Казанский федеральный университет

Институт физики

Акчурин А.Д., Юсупов К.М.

Программирование на языке Verilog

Казань 2016

Печатается по решению Редакционно-издательского совета Института физики Казанского федерального университета

УДК 004.4, 004.5

Принято на заседании кафедры радиоастрономии КФУ
Протокол № 8 от 31 мая 2016 года

Акчурин А.Д., Юсупов К.М. Программирование на языке Verilog. Учебное пособие. – Казань, 2016. – 90 с.

Настоящее пособие предназначено для использования при выполнении практических работ по курсу «Программирование на языке Verilog» студентами кафедры радиоастрономии.

В пособии излагаются типовые конструкции языка описания аппаратуры Verilog HDL и приводятся некоторые примеры логических схем, собранных из типовых конструкций. Учитывается, что современные студенты начинают изучать программирование на примере программирования процессоров (микроконтроллеров), и им очень трудно переключиться на языки программирования аппаратуры (или коротко HDL-языки). Начинающего HDL-программиста сбивает с толку то, что большая часть ключевых слов в HDL-языках и в популярных языках программирования C и Pascal совпадают. Иными словами, тексты на HDL-языках, выглядящие как «настоящие программы» для процессоров, на самом деле таковыми не являются. Они скорее являются способом описания структуры и поведения схемы, которую должна воссоздать программа под общим названием Схемогенератор по тексту HDL-программы. Все существующие схемогенераторы не являются универсальными и способными синтезировать все возможные схемы, поэтому при написании HDL-программы надо держать в уме элементную базу, на которую опирается схемогенератор при сборе общей схемы. Для того чтобы понять принципы работы схемогенератора, пособие содержит исходный код на языке Verilog HDL вместе со схемой, образовавшейся после синтеза и изображенной в так называемом RTL-представлении. Задания, представленные в пособии, охватывают синтез большинства типовых схем (от счетчиков/регистров до цифровых автоматов) и позволяют получить практические навыки программирования на языке Verilog.

Рецензент: Латыпов Р.Р. – к.ф.-м.н., доцент кафедры радиофизики КФУ

Введение

Несмотря на то, что в настоящий момент информация о базовых конструкциях языка Verilog HDL широко представлена и в русскоязычных книгах, и в интернет-документах, она по большей мере является общетеоретической и не содержит примеров конкретных практически реализуемых цифровых схем. Ситуация с освоением языка также осложнена тем, что достаточно большое число операторов языка Verilog HDL не участвуют непосредственно в синтезе цифровых схем и предназначены лишь для тестирования создаваемых схем. Начинающему работать на языке Verilog довольно трудно почувствовать эту разницу. Другой трудностью освоения языка является использование в нем тех же по названию конструкций (ключевых слов), что и в распространенных языках для программирования процессоров (Си, Паскаль и т.д.), толкающих новичков на неверную интерпретацию «знакомых» ключевых слов. С учетом того, что ныне языки программирования процессоров вошли практически во все программы обучения технических вузов, освоение языка Verilog сопровождается борьбой скорее с ненужными стереотипами, почерпнутыми из классических языков, чем с постижением мотивов, положенных в основу синтеза цифровых схем по текстовым конструкциям.

На этом, однако, все трудности начинающего не заканчиваются, еще одну трудность составляет освоение среды программирования, в которой выполняются все работы по разработке цифровой схемы. Из-за того, что такие среды помимо синтеза «абстрактной» цифровой схемы (т.е. не привязанной к конкретной микросхеме ПЛИС – интегральной схеме с программируемыми логическими характеристиками) выполняют достаточно много задач, в частности, «подгонку» (выбор конкретных элементов в конкретной ПЛИС и линий их соединения) полученной при синтезе «абстрактной» схемы к архитектуре (топологии) конкретной ПЛИС, выполняют временной анализ схемы, реализованной в конкретной ПЛИС, помогают в отладке схемы в ПЛИС и т.д. Такая многозадачность среды программирования ПЛИС приводит к тому, что ее интерфейс изобилует множеством инструментов и не всегда является интуитивно понятным, из-за чего начинающий просто теряется.

Вследствие такого положения освоение языка Verilog лучше начинать не с описания ключевых слов языка, как это принято в классических языках программирования процессоров, а с готовых примеров цифровых схем, опирающихся на базовые конструкции синтезируемой части языка Verilog. И здесь самым важным является понимание основных принципов программы, синтезирующей общую логическую схему по тексту Verilog кода. Такую программу называют либо «схемогенератор», либо «схемосинтезатор» (форма наиболее близкая к английскому названию). Для понимания логики работы схемосинтезатора обязательным условием является сопровождение Verilog кода «абстрактной» схемой (не привязанной к аппаратной платформе конкретной ПЛИС), созданной по нему и изображенной графически на так называемом уровне регистровых передач (RTL - register-transfer level). Уровень RTL – это уровень графического изображения схемы, достаточно удаленный от низового уровня транзисторов, когда схема изображается, в основном, с помощью регистров, триггеров, мультиплексоров, дешифраторов, цифровых компараторов, сумматоров и буферных элементов с

минимальным привлечением простейших схем И, ИЛИ, ИсклИЛИ к рисованию RTL-схем. Программное обеспечение современных САПР для больших структурируемых (иерархических) проектов пополняет свои RTL-схемы дополнительными блоками, скрывающими RTL-схему младшего по проектной иерархии модуля и изображаемыми в виде прямоугольников. В таком ключе изображаются цифровые автоматы (или в англоязычной традиции state-machines - машины состояний) без доступа к их внутренней RTL-схеме.

Начинающему HDL-программисту (или FPGA-программисту – по названию технологии самых популярных сейчас программируемых ПЛИС) бывает трудно сориентироваться, как выработать «хороший стиль» программирования, позволяющий получать надежную и компактную схему. Одним из простых способов для формирования такого стиля является просмотр RTL-схемы, создаваемой схемогенератором. В частности, при написании HDL-программ нежелательно «бездумно» наращивать число вложенных или многочисленных ветвей в условных операторах, так это приводит к формированию многочисленных каскадов мультиплексоров, снижающих компактность схемы.

Синтезируемые RTL-схемы у различных производителей программ-схемосинтезаторов могут немного различаться. В данном пособии RTL-схемы получены с помощью САПР Quartus II (от фирмы Altera). Такой выбор связан, в основном, с тем, что данная САПР позволяет не только синтезировать схемы, но также программировать (конфигурировать) ПЛИС, используемые в практических работах, и нет необходимости в изучении программ сторонних производителей. Разработчики САПР уделяют большое внимание выразительности RTL-схем, и не только в смысле удобной компоновки всех элементов и соединительных проводов, но и в смысле красочности их представления. Чтобы читатель в полной мере мог оценить эту работу разработчиков САПР, часть RTL-схем, представленных в пособии, получена в 9-й версии Quartus II (с функциональными узлами без цветовой заливки), а часть – в 13-ой версии (точнее в версии 13.1 с функциональными узлами с сине-зеленой заливкой), вышедшей пять лет спустя после 9-й версии.

Цель этого пособия состоит в том, чтобы продемонстрировать особенности и возможности Verilog для описания поддающихся синтезу цифровых схем и ограничиться использованием только тех конструкций, которые полезны для логического синтеза схем. Однако эти, или очень схожие конструкции, поддерживаются другими средствами разработки. При желании читатель может сравнить проекты Verilog с подобными проектами, используя языки VHDL и AHDL, и увидеть как сильные, так и слабые стороны языка, сравнивая его с другими.

1. Спецификация Verilog HDL в среде разработке Quartus II

Verilog HDL является модульным языком высокого уровня, который полностью интегрирован в среду разработки Quartus II. Файлы проекта Verilog (с расширением *.v) могут быть созданы, используя существующий текстовый редактор Quartus II. Затем они могут быть откомпилированы и промоделированы перед загрузкой в ПЛИС Altera. Файлы с Verilog проектом могут содержать любую комбинацию конструкций, поддерживаемых САПР Quartus II. Они могут также содержать предоставленные Altera логические

функции, включая примитивы, мегафункции и макрофункции и определенные пользователем логические функции. Конструкции Verilog HDL позволяют создавать иерархические проекты как из модулей, написанных только на Verilog-коде, так из проектных файлов Verilog, смешанных в иерархическом проекте с другими типами файлов. Verilog HDL-проекты также легко включаются в другие иерархические (не Verilog) проекты. В текстовом редакторе вы можете автоматически создать символьное изображение схемы, описанной в файле на языке Verilog (Verilog Design File), которое можете вставить в схему, изображенную графически в GDF-файле(ax) (Graphic Design Files). Точно так же в любой файл проекта Verilog вы можете включить как определяемые пользователем функции, так и предоставленные фирмой Altera логические функции.

Компилятор Quartus II позволяет быстро проверять синтаксис Verilog-кода или выполнять полную компиляцию для отладки и обработки проекта. Процессор сообщений Quartus II может использоваться для автоматического определения местонахождения ошибок с их подсветкой в окне редактора текста. После того, как проект успешно откомпилирован, могут быть выполнены дополнительное моделирование и временной анализ. Однако с ростом плотности ПЛИС фирма Altera оказалась не в состоянии поддерживать моделирование внутри САПР Quartus II (с 10-й версии), предложив пользователям обращаться к услугам сторонних производителей. Для этой цели компилятор может также создать выходные Verilog-файлы и файлы в формате Standard Delay Format (SDF - Формат Стандартной Задержки).

HDL-программист (проектировщик схем) может сам определить способ разбиения и назначения устройств для проектного Verilog-файла, чтобы вести логический синтез и отладку проекта, или может поручить компилятору/схемосинтезатору автоматическую реализацию проекта в лучшей комбинации устройств и распределить ресурсы внутри проекта.

Программное обеспечение Quartus II поддерживает подмножество конструкций, определенных стандартом IEEE 1364-2001, то есть оно поддерживает **только** те конструкции, которые относятся к логическому синтезу. Список поддерживаемых конструкций может быть найден в документации Altera на Quartus II.

Небольшие комментарии к используемым в языке Verilog типам данных

Важной составляющей любого языка программирования являются типы данных. По причине широкого распространения процессоров вместе со связанными с ними языками программирования за типами данных в массовом сознании закрепился совокупный образ форм хранения/представления информации с возможными видами операций над ними. Так, для хранения чисел целого типа используется двоичный весовой код определенной разрядности, а для выполнения операций над ними – библиотека целочисленных вычислений и сравнений. Также дело обстоит и с другими популярными типами данных – вещественными и булевскими. Несмотря на различия по форме упомянутых типов, их всех объединяет то, что они хранятся в ячейках памяти, между которыми они «неведомым образом» перемещаются под управлением процессора, причем все изменения их значений происходят, в основном, внутри процессоров, как одного из этапов перемещений данных. В таких условиях, когда данные всех типов лежат в одних и тех же по структуре ячейках

памяти, тип данных всецело задает программа на процессоре, прикладывающая различные библиотеки к конкретным ячейкам. По этой причине тип данных, определяющий методы обработки и явно указанный в программе для процессора, имеет важнейшее значение.

Совсем другая ситуация с некоторыми HDL-языками, где деление между типами опирается на «противостояние» между чисто комбинационными (соединительными) и регистровыми (стробируемыми) схемами, образующими общую проектируемую схему. Несмотря на то, что такое разделение имеется не во всех HDL-языках, мы будем поначалу придерживаться его для простоты изложения работы схемогенератора, крайне необходимой начинающему HDL-программисту. В языке Verilog вышеуказанные типы обозначаются ключевыми словами **wire** и **reg**. Кроме этих типов следующими по востребованности типами являются *константы*, вводимые с помощью ключевого слова **parameter**, и *целые числа*, вводимые с помощью ключевого слова **integer**. Основное назначение типа **integer** – это введение переменных, задающих число повторений в многократном размножении какой-либо части схемы при синтезе общей схемы (но не во время ее работы) внутри циклов (например, **for**, **repeat**). Использование констант (**parameter**) в Verilog также имеет особенности, отсутствующие в обычных языках программирования. Этот тип широко используется в так называемой параметризации модулей, когда сам модуль описан, опираясь (вроде бы) на константу (см., например, примеры 14, 21 и 34 настоящего пособия), однако имеется возможность включения экземпляра (instance) этого модуля в старший по проектной иерархии модуль с изменённым значением «константы». Иными словами, если младший модуль (например, счетчика) был описан через конструкцию `#(parameter SIZE =4)` в заголовке модуля, то по умолчанию он будет четырехразрядным. Однако при включении экземпляра этого модуля в старший по проектной иерархии модуль можно легко переопределить разрядность «вызываемого» счетчика, заменив значение «константы». Разумеется, сам параметризируемый модуль должен быть написан так, чтобы изменение величины **`parameter`**, не приводило к логическим ошибкам. В Verilog также имеется ниша для обычного использования констант, дающая возможность использования тематически (контекстно) понятных имен вместо безликих цифр с помощью того же ключевого слова **`parameter`**, расположенного вне заголовка (см., например, примеры 24 настоящего пособия).

Немного поясним разделение двух основных типов **wire** ↔ **reg** в Verilog. При проектировании «беспроцессорных» цифровых схем (т.е. схем, где отсутствует процессор, управляющий всеми перемещениями и изменениями данных) у разработчика имеется небольшой арсенал функциональных узлов в виде регистров (стробируемых либо фронтом, либо уровнем) и комбинационных схем (собираемых на простейших транзисторных схемах И и ИЛИ, называемых в англоязычной литературе AND-, OR-gates). При таком подходе комбинационные схемы не только выполняют математические операции (что самоочевидно), но и осуществляют маршрутизацию данных, соединяя выходы одного регистра с входом другого (так как в таких схемах нет «всевидящего старшего брата» в виде процессора, забирающего данные с одного порта и отправляющего в другой). Очевидно, такая компоновка цифровых схем без «старшего брата» позволяет организовывать параллельную обработку сигналов, теряя в универсальности схемы. Таким образом, средства соединения/коммутации и промежуточного регистрового

хранения в равной степени делят ответственность за работоспособность схемы, смещая акцент в разделении используемых типов в направлении цепь ↔ регистр (или **wire** ↔ **reg**), вместо традиционного направления, разделяющего формы хранения данных и их обработку (например, целые, вещественные, комплексные, строковые и др.). Учитывая, что «правильность» выполняемых математических операций никто не отменял, то при таком взгляде за правильностью использования типов должен следить не столько компилятор (например, по соответствию типов в операциях присваивания), сколько сам HDL-программист, добавляя ключевое **signed** к базовым типам **wire** или **reg**, чтобы изменить соответствующую комбинационную схему (например, сумматор знаковых целых чисел вместо беззнаковых).

Тут уместно вспомнить ситуацию конца 90-х прошлого столетия, когда в руках отечественного разработчика были микросхемы низкой степени интеграции (К155, К176, К555, К561 и ряда других серий) вместе с ограниченным доступом к микропроцессорным комплектам (К580, К1810 и др.) и связанными с ними микросхемами памяти (К565 и др.). В таких условиях относительно простые схемы (часы, таймеры, частотомеры, секвенсоры и др.) было проще делать из микросхем низкой степени интеграции. С появлением достаточно емких микросхем ПЛИС, которые с избытком изобилуют базовыми логическими блоками (ячейками), примерно близкими функциональным узлам микросхем низкой степени интеграции, ситуация в некотором смысле снова вернулась к концу 90-х, когда разрабатываемую схему можно собирать из микросхем «старой и доброй» ТТЛ-серии (К155 и К555), просто соединяя контакты в графическом редакторе (и никакой трассировки печатных плат и их пайки!). Для «неуемных патриотов» этой серии (или 74-й серии у зарубежного производителя) некоторые производители схемосинтезаторов предоставляют возможность им «тряхнуть стариной» и рисовать/разрабатывать свою схему прямо в функциональных узлах указанных серий. В частности, такая возможность имеется в САПР Quartus II. Правда, без подпитки в виде реальной работы с морально устаревшими ТТЛ-микросхемами и постоянно возрастающими требованиями к скорости разработки такой подход к разработке схем выглядит в настоящее время архаичным.

Разумеется, охватить одним взглядом весь спектр возможных электрических (логических) схем и тем более представить их в одном пособии не представляется возможным, поэтому упомянем лишь одно бурно развивающееся направление – цифровую обработку. Оно напрямую связано с тем фактом, что дешёвые микросхемы АЦП/ЦАП просто «наводнили» почти все выпускаемые ныне электрические схемы. Удаление «лишней» информации (выделение необходимой) на выходе АЦП (особенно в случае его работы со сверхдискретизацией) стало обычной практикой большинства цифровых схем. Самым распространенным узлом таких систем стали цифровые фильтры. Схема практически любого цифрового фильтра представляет яркую иллюстрацию комбинации регистров (параллельных и сдвиговых) и комбинационных схем (сумматоров, перемножителей).

Завершая эту информацию о типах HDL-языков, повторяем, что в языке Verilog базовыми типами являются **wire** и **reg** с соответствующими устоявшимися русскими названиями *цепи* и *регистры*. Значениями переменных данного типа являются значения (хотя сам подход, когда у типа «цепь» имеются переменные данного типа, выглядит немного «притянутым за уши», так как сама по себе цепь/провод ничего хранить не может и

является просто отражением тех регистровых переменных, к которым она косвенно или явно подключена), реально присутствующие на выходах цифровых схем: логические **0** и **1**, и необязательное состояние высокого импеданса **Z** (только для буферов и только в схеме самого верхнего уровня с внешними контактами-портами). Кроме них используется значение неизвестного состояния **X**, непосредственно не существующего на выходах реальных схем. Оно нужно HDL-программисту главным образом для указания безразличных состояний схемы на тех контактах (регистровых переменных) проектируемой схемы, где компилятор/схемосинтезатор выставит либо на 1, либо на 0, исходя из задач оптимизации схемы при ее синтезе. Другой вариант использования состояния **X** – это указание тех входов, чье значение не оказывает влияние на работу разрабатываемой схемы.

В языке Verilog кроме упомянутых выше типов **wire**, **reg**, **parameter** и **integer** имеются и другие типы (с соответствующими им возможными значениями), которые мы здесь не рассматриваем по причине относительной редкости использования или причастности их к другим приложениям языка – тестированию схем. Отметим лишь производные типы, агрегирующие простейшие **wire** и **reg**. Это массивы цепей (шины или многоразрядные (векторные) цепи) и массивы регистров (регистровая память). Последнюю не нужно путать с «полноценной» памятью (память с произвольным доступом, или RAM – random access memory), являющейся спутницей процессоров - массивом регистров (ячеек памяти) со встроенным оптимальным быстродействующим дешифратором адреса регистра (или ячейки памяти). С помощью схемосинтезатора по Verilog-коду можно организовать разнообразный доступ к отдельным регистрам массива регистров, от уже упомянутого произвольного доступа до последовательного, когда информационный поток проходит все регистры, соединенные последовательно при подаче тактовых импульсов. Для схем с такими последовательно включенными регистрами используется несколько названий: FIFO-буфер, многоразрядная линия задержки.

Если на языке Verilog создавать полноценную RAM память (массив регистров вместе с дешифратором), то из-за структуры базовых логических блоков (ячеек) в самой популярной ныне FPGA-топологии в конкретной ПЛИС будет создана неуклюжая громоздкая структура из регистров и мультиплексоров, неудержимо разрастающаяся с ростом объема RAM-памяти. Учитывая, что RAM-память – это очень востребованный элемент в цифровых схемах с высокими требованиями к быстродействию (особенно на стыке интерфейсов), разработчики ПЛИС кроме базовых логических блоков (ячеек) размещают на кристалле «островки» памяти со встроенным дешифратором адреса в «океане» базовых логических блоков (ячеек). Для таких ПЛИС схемосинтезатор должен уметь идентифицировать в Verilog-коде традиционную память и использовать в качестве нее те самые островки (внося в проект зависимость от аппаратной платформы и снижая переносимость проекта на другую ПЛИС), а не «пухлую» схему из базовых логических блоков (ячеек) с неэффективным использованием элементов. Отметим, что из базовых логических блоков (ячеек) более-менее оптимально создаются сдвиговые линии задержки, активно используемые в цифровой обработке сигналов.

