех стандартных ячеек в цепи:

$$P_{stat} = \sum (P_{leak}).$$

Мощность утечки в ячейке определяется на основе атрибутов библиотеки. **IC Compiler** может уменьшить мощность утечки, компенсируя ее использованием ячеек с низким порогом утечки вместо стандартных, если они доступны в библиотеке. Ячейки с низким порогом медленнее, чем стандартные, но менее подвержены утечкам. Команда разработчиков может использовать эти методы, для того чтобы уменьшить как статический, так и динамический вклад в общее рассеивание мощности.

А.4.4 Этап размещения (placement)

Следующий шаг после создания плана размещения кристалла – пробное размещение ячеек. Тысячи стандартных ячеек – логических элементов, мультиплексоров, полных сумматоров, триггеров – помещаются в доступные области кристалла в соответствии с планом размещения. По умолчанию ячейки располагаются в равномерно расположенных горизонтальных рядах. Размещение регулируется заданным коэффициентом заполнения. В соответствии с рис. А.7 команда размещения ядра логики следующая:

```
icc> place_opt
```

У данной команды имеется много опций, однако ее настройки и по умолчанию работают достаточно хорошо. Алгоритм размещения приблизительно расставляет стандартные ячейки по наложенной сетке. Предварительно расположенные ячейки могут позже быть переставлены другими командами для уменьшения слэка или исправления нарушений **DRC**.

IC Compiler выполняет размещение, опираясь на временные требования. Основываясь на файлах списка соединений на **Verilog**, он оценивает (до начала какой-либо маршрутизации) длины межсоединений. Затем он предсказывает задержку, используя приближенную формулу **t = RC**. Руководствуясь гистограммой путей задержки, аналогичной гистограмме на рис. А.6 (профиль слэков), он расставляет приоритеты оптимизации для путей прохождения сигналов. Те пути, которые после маршрутизации (p**ost-route**) с наибольшей вероятностью могут нарушить временные параметры, можно сократить, поместив ячейки источника и назначения ближе друг к другу.

Команда **place_opt** может исправить ожидаемые нарушения временных параметров путем вставки ячеек буферов в критические пути. Данная команда уделяет особое внимание путям с большими разветвлениями (**high-fanout nets**). Такие пути управляют множеством входных выводов, расположенных далее по пути распространения сигнала, что приводит к образованию большой емкостной нагрузки для ячейки, являющейся источником сигнала. Широко распространенным решением для уменьшения емкостной нагрузки является вставка буфера. Буфер обладает необходимой мощностью для подачи достаточного уровня тока, что обеспечивает быструю передачу сигнала даже по сильно загруженной линии.

Целью по умолчанию для команды **place_opt** является уменьшение **TNS** до нуля. Одновременно с этим происходят проверка расположения размещенных ячеек на серьезные нарушения проектных норм **DRC** и их исправление. Следует отме-

тить, что алгоритм размещения прекращает оптимизацию, как только достигнуто соответствие временным требованиям. Если в логическом ядре слишком много места, стандартные ячейки могут оказаться на большом расстоянии друг от друга. В этом случае команде разработчиков рекомендуется уменьшить размер кристалла, что может значительно снизить затраты на конечный чип.

Также следует обратить внимание на то, что на этапе размещения сети тактирования по умолчанию не буферизируются. Эта важная задача физического проектирования (рис. А.7) выполняется на следующем шаге – на этапе маршрутизации и буферизации тактового сигнала, называемом **синтезом тактового дерева** (**CTS**).

Оценка результатов пробного размещения ячеек

Главная задача пробного размещения – проверка, все ли ряды стандартных ячеек вписываются в доступное пространство, выделенное планом размещения. Если это так, на следующем шаге происходит проверка результатов размещения на предмет их соответствия временным требованиям. **IC Compiler** поддерживает ряд команд, аналогичных тем, что использовались ранее для **Design Compiler**. Существует возможность запросить сводный отчет по **QOR** по аналогии с тем, как это делается в разделе А.3.5. Данный отчет содержит краткую информацию о длине критического пути и величине слэка. Для получения детального отчета о наличии задержек прохождения сигнала между логическими элементами вдоль критического пути и их слэка необходимо использовать команду

```
icc> report_timing
```

Пример выдержки из типичного отчета о временных параметрах проекта приведен ниже:

```
. . . . . .
I_ALU/sub_88/U2_13/CO (ad01d0) 0.31 *
I_ALU/sub_88/U2_14/CO (ad01d0) 0.31 *
I_ALU/sub_88/U2_15/S (ad01d0) 0.37 *
I_ALU/sub_88/DIFF[15] (ALU_DW01_sub_1) 0.00
I_ALU/U156/Z (aor222d1) 0.27 *
. . . . . .
```

Листинг А.7 Выдержка из подробного отчета о временных параметрах проекта

Он отражает часть критического пути, который проходит через блок **ALU** на чипе процессора. Критическим путем является наихудшее узкое место по скорости в конструкции **ASIC**. Этот конкретный путь включает в себя пятнадцать ячеек, реализующих полный сумматор (рис. А.8). Ячейки расположены каскадно, образуя цепочку шестнадцатиразрядного вычитателя – очень распространенного узкого места в логике тракта данных (**datapath**) на стандартных ячейках.

Общая задержка в этом сегменте логики **ALU** составляет **1,26 нс**. Каждая добавочная задержка в правом столбце листинга А.7 – это задержка прохождения сигнала через ячейку **ad01d0** (или другую) плюс чистая задержка межсоединения, управ-

ляемого его выходом. Согласно рис. А.7, внутренние задержки между ячейками считываются из логической библиотеки стандартных ячеек, но следует обращать внимание на звездочки (*). Их наличие указывает, что инструмент изменил задержки, приблизительно рассчитанные на этапе логического синтеза.

Ранние статистические оценки задержек были заменены вычисленными задержками межсоединений с помощью модели нагруженности соединений (**RC**-модели). Паразитные значения **R** и **C** рассчитываются при размещении на основе специфичных для технологии атрибутов металлических и поликремниевых слоев. Этот процесс пересчета задержек известен как обратная аннотация (**back-annotation**). По мере перехода от логической к физической области данные о временных параметрах становятся все более и более реалистичными (рис. А.11).

Этот профиль слэков аналогичен столбчатой диаграмме в **Design Compiler** (рис. А.6), но более точен. Он помогает оценить, насколько проект соответствует временным требованиям. По рисунку видно, что в данной **ASIC** имеется **18** худших путей со слэком от **0,396 нс** до **0,422 нс**, отмеченных фиолетовым цветом. Все они находятся в правом нижнем углу увеличенной области кристалла. Все слэки положительные, поэтому это пробное размещение соответствует временным требованиям.

Часто проект, ранее соответствовавший временным требованиям (**timing-clean**), после логического синтеза из-за обратной аннотации более реалистичных задержек путей получает большой положительный ненулевой **TNS**. Такой проект нарушает временные требования, и, прежде чем перейти к следующим этапам проектирования, необходимо устранить образовавшиеся нарушения, желательно без значительных переделок. Рассмотрим несколько методов, которые имеются в **IC Compiler**, для решения такой задачи.

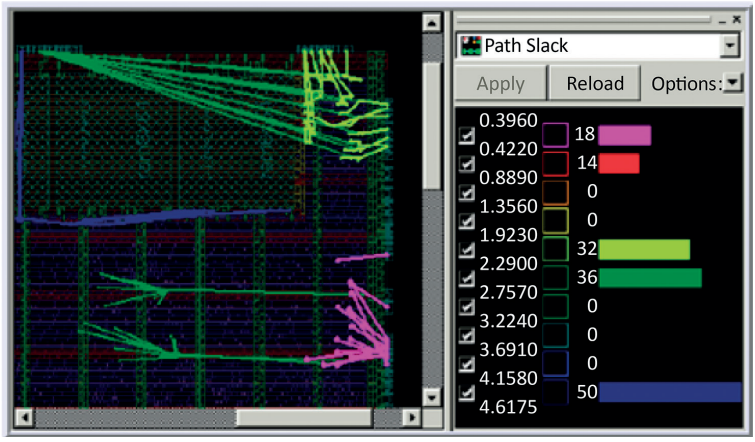


Рис. А.11 Гистограмма пути (ICC)

Уточнение размещения

Иллюстрация, представленная на рис. А.11, указывает на один из возможных способов уменьшения слэка путем изменения размещения. Все фиолетовые пути

(наихудшие случаи) на левой панели сходятся в нижнем правом углу. С помощью функции **magnet_placement** стандартные ячейки на концах пурпурных дорожек могут быть смещены ближе к неподвижному объекту, например такому, как пин ввода-вывода. Пользователи могут указать любой объект плана размещения, представляющий собой фиксированный макроблок или пин, в качестве магнита (**magnet**). Ячейки, привязанные к магниту, также могут быть указаны в явном виде, например как экземпляр модуля в тракте данных. Эта функция – хорошее компромиссное решение и позволяет вручную задать магнит, но автоматически перемещает ячейки в область его влияния.

IC Compiler предлагает и множество других средств визуализации, которые позволяют оценить: плотность размещенных ячеек; плотность соединений в данной зоне; статическую или динамическую плотность распределения питания; падение напряжения питания по линиям связи.

Более грубым методом уточнения размещения является повторный запуск пробного размещения с опцией более высокого уровня оптимизации. Данная техника уже была проиллюстрирована на этапе синтеза (раздел А.3.4).

Замечание 10: если пробное размещение удовлетворительно, но выявляет многочисленные нарушения синхронизации, имеет смысл попробовать повторно выполнить ту же команду, но с параметрами, заданными вручную. Например, добавление опции **-effort high** может улучшить результаты размещения за счет большего времени выполнения:

```
icc> place_opt -effort high
```

Повторный запуск команды использует первое размещение в качестве отправной точки и может пересмотреть расположение любой ячейки. Как только будет достигнуто размещение, удовлетворяющее требованиям команды разработчиков, можно переходить к следующему шагу на рис. А.7 – **синтезу тактового дерева** (**CTS**).

А.4.5 Тактовое дерево (clock tree)

Сигналы тактирования, особенно работающие с высокой частотой, всегда являются наиболее важными в конструкции **ASIC**. Их маршрутизация должна быть проведена до размещения линий данных и управляющих сигналов путем использования специализированных инструментов синтеза и маршрутизации. Этот этап называется **синтезом тактового дерева** (**CTS**). Но еще до проведения **CTS** необходимо определить источник каждого тактового сигнала.

Команда **create_clock**, которая использовалась вместе с входным портом **CLK** в разделе А.3.3, является наиболее распространенным способом определения внешнего источника тактирования (на рис. А.12 источник сигнала изображен в левом верхнем углу). Затем алгоритм **CTS** создает соединения между источником синхросигнала и всеми его приемниками, синтезируя промежуточные буферы по мере необходимости.

Приемник сигналов может быть триггером или макросом тактируемой памяти. Каждый размещенный триггер на данном этапе уже имеет известное **(x, y)** место на кристалле, поскольку у каждого макроса есть точное расположение в плане размещения. Это позволяет при синтезе тактового дерева точно вычислять длину соединения между источником и приемником сигнала. Чем длиннее сеть сигналов, тем позже фронт тактового сигнала прибывает в приемник сигналов.