2. Реализация комбинационной логики

Комбинационные логические цепи обычно используются как в линиях данных, так и в линиях управления в более сложных системах. Они могут быть созданы/смоделированы по-разному, используя операторы непрерывного присваивания (continuous assignment statements), которые включают выражения с логикой, арифметическими операторами и операторами сравнения. Отличительной особенностью непрерывного присвоения является немедленное изменение состояния присваиваемой цепи (**wire**) при изменении состояния входных цепей, влияющих на нее. Все операторы непрерывного присвоения всегда начинаются с ключевого слова **assign**. Как упоминалось выше, цепи (**wire**) являются средствами соединения/коммутации между регистрами (или, более правильно, между блоками, выполняющими различные функции), поэтому код, начинающийся с ключевого слова **assign**, является элементом структурного описания разрабатываемой схемы.

Для функционального или поведенческого описания в языке Verilog предусмотрены рассмотренные ниже блоки **always**. Имеются, правда, и другие варианты функционального описания – это блоки задач (**task**) и функций (**function**), позволяющие структурировать код в стиле создания подпрограмм в обычных языках программирования с примерным соответствием паскалевским **procedure** и **function** (правда, с более произвольным расположением их в теле программы). В данном пособии они не рассматриваются, отметим лишь, что блоки **task** могут иметь порты **output** и **input**, а блоки **function** – нет.

Кроме выражений с операторами сравнения комбинационные логические цепи могут быть также определены, используя условные операторы (**if** и **case**). Комбинаторная логика, задаваемая в Verilog HDL, использует блоки **always**, которые описывают исключительно комбинаторное поведение, то есть поведение, не зависящее от фронтов тактовых импульсов, используя операторы процедурного (последовательного) присвоения (procedural (sequential) statements). Отличительной особенностью процедурного присвоения является то, что изменение состояния присваиваемой регистровой переменной (**reg**) происходит только под управлением некоторой процедуры, которую в Verilog олицетворяет блок **always**.

Оба эти вида операторов должны быть помещены в теле модуля так, как в следующем шаблоне:

```
module module_name(ports);
     ....
   [continuous_assignments]
   [always_blocks]
     ....
   [continuous_assignments]
   [always_blocks]
     ....
endmodule;
```

или более точно

```
module module_name(ports);
     ....
    [assign operator]  // структурное описание
    [assign operator]
    [always_blocks]    // поведенческое описание
     ....
    [assign operator]
    [assign operator]
    [always_blocks]
     ....
endmodule
```

2.1 Логические и арифметические выражения

И логические, и арифметические выражения могут быть реализованы на кремниевом кристалле ПЛИС, используя логические, сравнительные (относительные) и арифметические операторы. Выражения принимают форму непрерывных присвоений (или назначений в английском варианте - continuous assignments) информации потокового типа. Под непрерывностью присвоений здесь, в первую очередь, понимается «независимость» таких присвоений от тактовых импульсов, так характерная любой комбинаторной логике. «Зависимость» от тактовых импульсов в электрических схемах реализуется различными триггерными и регистровыми схемами, которые будут рассмотрены ниже, начиная с третьей главы.

Логические операторы

Стандартные логические операторы Verilog могут использоваться, чтобы синтезировать комбинационные схемы. Примеры 1 и 2 реализуют логические операторы. Здесь и далее ключевые слова языка Verilog выделены жирным шрифтом. Другие английские термины, не являющиеся ключевыми словами, в тексте пособия помечены серым цветом, и их наличие поможет читающему ориентироваться в англоязычных мануалах. Приводимые примеры являются полностью законченными и работоспособными модулями (с примерным соответствием ключевого слова **module** ключевым словам **main** в C и **program** в Паскале). Иными словами, текст Verilog-кода без ключевых слов **module** и **endmodule** не будет компилироваться (обрабатываться схемосинтезатором). После ключевого слова **module** следует его имя (идентификатор), назначаемое HDL-программистом, с последующим перечислением входных и выходных портов модуля, которые после синтеза схемы (и ее последующей прошивки в ПЛИС) уже можно будет называть входными и выходными портами (контактами) разработанной схемы. Такое различие между модулем и схемой указано не случайно, так как схемосинтезатор всегда занимается оптимизацией схемы и в конце синтеза удаляет входы и внутренние сигналы (цепи и регистры), от которых ничего не зависит. По этой причине синтез схемы будет не возможен, если в объявлении модуля отсутствуют выходные сигналы. Внутренние сигналы (цепи и регистры) и схемы, их формирующие, могут быть удалены в окончательной схеме просто по причине избыточности, обнаруженной при оптимизации схемы.

Пример 1. Языковая конструкция для синтезирования простой логики с однобитовыми входами. Здесь и далее текст на Verilog приводится моноширинным шрифтом.

```
module logic_operators
( input a, b, c, d, // объявление входных портов или входов
 output y);  // объявление выходных портов или выходов
 wire e; // объявление внутренней цепи, необходимой для работы
           // схемы и не являющейся входным/выходным портом
assign y = (a & b) | e;
assign e = (c | d) ;
endmodule
```

Схема, соответствующая этому примеру, показана на рис. 1. Отметим, что схемосинтезатор удалил при синтезе схемы внутренний сигнал (цепь) `e` вместе с ее формирующей схемой 2ИЛИ, заменив последнюю на общую схему 3ИЛИ.

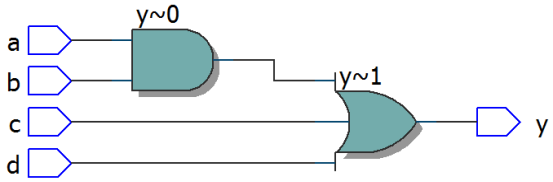


Рис. 1. RTL-схема, соответствующая примеру 1

Пример 2. Языковая конструкция для синтезирования простой логики с четырехразрядными шинами или векторными (многоразрядными) входами.

```
module logic_operators_2
( input [3:0] a, b, // объявление входных векторных портов
 output [3:0] y, // объявление выходного векторного порта
 output z);
assign y = a & b;   // bit-wise AND
assign z = & y;     // reduction AND
endmodule
```

Необходимо обратить внимание, что по отношению к шине(ам) операторы AND и OR могут быть как бинарными (или побитовыми) операторами, так и унарными (или редукционными). Их различие легко понять по рис. 2, глядя на выходные сигналы: шину `y` и однобитовый сигнал z.

Схема, соответствующая этого примеру, показана на рис. 2.

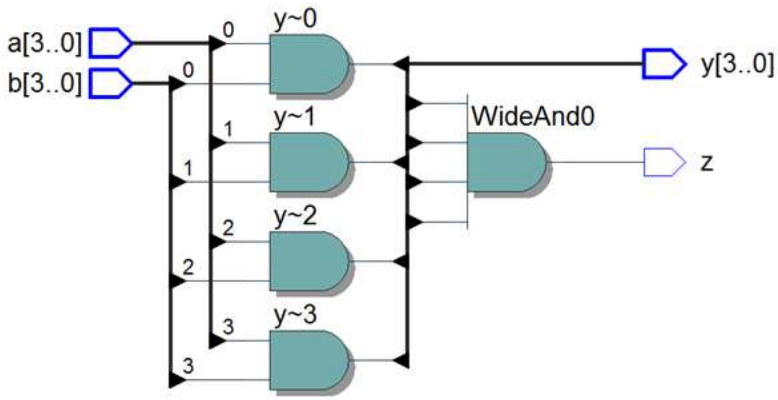


Рис. 2. RTL-схема, соответствующая примеру 2

Применительно к шинам (векторными входами/выходам) здесь необходимо упомянуть очень полезную операцию конкатенации, позволяющую с помощью фигурных скобок объединить множество сигналов в общую шину (хотя возможны записи, когда такое объединение чисто формально, и реального объединения не происходит). Гибкость операции конкатенации придает возможность многократного тиражирования заданной комбинации сигналов. Поясняющий пример

```
{ a, b, c, {3{d, e}}},
```

что является экономной записью «полного» выражения

```
{a, b, c, d, e, d, e, d, e}
```

Пример формального объединения нескольких одноразрядных в одну многоразрядную шину для уменьшения кода в операции непрерывного присвоения (назначения)

```
wire a, b, c; // объявление внутренних цепей, не являющихся
              // входными/выходными портами
assign {a, b, c} = 3'b010;
```

что эквивалентно трем записям

```
assign a = 1'b0;
assign b = 1'b1;
assign c = 1'b0;
```

Другой пример непрерывного присвоения (назначения) с помощью конкатенации многоразрядной шины (векторной цепи)

```
wire [3:0] a; // объявление внутренней 4-х разрядной шины,
              // не являющейся входной/выходной портовой шиной
wire b, c;
assign a = {1'b0, b, 1'b1, c};
```

что эквивалентно четырем записям

```
assign a[0] = c;
assign a[1] = 1'b1;
assign a[2] = b;
assign a[3] = 1'b0;
```

Операция конкатенации очень полезна для описания сдвиговых регистров, приведенного в третьей главе.

Операторы сравнения

Простые операторы определения равенства ("= =" и "! =") определены для всех типов. Получающийся результат для всех этих операторов является булевым. И хотя формально такой тип в Verilog отсутствует, проблем с булевскими данными нет, так как для цифровых (электрических) схем с их рождения было принято связывать логическую 1 (высокое напряжение) с булевским значением True, а логический 0 (низкое напряжение) – с False. Простое сравнение (равно и не равно) аппаратно легче осуществить, чем операторы ранжирования. Пример 3 иллюстрирует использование оператора равенства, чтобы сравнить два 4-битовых входных вектора (две 4-разрядные шины). Соответствующее схемное изображение представлено на рис. 3.

Пример 3. Синтезируемая логика операторов сравнения.

```
module relational_operators_1
(input [3:0] a, b,
 output y);
assign y = (a == b);
endmodule
```

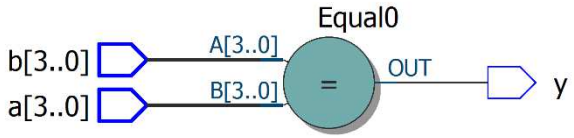


Рис. 3. RTL-схема, соответствующая примеру 3

Пример 4 использует оператор, "больше или равно" ("> ="), и его схематическая диаграмма приведена на рис. 4.

Пример 4. Синтезируемая логика операторов сравнения.

```
module relational_operators_2
( input [3:0] a, b,
 output y);
  assign y = (a >= b);
endmodule
```

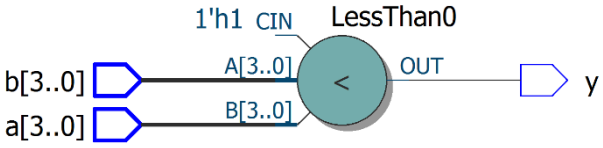


Рис. 4. RTL-схема, соответствующая примеру 4

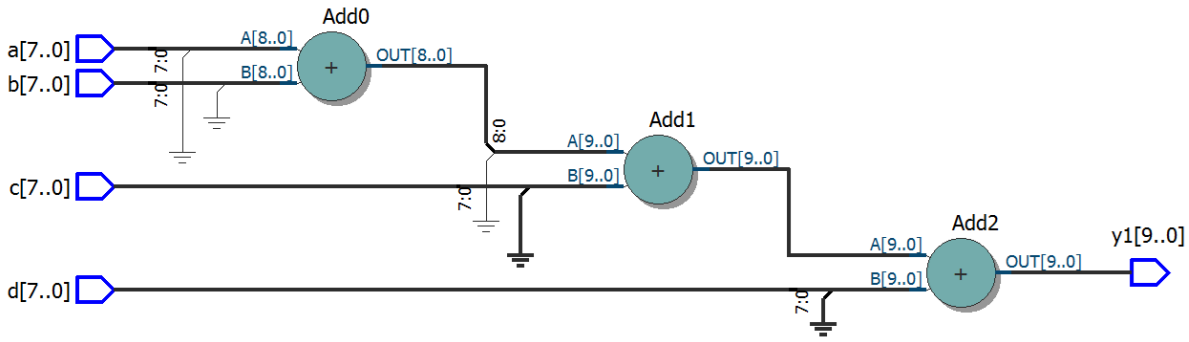
Арифметические операторы

Выполнение этих операторов очень зависит от технологии на целевом устройстве (конкретной ПЛИС). Пример 5 иллюстрирует использование арифметических операторов и круглых скобок, чтобы управлять синтезируемой логической структурой. Сравнение рис. 5а и 5б показывает разницу между цепями y1 и y2, имеющими одинаковый арифметический результат, но разные синтезируемые схемы из-за перерасстановки приоритетов выполнения с помощью скобок. Обратите внимание, что схемосинтезатор «обнаружил» на выбранной HDL-программистом микросхеме ПЛИС готовые сумматоры с большей разрядностью (16-разрядов) и заземлил лишние выводы (старшие разряды).

Пример 5. Применение арифметических операторов

```
module arithmetic_operators
( input [7:0] a, b, c, d,
 output [9:0] y1, y2);
assign y1 = a + b + c + d;
assign y2 = (a + b) + (c + d);
endmodule
```

а)



б)

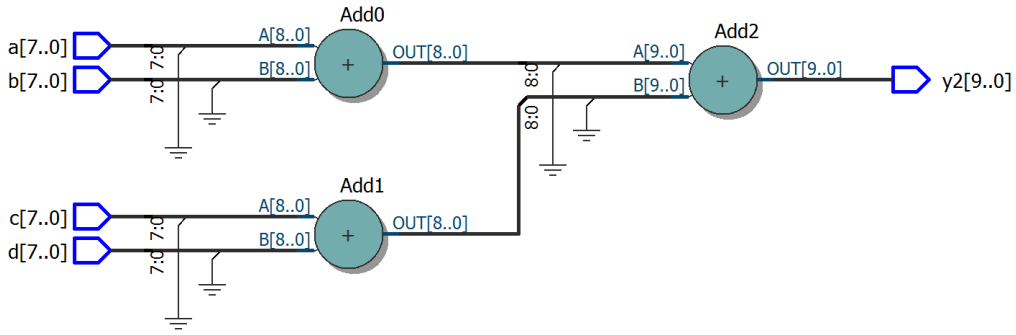


Рис. 5. RTL-схема для примера 5: а) схема, синтезируемая без указания приоритетности выполнения с помощью скобок; б) схема, синтезируемая с указанием приоритетности с помощью скобок

Другая возможность организовать арифметические действия (без стробирования тактовыми импульсами) состоит в том, чтобы заключить операторы присваивания (назначения) значений сигналам в блок **always** со всеми входными сигналами в списке чувствительности оператора **always** (напомним это так называемое процедурное назначение в рамках поведенческого описания). С точки зрения синтеза схемы не будет никакого различия. Однако моделирование (временных задержек) может быть более простым, если блок **always** используется для описания той же самой цепи (выходные сигналы `y1` и `y2`, правда, придется переопределить как **reg**, а не **wire**, как было в примере 5). В этом случае пример 5 может быть переписан и представлен в соответствии с описанием, данным в Примере 6. Синтезируемая логическая схема этого примера будет одинаковой с примером 5.

Пример 6. Использование конструкции **always** для описания арифметической схемы. Обратите внимание на то, что в данном примере использованы операторные скобки **begin** .. **end** для включения нескольких операторов в один **always** блок.

```
module arithmetic_operators_1
( input [7:0] a, b, c, d,
output reg [9:0] y1, y2);
always @(a or b or c or d)
 begin
    y1 = a + b + c + d;
    y2 = (a + b) + (c + d);
 end
endmodule
```

2.2 Условная логика

Verilog дает три типа операторов для создания условной логики:

- оператор **if**,

- оператор **case**,
- тернарный оператор **?:** (работающий и в непрерывных, и в процедурных присвоениях).

Пример 7 иллюстрирует использование оператора **if** для создания условной логики. В этом примере в комментариях показан образец использования тернарного оператора.

Пример 7. Применение условных назначений сигналов

```
module condit_stmts_1
( input [7:0] a, b,
 input sel,
output reg [7:0] y);
always @(a or b or sel)
 begin
        if (sel==1)    y = b;
        else           y = a;
  // эквивалентная запись с помощью тернарного оператора
  //  y = sel? b:a;
 end
endmodule
```

Схема, сгенерированная вышеприведенным примером, показана на рис. 6. Как видим – это множественный мультиплексор 2-в-1 (8 копий одноразрядного мультиплексора). Превращение двухветвевого оператора **if** в мультиплексор 2-в-1 – это стандартный ход HDL-схемосинтезатора (исключение составляют буферные элементы, имеющие высокоимпедансное Z-состояние, см. ниже). Особенности работы схемосинтезатора требуют от HDL-программиста повышенного внимания к работе оператора **if**: если вы не воспользуетесь второй ветвью оператора, то вместо мультиплексора получите триггер, управляемый уровнем (защелку или latch). Иными словами, схемосинтезатор не идет по линии простого игнорирования не указанной в коде ветки оператора **if**, а просто вместо мультиплексора устанавливает триггер, управляемый уровнем (см. пример 15). При таком подходе для отсутствующей ветви оператора **if** на выходе устройства (или той переменной, что указана в существующей ветви оператора **if**) сохраняется последнее значение, установившееся при последнем активном состоянии селектора.

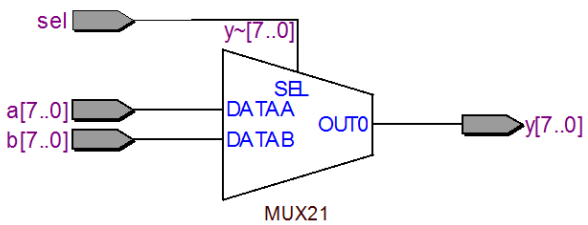


Рис. 6. RTL-схема, сгенерированная на основе кода в примере 7 с применением условного оператора **if**

Пример 8 показывает вариант использования оператора **case** для создания мультиплексора. Все возможные комбинации должны использоваться для выбранных назначений сигнала. Программист/проектировщик может убедиться в этом при использовании ветви **default** у оператора **case** (ветка оператора **case** по умолчанию).

Пример 8. Синтезирование мультиплексора с использованием конструкции **case**

```
module condit_stmts_2
( input a, b, c, d,
 input [1:0] sel,
 output reg y);
always @(sel or a or b or c or d)
  case(sel)
        0: y = a;
        1: y = b;
        2: y = c;
        3: y = d;
  default: y = a;
  endcase
endmodule
```

Схема, соответствующая примеру 8, показана на рис. 7. Видно, что теперь используемый мультиплексор уже не 2-в-1, а 4-в-1. Однако не стоит думать, что именно такой мультиплексор будет задействован на конкретной ПЛИС, так как его может не оказаться в ее топологии, и он будет заменен каскадом последовательных мультиплексоров 2-в-1.

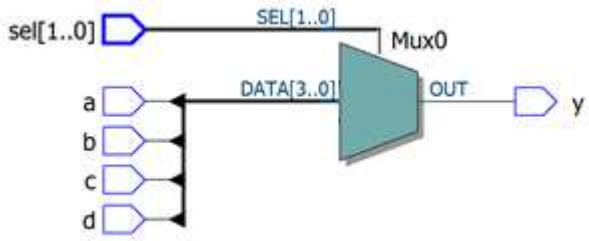


Рис.7. RTL-схема мультиплексора 4-в-1, синтезируемая на основе кода примера 8

2.3 Логика с тремя состояниями

Когда данные от множественных источников должны быть направлены к одному или нескольким приемникам, обычно используются или мультиплексоры, или буферы с тремя состояниями на выходе. Буферы с тремя состояниями реализуются в Verilog с использованием условных операторов:

- оператор **if**,
- оператор **case**,
- тернарный оператор **?:** (работающий и в непрерывных, и в процедурных присвоениях).

Буфер с тремя состояниями подразумевает присвоение высокоимпедансного значения ‘Z’ объекту данных в специальной ветви условного оператора. В случае моделирования множественных буферов, которые связаны с одним и тем же выходом, каждый из этих буферов должен быть описан в отдельном параллельном операторе. Пример 9 показывает четырехбитовый буфер с тремя состояниями, задаваемый тернарным оператором (**?:**).

Пример 9. Синтезирование буфера с тремя состояниями

```
module tbuf4
( input [3:0] a,
 input enable,
 output reg [3:0] y);
assign y = enable? a: 4'bZ;
endmodule
```

Аналогичный результат может быть достигнут с использованием оператора **if**:

```
module tbuf4
( input [3:0] a,
 input enable,
 output [3:0] y,
 reg [3:0] y);
always @(enable or a)
  if (enable == 1)    y = a;
  else                y= 4'bZ;
endmodule
```

Схема, соответствующая примеру 9, показана на рис. 8.

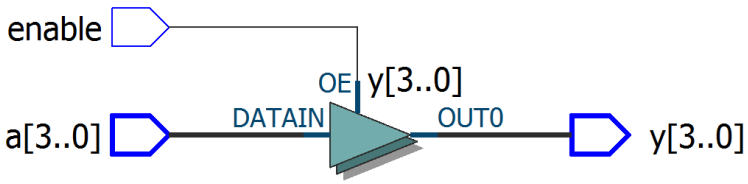


Рис. 8. RTL-схема буфера с тремя состояниями, синтезируемая на основе кода примера 9

2.4 Примеры типовых комбинационных блоков

В этом подразделе мы представим ряд стандартных комбинационных блоков и различные способы описания их в Verilog-коде. Эти блоки обычно представляют конструкции, которые используются для формирования более сложных цифровых проектов. Все эти проекты легко поддаются изменению, чтобы удовлетворить потребностям специфического применения. Различные подходы к синтезу/моделированию используются, чтобы продемонстрировать и гибкость, и мощь Verilog.

Примеры 10а и 10б показывают две различные поведенческие архитектуры шифратора 8-на-3. Первая архитектура использует оператор **if**, в то время как вторая использует оператор **case** внутри блока **always**. Использование оператора **if** вводит б*о*льшие задержки, потому что подразумевает вычисление выражения для сигналов (при моделировании работы схемы) в том порядке, в котором они появляются в модели (выражение в конечной ветви оператора **if** вычисляется последним). Поэтому рекомендуется использование оператора **case**. Это также обеспечивает лучшую удобочитаемость. Отметим тот же прием схемосинтезатора – заменять двухветвевые операторы **if** на мультиплексоры 2-в-1 (в данном случае на каскад последовательных мультиплексоров).

Пример 10а. Шифратор 8-на-3

```
module encoder83
( input [7:0] a,
output reg [2:0] y);
always @(a)
 begin
 if (a == 8'b00000001) y = 0;
 else if(a == 8'b00000010) y = 1;
      else if (a == 8'b00000100) y = 2;
           else if ( a == 8'b00001000) y = 3;
                else if ( a == 8'b00010000) y = 4;
                     else if ( a == 8'b00100000) y = 5;
                          else if ( a == 8'b01000000) y = 6;
                               else if (a == 8'b10000000) y = 7;
                                    else y = 3'bX;
 end
endmodule
```

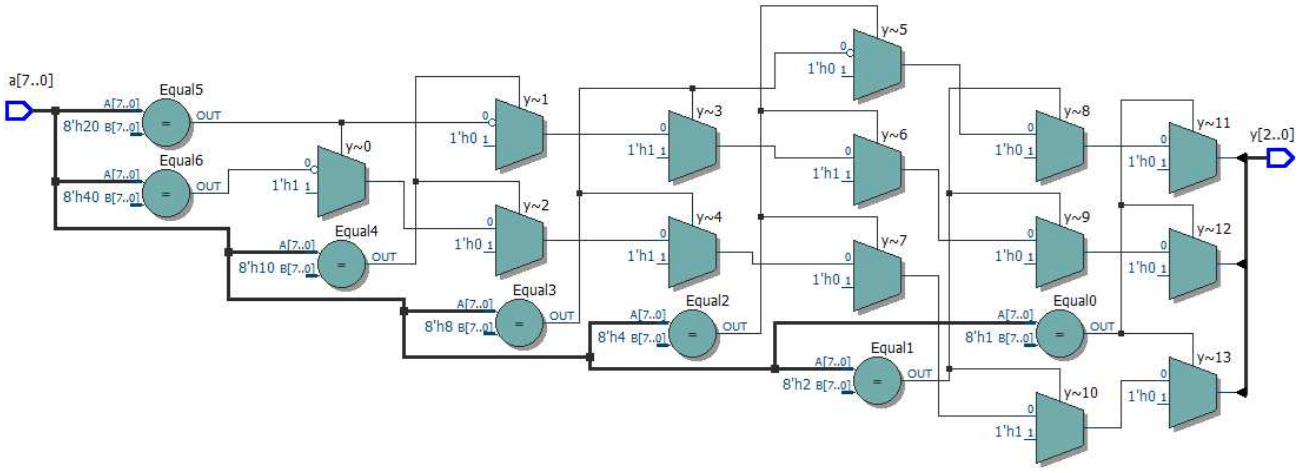


Рис. 9. RTL-схема, сгенерированная на основе кода примера 10а

Пример 10б. Шифратор 8-на-3

```
module encoder83
( input [7:0] a,
output reg [2:0] y);
always @(a)
  begin
   case(a)
       8'b00000001: y = 0;
       8'b00000010: y = 1;
       8'b00000100: y = 2;
       8'b00001000: y = 3;
       8'b00010000: y = 4;
       8'b00100000: y = 5;
       8'b01000000: y = 6;
       8'b10000000: y = 7;
           default: y = 3'bX;
    endcase
   end
endmodule
```

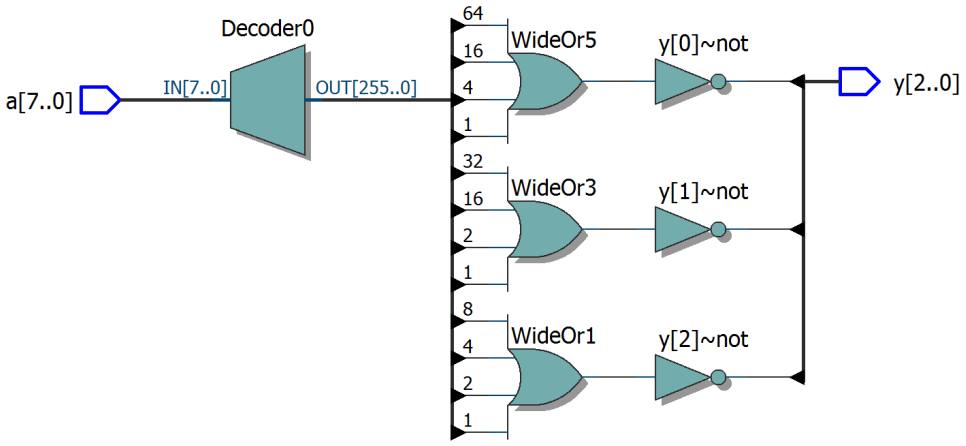


Рис.10. RTL-схема, сгенерированная на основе кода примера 10б

Следующая модель приоритетного шифратора 8-3, представленная в примере 11, использует оператор **for**, чтобы описать его поведение. Выходной порт `valid` указывает, что имеется, по крайней мере, один входной бит со значением логической 1. Отметим, что оператор **for** не имеет обычного для языков программирования процессоров смысла прямого повторения действий при обработке входного сигнала. Здесь оператор **for** несет лишь оттенок послойного наращивания числа элементов (операторов) при синтезе схемы (но не во время ее работы после синтезирования), вложенных в оператор **for** с учетом приоритета (порядка) выполнения повторяющихся операций. В частности, в данном случае схемосинтезатор нарастил слои мультиплексоров (от операторов **if**) до 7 с учетом приоритета выполнения.

Пример 11. Приоритетный шифратор 8-на-3

```
module priority83
(  input [7:0] a,
  output reg [2:0] y,
  output reg valid);
   integer N;
always @(a)
      begin
         valid =0; y = 3'bX;
         for (N=0; N<=7; N=N+1)
              if (a[N])
                   begin
                      y = N; valid = 1;
                   end
     end
endmodule
```

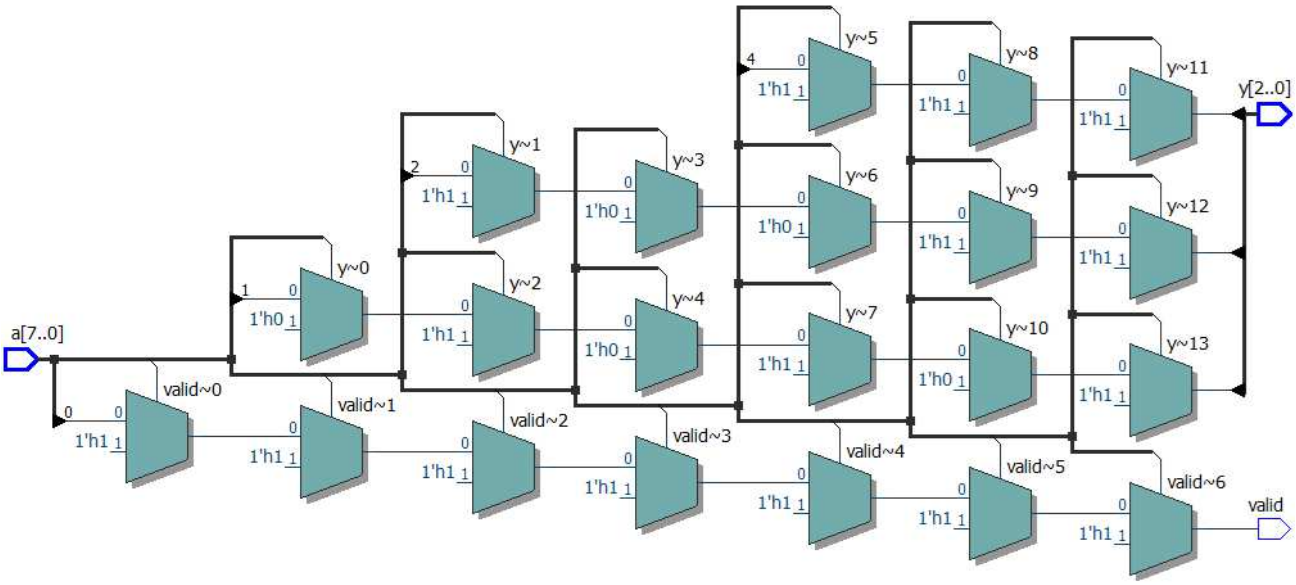


Рис. 11. RTL-схема, сгенерированная на основе кода примера 11

Пример 12 показывает дешифратор 3-на-5, описанный оператором цикла **`for`**, который многократно «выполняет» оператор **`if`** (некоторым образом размножает оператор **`if`**, чтобы получить схему в духе примера 10а, но при этом оператор **`if`** записан только один раз), когда первые пять вариантов входной шины `a` перебираются с помощью переменной `N` в указанном порядке (заданной приоритетности).

Пример 12. Двоичный дешифратор 3-на-5 с входом разрешения

```
module decoder35
(  input [2:0] a,
  output reg [4:0] y);
  integer N;
always @(a)
     begin
        for (N=0; N<=4; N=N+1)
            if (a == N)    y[N]=1;
            else           y[N]=0;
     end
endmodule
```

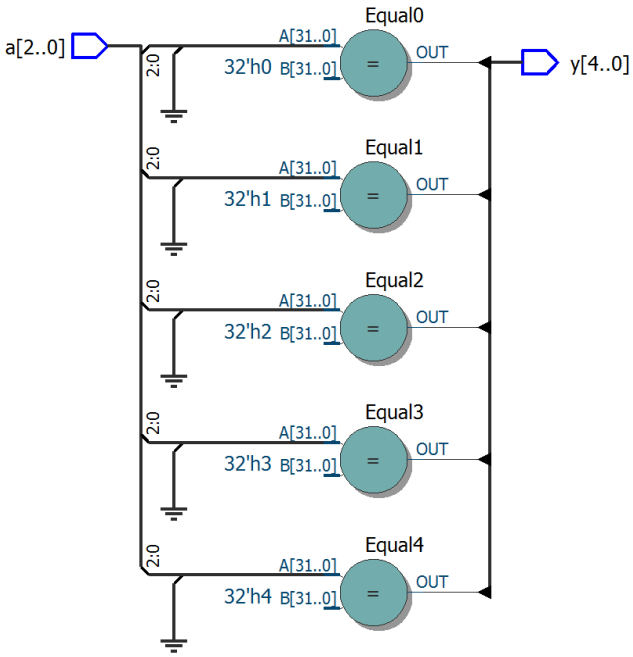


Рис. 12. RTL-схема, сгенерированная на основе кода примера 12

Задание для самостоятельной работы. Разберитесь в схеме, описанной следующим Verilog-кодом. Какую задачу она выполняет?

```
wire [4:0] a, b;
always @(a or b)
      begin
        for (N=0; N<=4; N=N+1)
              out_loop[N] = a[N] && b[N];
                // побитное четырех кратное логическое И
end
```

Задание для самостоятельной работы. Разберитесь в схеме, описанной следующим Verilog-кодом (конвертор кода Грея в двоичный весовой код). Почему это код не рабочий?

```
module gray2bin_always
#(parameter  SIZE =4)
 (output reg [SIZE-1:0]    bin,
  input  [SIZE-1:0]  gray);

 integer  i;
// Syntax Error - variable Index range
always @(gray)
     for (i=0; i<SIZE; i=i+1)
        bin[i] = ^gray[SIZE-1:i];
endmodule
```

Следующий Verilog-код аналогично коду в последнем задании для самостоятельной работы описывает конвертор кода Грея в двоичный весовой код, только реализован на основе конструкции **generate**, а не **always**.

```
module gray2bin_generate
#(parameter  SIZE =4)
(  input  wire [SIZE-1:0]gray,
   output wire [SIZE-1:0]bin
   );

genvar i;

generate
 for (i=0; i<SIZE; i=i+1)
 begin: bit
   assign bin[i] = ^gray[SIZE-1:i];
 end
endgenerate
endmodule
```

Пример 13 показывает дешифратор адреса, который обеспечивает сигналы выбора для сегментов памяти. Адресное пространство памяти содержит 1024 ячейки, представленные 10 адресными битами. Первые два сегмента имеют по 256 ячеек каждый, и третий – по 512 ячеек.

Пример 13. Реализация дешифратора адреса

```
module address_decoder
( input [9:0] address,
 output reg select0, select1, select2);
 always @(address)
   begin        //first segment
     if(address>=0 && address<=255)  select0=1;
     else                            select0=0;
                 //second segment
     if(address>=256 && address<=511) select1=1;
     else                              select1=0;
               //third segment
     if(address>= 512)     select2=1;
     else                  select2=0;
   end
endmodule
```

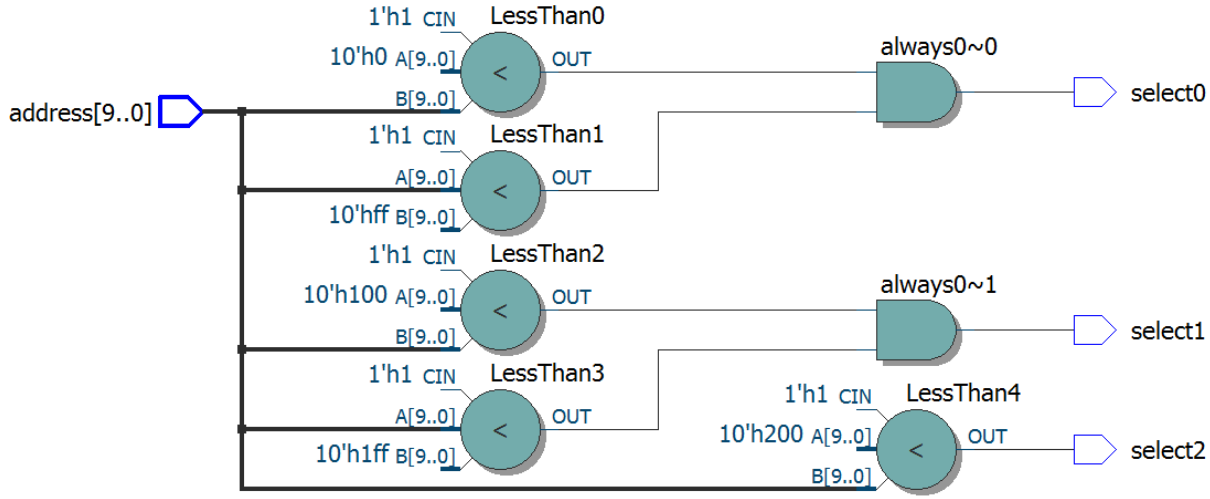


Рис. 13. RTL-схема, сгенерированная на основе кода примера 13

Пример 14 введен только для того, чтобы проиллюстрировать подход к описанию простого арифметико-логического устройства (АЛУ) как более сложной комбинационной схемы. Однако большинство проблем в проекте реального АЛУ связано с эффективным выполнением основных операций (арифметические операции, такие как сложение, вычитание, умножение и деление, операции сдвига и т.д.). АЛУ в этом примере выполняет операции с одним или двумя операндами, которые поступают от двух 8-битовых шин (`a` и `b`), и выводит результат на 8-битовую шину (`f`). Операция, выполняемая АЛУ, определяется кодом, поступившим на входные линии (`opsel`). Входной и выходной сигнал переноса проигнорированы. Коды операции определены в разделе параметров, которые можно легко изменить в начале описания.