Разброс во времени поступления сигналов в разных приемниках в большом временном домене (**clock domain**) никогда не бывает совершенно нулевым. Этот разброс называют расфазировкой тактовых сигналов (**clock skew**), и целью **CTS** по умолчанию является ее минимизация. **CTS** также должен соответствовать технологически ориентированным правилам проектирования, таким как максимальные ограничения ветвления (**maximum fan-out limits**). Приведенные ниже команды ограничивают расфазировку до 50 пс и ограничивают разветвление синхросигнала до 2000 приемников (значение по умолчанию):

```
icc> set_clock_tree_options -target_skew 0.050 -max_fanout 2000
icc> clock_opt
```

При этом следует отметить, что при синтезе тактового дерева для подключения каждого источника синхронизации к его многочисленным приемникам недостаточно просто минимизировать общую длину сети, как это делает маршрутизатор. Иногда, чтобы преодолеть расфазировку, при синтезе тактового дерева приходится удлинять линию тактового сигнала. Простая, но эффективная стратегия для достижения целей **CTS** – это построение сбалансированного тактового дерева (рис. А.12).

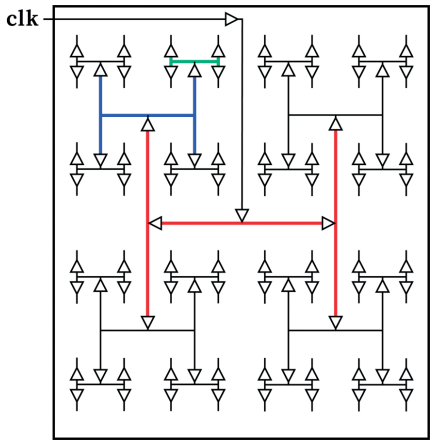


Рис. А.12 Сеть сигналов (3 H-Tree уровни)

Сеть тактового дерева

Типичное тактовое дерево (рис. А.12) начинается с внешнего источника (на рисунке находится слева вверху). Входящий тактовый сигнал сначала дважды буферизуется, а затем поступает в **H**-образное дерево (красного цвета) в центре кристалла. Он следует по маршрутам равной длины на четыре ветви **H**. Каждая ветвь

центральной **H**, в свою очередь, питает середину меньшей **H**. Эта небольшая сеть имеет три уровня **H**-деревьев, каждый из которых более локален. Данная схема может быть расширена до бесконечности.

Сеть H-дерева обеспечивает сбалансированную начальную точку с маршрутизацией равной длины к любой из **64** конечных точек на рисунке. Начиная с этой начальной точки, команда **clock_opt** может систематически выравнивать расфазировку при передаче сигналов к отдельным приемникам (не показаны), которые сгруппированы вокруг определенной конечной точки. Балансируются не только длины маршрутов; каждый приемник получает тактовый сигнал, прошедший ровно шесть этапов буфера, что помогает обеспечить единое время поступления сигналов.

Некоторая расфазировка все еще будет сохраняться из-за неизбежных различий в длине межсоединений, ступенек и металлических слоев. При синтезе тактового дерева можно использовать локализованные методы, такие как изменение размера буфера (**buffer resizing**) и вставки задержек (**delay insertion**), для более точной настройки сети и борьбы с расфазировкой. При этом вычисляется точное время поступления сигналов с использованием **RC**-расчетов и учетом правил проектирования от завода. Поскольку изменение параметров внутри кристалла неизбежно, синтез тактового дерева не должен быть чрезмерно ограничен, т. к. основной целью является достижение расфазировки в **0,0 пс**.

Данная схема распределения сигналов в виде **H**-дерева лучше всего работает, когда в плане размещения нет больших зон, запрещенных для маршрутизации. Такие зоны могут возникать из-за наличия ограничений, накладываемых на ячейки или макросы. Например, пользователь может запретить использовать слой **METAL5** в пределах определенной области, чтобы защитить чувствительный макрос. Внутри этой ограничительной рамки маршрутизация на слое **METAL5** будет невозможна.

Замечание 11: синтез оптимального тактового дерева возможен при отсутствии больших областей блокировки в плане размещения (таких как чувствительные аналоговые макросы) между глобальным источником синхронизации и его приемниками. Следует избегать наличия областей, в которых приемники тактовых сигналов окружены узкими каналами маршрутизации.

По умолчанию при синтезе тактового дерева можно использовать любые буферы или инверторы, имеющиеся в библиотеке логических ячеек (рис. А.7). Поэтому нужно стремиться применять наиболее полные библиотеки ячеек, содержащие большой набор буферов с широким диапазоном мощностей питания.

Замечание 12: по причине того, что при синтезе тактового дерева использовались буферы с недостаточной мощностью питания, у тактового дерева может быть слишком много этапов буферизации. Чтобы решить эту проблему, следует ограничить список буферов и инверторов, используемых при синтезе тактового дерева, ячейками с более высокой мощностью питания. Чтобы указать список ячеек библиотечных буферов явным образом, нужно использовать команду **set_clock_tree_references**.

Оценка качества результатов CTS

После выполнения синтеза тактового дерева используют команду **report_clock_tree** для оценки буферизации, а на следующем шаге генерируют стандартный **QOR** или отчет о временных параметрах, чтобы проверить **TNS**. Переход к следующем шагу (маршрутизации сигналов данных) возможен при условии достижения слэка, близкого к **0 нс**.

A.4.6 Маршрутизация (routing)

После того как завершены этапы физического синтеза (рис. А.7), которые заключаются в размещении всех макроблоков, синтезе кольца площадок ввода-вывода, размещении стандартных ячеек и создании тактовых деревьев, следующим шагом является реализация соединений между выводами ячеек, макросами и контактными площадками. Эту задачу выполняет программа-маршрутизатор (**router**), которая:

- выбирает оптимальные двухточечные пути для всех соединений, особенно для тех, которые проходят через весь кристалл;
- размещает каждый двухточечный путь на определенных слоях металла, вставляя сквозные межсоединения (**vias**) по мере необходимости для перехода между слоями;
- следит, чтобы каждое металлическое межсоединение и сквозное межсоединение отвечали проектным требованиям по ширине и зазору в соответствии с правилами завода (были **DRC**-чистыми).

Таким образом, при решении задачи маршрутизации требуется высокая загруженность процессора в течение длительного времени. Большинство маршрутизаторов разбивают эту сложную задачу на отдельные этапы. Каждый следующий этап выполняется на более низком уровне абстракции. **IC Compiler** по умолчанию использует маршрутизатор **Zroute**. Основная команда **IC Compiler**, которая запускает этот процесс многоэтапной маршрутизации (рис. А.7):

```
icc> route_opt
```

После запуска **Zroute** последовательно выполняются следующие три этапа:

1) **глобальная маршрутизация** – прокладывается абстрактный двухточечный путь для каждого пути сети (такой, к примеру, как на картах местности);
2) **распределение дорожек** – каждому сегменту пути на заданном металлическом слое назначается дорожка нулевой ширины;
3) **детальная маршрутизация** – каждый назначенный сегмент анализируется и преобразуется в соответствующую проектным нормам металлическую полосу.

Поскольку маршрутизация может занимать много часов, инструмент отображает подробную расшифровку в окне консоли, что позволяет просматривать текущий ход выполнения задачи.

Основная цель этапа маршрутизации – создать **100 %** соединений, устраняя перегрузки на каждом участке, после чего маршрутизатор стремится обеспечить исходные временные ограничения, содержащиеся в файле **SDC**. По умолчанию все три этапа маршрутизации работают в режиме оптимизации временных параметров. Несмотря на то что проект был проверен на соответствие временным требованиям в начале этапа маршрутизации, часто из-за соединений, которые прокладываются по обходным путям, чтобы избежать перегрузки, возникают нарушения временных параметров.

Во время работы маршрутизатора формируется отчет (листинг А.8), который полезен для мониторинга настроек, расчета статистики и отслеживания хода выполнения маршрутизации.

```
route_opt:
Routeopt: Using zrt (Zroute) router.
Information: Running timing-driven global route (ROPT-020)
Information: Running timing-driven track_assign (ROPT-020)
Information: Running timing-driven detail route (ROPT-020)
. . . .
phase3. Total Wire Length = 16820202
```

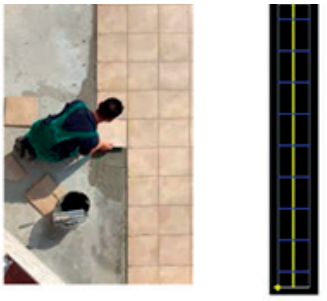
Листинг А.8 Пример выдержки из отчета маршрутизации

Если достигнуто соответствие проекта временным параметрам, маршрутизатор стремится минимизировать общую длину всех маршрутизируемых межсоединений. Общая длина соединений – удобный показатель оценки качества итераций маршрутизации. Общая длина выражается в мкм и содержится в отчете маршрутизации (листинг А.8).

На последнем этапе маршрутизатор стремится максимально обеспечить соответствие итогов маршрутизации геометрическим **DRC**-требованиям завода, указанным в технологическом файле. Поскольку выполнение данной задачи менее приоритетно, чем устранение перегрузок, некоторые нарушения проектных норм все же нередко остаются и исправляются вручную, о чем рассказывается в разделе А.5.2. Далее более подробно рассмотрены три этапа маршрутизации и даны некоторые практические рекомендации.

Глобальная маршрутизация (global routing)

Маршрутизация сотен миллионов соединений является сверхсложной задачей, требующей интенсивного использования процессора. Таким образом, большинство инструментов используют упрощения допущений. Маршрутизатор покрывает площадь кристалла прямоугольной сеткой, состоящей из базовых ячеек (**unit tiles**). На рис. А.13(а) показана простая аналогия: обычный пол покрыт тротуарной плиткой. На рис. А.13(б) представлена базовая ячейка, которая используется в маршрутизации:



а) б)

Рис. А.13 Аналогия: квадратная тротуарная плитка

Базовая ячейка на рис. А.13(б) не квадратная. Ее высота соответствует стандартной высоте ячеек в библиотеке (в данном примере – **3,69 мкм**). Ее ширина – это минимальный шаг (**minimum pitch**) между металлическими межсоединениями нижнего уровня для данной технологии. Эти размеры определены в технологическом файле (листинг А.9), который упоминался в разделе А.4.1.

```
Tile unit {
   width = 0.41
   height = 3.69
}
```

Листинг А.9 Пример описания базовых ячеек в технологическом файле

На рис. А.13(б) линиями показаны треки (металлические дорожки нулевой ширины, **tracks**), доступные для сигналов, которые проходят через эту ячейку. Чтобы избежать наложений, показаны только два из шести возможных металлических слоев. Следует обратить внимание на то, что треки расположены на большом расстоянии друг от друга. Расстояние определяется проектными нормами, которые задают шаг металлизации и содержатся в технологическом файле. Когда маршрутизатор прокладывает соединения через соседние элементы базовых ячеек, они автоматически приводятся в соответствие с требованиями минимального шага металлизации, поскольку плотность треков ограничена проектными нормами.

Глобальный маршрутизатор использует данную сетку базовых ячеек, чтобы проложить маршрут от одного элемента к другому для каждого соединения. Как и маршруты на географических картах, глобальный маршрут является абстрактным. Он перемещается по ячейкам, оставляя задачу детализации этапу распределения дорожек.

Ключевой деталью является то, что, как показано на рис. А.13(б), треки проходят горизонтально или вертикально в зависимости от слоя металла. Синие дорожки, представляющие **METAL1**, ориентированы преимущественно в горизонтальном направлении. Соседние слои, такие как желтый трек **METAL2**, ориентированы преимущественно в вертикальном направлении. Ортогональные направления для соседних слоев позволяют избежать коротких замыканий. Горизонтальная полоска **METAL1** вряд ли пересечет несвязанный сигнал, проложенный верти-

кально на этом же слое (здесь рассматривается пример параметров по умолчанию, которые иногда могут быть переопределены маршрутизатором).