Пример 14. Реализация простого АЛУ

```
module alu
#(parameter addab = 4'b0000, inca = 4'b0001, incb = 4'b0010,
            andab = 4'b0011, orab = 4'b0100, nega = 4'b0101,
            shal = 4'b0110, shar = 4'b0111,
            passa = 4'b1000, passb = 4'b1001)
( input [7:0] a, b,
  input [3:0] opsel,
  output reg [7:0] f);
always @(a or b or opsel)
  begin
      case (opsel)
           addab: f = a + b;
            inca: f = a + 1;
            incb: f = b + 1;
           andab: f = a & b;
            orab: f = a | b;
```

```
            nega: f = !a;
            shal: f = a << 1;
            shar: f = a >> 1;
           passa: f = a;
           passb: f = b;
         default: f = 8'bX;
      endcase
  end
endmodule
```

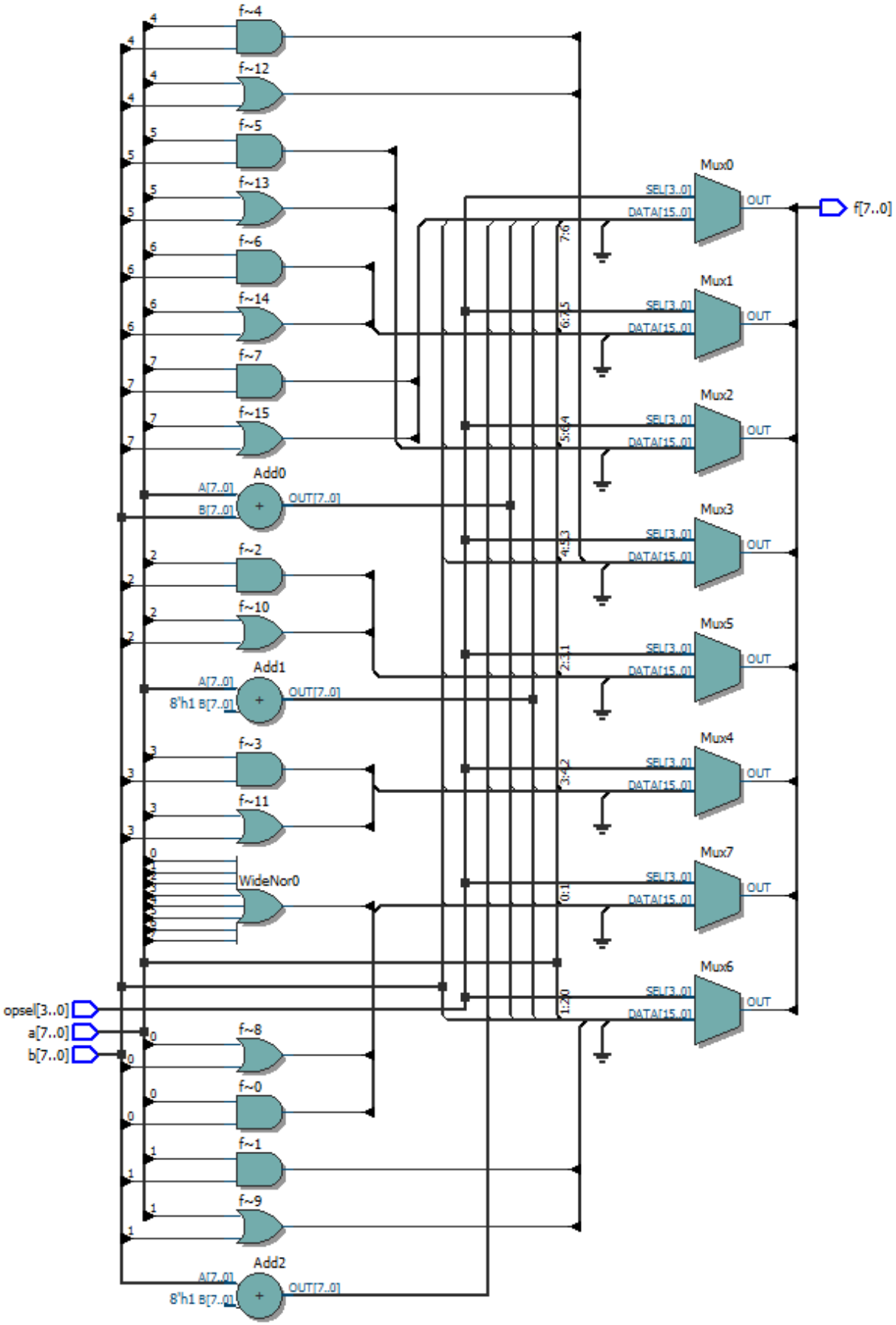


Рис. 14. RTL-схема, сгенерированная на основе кода примера 14

3. Синтез последовательностной логики

Verilog позволяет нам описывать поведение простых логических устройств, таких как регистр и триггер, а также поведение более сложных последовательностных устройств (сменяющих свое состояние по определенной последовательности). Эта глава показывает, как, используя Verilog, можно создавать простые элементы, такие как триггеры, стробируемые как уровнем, так и фронтом, или более сложные стандартные последовательностные блоки, такие как регистры и счетчики.

Для этих целей используются блоки функционального или поведенческого описания схемы в виде блоков **`always`**. В таких **`always`**-блоках имеется возможность описывать поведение схемы, т.е. задавать определенные последовательности сменяемых состояний, что подразумевает наличие в схеме тактируемой памяти, которая обновляется только по приходу определенного тактового сигнала. В остальное время (при неактивном тактовом сигнале) память должна сохранять свое состояние в течение долгого времени. В ходе эволюции простейших цифровых схем памяти, триггеров, сформировались два вида их тактирования – по уровню и по фронту. Но необходимо отметить, что стробируемые фронтом триггеры всегда были гораздо распространённее своих собратьев, стробируемых уровнем. Особенно это хорошо видно в номенклатуре микросхем ТТЛ-серии, где стробируемых уровнем триггеров (и регистров) – раз-два и обчёлся. Такое состояние дел объясняется тем, что у таких триггеров непонятен момент (именно момент) времени, когда запишется сохраняемое значение. Тогда как у стробируемых фронтом триггеров момент фиксации сохраняемого значения длится несколько наносекунд и строго задается конкретным для данного триггера фронтом тактирующего сигнала (передним или задним). Данная информация приведена здесь только для того, чтобы при проектировании быстродействующих схем вы стремились использовать только стробируемые фронтом триггеры (регистры).

Но начнем мы свое описание тактируемых схем все-таки со схем, тактируемых уровнем. Такие схемы могут создаваться схемосинтезатором помимо вашего желания, если вы будете невнимательно относиться к написанию оператора **`if`**. Иными словами, если ваша цель состоит в том, чтобы создать последовательностную логику на основе триггеров, стробируемых уровнем (а не комбинационную логику), то проект должен быть описан, используя одно или даже несколько из приведенных ниже правил:

1. Используйте **не** полностью определенную **`if-then-elseif`** логику, чтобы подразумевалось, что один или более сигналов на выходе схемы должен(ы) сохранять свои значения, полученные при последнем по времени активном значении селектора в операторе **`if`**.

2. Пишите блок **`always`** так, чтобы он **не** включал все входы модуля в списке чувствительности (иначе будет подразумеваться реализация комбинационной схемы).

3. Используйте одну или более переменных таким способом, чтобы они сохраняли значения между повторениями (итерациями) блока **`always`**.

Отметим, что самым сильным, всегда работающим, является первое правило. Остальные правила различными схемосинтезаторами могут не поддерживаться.

3.1 Поведенческое описание фундаментальных стробируемых элементов

Если исключить из рассмотрения RS-триггер, то в большом семействе триггеров самым простым является D-триггер. Остальные триггеры изготовителями микросхем в свое время модернизировались с прицелом для работы в счетчиках, обязательно стробируемых фронтом. Особенностью D-триггеров является то, что они, единственные из триггеров, имели две модификации по способу стробирования: и по уровню, и по фронту. Это означает, что при синтезе триггеров, стробируемых уровнем, на RTL-схеме (как впрочем и на реальной) мы получим D-триггер.

В языке Verilog имеются два главных метода, чтобы описать поведение основных элементов памяти:

• с использованием условных операторов **`if`** и **`case`**, или

• с использованием оператора **`wait`**.

Второй метод использования оператора **`wait`**, однако, не поддерживается схемосинтезаторами и не будет использоваться здесь. Кроме того, в первом методе лучше избегать использования оператора **`case`**, так как нет никакой возможности явно определить сигнал разрешения (`enable`).

Пример 15 описывает триггер, чувствительный к уровню входных сигналов a и b, связанных с его входом. Во всех этих случаях, сигнал "y" сохраняется, если текущее значение сигнала `enable` отлично от ‘1’.

Пример 15. Триггер, чувствительный к уровню входных сигналов

```
module latch1
( input a, b,
  input enable,
  output reg y);
always @(enable or a or b)
  begin
      if (enable)
        y = a & b;      //blocking signal assignment
  end
endmodule
```

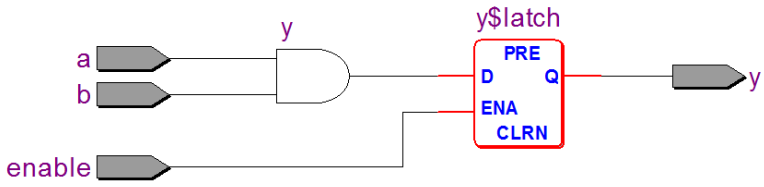


Рис. 15. RTL-схема, сгенерированная на основе кода примера 15

Задание к самостоятельной работе. Сравните пример 7 с примером 15. Почему в одном случае синтезируется мультиплексор, а во втором триггер?

Этот пример может быть легко усложнен внесением дополнительной логики на входах триггера, реализующей любую Булеву функцию, или внесением дополнительных входов, таких как асинхронная предустановка (`preset`) и асинхронный сброс (`clear`). Ранее у изготовителей ТТЛ-микросхем аналогичные входы назывались S (set) и R (reset).

Пример 16 показывает ряд триггеров, смоделированных в пределах функционального блока. Работа всех триггеров разрешается общим входом `enable`. Стоит отметить, что схемосинтезатор никак не изменит схему, если указанный в примере блок **`always`** будет разбит на три отдельных блока **`always`** с соответствующим списком чувствительности. Отметим, что в случае синтеза комбинационной схемы должен использоваться только отдельный блок **`always`**.

Пример 16. Триггеры, выполненные внутри одного функционального блока

```
module latches
(input a1, preset1,
 input a2, clear2,
 input a3, preset3, clear3,
 input enable,
 output reg y1, y2, y3);
always @(enable, a1, a2, a3, preset1, clear2, preset3, clear3)
     begin
       if (preset1)          y1 = 1;
        else if (enable)     y1 = a1;

       if (clear2)           y2 = 0;
        else if(enable)      y2 = a2;

       if (preset3)           y3 = 1;
        else if (clear3)      y3 = 0;
        else if (enable)      y3 = a3;
      end
endmodule
```

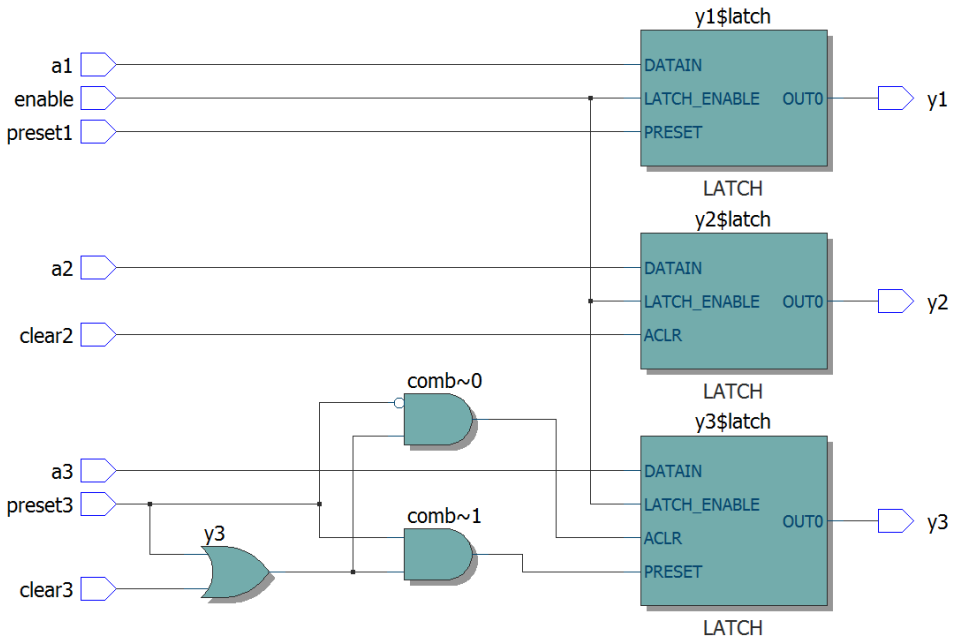


Рис. 16. RTL-схема, сгенерированная на основе кода примера 16

3.2 Синтез регистров и счетчиков

Чувствительность к фронтам сигнала появляется у триггеров и регистров, когда в список чувствительности блока **always** включено ключевое слово либо **posedge**, либо **negedge**, приводящее к чувствительности либо к переднему, либо к заднему фронту соответственно. На самом деле это не совсем верное высказывание, так как особенности схемосинтеза требуют появления в списке чувствительности двух (и более) этих ключевых слов, если потребуется наличие у регистра дополнительного входа для его асинхронного (нетактируемого) сброса. Это, конечно, может удивить начинающего HDL-программиста, но таковы правила.

На языке Verilog у регистра можно создать различные предустановочные входы: сброс, предустановка, разрешение, асинхронная загрузка. Однако это происходит не для всех их комбинаций, так, в частности, нельзя одновременно на одном регистре (триггере) получить и сброс, и предустановку. Это относится к случаю несинтезируемого подмножества схем на языке Verilog. Для создания асинхронных входов сброса и предустановки в списке чувствительности оператора **always** второе ключевое слово **posedge** или **negedge** должно использоваться так, как это показано в Примере 17. Пример 17 показывает несколько способов создания регистров, которые тактируются по сигналу `clk` и управляются по асинхронным сигналам сброса, предустановки и загрузки (`clear`, `preset`, `load`).

Пример 17. Реализация логических регистров

```
module register_inference
(  input d, clk, clear, preset, load,
  output reg q1, q2, q3, q4, q5);
```

```
          // register with active-low clock
always @(negedge clk) q1 = d;
       // register with active-high clock and asynchronous clear
always @(posedge clk, posedge clear)
        if (clear)   q2 = 0;
        else         q2 = d;
      // register with active-high clock and asynchronous preset
always @(posedge clk, posedge preset)
        if (preset)   q3 = 1;
        else          q3 = d;
// register with active-high clock and synchronous load
always @(posedge clk)
        if (load)   q4 = d;
        else        q4 = q4;
// register with active-low clock and asynchronous clear and
preset
always @(negedge clk, posedge clear, posedge preset)
        if (clear)           q5 = 0;
        else if (preset)          q5 = 1;
             else                 q5 = d;
endmodule
```

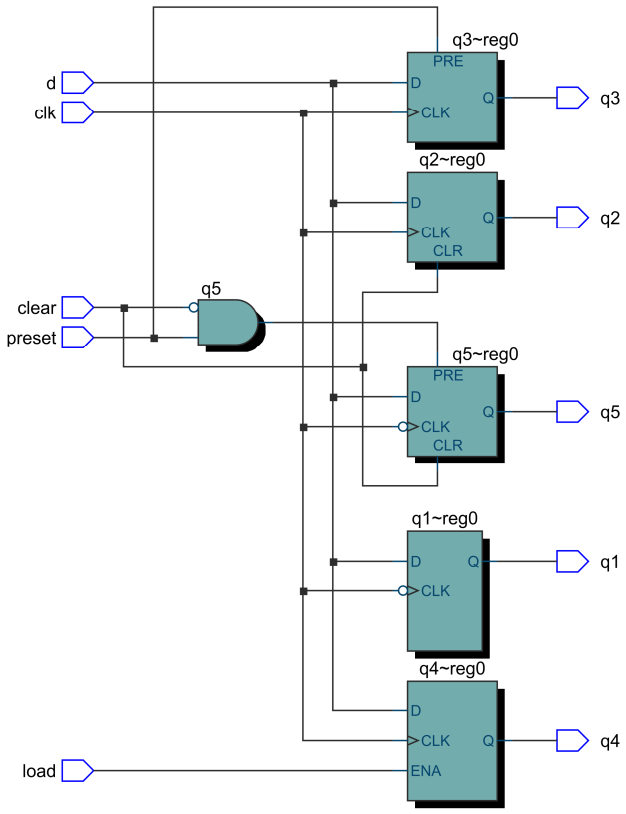


Рис. 17. RTL-схема, сгенерированная на основе кода примера 17

От создания регистра до создания счетчика – один шаг. Счетчик создается схемосинтезатором внесением инкрементируемой регистровой переменной (например, q) внутрь оператора **always**, чувствительного к фронту тактового сигнала clk. Продвинутый счетчик со сбросом или предустановкой создается оператором **if**, что позволяет схемосинтезатору установить мультиплексор, переключающий регистровую переменную q между выходом логики, которая прибавляет или вычитает значение, к текущему значению переменной или к фиксированному значению, выбранному по желанию HDL-программиста (чаще всего это или нулевое, или максимальное значение). Оператор **if** и дополнительная логика должны быть внутри оператора **always**. Пример 18 показывает некоторые Verilog-коды 8-разрядных счетчиков, управляемых сигналами clk, clear, ld, d, enable, и up_down, что реализованы с помощью оператора **if**.

Пример 18. Verilog реализация логических счетчиков

```
module counters
( input [7:0] d,
  input clk, enable, clear, load, up_down,
  output reg [7:0] qa, qb, qc, qd, qe, qf);

    integer direction;
                              // An enable counter
always @(posedge clk)
              if (enable) qa = qa + 1;

                              // A synchronous load counter
always @(posedge clk)
      begin
           if (load)    qb = d;
           else         qb = qb + 1;
      end
                              // A synchronous clear counter
always @(posedge clk)
         begin
             if (clear)  qc = 0;
             else        qc = qc + 1;
         end
                               //An up/down counter
always @(posedge clk)
         begin if (up_down)    direction = 1;
               else            direction = -1;
               qd = qd + direction;
         end
                             // A synchronous load clear counter
always @ (posedge clk)
         begin
```

```
            if (clear) qe = 0;
            else if (load) qe = d;
                 else      qe = qe + 1;
        end
                    // A synchronous load enable up/down counter
always @(posedge clk)
       begin
           if (up_down)         direction = 1;
           else                 direction = -1;

           if (load)       qf = d;
           else if (enable)
                qf = qf + direction;
       end
endmodule
```

Отметим, что все приведенные счетчики являются свободно работающими (если не пользоваться сигналами сброса) во всем диапазоне от нуля до модуля счета, определяемом разрядностью инкрементируемых регистровых переменных `qa`, `qb`, `qc`, `qd`, `qe`, `qf`. Она равна 8, а это значит их модуль счета (максимальное число+1) будет 256. Для таких счетчиков с максимально возможным модулем счета в англоязычной литературе используется специальный термин free-running. Под несвободно работающими (non-free-running) счетчиками понимаются счетчики, работающие с меньшим модулем счета, для этого в Verilog-коде ставится оператор **if**, который по достижению определенного модуля счета (меньшего максимального) сбрасывает состояние счетчика в нулевое.

```
always @(posedge clk)
             if (qa<14) qa = qa + 1;
             else       qa = 0;
```

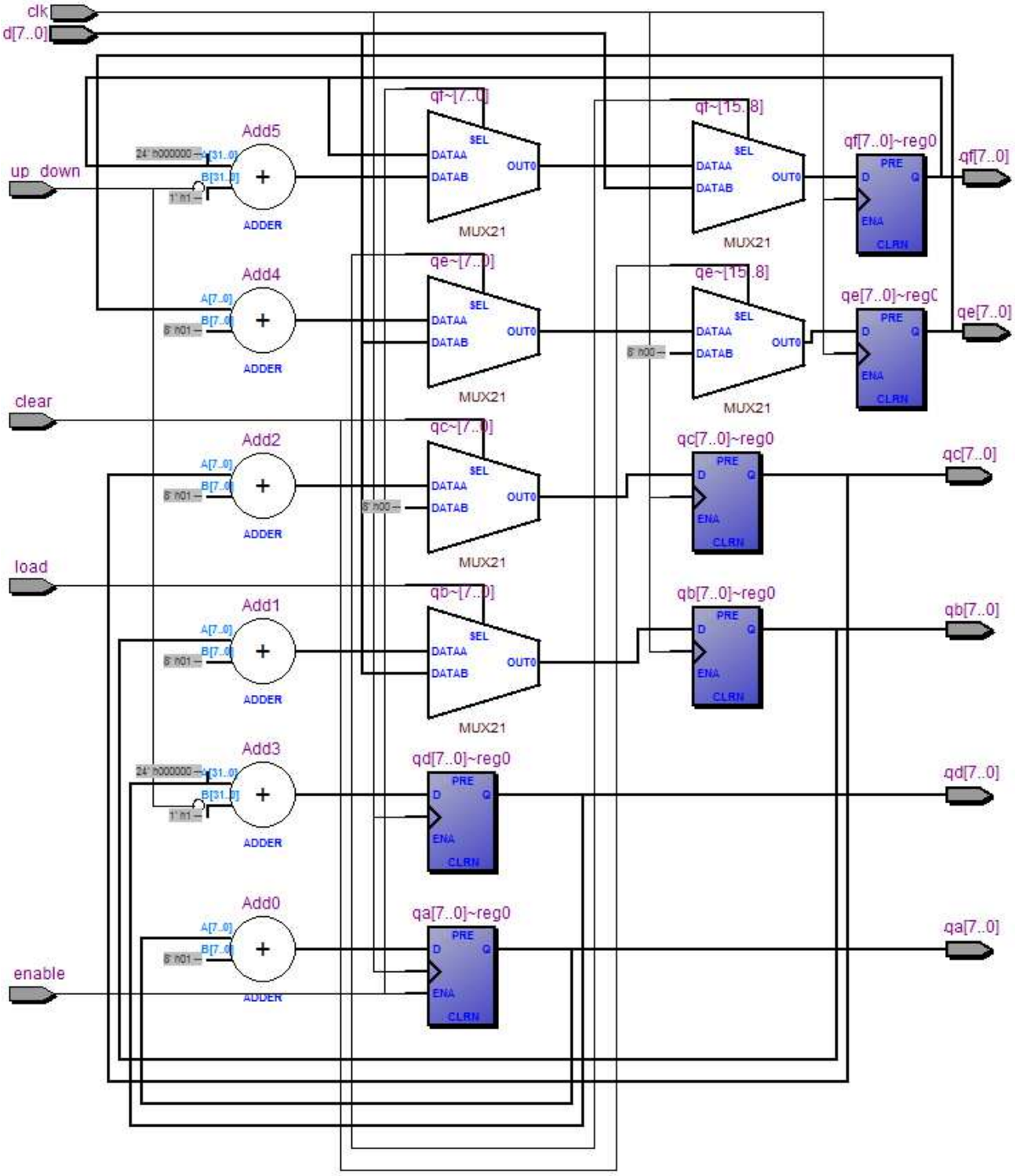


Рис. 18. RTL-схема, сгенерированная на основе кода примера 18

Все операторы **`always`** в примере 18 чувствительны к изменениям только одного входного сигнала `clk`. Все другие управляющие сигналы являются синхронными (т.е. активизируются только по сигналу `clk`).