Горизонтально-вертикальная маршрутизация известна как **Манхэттенская геометрия**. Происхождение термина связано с деловым и торговым районом Манхэттен в Нью-Йорке, где проспекты идут вертикально с севера на юг, а нумерованные улицы – горизонтально с востока на запад (похожая организация улиц существует и на Васильевском острове в Санкт-Петербурге). На рис. А.14 **IC Compiler** измеряет манхэттенское расстояние между двумя стандартными ячейками. Маршрут между размещенными ячейками, показанными белым цветом, проходит горизонтально на **81 мкм**, а затем вертикально на **48 мкм**. Общая протяженность сети равна манхэттенскому расстоянию **129 мкм**, а не евклидову расстоянию в **94 мкм**.

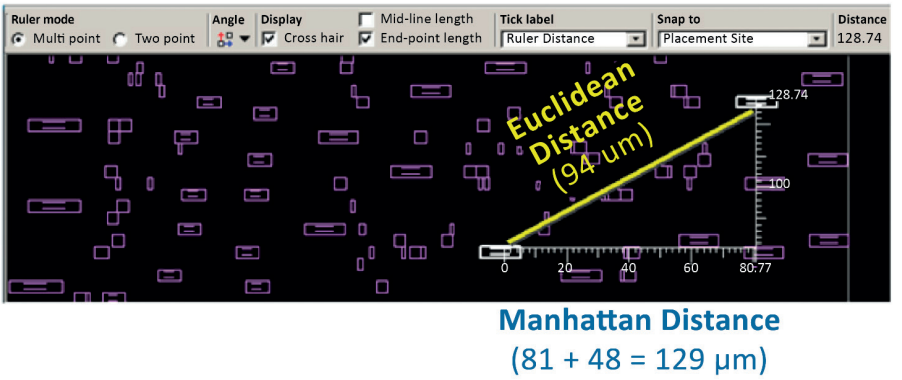


Рис. А.14 Манхэттенское расстояние

Замечание 13: предпочтительные направления для металлических слоев: горизонтальные – для нечетных слоев; вертикальные – для четных. Таким образом, пользователям следует подбирать пары слоев для маршрутизации длинных соединений, пересекающих кристалл. Например, слои **METAL5** и **METAL6** могут быть оптимальными для реализации высокоскоростной шины между **IP**-ядрами сопроцессора на кристалле. Вышележащие металлические слои в металлическом стеке обычно толще и шире; они обеспечивают меньшее паразитное сопротивление (R) и, соответственно, большую скорость.

Манхэттенский маршрут, изображенный на рис. А.14, требует наличия электрического контакта в металлических межсоединениях при их пересечении (правый нижний угол рис. А.14). При этом в технологии **CMOS** два последовательных металлических слоя всегда изолируются слоем диэлектрика (**intermetal dielectric**, **IMD**). Таким образом, для их электрического соединения требуется наличие сквозного отверстия (**through-hole**).

На рис. А.15 изображен поперечный разрез небольшого квадратного сквозного отверстия, протравленного в **IMD** при изготовлении. Отверстие заполняется вольфрамовым сплавом, который электрически соединяет межсоединения **METAL1**

и **METAL2**. Чтобы обеспечить прочное соединение (в том числе при возможном смещении), это малое отверстие накрывается квадратными контактными площадками из **METAL2** (сверху) и **METAL1** (снизу). Полученную трехслойную структуру называют сквозной перемычкой. Маску **CMOS**, определяющую все необходимые сквозные отверстия между слоями **METAL1** и **METAL2**, называют **VIA12**.

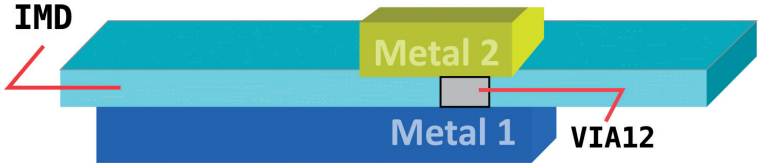


Рис. А.15 Сквозная перемычка между слоями металла в поперечном разрезе

Каждая сквозная перемычка добавляет некоторые паразитное последовательное сопротивление **R** и емкость **C** на пути прохождения сигнала. Таким образом, вторичная цель маршрутизатора состоит в том, чтобы уменьшить общее количество этих перемычек. Количество сквозных перемычек в каждом слое отображается в отчете маршрутизации.

Еще одним упрощением является маршрутизация с использованием шаблонов. Маршрутизатор может уменьшить сложность вычислений, концентрируясь на кластерах ячеек и прокладывая через них пути с использованием простых шаблонов. **L**-образный шаблон имеет один изгиб или перемычку между слоями. **Z**-образный шаблон имеет два изгиба. **Zroute** также может в некоторых местах отходить от Манхэттенской геометрии и при необходимости выходить за пределы сетки, чтобы сократить длину критического соединения или избежать геометрических нарушений проектных норм.

Распределение путей (track assignment)

Этап распределения путей является промежуточным звеном между глобальной и детальной маршрутизацией. Этот этап маршрутизации используется для уменьшения перекрестных помех. Если две сигнальные шины проложены друг рядом с другом и проходят через несколько базовых ячеек, это потенциально может приводить к возникновению помех. Чтобы уменьшить помехи, пути распределяют таким образом, чтобы они находились как можно дальше друг от друга.

При распределении путей **Zroute** также пытается устранить перегрузки, для чего используется схема с первым среди худших (**worst-first scheme**). При глобальной маршрутизации области с плотно расположенными контактами идентифицируются как перегруженные горячие точки (**hot spots**), поэтому отдельные треки в первую очередь распределяются именно им. При этом кристалл остается все еще относительно незагроможденным. Затем используются наиболее доступные соединения, после чего количество вариантов уменьшается, и средствам маршрутизации остается только распределить соединения по наименее перегруженным областям кристалла.

В случае если проблема с перегрузкой не исправляется (горячая точка возникает в любой области кристалла, где число требуемых дорожек превышает их коли-

чество, доступное в ячейках), незначительная перегрузка может быть устранена с помощью изменения настроек команды **route_opt**.

Для того чтобы устранить сильную перегрузку, может потребоваться еще одна итерация синтеза плана размещения. Перемещая элементы, можно расширить каналы маршрутизации и уменьшить перегрузку. Любая горячая точка – это неисправность, приводящая к неработоспособности системы и свидетельствующая о том, что маршрутизатору не удалось охватить все соединения в этой области. Соединение, которое маршрутизатор не смог проложить, почти всегда является фатальной ошибкой в проекте.

Замечание 14: современные чипы обычно содержат от **6** до **10** слоев металлизации (зависит от конкретной **CMOS**-технологии). Чем больше металлических слоев, с которыми должен работать маршрутизатор, тем меньше вероятность перегрузки. Если средняя длина маршрута сокращается, то уменьшается и **RC**-задержка соединения и, таким образом, увеличивается скорость работы чипа.

Детальная маршрутизация

На третьем этапе выполнения команды **route_opt** происходит окончательная детальная маршрутизация, во время которой каждый сегмент маршрута, присвоенный треку нулевой ширины с металлическим слоем, трассируется (**trace**) в соответствии с проектными нормами. Его ширина и интервал между соседними треками также должны соответствовать проектным нормам. На этапе подробной маршрутизации прокладываются соединения между последовательными металлическими сегментами в виде сквозных перемычек, отвечающих проектным нормам.

На рис. А.16 приведена ячейка полного сумматора **ad01d0**, рассмотренного ранее. На рис. А.8 вывод **S** ячейки подключен к абстрактной линии соединения. После детальной маршрутизации это соединение, обозначенное как **net N207**, реализуется полностью и занимает ячейку в желтом сегменте **METAL2**. Этот слой преимущественно проходит в вертикальном направлении. Следует обратить внимание на то, как детальный маршрутизатор добавляет излом соединительной линии, чтобы выровнять ее по точкам сетки. Соединение на слое **METAL2** имеет минимальную ширину **0,20 мкм**. Оно соответствует требованию обеспечения расстояния к ближайшей соединительной линии на уровне **METAL2** не менее **0,21 мкм**. Сквозные перемычки обозначены маленькими серыми квадратами (рис. А.16). Таким образом, сквозной контакт S соединяет синюю контактную площадку **METAL1** с желтой контактной площадкой **METAL2** чуть выше него. Сквозные перемычки должны удовлетворять нормам **DRC**: быть минимальными по ширине; иметь металлические контактные площадки; обеспечивать необходимое расстояние до других перемычек. Кроме того, существует ограничение на вертикальное размещение перемычек в одном месте.

На рис. А.16 **Zroute** прокладывает соединение с выходным контактом **S**, используя межсоединение в слое **METAL2**, а затем добавляет перемычку. По умолчанию

Zroute может расположить перемычку и добавить соединение в любом месте полигональной формы выходного контакта.

Следует отметить, что вид ячейки с полным сумматором **ad01d0** на рис. А.16 менее загроможден, чем на рис. А.8. На предыдущем рисунке используется полное представление макета (**CEL**), включая базовые слои (такие как **ploy** и контактный) и **METAL1**. На рис. А.8, напротив, показан вид **FRAM**, представляющий собой изорамное представление ячейки **ad01d0**. Вид **FRAM** содержит только детали, представляющие интерес для маршрутизатора: расположение выводов, сквозные перемычки, металлические блокировки и границы ячеек.

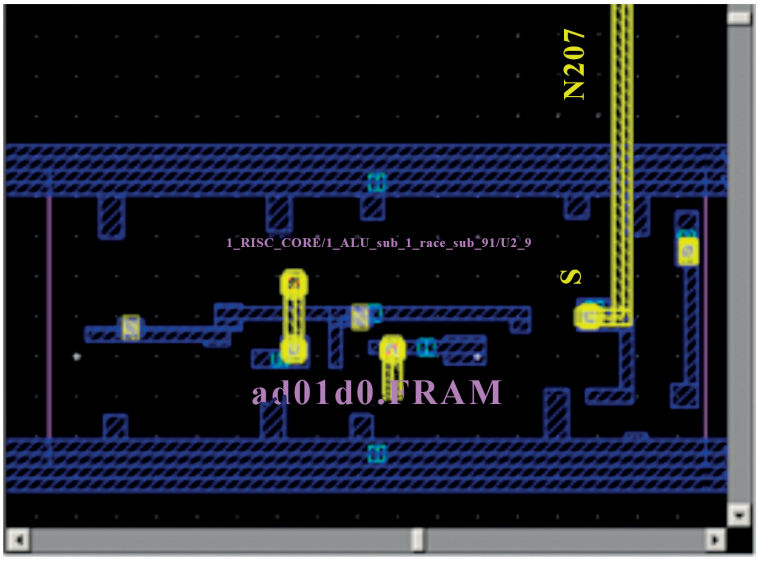


Рис. А.16 Детально маршрутизированная Net N207

База данных **Milkyway** включает в себя как **CEL-**, так и **FRAM**-представления каждой стандартной ячейки и площадки ввода-вывода. Представление **FRAM** используется при маршрутизации, что еще раз иллюстрирует важность уровней абстракции при проектировании **ASIC**. Благодаря скрытию низкоуровневых физических данных от маршрутизатора обеспечивается эффективная маршрутизация и быстрое отображение расположения компонентов, не обремененное ненужной информацией (такой как расположение внутреннего транзистора в каждой стандартной ячейке).

Оценка результатов маршрутизации (route evaluating)

На рис. А.17 приведена полностью маршрутизированная область кристалла **ASIC**. Это окно представляет собой расширенный отчет после детальной маршрутизации. В технологии кристалла, изображенного на рисунке, существует шесть металлических слоев. Справа расположено диалоговое окно **Layers**, которое отображает маскирующие слои **CMOS**, а также их цветовое кодирование. Каждый металлический слой имеет предпочтительное направление, показанное стрел-

кой. Слои в **VIA** (изолирующие слои) изображены между металлическими слоями. Целое число в скобках рядом с каждым слоем указывает номер маски. Таким образом, **METAL6** изготавливается с использованием маски под номером **34**. **11** масок используются для металлических слоев и промежуточных слоев, составляющих металлический стек кристалла, в то время как процесс изготовления **CMOS** задействует от **30** до **50** масок.

Более широкие металлические межсоединения **METAL3** и **METAL4** в энергосистеме служат для подвода **VDD** и **VSS**, о которых говорилось в разделе А.4.2. Слой **METAL1** в основном используется для разводки более тонких горизонтальных каналов, которые распределяют энергию от сети питания на отдельные ряды стандартных ячеек.

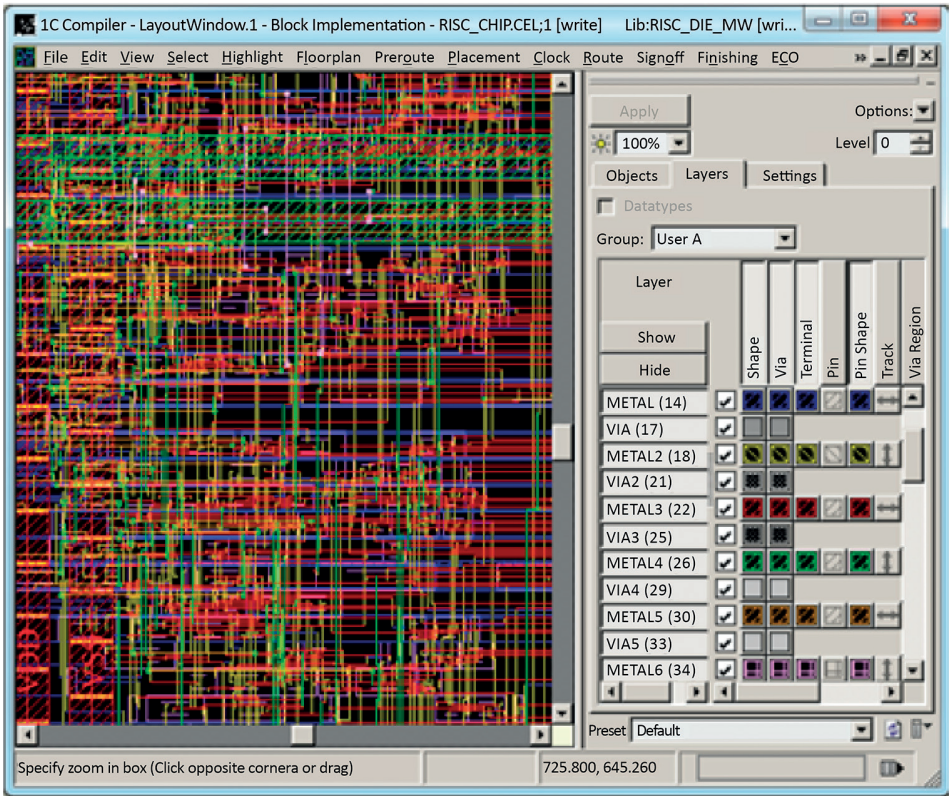


Рис. А.17 Маршрутизированная область кристалла

Поскольку каждая стандартная ячейка использует каналы, межсоединения и выводы в слое **METAL1**, маршрутизатор для прокладки большинства соединений, изображенных на рис. А.17, предпочитает использовать другие слои – от **METAL2** до **METAL5**.

Визуальное представление маршрутизированного кристалла позволяет разработчику выработать некоторые интуитивные идеи о **QOR** маршрутизации. Для

получения более подробных результатов нужно анализировать обычные отчеты на предмет соответствия проекта следующим требованиям:

1. Не должно быть скоплений горячих точек. Процент маршрутизируемых соединений должен составлять **100 %**.
2. Не должно оставаться никаких нарушений временных требований. Сумма всех отрицательных слэков (**TNS**) должна быть равна нулю до передачи в производство.
3. Общая длина соединений и общее количество межуровневых перемычек являются информативными параметрами для сравнения различных итераций маршрутизации.

A.4.7 Маршрутизация в соответствии с указаниями о технических изменениях (ECO)

Микроэлектронные компоненты часто в последний момент нуждаются в ручной модификации. Например, печатная плата может быть настроена с помощью переключения перемычек. Разработка чипа **ASIC** не является исключением. Изменения в последний момент, осуществляемые вручную, – будь то синтезированный список соединений или маршрутизированный чип – становятся все более распространенными, поскольку разработка на уровне наноразмерных технологий постоянно усложняется. Поскольку для их осуществления необходимы письменное разрешение и документальное подтверждение. Эти изменения регламентируются указаниями о технических изменениях (**engineering change orders**, **ECO**). Большинство **ECO** осуществляются вне рамок обычного процесса разработки.

Необходимость в **ECO** может возникнуть во время проверки первых изготовленных чипов (**post-silicon validation**), если обнаружится, что микросхемы имеют конструктивную ошибку. Также **ECO** требуется после компоновки (post-layout) в случае, если команда разработчиков сталкивается с трудно исправляемыми нарушениями временных требований и хочет устранить их без повторного проведения этапов размещения и маршрутизации. Многие технические изменения включают в себя и модификацию маршрутизации. Для этого в меню инструмента маршрутизации (показано на верхней части рис. A.17) содержится пункт **ECO**.

Еще одна причина технических изменений – необходимость в продлении срока службы изделия. Зачастую это можно сделать с помощью незначительного улучшения конструкции без излишних затрат на переход к следующему уровню технологии. Затраты на создание масок в настоящее время превышают однократные расходы на исследование по улучшению изделия (**non-recurring engineering**, **NRE**), поскольку, как упоминалось ранее, переход от **90-нм** к **32-нм** технологии может утроить стоимость итогового изделия.

Рисунок A.18 иллюстрирует пример типичной модификации проекта в соответствии с указаниями о технических изменениях. Такая модификация включает в себя только локализованную область кристалла. В частности, существующая схема адресации для блока **RAM** должна была быть изменена путем добавле-

ния сквозных соединений (**байпассирование**, **bypass**) в большую часть ее логики выбора адреса для реализации более прямого соединения. Эта модификация выполнялась после сдачи в производство. Чтобы снизить затраты, команда разработчиков использовала только существующие запасные элементы и перенаправила несколько металлических слоев, сохранив при этом неизменными дорогие маски базовых слоев.

Данный вид модификации получил название «модификация только металлических межсоединений» (**metal-only fix**). В соответствии с рис. А.18 изменение маршрута ограничено лишь слоями **METAL2** и **METAL3**. Байпассирование выполнено путем предварительного редактирования списка соединений. После того как соединения списка были скорректированы вручную, потребовалось изменить маршрутизацию сетей.

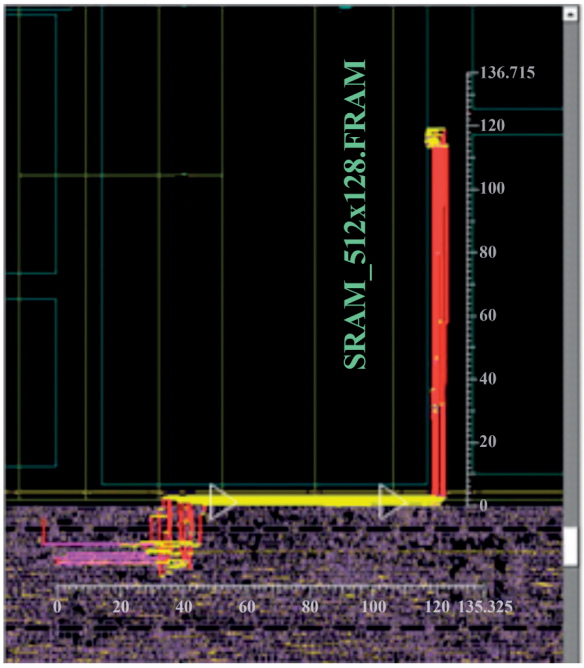


Рис. А.18 Указания о технических изменениях только металлических межсоединений

Первая команда ограничивает перенаправление соединений лишь слоями **METAL2** и **METAL3**. Поскольку манхэттенская длина байпасных соединений на рис. А.18 только по вертикали составляет более **120 мкм**, для управления большой емкостной нагрузкой этих длинных соединений и для избегания нарушений временных требований в перенаправленные адресные линии вставлены два дополнительных буфера. Их расположение на рисунке выделено белым цветом.

Для того чтобы избежать повторного размещения ячеек, эти **ECO**-буферы не были добавлены в проект; вместо них использовались существующие, неподключенные запасные ячейки (**spare cells**; их вставка рассмотрена в следующем раз-

деле). Перенаправленные адресные строки, изображенные в левом нижнем углу рис. А.18, берут начало от расположения этих запасных буферных ячеек.

ECO-модификация только металлических межсоединений – приемлемое решение для внесения изменений в проект в последний момент. Она позволяет снизить затраты за счет замены одной или нескольких масок для металлических слоев, избегая при этом каких-либо изменений в более дорогих масках базовых слоев. Данная модификация часто используется из-за сложностей разработки наноразмерной **ASIC** при технологии **65 нм** и ниже по причине того, что:

1) этапы физического проектирования становятся все более сложными и гораздо менее предсказуемыми, что делает ручные изменения привлекательнее. Они требуют усилий, однако более контролируемы, чем полная повторная маршрутизация;
2) размер кристалла **ASIC** и пропускная способность становятся больше. Добавление тысяч запасных ячеек вполне осуществимо;
3) комплекты масок становятся дороже с каждым уровнем технологии. **ECO**-модификация только металлических межсоединений помогает снизить стоимость масок.

А.4.8 Доводка кристалла (chip finishing)

В предыдущих подразделах основное внимание уделено реализации логической функциональности **ASIC**: был создан план размещения для площадок ввода-вывода и макросов с жесткой структурой; тысячи стандартных ячеек размещены в ряды, реализуя функциональную логику, а затем проведена маршрутизация всех соединений между элементами в соответствии с исходным списком соединений на **Verilog**. Все эти действия во время физического синтеза (рис. А.7) повлекли за собой изменение исходного списка соединений на **Verilog**: в тактовые деревья и сигнальные маршруты были вставлены буферы; увеличены или уменьшены ячейки, чтобы лучше соответствовать временным требованиям и ограничениям по мощности; опционально выполнена **ECO**-модификация. В результате получился модифицированный список соединений на **Verilog**, который используется на последующих этапах проектирования (например, для проверки соответствия топологии схеме (**LVS**) в разделе А.5.3).