3.3 Примеры типовой последовательностной логики

Пример 19 демонстрирует проект 16-битового счетчика, который позволяет инициализацию нулевого значения (сброс), и управление режимом счета через выбор шага счета: суммирование с шагом 1 или 2 и вычитание с шагом 1. Он также демонстрирует использование различных типов данных (в частности, типа **integer**).

Пример 19. Verilog реализация 16-битного счетчика со входом разрешения и дополнительным управлением.

```
module flexcount16
( input up1, up2, down1, clk, enable, clear, load,
  input [15:0] d,
 output reg [15:0] q);
 integer direction;
always @(posedge clk or posedge clear)
    begin
       if ((up1 == 1) & (up2==0) & (down1==0))
           direction = 1;
       else if ((up1 == 0) & (up2==1) & (down1==0))
                direction = 2;
            else if ( ( up1 == 0 ) & (up2==0) & (down1==1))
                       direction = -1;
                 else
                       direction = 0;

       if (clear)
           q = 16'b0000_0000_0000_0000;
// для пущей наглядности Verilog разрешает разбивку длинной
// цифровой последовательности с помощью символа _
       else if (load)
                     q = d;
            else if (enable)
                  q = q + direction;
// альтернативный вариант для случая, когда переменная, задающая
// направление счета имеет только два значения
    //    q = q + (direction_up_down? 1 : -1);
     end
endmodule
```

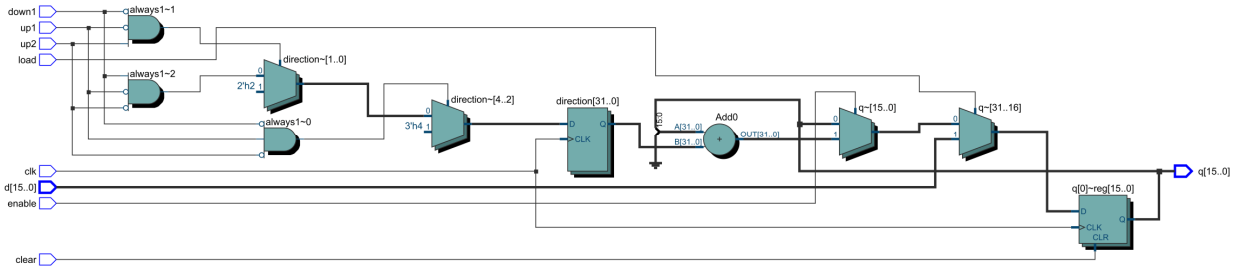


Рис. 19. RTL-схема, сгенерированная на основе кода в примере 19

Пример 20 демонстрируется как представитель делителей частоты (в данном случае делит на 12). Выходной импульс должен появиться после 11-го импульса `clk`, полученного схемой.

Пример 20. Verilog реализация делителя частоты или (non-free-running) счетчика с модулем счета 12.

```
module divider11
( input clk, reset,
  output reg clkdiv11);
  reg [3:0] cnt;
  reg n;

always @(posedge clk, posedge reset) //
    begin
        if (reset)
              begin
               cnt = 0; n = 0;
              end
        else   cnt = cnt + 1;

        if (cnt ==11) n = 1;
        else          n = 0;

        if (n == 1)   cnt = 0;
     end

always @(n)    clkdiv11 = n;
// эквивалентно  assign    clkdiv11 = n;
endmodule
```

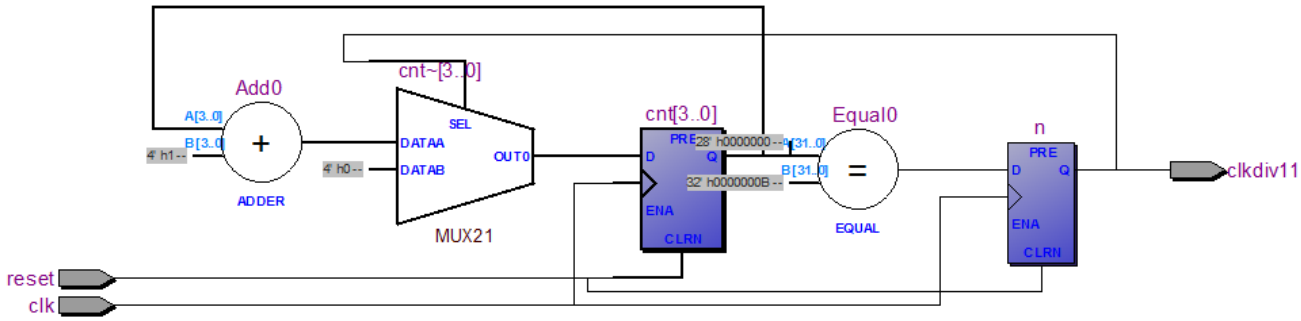


Рис. 20. RTL-схема, сгенерированная на основе кода в примере 20

Помимо формирования счетчиков с заданным модулем счета часто от счетчиков требуется формирование так называемых сигналов переноса (carry), свидетельствующих о достижении счетчиком максимального значения (модуль счета–1). Пример такого счетчика приведен ниже (пример 21). В этом примере сознательно не используется оператор **if** с явной записью максимального числа (например, **if** (count==255) carryN <= 1;), так как это, с одной стороны, позволяет описывать универсальный (параметризуемый) счетчик с переносом, а с другой стороны, демонстрирует возможности конкатенации и возведения в степень.

Пример 21. Verilog-описание параметризуемого счетчика с переносом

```
module  counter_with_carry
#( parameter SIZE = 3 )
( input   rst,
  output reg  [3:0] count,
  output reg  carryN );

always @(posedge clk)
 begin
      if (rst) count <= 0;
      else     count <= count + 1;
 //       первый вариант с использованием конструкции if
      if (count=={SIZE{1'b1}}) carryN <= 1;
 // здесь использована конкатенация 1'b1 единичек SIZE раз
      else                  carryN <= 0;
//       второй вариант с использованием конструкции if
//      if (count== 2**SIZE -1)  carryN <= 1;
 // здесь использовано возведение 2  в степень SIZE
//      else                  carryN <= 0;
//третий вариант с использованием условного тернарного оператора
//        carryN <= (count == 2**SIZE -1)? 1'b1:1'b0;
 end
endmodule
```

Еще одна полезная схема – таймер. Таймер – это схема, задающая точные временные интервалы, основанная, например, на подсчете тактов внешнего опорного сигнала с заданной частотой (периодом). Временной интервал задается как кратное число периодов тактового сигнала. Если таймер строится на вычитающем счетчике, то начальное значение, задающее временной интервал, хранится во внутреннем регистре. При запуске таймера начинается вычитание текущего значение таймера, этот процесс декрементации выполняется при каждом или положительном или отрицательном фронте тактового сигнала. Когда значение внутреннего счетчика таймера достигает нуля, желаемый временной интервал истекает. Процесс декрементации является активным, пока внешний сигнал enable , управляемый внешним процессом, является активным. Блок-схема таймера представлена на рис. 21. Описание таймера на Verilog дано в Примере 22.

Пример 22. Verilog-описание таймера или вычитающего счетчика с модулем счета, код которого содержится на входной шине `data`.

```
module timer
( input clk, load, enable,
  input [15:0] data,
  output reg timeout);
 reg [15:0] cnt;
always @(posedge clk)
       begin
          if (load & !enable)          cnt = data;
          else if (!load & enable)     cnt = cnt - 1;
               else                    cnt = cnt;

         if ( cnt == 0)  timeout = 1;
         else            timeout = 0;
       end
endmodule
```

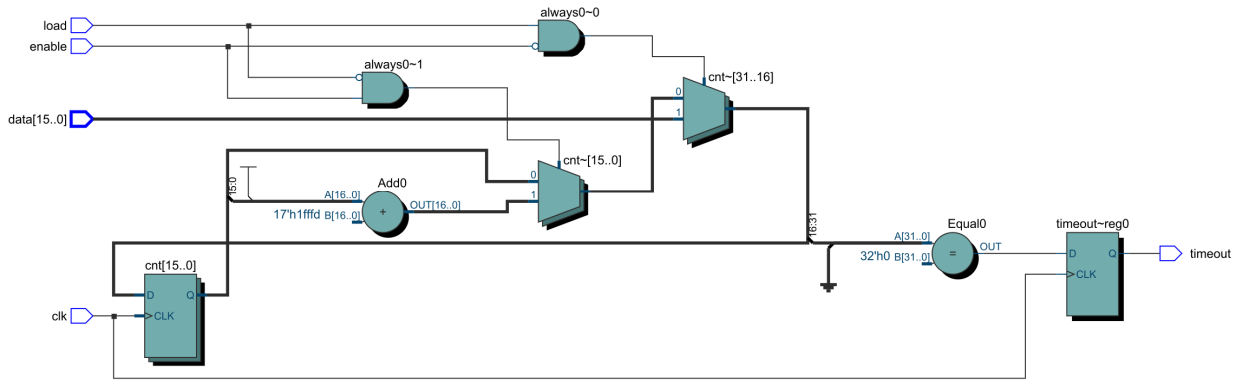


Рис. 21. RTL-схема, сгенерированная на основе кода в примере 21

Приведенный пример с таймером не совсем полноценный, так как на практике таймером считают такое устройство, которое по одному запуску определит только один временной интервал, а не будет делать это непрерывно, как приведенная выше схема, если ее не остановит внешняя схема через вход `enable`. И, тем не менее, в огромном количестве приложений необходима схема, выполняющая свое задание только один раз. В англоязычной литературе для таких схем часто используют термин one-shot. Разработчики схемосинтезатора для таких целей рекомендуют создавать цифровой автомат (см. главу 4 о их синтезе), а не пытаться делать их счетчиках.

Далее мы приведем одну полезную схему, запускающуюся только один раз, – схему, выделяющую передний фронт. Для ее названия используют термин «одновибратор», который вырабатывает одиночный импульс, равный 1 такту, по приходу запускающего сигнала большой длительности (>> такта). Конечно, от идеального одновибратора

(monostable or one-shot multivibrator), который выделяет фронт в момент его появления, он может отстать на 1 такт, но что делать – таковы правила игры.

Пример 23. Verilog-описание одновибратора.

```
module monostable_multivibrator
(input clk, reset,
output reg reset_short);
always @(posedge clk)
  if (reset_short == 1) reset_short <= 0;
  else if  (reset == 1) reset_short <= 1;
       else             reset_short <= 0;
endmodule
```

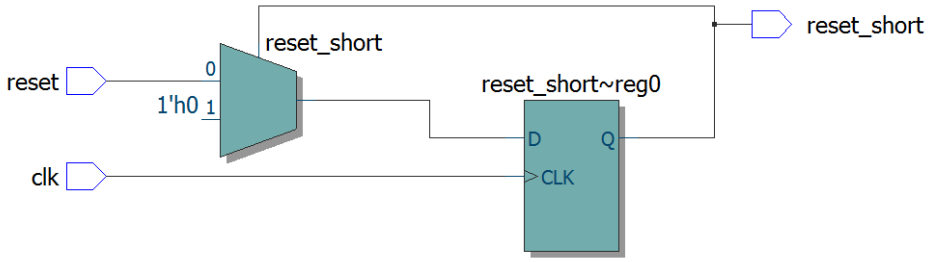


Рис. 22. RTL-схема, сгенерированная на основе кода в примере 23

4. Синтез машин с конечным числом состояний (Finite State Machines)

Кроме счетчиков и комбинационных схем важную часть проекта цифровой системы составляют устройства, способные иметь некоторые состояния, пребывать в них и переходить между ними в заданном (и необязательно циклически как у счетчика, см. рис. 23). В отечественной литературе до 90-х годов для таких устройств использовался термин *цифровой автомат*, а в англоязычной литературе *машины с конечным числом состояний* (Finite State Machines, FSMs) или просто машины состояний. Ввиду доминирования английской литературы по языку Verilog мы будем придерживаться английского термина «машина» и ее английской аббревиатуры FSM.

а) б)

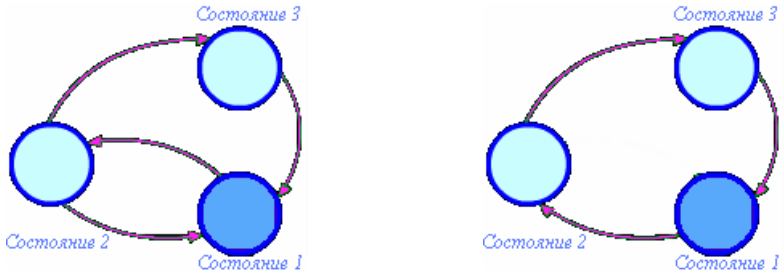


Рис. 23. Диаграмма состояний (а) цифрового автомата и (б) счетчика

В этой главе мы только упоминаем некоторые специфические особенности описания FSM-машин в языке Verilog. Код, описывающий FSM-машину, может быть структурирован в виде трех частей, соответствующих логике *следующего состояния*, логике *текущего состояния* и *выходной* логике. Эти части могут быть сгруппированы по-разному, когда описываются в HDL-коде.

Логика *текущего состояния* задается одним регистром с разрядностью, достаточной, чтобы охватить все возможные состояния разрабатываемой FSM-машины. Поскольку большинство FSM-машин требует, чтобы схемосинтезатор установил FSM-машину в известное начальное состояние, то для этой цели может быть использован или асинхронный, или синхронный сброс. В Verilog только оператор **`if`** может быть использован, чтобы описать поведение этого типа, и в случае асинхронного сброса в список чувствительности оператора **`always`** должны быть включены ключевые слова или **`posedge`** или **`negedge`** по сигналу, тактирующему FSM-машину.

Логика *следующего состояния* лучше всего задается в Verilog, используя оператор **`case`** в блоке **`always`** для комбинационных схем. Данный оператор с селектором в виде *текущего состояния* должен описать все правила необходимых переходов из *текущего состояния*. Ветка по умолчанию этого оператора **`case`** освобождает вас от необходимости явно определять все возможные комбинации переменных состояний, поскольку обычно остаются несколько неиспользованных состояний, что не являются частью жизни FSM-машины. Это значительно упрощает борьбу с «тупиковыми» состояниями: куда попав однажды, FSM-машина выйти самостоятельно не может.

Способ генерации *выходной* логики, зависит от того, какого типа FSM-машину мы используем (Moore или Mealy), что покажем в следующих параграфах.

Для описания состояний и их кодирования может использоваться оператор **`parameter`**, поскольку он позволяет вносить изменения в назначении состояния в единственном месте, если это требуется при настройке кода. В этой главе мы покажем описание FSM-машин типа Moore и Mealy, используя Verilog-код.

4.1 Пример FSM-машины на языке Verilog

FSM-машина, представленная в Примере 24, при активации асинхронного сигнала сброса должна переводить машину к известному начальному состоянию. FSM-машина имеет единственный управляющий вход (`up_down`), два выхода (`lasb` и `msb`) и вход сброса.

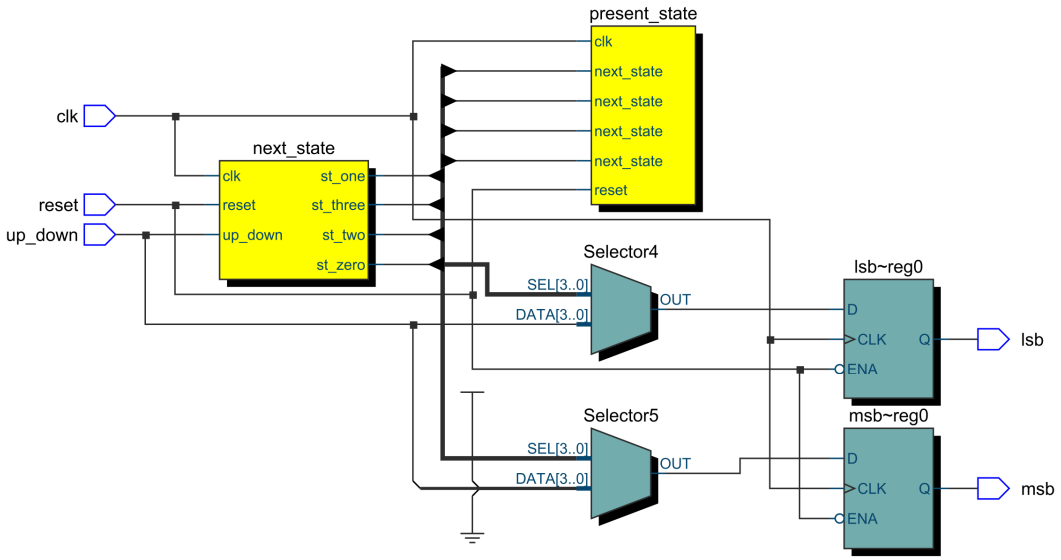
Она может быть в четырех состояниях, которые назначаются двоичными значениями, используя оператор **parameter**. Внутренние регистровые переменные present_state и next_state используются, чтобы описать переходы между состояниями. Переходы между состояниями описаны, используя оператор **case** в пределах блока **always**, который активизирован всякий раз, когда происходит изменение входного сигнала управления или текущего состояния. Другой оператор **always** используется, чтобы синхронизировать переходы между состояниями с тактовыми импульсами (по событию **posedge**) также как переход FSM в начальное состояние (когда происходит сброс). Схема такой FSM-машины приведена на рис. 24а. Обратите внимание на то, что FSM-машина сгенерирована в виде буферной выходной логики и двух закрытых блоков: комбинационного блока с названием next_state и регистра с названием present_state. Для понимания принципов работы этого блока в САПР Quartus II необходимо двойным щелчком левой кнопки мышки «зайти внутрь» блока FSM-машины present_state или next_state, после чего Quartus II сгенерирует схему переходов состояний данной машины конечных состояний, которая приведена на рис. 24б.