Следующим этапом после разработки логического функционала **ASIC** является доведение ее электрических и физических характеристик до соответствия заданным требованиям. На этом этапе решаются многие рутинные производственные задачи, называемые доводкой кристалла (**chip finishing**) или проектированием для производства (**DFM**, **design for manufacture**). В данном разделе рассмотрено несколько типичных примеров таких задач.

Вставка ячеек-заполнителей

На этапе размещения (особенно если область ядра используется на 100 %), как правило, в строках стандартных ячеек остаются пустые места, что хорошо видно на рис. А.14. Рисунок представляет собой вид **FRAM**, увеличенный до нескольких

рядов ячеек, которые обозначены фиолетовым цветом. Эти ряды не полностью заполнены и напоминают кафельный пол со множеством недостающих плиток. Пустые пространства можно заполнить специальными ячейками-заполнителями (**filler cells**), имеющими электрическое, физическое или иное нефункциональное назначение. Существует несколько видов таких ячеек. Для вставки ячейка-заполнитель должна быть доступна в библиотеке.

Некоторые виды заполнителей лучше всего вставлять сразу после этапа размещения. Другие вставляются после завершения трассировки. Поскольку ячейки-заполнители не являются частью исходной функциональной логики, вопрос их вставки рассматривается в данном разделе. Команда **insert_stdcell_filler**, используемая для вставки ячеек, имеет множество вариантов.

Один из вариантов ячеек-заполнителей вставляется как продолжение каналов питания и заземления в слое **METAL1** через зазор в горизонтальном ряду стандартных ячеек. Другой вариант – делает то же самое, но также содержит небольшой развязывающий конденсатор (**decoupling capacitor, decap**) между **VDD** и **VSS**. Развязывающие конденсаторы работают в качестве фильтров нижних частот (**low-pass filter**) в сети питания и минимизируют шум питания из-за переходных токов. Такие токи возникают при переключении сигналов где-либо на кристалле, в результате чего колебания тока вызывают внезапные локальные падения напряжения питания. Развязывающие конденсаторы, расположенные рядом со стандартными ячейками, поглощают эти переходные напряжения.

Рассмотрим вставку запасных ячеек для обеспечения **ECO**-модификации, описание которой дано в разделе А.4.7. Запасные ячейки могут быть вставлены после размещения. Для того чтобы вставить набор полезных запасных ячеек в одну ограничивающую область за раз, обычно используется скрипт. Наиболее полезные запасные ячейки, как правило, простые: **BUF**, **INV**, **NOR**, **NAND**, **MUX** и триггеры. В наборе применяют различные мощности питания, а запасные ячейки равномерно распределяют по кристаллу, чтобы предвидеть возможную **ECO**-модификацию. Запасные ячейки остаются фактически не подключенными до тех пор, пока не будут использоваться в **ECO**-маршрутизации, а их входы должны быть заземлены, чтобы предотвратить возникновение нежелательного шума.

Это простой пример задачи доводки кристалла. В следующем разделе рассмотрена задача, которая иллюстрирует проектирование для производства. Она ничего не добавляет к функциональности чипа, но повышает его технологичность.

Вставка избыточных перемычек

При использовании наноразмерных технологий межуровневые перемычки очень малы. Ширина сквозного отверстия измеряется сотнями атомов кремния. На рис. А.19(а) изображена одинарная перемычка, используемая для соединения двух пересекающихся каналов **METAL1** и **METAL2**. Она удовлетворяет (упрощенным) **65-нм** правилам проектных норм. Сквозное квадратное отверстие должно иметь ширину не менее **65 нм**. Вокруг него также необходимы минимальные металлические контактные площадки (enclosure). Этому параметру соответствуют металлические контактные площадки площадью не менее 1**30 нм**.

Данные проектные нормы определяются минимальным разрешением, достижимым с помощью **65-нм CMOS**-литографии. Они также позволяют обеспечить прочное электрическое соединение между двумя металлическими каналами. При изготовлении **CMOS** эти малые перемычки подвержены дефектам обработки. Распространенными дефектами являются неправильный разрез (**open-via defect**) и неполное заполнение сквозного отверстия вольфрамовым сплавом (**void defect**). Такие дефекты перемычек могут привести к обрыву электрического соединения. При этом даже один дефект может быть фатальным для чипа.

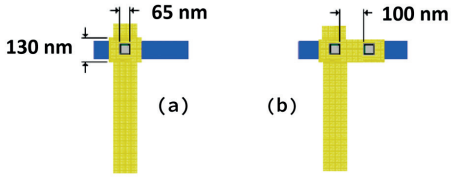


Рис. A.19 Вставка избыточных перемычек

Для уменьшения потерь на выходе годных изделий при проектировании для производства используется профилактический метод. Он состоит в том, чтобы вставлять одну или несколько дополнительных перемычек туда, где это позволяет пространство. На рис. A.19(б) показаны две перемычки для одного и того же соединения. Даже если одна из них оборвана, в случае если вторая перемычка не имеет дефектов, соединение остается рабочим. Этот прием значительно уменьшает количество дефектов, связанных с обрывом соединений.

Следует обратить внимание на то, что дополнительная перемычка является избыточной и не относится к первоначальному проектированию. Она добавляется только для уменьшения потерь при выходе годных изделий. Команда посттрассировки **insert_zrt_redundant_vias**, используемая для вставки этих дополнительных перемычек, имеет множество опций, для того чтобы помочь пользователям с выбором оптимального решения. Одним из таких решений является более узкое размещение, как на рис. A.19(б). Поскольку пара перемычек занимает больше места для реализации одного электрического соединения, когда дополнительные перемычки вставляются в миллионы мест на кристалле, компоновка становится более жесткой. Маршрутизатору затем намного сложней привести проект в соответствие проектным нормам. Поэтому, чтобы облегчить ему работу, смежные перемычки по умолчанию располагают на расстоянии **100 нм** друг от друга, уменьшая занимаемую площадь.

Избыточные перемычки могут также влиять на паразитное сопротивление и емкость соединений. На рис. A.19(б) **R** делится пополам, так как два сквозных отверстия проводят ток по двум параллельным соединениям. Но **C** почти удваивается, потому что теперь у соединения есть две металлические контактные площадки, а не одна. Короткие соединения имеют тенденцию замедляться из-за более высокой **C**, в то же время длинные соединения могут ускоряться из-за более низкого **R**. Таким образом, посттрассировка, связанная со вставкой дополнительных соединений, влияет на задержки. Поэтому приведенная выше команда содержит параметры для сохранения временных характеристик в рамках требований.

А.5 Этап передачи в производство

Когда физическое проектирование закончено, **ASIC** практически готова к передаче в производство (**tape-out**). Данный термин (**tape-out** – буквально: выдать на магнитную ленту-носитель) возник еще на заре развития электроники. Позднее, даже когда база данных по проектированию микросхем стала занимать одну или несколько катушек с магнитной лентой, название не изменилось. Сегодня под **tape-out** по-прежнему подразумевается отправка окончательной базы данных по разработанной микросхеме на завод (обычно путем передачи файлов по защищенному протоколу **ftp**).

На рис. А.20 представлен этап передачи в производство. Он включает в себя несколько подробных проверок маршрутизированной базы данных готового чипа. Эти этапы называются физической проверкой, поскольку они проверяют геометрические и электрические, а не функциональные ошибки. Все найденные ошибки должны быть исправлены, и зачастую исправление осуществляется вручную.

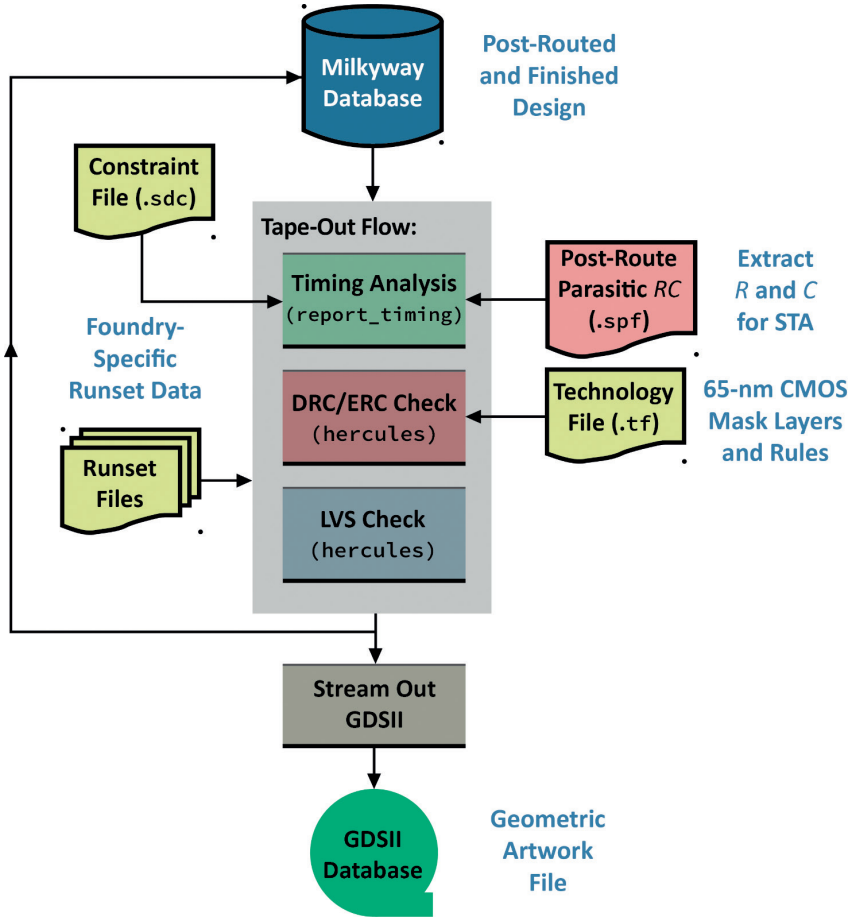


Рис. А.20 Этап передачи в производство

Чтобы не загромождать изображение, на рисунке показан только один типичный цикл итерации, при котором, путем возврата в **IC Compiler**, выполняется исправление ошибок. При этом по возможности следует избегать повторения этапов размещения или маршрутизации. Вместо этого можно использовать низкоуровневые команды перемещения, такие как **set_cell_location**, для исправления ошибок и изменения базы данных проектирования **Milkyway**. После внесения исправлений происходит повторный запуск проверок.

На последней стадии, когда все итерации завершены, проводится окончательная проверка перед передачей в производство (выходная проверка, **sign-off check**). Средства физической верификации проекта стоят больших денег, но они необходимы для контроля затрат на проектирование **ASIC**, поскольку, как уже было сказано в разделе А.1.1, генерация набора масок **CMOS** может стоить до **1** млн долларов США. Поэтому проверка базы данных проекта перед созданием масок и исправление любых электрических или геометрических ошибок очень важны. В следующих разделах рассмотрены три заключительных этапа проверки: статический временной анализ (**static timing analysis**, **STA**), проверка соответствия проектным нормам (**DRC** / **ERC**) и проверка соответствия топологии схеме (**LVS**).

А.5.1 Проверка временных параметров

На каждом этапе разработки проекта в разделах А.3 и А.4 осуществляется постоянная проверка отчетов по временному анализу, а для визуальной проверки используются гистограммы слэков (рис. А.6 и А.11). С каждым этапом проекта результаты временного анализа становятся все реалистичнее, поскольку задержки в сети вычислялись все более и более точно. Естественно, когда фаза физического проектирования завершена, чип полностью маршрутизирован, все **ECO**-модификации выполнены, а **TNS** равен нулю, требуется окончательная выходная проверка временных параметров.