Пример 24. Машина с конечным числом состояний

```
module state_machine
(input up_down, clk, reset,
 output reg lsb, msb);

 reg [1:0] present_state, next_state;
 parameter [1:0] st_zero = 2'b11, st_one = 2'b01, st_two =
2'b10, st_three = 2'b00;

//Combinational part
always @(up_down or present_state)
  begin
    case (present_state)
      st_zero: if (up_down == 0) begin
                          next_state = st_one; lsb = 0; msb = 0;
                                 end
               else
                                  begin
                         next_state = st_three; lsb = 1; msb = 1;
                                  end
      st_one: if (up_down == 0)   begin
                          next_state = st_two; lsb = 1; msb = 0;
                                  end
               else
                                  begin
                          next_state = st_zero; lsb = 0; msb = 0;
                                  end
      st_two: if (up_down == 0)    begin
                        next_state = st_three; lsb = 0; msb = 1;
```

```
                                    end
                else
                                    begin
                                next_state = st_one; lsb = 1; msb = 0;
                                    end
        st_three: if (up_down == 0)  begin
                                next_state = st_zero; lsb = 1; msb = 1;
                                    end
                else
                                    begin
                                next_state = st_two; lsb = 0; msb = 1;
                                    end
    endcase
end

//Sequential part
always @(posedge clk or posedge reset)
          begin if (reset) next_state = st_zero;
                else
                                present_state = next_state;
          end
endmodule
```

а)



б)

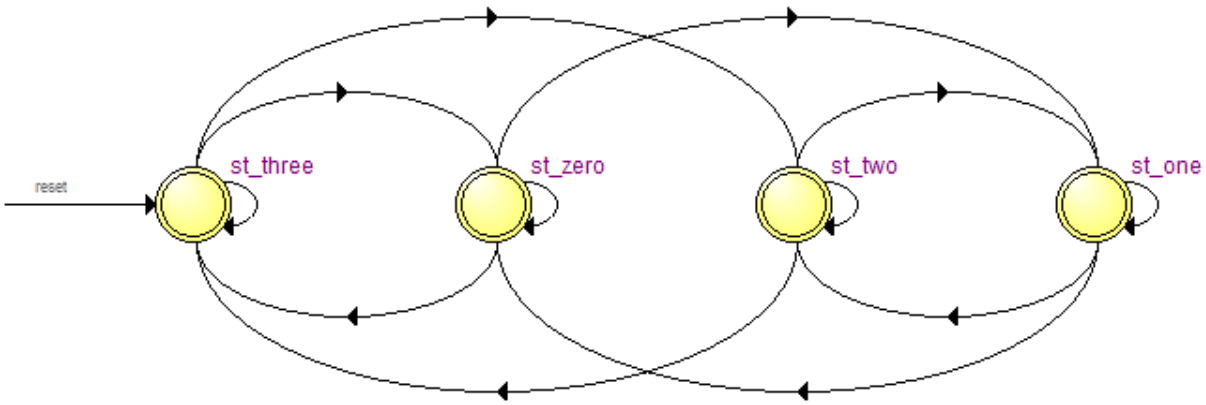


Рис. 24. а) RTL-схема машины с конечным числом состояний, сгенерированная на основе кода примера 24. б) диаграмма переходов, поясняющая логику переходов состояний FSM-машины present_state

4.2 FSM-машины типа Moore

Машина состояний типа Moore имеет выходы, которые являются функцией только текущего состояния. Общая структура FSM-машины типа Moore представлена на рис. 25 с двумя функциональными блоками, которые могут быть осуществлены как комбинационные схемы:

- логикой следующего состояния, которая может быть представлена функцией next_state_logic;
- логикой выхода, которая может быть представлена функцией output_logic.

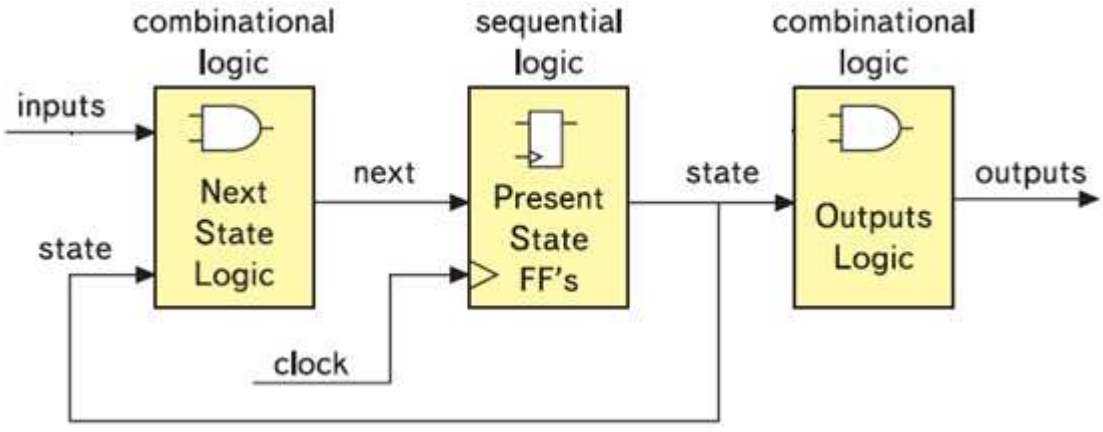


Рис. 25. RTL-схема машины состояний Moore

Выходы обеих этих функций (next_state_logic и output_logic) - это функции соответствующих текущих состояний на их входах. Третий блок - это регистр, который

сохраняет текущее состояние FSM-машины. FSM-машина типа Moore может быть представлена тремя **always** блоками так, что каждый из них соответствует одному из функциональных блоков на рис.25:

Пример 25. FSM-машина типа Moore

```
module moore1(d, a, clk)
( output reg d,
 input a,
 input clk,
 input e);

   reg b; // выход логики, определяющей следующее состояние
   reg c; // текущее состояние

       //    [..] обозначают размер выходной шины
function [..] next_state_logic;
    input a, c;
 begin
       next_state_logic=a+c;
 end
endfunction

function [..] output_logic;
 input c, e;
 begin
       output_logic=c+e;
 end
endfunction

// генерация следующего состояния
always @(a or c) begin
                   b = next_state_logic(a, c);
                 end
// system output
always @(c or e) begin
               d = output_logic(c, e);
            end
// state transition
always @(posedge clk) begin
                        c = b;
                      end
endmodule
```

Более компактное описание этой архитектуры могло быть написано следующим образом:

Пример 26. Verilog-описание FSM-машины типа Moore

```
module moore2
( output reg d,
 input a,
 input clk,
 input e);
   reg c; // текущее состояние

       //    [..] обозначают размер выходной шины
function [..] next_state_logic;
    input a, c;
 begin
       next_state_logic= a+c;
 end
endfunction

function [..] output_logic;
 input c, e;
 begin
       output_logic= c+e;
 end
endfunction

// system output
always @(c) begin
              d = output_logic ( c ) ;
            end
// state transition
always @(posedge clk) begin
                        c = next_state_logic(a, c);
                      end
endmodule // moore2
```

Результат синтезирования схемы этих двух кодов будет одинаков.

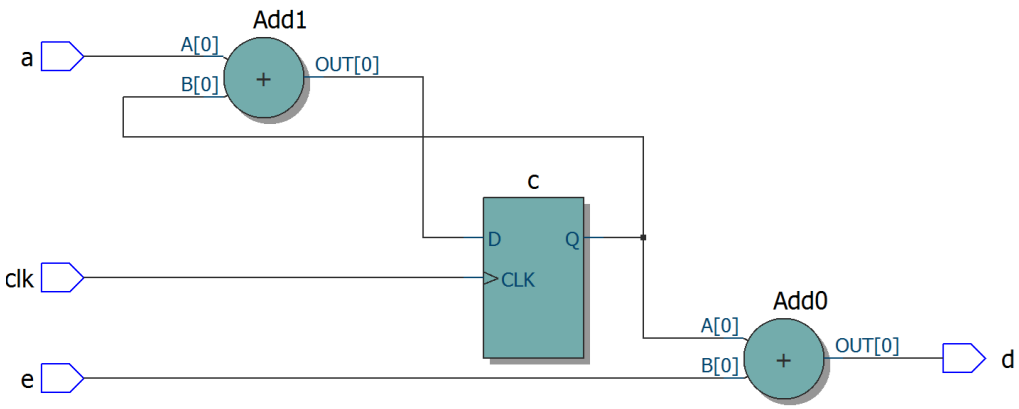


Рис. 26. Машина состояний Moore, реализованная на основе примеров 25 и 26

Фактически FSM-машина типа Moore может часто определяться в единственном функциональном блоке, когда система не требует никакой логики между входами системы и регистрами, или никакой логики между выходом системы и регистрами. В обоих из этих случаях достаточно единственного **always** блока для описания поведения FSM-машины в стиле, показанном в примере 26.

В обоих этих примерах используются функции, чтобы описать схемы, формирующие следующие состояния и выхода. Они могут быть также реализованы, используя любые блоки поведенческого описания, которые комбинируют входы и местные переменные для формирования ее выходной логики. Скобки [..], приведенные в примерах 25 и 26, позволяют обозначить необходимый размер (диапазон битов) выходной шины в рабочем коде.

Различные блоки **always** используются, чтобы отделить описание комбинационных логических блоков от последовательностных логических блоков, которые вместе образуют одну FSM-машину.

4.3 FSM-машины типа Mealy

FSM типа Mealy имеет выходы, которые являются функцией и текущего состояния, и основных входов системы. Общая структура FSM типа Mealy представлена на рис. 27.

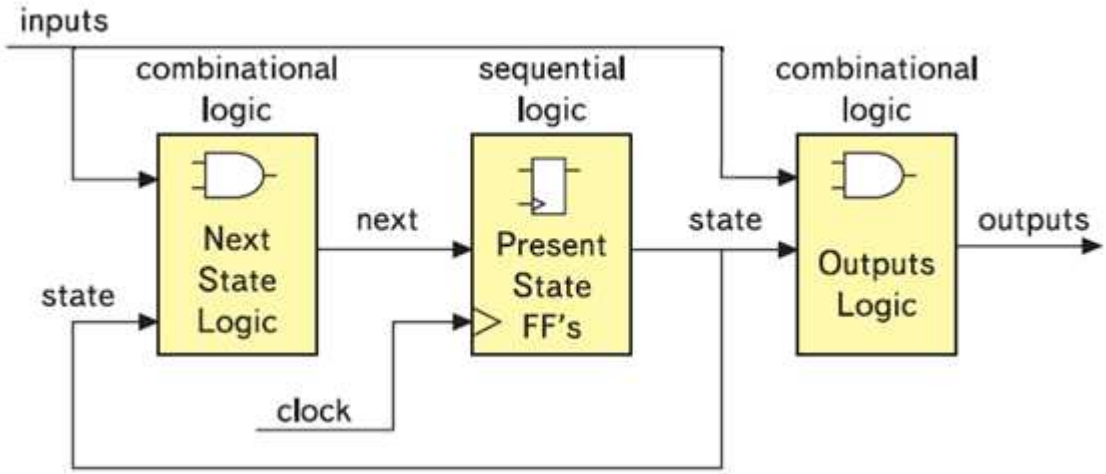


Рис. 27. RTL-схема машины состояний Mealy

FSM-машина типа Mealy может быть представлена следующим общим примером на Verilog подобно машине Moore (пример 27).

Пример 27. Verilog-описание FSM-машины типа Mealy

```
module mealy (d, a, clk)
 output d;
```

```
 input a;
 input clk;
 reg d;
   reg c; // текущее состояние

function next_state_logic;
 input a, c;
   begin
         next_state_logic= a+c;
   end
endfunction

function output_logic;
 input a, c;
    begin
          output_logic= a-c;
    end
endfunction

always @(posedge clk)
       c = next_state_logic(a, c);

always @(a, c)
        d = output_logic(a, c) ;

endmodule // mealy
```

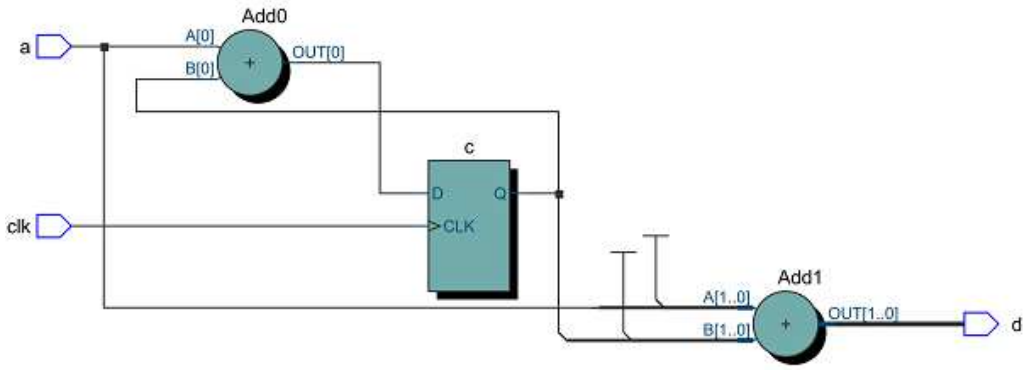


Рис. 28. RTL-схема машины состояний Mealy, реализованная на основе примера 27

Она содержит, по крайней мере, два блока **always**, один из которых предназначен для генерации следующего состояния, и другой – для генерации выхода FSM-машины.

5. Иерархические проекты в Quartus II

Файл проекта Verilog может быть объединен с другими файлами проекта Verilog и с другими файлами проекта от различных инструментов в иерархический проект на любом уровне проектной иерархии.

Помимо примитивов Verilog интегрированная среда проектирования Quartus II обеспечивает множество других примитивов и шин, макрофункций и функций библиотек параметризованных модулей (library of parameterized module, LPM), оптимизированных по архитектуре и для определенных задач. Программист/проектировщик может использовать операторы вставки экземпляра (instance) компонента, чтобы вставить случаи примитивов, макрофункций, LPMs также, как предварительно определенные пользователем компоненты. Более полный список компонентов может быть найден в соответствующих документах Altera, включая файлы помощи в среде Quartus II. В этой главе мы показываем ряд простых примеров, как эти компоненты могут быть вставлены в пользовательский модуль. Цель этого представления – только простое ознакомление с соответствующим синтаксисом Verilog и механизмом внесения экземпляра.

5.1 Функции, определенные пользователем

Verilog позволяет создавать определенные пользователем функции. Любой проектный файл Verilog может стать определенной пользователем функцией после компиляции и генерации. Пример 28 показывает файл reg8.v, в котором содержится проект 8-битового регистра. В случае, если вы захотите использовать его в Verilog-описании другого старшего по проектной иерархии Verilog-файла, то вам придется выполнить ряд несложных записей, показанных в примере 29.

Пример 28. Verilog описание 8-битного регистра

```
module reg8
( output reg [7:0]q,
 input [7:0] d,
 input ena, clk);
always @(posedge clk)
 if (ena) q = d;
endmodule
```

Пример 29 показывает файл reg24.v, что является проектом на языке Verilog, который описывает модуль `reg24`, «вызывающий» (младший по проектной иерархии) модуль `reg8`, не требуя никакой декларации модуля (кроме его имени) внутри себя. Три экземпляра (instances или представителей) модуля `reg8`, включенные в модуль `reg24`, названы как `regA`, `regB` и `regC`. При обработке такого иерархического проекта программа чтения соединения узлов (netlist reader) в Verilog-коде в среде Quartus II автоматически обращается к файлу reg8.v для информации относительно названий входов и порядка их перечисления. Когда же вы не хотите перечислять портовые переменные в

том порядке, как указано в исходном модуле (просто по незнанию этого порядка или вам нужны не все портовые переменные), у вас имеется возможность соединения портов с помощью точки и имени исходного порта (или порта, младшего по проектной иерархии модуля). После чего в скобках указывается порт старшего по проектной иерархии модуля. Примеры такого описания также показаны в примере 29.

Пример 29. Verilog-описание 24-битного регистра, подключающего экземпляры (instances или представителей) 8-битного регистра, расположенного (описанного) в другом файле reg8.v

```
module reg24
(input   [23:0] data,
input  enable, clk,
output  [23:0] out);
// включение экземпляра reg8
// при строгом соответствии порядка перечисления портовых
переменных
 reg8 regA(out[7:0], data[7:0], enable, clk);
// при возможности произвольного порядка перечисления, но не
реализованной до конца
 reg8 regB(.q(out[15:8]), .d(data[15:8]), .ena(enable),
.clk(clk));
// при реализации произвольного порядка перечисления
 reg8 regC(.ena(enable), .clk(clk), .q(out[23:16]),
.d(data[23:16]));
endmodule
```

5.2 Применение параметризированных модулей и Мегафункций

Altera обеспечивает другую абстракцию в форме библиотеки проектных блоков, которые используют параметры для достижения масштабируемости, адаптируемости и эффективного использования кремниевой начинки ПЛИС. Изменяя параметры, пользователь может настроить проектный блок для определенного применения. Например, в функциях памяти параметры используются, чтобы определить: ширину входных и выходных данных; число слов данных, сохраненных в памяти; наличие входных данных, входы адреса/контроля; стробируемы ли выходы; должен ли начальный файл содержимого памяти быть включен в блок RAM; и так далее. HDL-программист (проектировщик) должен объявить названия параметра и значения для функций RAM или ROM при использовании аспектов отображения параметров настройки (generic map aspects). Пример 30 показывает функцию (модуль) lpm_ram_dq со структурой 512 × 8 битов с отдельными входными и выходными портами.

Пример 30. Применение мегафункции памяти

```
module ram256x8
( output[7:0] dataout,
 input [7:0] datain,
 input [8:0] address,
 input we, inclock, outclock);

lpm_ram_dq  ramA(.q(dataout), .data(datain), .address(address),
.we(we), .inclock(inclock), .outclock(outclock));

 defparam  ramA.lpm_width = 8;
 defparam  ramA.lpm_widthad = 9;
endmodule
```

Программист/проектировщик задает (назначает) значения всем параметрам данного экземпляра (instance или представителя) логической функции, используя Altera-определенный оператор **defparam**. Некоторые параметры не требуют определения пользователем значений. Если какое-либо значение параметра не определено, то компилятор ищет значение по умолчанию согласно порядку поиска значения параметра.

6. Сдвиговые регистры

Сдвиговый регистр представляет собой разновидность регистра, для которого по фронту тактового сигнала происходит сдвиг содержимого на один или несколько разрядов в какую-либо сторону (либо в сторону младших разрядов, либо старших). Но прежде чем рассмотреть базовые разновидности этих регистров, необходимо сделать несколько замечаний о двух видах процедурных присвоений (в блоке **always**).

Небольшой комментарий о блокирующих и неблокирующих присвоениях (назначениях)

В языке Verilog имеется одна разновидность присвоения (назначения) в **always** блоках, которую мы пока сознательно обходили, так как для начинающих HDL-программистов она может представлять довольно серьезное затруднение. Она связана с порядком обработки и назначения сигнала на RTL-уровне (на уровне регистров). Дело в том, что начинающие HDL-программисты обычно уже имеют некоторый опыт в написании программ на обычных языках программирования (Си или Паскаль). Там строки программы обрабатываются (процессором) последовательно, одна за другой. И такое положение дел у программиста – в порядке вещей. Однако, и в чем абсолютно уверен любой радиолюбитель («от цифровой схемотехники»), имевший любой опыт работы с микросхемами малой степени интеграции (например, ТТЛ-серий) при наличии параллельных веток в схемах, ни о какой последовательности действий не может быть и речи. Такой радиолюбитель будет уверен, что прохождение всех сигналов по параллельным веткам своей электрической схемы происходит **только** одновременно (в чем несложно убедиться любому вооружившемуся многоканальным осциллографом). В результате такой радиолюбитель (и одновременно начинающий HDL-программист) будет настороженно относиться к последовательному стилю изложения Verilog-кода.

Так как же вразумить первых-несведущих и убедить вторых-предубежденных. Ответ сильно зависит от блока, где осуществляются присвоения (назначения). Самое простое толкование одновременности в блоках структурного присвоения/назначения (**assign**). Там все действия – всегда одновременны, так как время (в виде тактовых импульсов) «над ними не властно», разумеется, кроме переходных процессов и связанных с ними логических гонок. Такое положение дел – следствие роли типа **wire** как в некотором смысле листа соединений (netlist), формально не связанного со временем (за исключением выполняемых математических/булевых функций).

Значительно сложнее ситуация в блоках поведенческого присвоения/назначения (**always**). В этих блоках имеются два вида присвоений: (так называемые) *блокирующие* и *неблокирующие* (blocking и non-blocking). Они привлекаются к работе с помощью знаков соответственно “=” и “<=”. Практически все разъяснения работы этих операторов опираются на сложные описания того, как «симулятор обрабатывает и выполняет неблокирующие процедурные назначения». Безусловно, понимать то, как средства моделирования и оценки временных задержек, базирующиеся на анализе Verilog-кода, очень важно, но это никак не помогает логику работу схемосинтезатора. Поэтому мы в нашем пособии рассматриваем только синтезируемую часть языка, а в разъяснении блокирующих и неблокирующих присвоений мы будем опираться только на RTL-схемы и некоторые простые рекуррентные соотношения, проясняющие их работу.

В грубом приближении блокирующие и неблокирующие присвоения олицетворяют создание параллельного и последовательного включений регистров соответственно. Сначала рассмотрим простой Verilog-код, в котором смешаны все виды присвоений для четырех регистров. Чтобы схемосинтезатор не слишком «увлекался» оптимизацией схемы, все интересующие нас регистры (`a`, `b`, `c`, `d`) через непрерывные присвоения подключены к выходным портам (`Outa`, `Outb`, `Outc`, `Outd`). Выходные порты для схемосинтезатора всегда являются «ценными и незаменимыми» и никогда им не сокращаются.

Пример 31. Verilog-код для иллюстрации блокирующих и неблокирующих присвоений

```
module Blocking_ass
( input clk,
  input  [2:0] a_in,
  output  [2:0] Outa,
  output  [2:0] Outb,
  output  [2:0] Outc,
  output  [2:0] Outd);

reg [2:0] a;
reg [2:0] b;
reg [2:0] c;
reg [2:0] d;
assign Outa= a; assign Outb= b;
assign Outc= c; assign Outd= d;
```

```
always@(posedge clk)
begin
   a=a_in;  b=a;  c=b;  d=c;   //1 случай
//a<=a_in;  b=a;  c=b;  d=c;   //2 случай
//a<=a_in;  b<=a;  c=b;  d=c;  //3 случай
//a<=a_in;  b<=a;  c<=b;  d=c; //4 случай
//a<=a_in;  b<=a;  c<=b;  d<=c; // аналогично 4 случаю
end
endmodule
```

Поочередно выводя из комментариев различные варианты присвоений в блоке **always**, можно с помощью схемосинтезатора получить четыре варианта схем соединений регистров, которые изображены на рис. 29. Хорошо видны предельные случаи параллельного (случай 1) и последовательного (случай 4) включений (блокирующее и неблокирующее присвоение соответственно). Промежуточные случаи (2, 3) помогают понять, почему это происходит.

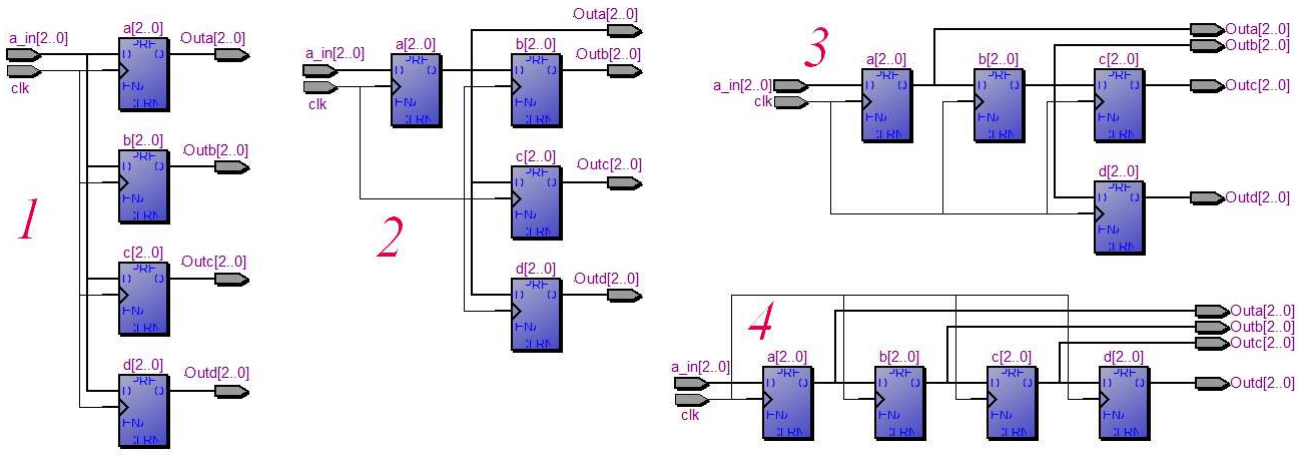


Рис. 29. RTL-схема соединений регистров для четырех вариантов присвоений в примере 31

Одним вариантом объяснения всех вышеприведенных регистровых соединений является такая трактовка одного отдельно взятого регистра (см рис. 30, 31), что на входе у него находится будущее состояние (относящееся к моменту времени t+1), на выходе – уже состоявшееся (относящееся к моменту времени t). При блокирующем присвоении `b=a` соблюдается такое соединение регистров, что $b_{t+1} = a_{t+1}$ (или равны все будущие значения). Это ведет к объединению входов регистров и блокированию их последовательного соединения, несмотря на то, что запись `b=a`, вроде бы, так и просит такое соединение.

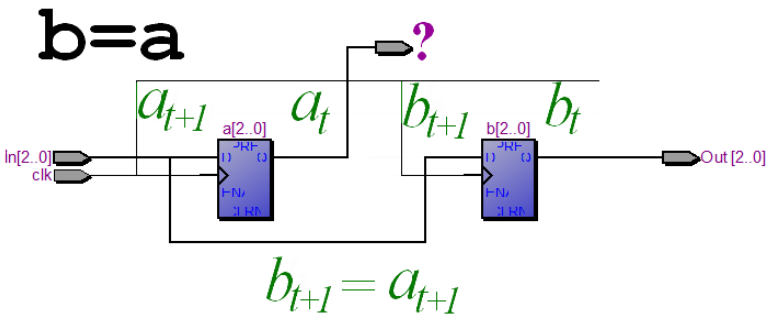


Рис. 30. RTL-схема соединений регистров для одного блокирующего присвоения

Объединение входов регистров ставит под вопрос смысл существования дублирующих друг друга выходов регистров (на рис. 30 один такой выход помечен знаком вопроса). Схемосинтезатор оставит такие «дублирующие» регистры, только если они подключены к выходным портам. В остальных случаях схемосинтезатор уничтожит все дублирующие регистры и оставит только один.

Теперь в том же ключе рассмотрим вариант неблокирующего присвоения `b<=a`. Тут мы видим случай, совпадающий с правилами выполнения расчетов в обычном программировании, а именно, $b_{t+1} = a_t$, где справа стоят прошлые значения, а слева – будущие. Применительно к регистрам это ведет к их последовательному включению (см. рис. 31).

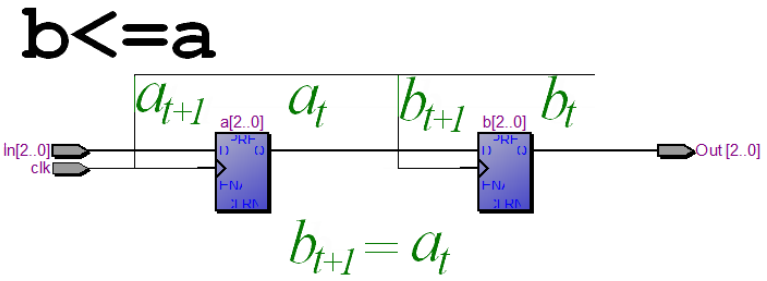


Рис. 31. RTL-схема соединений регистров для одного неблокирующего присвоения

Если подытожить все приведенные сведения о блокирующих и неблокирующих присвоениях, то можно сказать следующее. Блокирующие присвоения – это действительно полностью параллельные действия на параллельной шине, как это и ждал наш радиолюбитель от цифровой схемотехники. Не очень понятным является название такого присвоения, но для удобства запоминания его можно запомнить как «блокировка последовательного соединения». Другой вариант для запоминания блокирующего присвоения: оно блокирует задержку на такт между правой и левой сторонами присвоения, такую привычную для всех языков программирования процессоров.

Неблокирующие присвоения – это тоже параллельные действия, но на некой последовательной шине. Иными словами, при последовательном соединении регистров все действия между всеми отдельно взятыми парами регистров (выходом

предшествующего и входом последующего) происходят одновременно. Разумеется, не нужно полагать, что эти пары регистров должны быть простыми проводами. Если вместо проводов установить комбинационные схемы, выполняющие математические функции, то можно получить некий конвейер вычислений, по которому движется цифровой поток. В такой трактовке легко понять, что в любом наборе присвоений `b<=a`, `c<=b` и т.д. все выражения справа от присвоения относятся к одному своему времени *t*+1, а слева от присвоения – к одному, но уже другому своему времени *t*.

В таком случае мы получаем полную аналогию с обработкой на процессоре, когда все выполняется по шагам (в случае процессора) или от регистра к регистру (в случае неблокирующих присвоений). Т.е. вместо традиционной блок-схемы программы (без ветвлений) для процессора теперь мы получаем цепь регистров с необходимыми вычислительными действиями между регистрами. Единственным важным отличием будет то, что у процессора не будет лишних фоновых действий, так как он сам лично «проходит» все блоки в блок-схеме, оживляя в каждый момент только один блок и его переменные. В цепи регистров это не так, вычисления будут представлять цуг разбросанных по цепочке регистров арифметических действий. Этот цуг будет иметь свои «передний и задний фронты». Таким образом, регистры до и после этих фронтов будут выполнять никому не нужные фоновые вычисления со случайными числами.

Особым и важным случаем всех вышеуказанных присвоений является зацикливание регистровых цепочек. Простейший вариант такого зацикливания – это `b<=a; a<=b`. Это случай вечного обмена данных (swap), причем из-за особенностей структуры регистров никакая информация извне в них не поступает. В таких условиях важнейшее значение имеет самое первое начальное значение, находящееся в регистрах при включении. В языке Verilog для выше указанной инициализации предусмотрен блок **initial**. Пример использования такого блока ниже. Имейте в виду, что из-за особенностей некоторых схемосинтезаторов через блок **`initial`** можно инициализировать лишь внутренние регистровые (а не портовые) переменные.

```
reg a;
reg [1:0] b;

initial begin
  a = 1'b0;
  b = 3'b011;
end
```

Задание для самостоятельной работы. Разберитесь в RTL-схеме, созданной схемосинтезатором с участием более длинной/сложной цепочки присвоений, например, в виде `a=b+1; b=a+1;`. Сравните ее со RTL-схемой, созданной при неблокирующих присвоениях `a<=b+1; b<=a+1;`. Повторите эту работу уже с тремя регистрами (например, `a<=b+1; d<=a+1; b<=d+1;` или `a<=b-1; d<=a+2; b<=d+1;`). Обязательно меняйте стартовые значения регистров с помощью **`initial`** блока. Объясните различия.

И снова к сдвиговым регистрам

В примере 32 показан Verilog-код сдвигового регистра, сдвигающего по тактам `clk` содержимое внутреннего регистра `data` на один разряд влево. В младший разряд помещается значение, поданное на вход `d_in`, при этом старший разряд `data[15]` теряется. Работа регистра управляется входом `ce`. Обратите внимание, что в строке `data <= {data[15:1], d_in}` производится как запись, так и чтение переменной `data`. Так как для порта типа **`output`** допустима только операция записи, то эта переменная **не** может быть портом **`output`**. Дополнительно введена переменная `q`, которая является копией внутреннего сигнала `data`.

Пример 32. Сдвиговый регистр с перемещением битов в сторону старших разрядов

```
module shift_reg1x8
( input clk,
  input d_in,
  input ce,
  output q );

reg [7:0] data;
always @(posedge clk)
 begin
     if (ce)  data <= {data[6:0], d_in};
                              // конкатенация с входным d_in
  //  data = data  << 1; // альтернативный вариант с помощью  <<
  //  data[0] = d_in;    // удобен внутри оператора for
 end
assign q = data[7];
endmodule
```

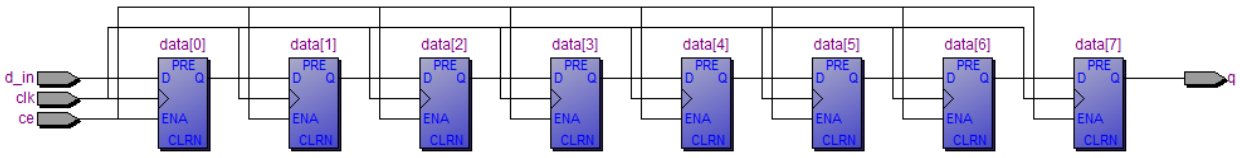


Рис. 32. RTL-схема сдвигового регистра с перемещением битов в сторону старших разрядов

Направление сдвига легко можно поменять, изменив лишь индексы в скобках. В примере 33 показан Verilog-код сдвигового регистра с управляемым направлением сдвига.

Пример 33. Сдвиговый регистр с управляемым направлением сдвига

```
module shift_leftright
( input  clk, d_in, left_right,
```

```
  output [7:0] d_out);

reg [7:0] tmp;

always @(posedge clk)
    if (left_right==1'b0)       tmp = {tmp[6:0], d_in};
    else                        tmp = {d_in, tmp[7:1]};
assign d_out = tmp;
endmodule
```

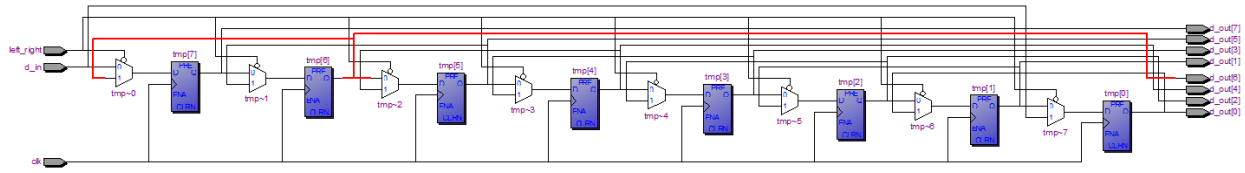


Рис. 33. RTL-схема сдвигового регистра с управляемым направлением сдвига для кода в примере 33

Более сложное описание сдвигового регистра с разрешением сдвига и асинхронным сбросом на основе оператора **for** дано в примере 34.

Пример 34. Сдвиговый регистр со сбросом

```
module shift_reg1x8
# (parameter W = 8) // разрядность регистра
(input in_data, // вход данных регистра
 input clk, // вход тактовых импульсов
 input en, //разрешение сдвига
 input reset_n, // сброс (активный уровень «0»)
 output q); // выход сдвигового регистра

 reg [W-1:0] q_int; // внутренняя переменная
 integer i; // переменная цикла

always @ (posedge clk or negedge reset_n)
 begin
   if (!reset_n) // сброс регистра
         q_int<=8'b0;
   else if (en) // разрешение сдвига
       begin
         q_int[0]<= in_data; // запись в мл. разряд регистра
         for (i=1; i<W; i=i+1) // реализация сдвига
           q_int[i]<= q_int[i-1];
       end
```

```
 end
assign q=q_int[W-1]; // старший разряд является выходом регистра
endmodule
```

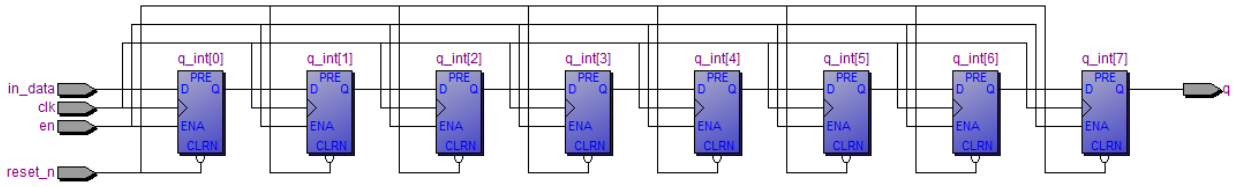


Рис. 34. RTL-схема сдвигового регистра со сбросом для кода в примере 34

Сдвиговые регистры не обязательно должны быть одноразрядными. Очень часто при цифровой обработке сигналов требуются N-разрядные сдвиговые регистры. Однако практики цифровой обработки сигналов такие устройства предпочитают называть *линией задержки*. По своей структуре цифровая линия задержки (N-разрядный сдвиговый регистр) совпадает со структурой FIFO-памяти (First In, First Out). С небольшими переделками одноразрядный сдвиговый регистр превращается в N-разрядный – для этого надо ввести регистровый массив в виде **reg** `[7:0] sr [63:0]`. В данном примере 8-разрядный регистровый массив из 64 элементов. После такого объявления регистров в Verilog доступ к отдельным битам отдельного регистра теряется, и остается лишь одномоментный доступ сразу ко всем битам отдельного регистра `sr[n]`. Verilog-код для линии задержки 8×64 или 8-разрядного FIFO-буфера (FIFO-памяти) на 64 значения с отводами (taps) для внешнего использования на 16, 32, 48 и 64 ячейках (звеньях) дан в примере 35.