На заре развития полупроводниковой промышленности логическое моделирование (**logic simulation**) было традиционным методом для выявления любых нарушений временных параметров перед передачей в производство. Логическое моделирование используется и в наши дни (моделирование **RTL**-кода осуществлялось во всех главах этой книги). Логический симулятор может также легко провести моделирование измененных **Verilog**-файлов списка соединений (рис. А.7) с помощью специализированной библиотеки ячеек для конкретного процесса производства на уровне логических элементов, содержащей точную информацию о задержках элементарных ячеек. При этом даже на последних этапах проектирования моделирование сигналов и использование средств проверки кода могут выявить неожиданные нарушения временных параметров, а также непредвиденные сбои (**glitches**) и всплески (**spikes**).

Однако в сложных **ASIC** выполнение полной симуляции на уровне логических элементов для всего чипа происходит очень медленно. Поэтому моделирование на уровне логических элементов обычно выполняют только для критических путей данных или для логики включения питания чипа.

Инструмент статического временного анализа позволяет обнаруживать нарушения временных параметров без проведения моделирования на интервалах в тысячи тактов. Вместо этого инструмент **STA** анализирует проверочный тактовый цикл с учетом временных ограничений. Таким образом, достигается ускорение процесса моделирования на несколько порядков по сравнению с использованием традиционных инструментов динамического моделирования. Ранее уже был рассмотрен типичный отчет, генерируемый командой **report_timing** в **Design Compiler** или **IC Compiler** (листинг А.7). Однако для окончательной выходной проверки временных параметров предпочтительным является отдельный инструмент статического анализа **Synopsys PrimeTime**. Он изображен первым блоком на рис. А.20 и может работать с базами данных для всего чипа в формате **Milkyway**.

Кроме обычных отчетов наподобие тех, что были проиллюстрированы ранее (листинг А.7), **PrimeTime** имеет большие возможности для анализа временных характеристик на уровне логических элементов всего кристалла. Он также предоставляет информацию о том, какое влияние на тракты синхронизации оказывают неизбежные отклонения на кристалле (**on-chip variations**, **OCV**) – имеются в виду отклонения от параметров технического процесса, напряжения питания и температуры. При этом основная задача данного инструмента – убедиться в том, что **TNS** по-прежнему равен нулю даже при ожидаемых отклонениях от нормального режима работы чипа. Временные требования к проекту содержатся в файле ограничений **.sdc**. **PrimeTime** также может оценивать задержки соединений путем расчета слэка с использованием сведений из базы данных в формате **Milkyway** или более точным методом обратного аннотирования соединений.

Файл с данными о паразитных свойствах соединений после маршрутизации (**.spf**, рис. А.7) используется **PrimeTime** для аннотирования каждого физического сегмента соединений, что позволяет вычислять более точные задержки **RC** в процессе работы. Это также позволяет проверить требования завода, такие как максимальная емкость соединений.

Следует отметить, что при маршрутизации паразитная индуктивность **L** не рассматривается. Наноразмерные металлические межсоединения на чипе имеют достаточное сопротивление, чтобы не стать сильно задемпфированными (**heavily-damped**). Чего не скажешь про полосы на печатной плате, которые имеют намного меньшее сопротивление и поэтому должны рассматриваться как линии передачи **RLC**.

Другой ключевой возможностью **PrimeTime** является анализ целостности сигнала (**signalintegrity**, **SI**). Маршрутизируемые металлические межсоединения (рис. А.17) размещаются все ближе друг к другу с каждым уровнем технологии производства **ASIC**. Это приводит к нежелательным перекрестным помехам (**crosstalk**), возникающим между несвязанными сигнальными линиями. В разделе А.4.6 демонстрируется, как инструменты маршрутизации борются за уменьшение перекрестных помех. Тем не менее электромагнитная энергия все равно может передаваться между соседними металлическими межсоединениями посредством паразитной емкостной связи. Энергии, передаваемой линией-агрессором ближайшей линии-жертве, может быть достаточно, чтобы повлиять на задержку прохож-

дения по ней сигнала. В результате может возникнуть непредвиденное нарушение времени удержания или установки сигнала.

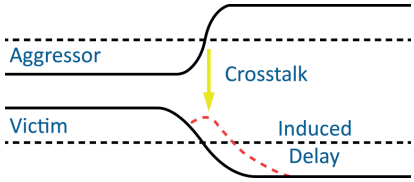


Рис. А.21 Нарушение целостности сигнала

PrimeTime вычисляет изменения задержки **SI** для худшего случая и отображает эту информацию в данных отчета. С его помощью можно точно определять места, где возникают чрезмерные перекрестные помехи, что позволяет команде разработчиков исправить соответствующие места. Когда **ASIC** прошла окончательную выходную проверку временных параметров – значит, что уже не осталось никаких известных нарушений синхронизации и не требуется никаких переделок. Следующим этапом окончательного выходного контроля является тщательная проверка размещенной и маршрутизированной топологии кристалла на соответствие геометрическим проектным нормам.

А.5.2 Проверка соответствия проектным нормам (DRC/ERC check)

Проверка на соответствие проектным нормам должна обеспечивать выполнение условий, при которых маршрутизируемая **ASIC** будет соответствовать всем требованиям разработки топологии для конкретного технологического процесса. В данном приложении уже рассмотрены некоторые простые правила, такие как соблюдение минимальной ширины металла и расстояния между перемычками. База данных проектирования **ASIC** должна удовлетворять всем специфическим правилам завода, чтобы гарантировать изготовление чипа с нормальным уровнем выхода годных изделий. В частности, проектные нормы характеризуют минимальное разрешение, достижимое с помощью литографии **CMOS**. Таким образом, если наименьшее разрешение составляет **65 нм**, средства проверки помечают любую ширину канала, контакта или соединений, которые слишком малы для данного разрешения.

Проектные нормы ужесточаются по мере продвижения от **65 нм** к **45 нм** и далее. Ранее было отмечено (раздел А.4.6), что **Zroute** по умолчанию может выходить за пределы сетки маршрутизации, что является компромиссным решением. Проектные нормы должны допускать неманхэттенскую геометрию, предусматривать проверку ширины и интервалов линий под углом **45°**, а также неправильных форм.

Проверка **DRC** требует больших вычислительных ресурсов. На заре развития полупроводниковой промышленности эти вычисления выполнялись на мейнфреймах. Сегодня требования проектных норм намного выше. Так, в соответствии с рис. А.20 инструменты проверки проектных норм должны обеспечивать соблюдение как правил, содержащихся в технологическом файле, так и тех, которые

описаны в специфичном для завода файле с настройками выполнения (**foundry-specific runset file**).

Основные проектные нормы содержатся в технологическом файле в декларативной форме. Например, минимальный шаг на слое **METAL1** описывается как **pitch = 0,41**. Файл с настройками выполнения проектных норм или обеспечения соответствия топологии схеме представляет собой более сложную программу управления (**runset file**), использующую операторы **if-else** и геометрические операторы для моделирования взаимодействий между отдельными элементами проекта на различных слоях.

Если **IC Compiler** имеет доступ ко всем проектным нормам в технологическом файле, то возникает вопрос: почему на этапе размещения и маршрутизации они иногда нарушаются? Ниже приведено несколько практических соображений.

1. Технологический файл использует простой декларативный синтаксис для описания проектных норм. Это может нивелировать более сложные нормы.
2. Размещение и маршрутизация определяются простыми правилами в технологическом файле. Но инструменты проверки проектных норм полагаются на файл с настройками выполнения. Это отступление от парадигмы может привести к неожиданным результатам.
3. При маршрутизации всегда отдается приоритет решению проблемы перегрузок. Соблюдение проектных норм происходит с более низким приоритетом. Таким образом, маршрутизатор может успешно проложить **100 %** соединений, но нарушить требования части проектных норм.

Итак, все больше и больше нарушений в настоящее время обнаруживается на поздних этапах проектирования **ASIC**. По этой причине многие итерации выполняются вручную, часто – незадолго до крайнего срока передачи в производство. На рис. А.22 приведен пример описания довольно сложной проектной нормы – нормы интервала конца межсоединений, которая включает относительно широкую полоску **METAL1**, близкую к узкой вершине ширины **W**. Согласно проектным нормам, по умолчанию расстояние **S** между этими примитивами **METAL1** составляет всего **0,12 мкм**. Однако норма обеспечения надежности изготовления становится более жесткой в условиях, показанных на рис. А.22(а). Если меньшая форма представляет собой конец межсоединения с узкой шириной **W ⩽ 0,2 мкм**, то расстояние **S** должно составлять не менее **0,15 мкм**.

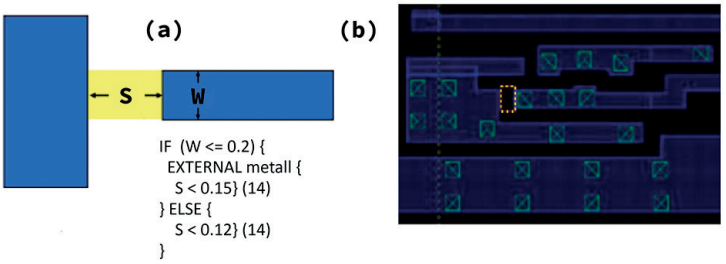


Рис. А.22 Проектная норма: интервал конца межсоединения

Целое число **14** в скобках указывает слой. Подсвеченная желтым цветом область называется полем ошибок. Код на рис. А.22(а), описывающий это условное правило, является частью файла настройки выполнения. Сложные нормы, подобные этой, непросто описать в традиционном технологическом файле. На рис. А.22(б) приведен фактический вид топологии, где норма нарушена. Большая металлическая форма слева от желтого поля ошибок на рис. А.22(а) слишком близка к концу узкого горизонтального межсоединения справа. Устранение такого нарушения обычно выполняется в **IC Compiler** с использованием ручных команд для увеличения расстояния **S**. Это явно утомительная и чреватая ошибками задача.

Замечание 15: имеет смысл настраивать приоритеты маршрутизации с помощью команды **set_route_zrt_global_options** так, чтобы приоритет **QOR** по отношению к проектным нормам был ниже. Это обеспечит соответствие проектным нормам и уменьшит количество нарушений, оставшихся неисправленными после маршрутизации. По умолчанию глобальный параметр маршрутизации установлен как **timing_driven_effort_level**.

На рис. А.23 приведено описание базового этапа проверки проектных норм. Инструмент проверки **DRC/ERC**, такой как **Hercules**, получает информацию о проекте либо уже в окончательном формате **GDSII**, либо непосредственно из базы данных **Milkyway**. В файле настроек выполнения содержатся инструкции о том, что проверять и какие выходные файлы генерировать.

Hercules – это иерархический инструмент физической проверки. Устаревшие инструменты проверки проектных норм игнорировали иерархию проекта, что приводило к ненужным повторам. Если стандартная ячейка, такая как **ad01d0**, использовалась в **1000** мест проекта, то те же самые проверки проектных норм повторялись **1000** раз. В результате этого проверка могла занимать несколько дней. Иерархический инструмент проверки проектных норм сначала идентифицирует подобные ячейки, а затем проверяет контрольную ячейку только один раз, за счет чего время выполнения значительно сокращается.

Hercules предназначен для окончательного выходного контроля проектных норм вплоть до **45-нм** технологии. Для еще более продвинутой технологии нужно использовать более новый инструмент **IC Validator**. Это одно из самых больших отличий, касающихся инструментов, используемых за пределами **65 нм**.

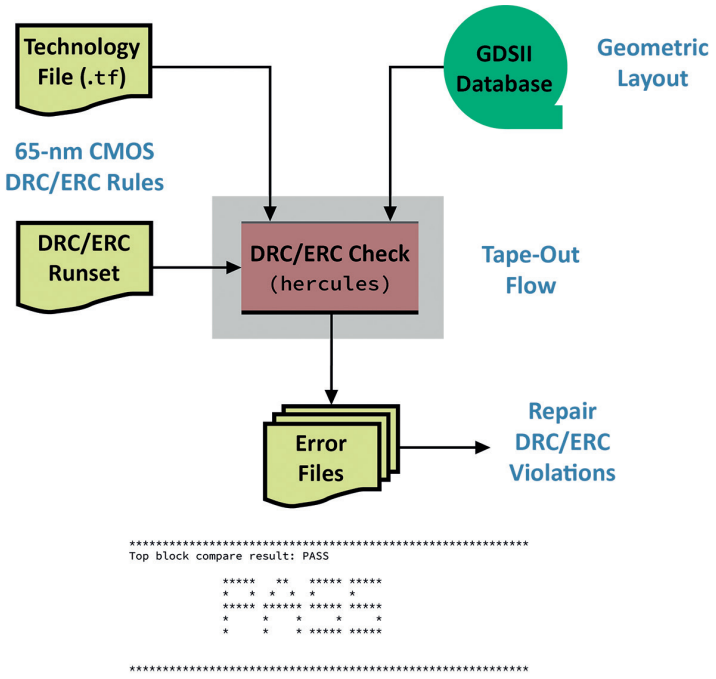


Рис. А.23 Проверка соответствия проектным нормам (DRC)

Следует обратить внимание, что для маршрутизации можно установить более жесткие проектные нормы, чем нормы по умолчанию в технологическом файле. Данный прием позволяет повысить технологичность и улучшить **QOR**. Так, пример ниже позволяет определить новые правила для маршрутизации более широких высокоскоростных линий тактирования на слоях **METAL5** и **METAL6**:

```
icc> define_routing_rule FAST_CLK_RULE \
   -widths {METAL5 0.3 METAL6 0.3} \
   -spacings {METAL5 0.4 METAL6 0.4}
icc> set_clock_tree_options -routing_rule FAST_CLK_RULE
```

Листинг А.10 Пример задания более жестких проектных норм на стадии маршрутизации

По причинам, которые были указаны ранее, маршрутизатор может оставить многие нарушения проектных норм неисправленными. Исправление их может быть утомительным. Из этого следует следующее замечание.

Замечание 16: команда физической проверки по своему усмотрению анализирует и исправляет каждое нарушение проектных норм, которое обнаруживает **Hercules**.

Для улучшения характеристик чипа или ради уменьшения размеров кристалла (правда, за счет сокращения выхода годных изделий) иногда практикуется некоторое отхождение или отказ от определенных норм проектирования. Данные шаги

рекомендуются только тем компаниям, которые имеют собственный завод по производству чипов и хороший контакт между командой разработчиков и инженерами фирмы, специализирующейся на выпуске интегральных схем. Компаниям без собственных производственных мощностей (**fabless**), которые полагаются на независимые заводы, такие как **TSMC** или **GLOBALFOUNDRIES**, рекомендуется исправлять все нарушения проектных норм.

На этом этапе также выполняется проверка на соответствие электрическим нормам (**ERC**). Простым примером проверки **ERC** служит анализ, все транзисторы, описанные в схеме, имели выходной тракт, и цепь не была разорвана.

Более сложная проверка на соответствие электрическим нормам состоит в том, чтобы убедиться, что все металлические межсоединения достаточно широки, чтобы проводить силу тока, которая, как ожидается, будет проходить через них. Нарушение этой нормы может не только снизить производительность при изготовлении, но и привести к проблемам, связанным с обеспечением надежности работы готовых чипов.

A.5.3 Проверка соответствия топологии схеме (LVS)

Когда все нарушения **DRC/ERC** устранены, дальнейших изменений в топологии кристалла не требуется. В соответствии с рис. A.24 следующий шаг – сравнение текущего списка соединений проекта со схемой, извлекаемой непосредственно из топологии. Этот последний шаг физической проверки называется проверкой соответствия топологии схеме (**LVS**).

Verilog-файл списка соединений, описание которого дано в разделе A.3.6, задает связи между стандартными ячейками. Так, в приведенном ниже фрагменте кода (листинг A.11) задается однобитный провод **Q1**, используемый для соединения выходного вывода **Q** триггера с входным выводом **A1** логического элемента **AND**. В сложной **ASIC** могут быть сотни миллионов таких контактов. Все они должны быть точно расположены на кристалле при физическом проектировании:

```
module NETWORK_3FF( . . . );
   . . . .
   wire Q1, N1, n5, n7;

   SDN_FSDPRBQ_1 Q1_reg(
      .D(PI1),...., .CK(CLK),.RD( RSTn),.Q(Q1)
   );
   . . . .
   SDN_AN2_1 U10(.A1(Q1), .A2(PI2), .X(N1));
endmodule
```

Листинг A.11 Фрагмент файла списка соединений на Verilog

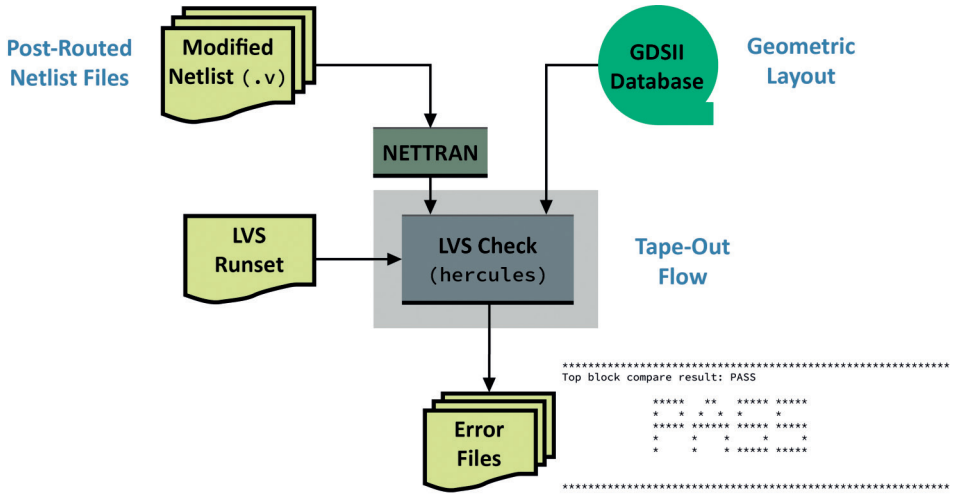


Рис. А.24 Проверка соответствия топологии схеме (LVS)

Согласно этапам проектирования, рассмотренным в разделе А.4, стандартные ячейки, такие как триггер и логический элемент **AND**, сначала помещаются на кристалл. После этого провод **Q1** прокладывают между их выводами, используя вначале дорожки нулевой ширины, а затем полосы, отвечающие проектным нормам. Получившееся после маршрутизации соединение **Q1** представляет собой серию металлических сегментов, которые проходят вдоль одного или нескольких слоев с межслойными перемычками, вставляемыми по мере необходимости. На этом этапе нарушения проектных норм уже устранены. При этом необходимо убедиться в том, что размещение после маршрутизации – низкоуровневая реализация соединения **Q1** – все еще соответствует первоначальному плану. То есть необходима тщательная проверка всей схемы на наличие расхождений, как показано на рис. А.24.

Данная проверка выполняется средствами **LVS**, такими как **Synopsys Hercules** или **Mentor Graphics Caliber**. Эти инструменты получают на вход **Verilog**-файлы списка соединений после маршрутизации (может потребоваться трансляция в специальный формат конкретного используемого инструмента) или геометрическое расположение компонентов **ASIC** в формате **GDSII**, либо непосредственно из базы данных **Milkyway**.

Проверка **LVS** происходит в два этапа. Сначала инструмент извлекает схему на уровне вентилей путем тщательного сканирования геометрической топологии. Идентифицируются отдельные транзисторы по их геометрическим характеристикам. Например, на рис. А.8 приведен вид ячейки полного сумматора с **28** транзисторами, каждый из которых создается путем наложения полоски металла слоя **poly** и диффузионной области. Далее извлеченная схема тщательно сравнивается с исходным списком соединений. Если обнаруживаются какие-либо ошибки, они идентифицируются по имени модуля. Рисунок А.24 представляет собой проход проверок от модуля верхнего уровня вниз, во время которого расхождений не обнаружено.

Стадия извлечения при проверке соответствия топологии схеме очень похожа на то, что происходит при проверке проектных норм. Обе программы анализируют ячейки (их размер, расстояние между перемычками, перехлест) путем сканирования геометрической топологии. Таким образом, **Hercules** выполняет проверку как **DRC**, так и **LVS**. Параметры командной строки **Hercules** и файла с настройками выполнения подробно информируют инструмент проверки, что нужно делать.

A.5.4 Геометрическая модель в формате GDSII (stream out GDSII)

После того как получена соответствующая временным требованиям и прошедшая все проверки база данных проектирования **Milkyway**, последний шаг в процессе разработки **ASIC** – реализация геометрической модели в формате **GDSII**. При этом создается огромный файл (часто терабайтного размера). Как упоминалось ранее, термин «потоковая передача» означает запись большого двоичного файла на несколько катушек магнитной ленты. Именно этот файл базы данных **GDSII** передается на завод для производства чипов **ASIC**. После того как команда разработчиков уже выполнила окончательную выходную проверку временных параметров, **DRC/ERC** и **LVS**, все, что необходимо заводу, – это точная послойная геометрия на транзисторном уровне (ее часто называют топологическим чертежом).

GDSII – это очень старый формат баз данных, но он может хранить огромный объем информации в относительно компактной форме. Он содержит двоичные записи переменной длины для описания каждой плоской формы – такой, например, как многоугольник, изготовленный на слое **METAL1**. Это **GDSII**-представление конструкции чипа является самым низким уровнем абстракции на всех этапах разработки. Оно сводит сверхсложную схему **ASIC** к одному лишь топологическому чертежу, состоящему из сотен миллионов плоских геометрических фигур, которые представляют собой любые элементы проекта от мельчайших перемычек до ячеек верхнего уровня.

Команда для потоковой передачи ячеек верхнего уровня имеет следующий вид:

```
icc> write_stream -cells ASIC_CHIP OUT/ASIC_DIE.gds
```

Представим простой пример, как будет выглядеть описание ячейки полного сумматора (рис. A.8). Наиболее критичным из всех маскирующих слоев является слой поликремния, который образует затвор любого **МОП**-транзистора. У рассматриваемого сумматора более дюжины сложных полиформ (на рис. A.25 окрашены в красный цвет).

В формате **GDSII** такая форма описывается с помощью структурного компонента **BOUNDARY**, который представлен на рис. A.25(a) и описывает замкнутый контур, имеющий три или более вершины. На рисунке имя структуры обозначено как **POLY1**, ее маскирующий слой имеет обозначение **5**, а контур задается парами целых чисел, представляющих собой координаты точек, которые образуют замкнутый контур. В контуре имеется двенадцать вершин. В описание добавляется последняя пара координат точки, повторяющей начальную точку, что и определяет контур. Такая запись переменной длины может описывать произвольно сложные многоугольники.

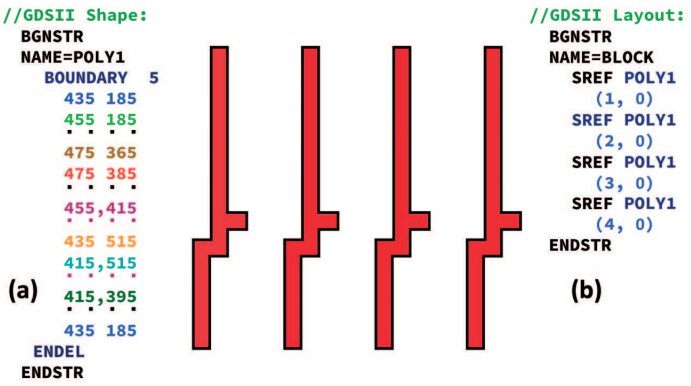


Рис. А.25 Описание GDSII

Для проведения соединений фиксированной ширины в **GDSII** используется синтаксис **PATH**.

После того как структура **POLY1** определена, она при необходимости может быть инстанцирована на кристалле. Код **GDSII** на рис. А.25(б) размещает четыре идентичных экземпляра структуры **POLY1**, располагая их в координатах **1**, **2**, **3**, **4**, что образует структуру более высокого уровня с именем **BLOCK**. Таким образом может быть описана (слой за слоем, вплоть до нанометрового масштаба) вся иерархия проекта **ASIC**.

А.6 Производство ASIC

При передаче в производство огромный файл **GDSII** отправляется в мастерскую по изготовлению масок. Там он разбивается на отдельные слои. Каждый слой данных **GDSII** записывается на хромированную прозрачную маску с использованием очень медленной, но имеющей высокое разрешение технологии. Она представляет собой нанесение рисунка с помощью точно контролируемого растровоориентированного электронного пучка. На рис. А.26 показано, как выглядит типичная поликремниевая маска, на примере переноса паттернов с рис. А.25 на хром. Запись одной маски с помощью электронного луча может занять **8** часов, что определяет высокую стоимость создания масок.

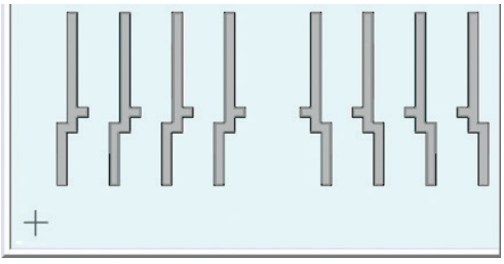


Рис. А.26 Поликремниевая маска (небольшая часть)

При формировании последовательных масок необходимо их точное выравнивание. Раньше для этого наносили отметку совмещения (слева внизу на рисунке),

которая помогает при визуальном выравнивании. В настоящее время для размещения масок используется лазерная интерферометрия.

Десятилетиями при обработке **CMOS** применялась одна маска на слой, однако по мере миниатюризации технологий ситуация меняется. В настоящее время обычно используются две маски для удвоения разрешения (за счет более высоких затрат на маскирование и большей сложности процесса). Данную технику, получившую распространение для **22-нм** технологии, называют двойной литографией (doublepatterning) и используют в первую очередь для критических слоев маски, таких как поликремний.

После того как набор масок **CMOS** сгенерирован, на заводе начинается изготовление пластин (**wafers**). Слой за слоем на поверхности голой пластины из монокристаллического кремния с помощью **IC Compiler** наносятся различные слои кристалла. Пластина со стандартным диаметром **300 мм** вмещает сотни одинаковых кристаллов. Не все получившиеся изделия полностью функциональны. Обязательно возникают производственные дефекты (незамкнутые сквозные перемычки или закороченные металлические межсоединения). Процент рабочих кристаллов от общего их количества на пластине называется выходом годных изделий. Кристаллы на каждой пластине сортируются с помощью электрического тестирования. Гарантированно работоспособные кристаллы (**KGD**, **known-good die**) корпусируются – помещаются в пластиковые или керамические корпуса и соединяются с выводами на корпусе.

А.7 Заключение

Представленная информация об этапах проектирования чипов **ASIC** позволяет считать первоначальную задачу этого приложения – ознакомление инженеров, прошедших обучение проектированию на **FPGA**, с применением имеющихся у них навыков к разработке **ASIC** для конкретных приложений – выполненной. Чип **ASIC**, предназначенный для конкретного изделия (нейропроцессора, сотового телефона, телевизионной приставки для кабельного телевидения или автомобильного контроллера), может часто превосходить аналогичное устройство, реализованное с помощью **FPGA**, и при этом потреблять меньше энергии.

В данном приложении внимание уделено в первую очередь **65-нм** технологии, но обозначенные фундаментальные концепции распространяются и на **32-**, **14-** и **7-нм** технологии. Эти базовые концепции также помогают лучше понять многие современные тенденции в проектировании **ASIC**:

- проектирование с низким энергопотреблением;
- многорежимный синтез;
- распределение высокоскоростных синхросигналов;
- маршрутизация с уменьшенными перекрестными помехами;
- использование инструментов редактирования проекта **ASIC** в соответствии с указаниями о технических изменениях (**ECO**);

- проектирование, ориентированное на производство (**design-for-manufacture**).

Разработчик современной **ASIC** ограничен только своим воображением, настойчивостью и находчивостью. Авторы приложения желают вам, читатель, успеха в проектировании вашего первого чипа **ASIC**!

Отвечает: Владислав Шаршин; интервьюер: Юрий Панчул;
редактор: Александр Романов

Цифровой синтез: практический курс

Приложение Б. История успеха победы российской команды на международном конкурсе Innovate FPGA от Intel

Данное приложение – небольшое интервью с участником российской команды, занявшей второе место на одном из основных международных конкурсов по проектированию на ПЛИС – Innovate FPGA от Intel.

Как вы начали изучать RTL и FPGA?

Изучение этих дисциплин я начал в СПбГЭТУ «ЛЭТИ». Предмет назывался, кажется, «Дискретная математика и цифровые устройства», читала его замечательный педагог Татьяна Яковлевна Новосельцева. Курс начинался с самых основ (булева алгебра, таблицы истинности и т. д.). Сначала было не понятно, как применять полученные знания на практике и где можно попробовать свои силы. И вот в конце семестра нам рассказали про ПЛИС, мы сделали несколько лабораторных работ, мне очень понравилась концепция: в одном маленьком чипе, не применяя пайку, можно сделать столько различных «устройств»! Я понял, что это – мое.

Как вы начали делать проект?

Информацию о начале конкурса[1] я увидел на сайте о FPGA https://marsohod.org/. Я сразу загорелся этой идеей и предложил моему коллеге Андрею Папушину зарегистрировать команду и попробовать свои силы; позже к нам присоединился третий участник – Елена Кириченко. Целью конкурса была разработка устройства на базе отладочной платы с FPGA Intel, которое продемонстрировало бы преимущества FPGA. Главное из преимуществ –возможность параллельной работы разных модулей внутри FPGA. Так появилась идея использовать две камеры одновременно, обрабатывая данные с каждой из них.

Процедура конкурса

Участвовать в конкурсе может любой. Среди участников были школьники и инженеры из России, профессор из Китая, кандидаты наук из Италии и многие другие – со всего мира. Необходимо было зарегистрировать команду и описать основную идею устройства, которое планируется реализовать на отладочной плате с FPGA. Далее следовал этап «народного голосования» – пользователи, зарегистрировавшиеся на сайте конкурса, могли отдать голос за понравившийся проект. Проекты, набравшие большее число голосов, прошли в следующий этап, и организаторы предоставили им отладочную плату для реализации их идеи. Далее в течение нескольких месяцев команды разрабатывали свои устройства. Затем судьи, основываясь на описании проекта, демовидео и ответах на вопросы, отобрали несколько наиболее сильных команд для участия в региональном финале. Затем был дан один месяц на доработку проекта, после чего судьи выбрали три команды из каждого региона, которые были приглашены в США на гранд-финал. В финале каждая команда выступала с презентацией, а потом демонстрировала реальную работу своего устройства. Мы очень рады, что приняли участие в таком событии. Особенно понравилась очная часть в Калифорнии, где была возможность пообщаться с разработчиками со всего мира, погулять по кампусу и музею Intel.

Какие проекты могут делать студенты?

Считаю, что для студентов надо выбирать проекты, результат которых можно «пощупать руками». Непередаваемы ощущения, когда твое устройство впервые

[1] Innovate FPGA – World FPGA Design Contest. URL: http://www.innovatefpga.com.

включилось, заработало и виден результат данной работы, – это отлично мотивирует для дальнейшей учебы. Я рекомендовал бы проекты, реализующие обработку видео.

Как проекты на FPGA отличаются от проектов на процессорах?

Про отличие FPGA от процессора сказано уже очень много разными людьми и на разных ресурсах. Я хотел бы ответить на этот вопрос не с точки зрения архитектуры или процесса разработки, а с точки зрения универсальности решения для обучения и работы. Отладочная плата с FPGA позволяет подключить практически любую возможную периферию, а такое невозможно при работе с микроконтроллером. Если на плате нет UART или I2C – не беда. Все можно реализовать в FPGA: работу с датчиками, моторами, экранами, камерами, АЦП, ЦАП и многое другое. Хочется попрактиковаться в Си – есть soft-процессоры, синтезируемые прямо в FPGA; для пользователя такой процессор не будет отличаться от обычного hard-процессора. Таким образом, FPGA покрывает больше задач, чем процессор; разработчик ничем не ограничен (в разумных пределах). В этом – главное отличие FPGA от процессора: в возможности выйти за рамки, которые установил производитель процессора или другого hard-устройства.

Каким образом упражнения на FPGA подготовят студентов к разработке массовых ASIC'ов?

Работа с FPGA – это отличная возможность разобраться в цифровой схемотехнике, освоить языки описания аппаратуры, временной анализ. При этом вы ничем не рискуете: FPGA можно многократно переконфигурировать, она простит практически любую ошибку.

Каким образом упражнения на FPGA подготовят студентов к разработке систем, в которых часть обработки данных выполняется программами, а другая часть – аппаратно?

FPGA позволяет поработать на низком уровне, разобраться с физикой процессов. Даже если в дальнейшем придется работать на более высоком уровне, знания о том, что передача по UART – это не просто функция в Си, или об архитектуре процессора не будут лишними. Отличную возможность для освоения связки CPU+FPGA дают soft-процессоры, реализованные в FPGA. Есть возможность увидеть – в симуляторе или в железе, – как строка на C или ассемблере, например, зажигает светодиод. Такие опыты дадут изучающему глубокое понимание происходящего и, несомненно, будут полезны в профессиональной деятельности.

Любые другие мысли, которые бы вы хотели сообщить студентам

Изучение цифровой схемотехники и упражнения на FPGA дают понимание основ работы любого цифрового устройства. Они пригодятся как будущему инженеру-железячнику, так и программисту.

Шаршин Владислав Вадимович,

Санкт-Петербург

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Оптовые закупки: тел. +7 (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru**.

Под общей редакцией А. Ю. Романова, Ю. В. Панчула

Цифровой синтез: практический курс

Главный редактор *Мовчан Д. А.*

dmkpress@gmail.com

Корректор *Синяева Г. И.*

Верстка *Орлов И. Ю.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитуры «PT Serif», «PT Sans», «PT Mono». Печать цифровая.
Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5.
Усл. печ. л. 45,18. Тираж 200 экз.
Веб-сайт издательства: **www.dmkpress.com**