Пример 35. Линия задержки 8×64 с отводами для внешнего использования на 16, 32, 48 и 64 ячейках

```
module shift_8x64_taps (
 input clk, shift,
 input [7:0] sr_in,
 output [7:0] sr_tap_one, sr_tap_two, sr_tap_three, sr_out );

   reg [7:0] sr [63:0];
   integer n;

always@(posedge clk)
   begin
     if (shift == 1'b1)
     begin
       for (n = 63; n>0; n = n-1)
       begin
          sr[n] <= sr[n-1];
       end
```

```
        sr[0] <= sr_in;
      end
    end
            // выходы входной последовательности sr_in
assign sr_tap_one = sr[15];   // с задержкой на 16 тактов
assign sr_tap_two = sr[31];   // с задержкой на 32 такта
assign sr_tap_three = sr[47]; // с задержкой на 48 тактов
assign sr_out = sr[63];       // с задержкой на 64 такта

endmodule
```

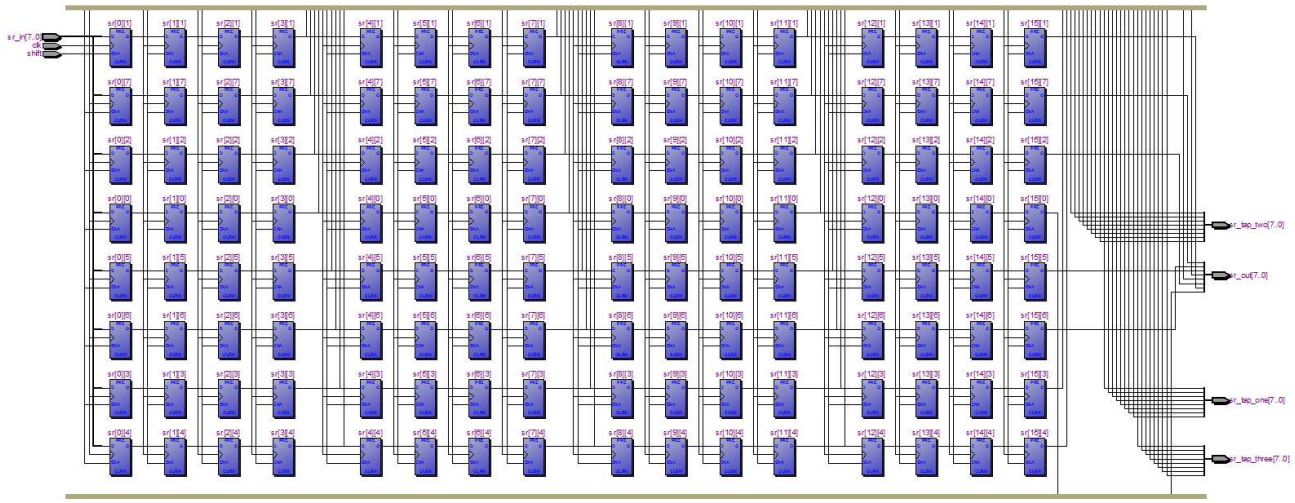


Рис. 35. RTL-схема линии задержки 8×16 с отводами для внешнего использования на 4, 8, 12 и 16 ячейках (упрощенная версия примера 32)

Задание для самостоятельной работы. Объясните работу устройств, описанных следующими Verilog-кодами

код 1

```
module shift5byte
( input clk, shift,
  input [39:0] sr_in,
  output [7:0] sr_out);

  reg [7:0] sr [4:0];
  integer n;

  always@(posedge clk)
    begin
```

```
          if (shift == 1'b1)
          begin
               for (n = 4; n>0; n = n-1)
               begin
                   sr[n] <= sr[n-1];
               end

               sr[0] <= 0; //sr_in;
          end
          else
          begin
              sr[0] <= sr_in[ 7: 0];
              sr[1] <= sr_in[15: 8];
              sr[2] <= sr_in[23:16];
              sr[3] <= sr_in[31:24];
              sr[4] <= sr_in[39:32];
          end
    end
 assign   sr_out = sr[4];         // с задержкой на 5 тактов
endmodule
```

код 2

```
begin
 reg [7:0] shift_reg;
 result = 0 ;
 shift_reg = rega;
while(shift_reg)
  begin
     if (shift_reg[0]) result = result + 1;
         shift_reg = shift_reg >> 1;
  end
end

begin: countls
reg [7:0] shift_reg;
 result = 0;
   for (shift_reg = rega; shift_reg!=0; shift_reg = shift_reg >>
1)
   if (shift_reg[0])  result = result + 1;
end
```

код 3

```
module multiply
```

```
( input clock, start,
  input a, b,
  output result, ready);

parameter size = 8, longsize = 16;
 wire [size:l] a, b;
 reg [size:l] opb;
 reg ready;
 reg [longsize:1] result;
 reg [longsizble:1] opa;

always @(posedge start)
 begin
   opa = a; // загрузка новых исходных данных
   opb = b;
   result = 0;
   ready= 0;
     repeat (size) // повторить для всех разрядов
        @(posedge clock) // блок инициируется фронтом clock
       begin
         if (opb[1])  result = result + opa;
         opa = opa << 1;
         opb = opb >> 1;
       end
   ready=l;
 end
endmodule
```

код 4

```
module task_example
( input [7:0] a,b,
  output reg [7:0] c);

task adder;
 input [7:0] a,b;
 output [7:0] adder;
 reg c;
 integer i;
begin
 c = 0;
 for (i = 0; i <= 7; i = i+1) begin
   adder[i] = a[i] ^ b[i] ^ c;
   c = (a[i] & b[i]) | (a[i] & c) | (b[i] & c);
 end
end
endtask
```

```
always
   adder (a,b,c); // c is a reg

endmodule
```

7. Линейный сдвиговый регистр с обратной связью (Linear Feedback Shift Register)

Очень часто процессы разработки схем в ПЛИС, их отладка, а также прошивка с помощью различных методов демонстрируются на примере линейного сдвигового регистра с обратной связью. Он нашел широкое распространение в цифровых схемах от самых быстродействующих счетчиков (с минимальным числом вентилей в обратной связи), генераторов случайных чисел до шифраторов, вычислителей циклических контрольных сумм (CRC), систем тестирования. Генератор случайных чисел нашел широкое применение в системах со скрытыми временными метками синхронизации, как, например, это организовано в GPS.

В общем случае Линейные Сдвиговые Регистры с Обратной связью (LFSR - Linear Feedback Shift Registers) формируются из простого сдвигового регистра с добавлением обратной связи, подключенной к двум или более точкам в регистровой цепочке, которые называют отводами или ответвлениями (taps). Грубо схемы LFSR-регистры можно разделить на две группы.

Первую группу составляют LFSR-регистры, внутрь которых внешние сигналы (кроме тактового сигнала) не поступают. Представителями таких схем являются счетчики/генераторы псевдослучайных чисел. Со счетчиками такую схему роднит циклическое прохождение всех своих состояний, однако эти состояния не являются возрастающими значениями двоичного весового кода. Ввиду того, что LFSR-счетчики собираются из сдвигового регистра с немногими ответвлениями со схемой Искл. ИЛИ в цепи обратной связи, они являются очень быстрыми счетчиками (у обычных счетчиков в обратной связи стоит более медлительный сумматор из-за большого числа задействованных в нем схем). Однако полноценными счетчиками их считать нельзя, так как они не считают последовательно. Из-за особенностей обхода своих состояний такие LFSR-счетчики могут использоваться как генераторы псевдослучайных чисел. Наиболее интересными являются те n-разрядные LFSR-счетчики, которые имеют все $2^n–1$ уникальных состояний и называются счетчиками с "максимальным счетом". Не существует простой формулы, по которой можно вычислить формулу обратной связи (задать схему соединений) для счетчика произвольной разрядности.

Вторую группу составляют LFSR-регистры, внутрь которых поступают внешние сигналы (кроме тактового сигнала) и учитываются в ходе работы LFSR-регистры. Представителями таких схем являются LFSR-шифраторы и вычислители циклических контрольных сумм (CRC - cyclic redundancy code). Последние широко используются при передаче больших фрагментов бинарной информации (по кабелю и при чтении

информации с внешнего накопителя), когда передаются **сразу** же после переданной информации.

Как отмечалось ранее, при создании схемы LFSR-счетчиков важным является получение такой схемы, которая позволит получить счет с максимальной длиной. Имеется несколько методов, которые могут использоваться, чтобы принудить LFSR-счетчик обойти все 2^n возможных состояний (а не 2^n–1, как обычно, когда «пропадает» нулевое состояние):

1. С помощью увеличения размера LFSR-регистра на один бит и установки *единиц* во все разряды счетчика, когда обнаружен конец счета. Значение конца счета отлавливается схемой ИЛИ, выход которой попадает вход каждого триггера LFSR-счетчика (чтобы переустановить все триггеры LFSR в единичное состояние). Эта техника экономит число сравнений (но далее не рассматривается).

2. Без увеличения размера LFSR регистра потребуется сброс регистра ко всем *нулям* во время прихода сигнала сброса, и когда окончание счета уже достигнуто. Это обеспечивается использованием высокоприоритетного входа асинхронного сброса Clear (или Reset) у триггера в логических блоках (ячейках) FPGA-микросхем. Высокий приоритет асинхронного сброса – особенность изготовления триггеров. Такой «аппаратный» сброс счетчика вызывает появление такого желанного дополнительного нулевого состояния. Так как сам LFSR-счетчик не способен перейти в нулевое состояние (и тем более выйти из него), важно так разработать LFSR-счетчик, чтобы он был способен выйти из *нулевого* состояния, в которое он попадет после принудительного «аппаратного» сброса. В примере 35 показан Verilog-код LFSR-счетчика с максимальным счетом. Последовательность состояний в шестнадцатеричном виде такого счётчика (см. также рис. 33 г) будет таковой (для удобства каждый новый виток повторения показан с новой строки, правда, без принудительного сброса в *нулевое* состояние):

0 (*после reset*), 1,8,C,E,F,7,B,5,A,D,6,3,9,4,2,

1,8,C,E,F,7,B,5,A,D,6,3,9,4,2,

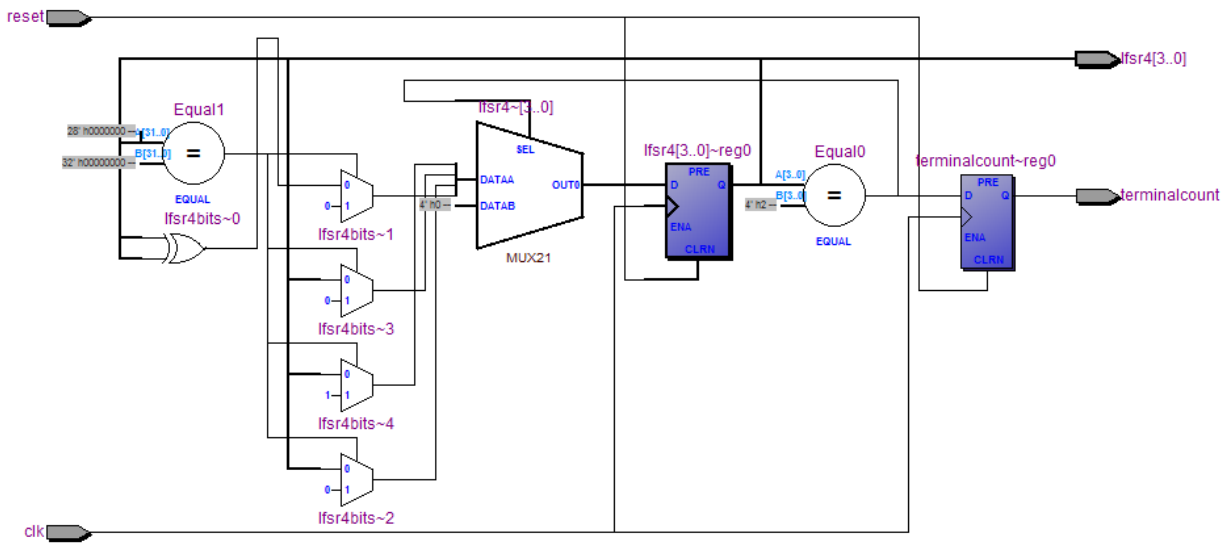
1,8,C, ..

Пример 36. 4-х разрядный LFSR-счетчик с максимальным счетом

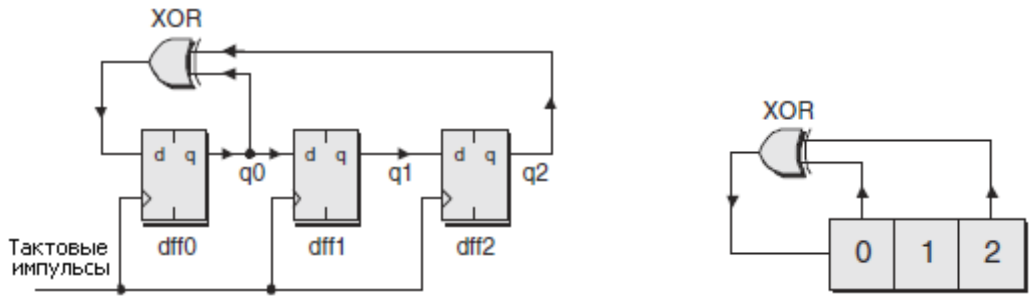
```
module lfsr4count (
 output reg [3:0] lfsr4,      // выход счетчика
 output reg  terminalcount, // окончание счета по lfsr4 = 4
 input       clk,        // внешние такты
 input       reset    // внешний сброс, активный уровень hi
);

function [3:0] lfsr4bits;
   input [1:4] lfsr;
begin
   if (lfsr == 0) lfsr4bits = 1;
```

```
    else            lfsr4bits = {(lfsr[4]^lfsr[1]),lfsr[1:3]};
 end
 endfunction // lfsr4bits

always @(posedge clk)
  begin
   if (reset)
    begin
      lfsr4 <= 4'h0;  // сброс в ноль по reset
      terminalcount <= 1'b0;
    end
    else if (lfsr4 == 4'h2) begin
           lfsr4 <= 4'h0; // коварная строчка сброса в ноль
           terminalcount<= 1'b1;
         end
         else begin
          lfsr4 <= lfsr4bits(lfsr4);
          terminalcount<= 1'b0;
         end
  end
endmodule // lfcr4count
```

а)



б) в)



г)

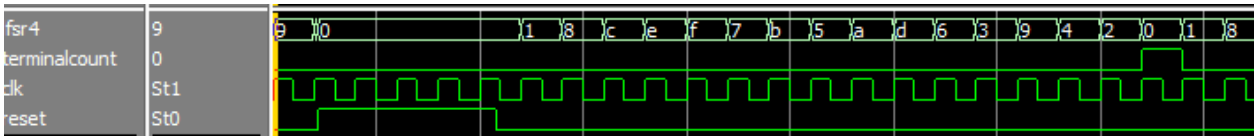


Рис. 36. а) RTL-схема 4-разрядного LFSR-счетчика с максимальным счетом, б) простая функциональная RTL-схема 4-разрядного LFSR-счетчика без нулевого состояния, в) его схематичное обозначение, г) временная диаграмма 4-разрядного LFSR-счетчика с максимальным счетом

При синтезе схемы по Verilog-коду lfsr4count (пример 36) с различными настройками оптимизации схемосинтезатора может случиться так, что счетчик не сможет выйти из *нулевого* состояния без использования утверждения "`if (lfsr == 0) lfsr4bits = 1;`". В этом случае счетчик никогда не начнет считать после сброса. Спроектированная схема, достигнув состояния "0000", застрянет на нем, поэтому при синтезе данной схемы надо уделять внимание этой ее особенности.

На рис. 36 а показана RTL-схема 4-разрядного LFSR-счетчика с максимальным счетом, созданная с помощью САПР Quartus II. Схема сильно усложнена из-за особенностей внутренней структуры ПЛИС типа FPGA, поэтому на рис. 36 б показана упрощенная RTL-схема (правда без нулевого состояния), а на рис. 36 б – его схематичное обозначение. Далее 4-х разрядный LFSR-счетчик будет рисоваться (рис. 37 и 38) по его схематичному обозначению (без сброса в ноль).

Подводя итог особенностям генераторов псевдослучайных чисел, отметим, что существуют различные методы генерирования псевдослучайных чисел, один из которых заключается в использовании LFSR-регистра с изменяемым местом отвода значений для обеспечения достаточно хорошего источника псевдослучайного сигнала.

Традиционной сферой применения LFSR-регистров является вычисление циклических избыточных кодов (CRC – cyclic redundancy code), которые используются для обнаружения ошибок в канале связи. В этом случае поток передаваемых данных используется для модификации значений формируемой LFSR-регистром последовательности при помощи подключения его к цепи обратной связи (рис. 37).

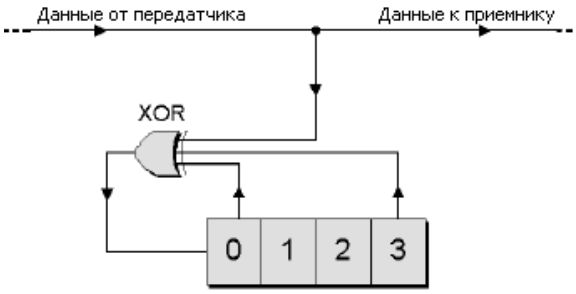


Рис. 37. CRC-блок на основе 4-разрядного LFSR-регистра

Итоговое CRC-значение, сохранённое в LFSR-регистре, называется *контрольной суммой*, и зависит от каждого передаваемого бита. После того, как все данные будут переданы, передатчик посылает приёмнику значение контрольной суммы. Приёмник также содержит аналогичный CRC-блок и при приёме данных осуществляет формирование своего собственного значения контрольной суммы. После приёма всех данных приёмник сравнивает внутреннее значение контрольной суммы с тем, которое было сформировано передатчиком, и на основании этого сравнения делает выводы о том, были ли искажения данных в процессе их передачи.

Этот вид обнаружения ошибок очень эффективен с точки зрения малого количества информации, который необходимо передавать вместе с основным блоком данных. Однако недостатком этого метода служит то, что при этом вы не узнаете о наличии ошибки до тех пор, пока не закончится передача данных, и в случае ошибки вам придётся передавать всю информацию заново.

В реальном мире 4-битные блоки подсчёта контрольных сумм не обеспечивают требуемый уровень достоверности о целостности передаваемых данных, так как могут формировать только $(2^4 – 1) = 15$ уникальных значений. Это может привести к проблеме, называемой *совмещением* (или aliasing), при которой итоговое значение контрольной суммы совпадает с ожидаемым, но на самом деле вызвано действием многочисленных ошибок, которые накладываются друг на друга и в итоге приводят к требуемому CRC-значению. При увеличении количества бит в блоке CRC увеличивается и количество генерируемых им уникальных значений, вследствие чего уменьшается вероятность того, что многочисленные ошибки спровоцируют появление такого значения контрольной суммы, которое совпадёт с ожидаемым. Поэтому на практике обычно используют как минимум 16-битные CRC-блоки, которые могут генерировать 65535 уникальных значений.

Различные протоколы связи используют разное количество бит для вычисления значения контрольной суммы и разные отводы. Отводы выбираются таким образом, чтобы ошибка в одном отдельном бите приводила к максимально возможному изменению итогового значения контрольной суммы. Таким образом, LFSR-регистры могут также классифицироваться не только по их *максимальной длине*, но и по их *максимальному смещению*.

Если не вводить информацию в LFSR-регистр, как в случае CRC-блока, а только складывать по модулю 2 с каким-либо битом LFSR-регистра, то можно получить простое

устройство для шифрования или дешифрования информации. Таким образом, необычные последовательности значений, генерируемые LFSR-регистрами, могут использоваться для шифрования (кодирования) и дешифровки (декодирования) данных. Пример такого устройства изображен на рис. 35.

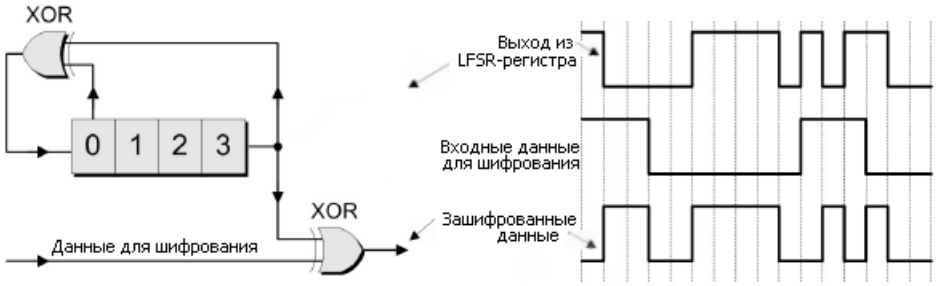


Рис. 38. Шифрование данных с помощью LFSR-регистра

8. Вопросы и задания

Ниже приведены задания как для начинающих знакомство с Verilog (задания 1-23) и выполняющих задания и проверку работоспособности своих проектов в программной среде (САПР Quartus и ModelSim-Altera), так и для уже имеющих некоторый опыт по реализации проектов непосредственно в ПЛИС (главным образом, задания 24-37). Последние задания выполняются на так называемых оценочных/отладочных платах (у нас используются DK-MAXII-1270N, DE2, DE2-115), где кроме микросхем ПЛИС размещены различные периферийные узлы, среди которых часто задействуются различные световые индикаторы. Наиболее простыми в управлении являются светодиодные, не требующие комментариев, индикаторы, тогда как информация по жидкокристаллическим индикаторам в документации к указанным оценочным платам отсутствует. Для того чтобы помочь обучающимся разобраться с работой имеющихся жидкокристаллических индикаторов в **Приложении** в конце пособия приведена вся необходимая информация, собранная из различных источников в интернете.

1. При каких условиях типичный Verilog-схемосинтезатор производит комбинационную схему по блоку **`always`**? Когда последовательностная схема синтезируется по блоку **`always`** Verilog-схемосинтезатором?

2. При каких условиях типичный схемосинтезатор создает комбинационную схему от процесса VHDL?

3. Адресное пространство памяти 64 КБ разделено на восемь сегментов шириной по 8 КБ. Используя Verilog, опишите дешифратор адреса, который распознает сегменты по 16-битовому адресу.

4. Адресное пространство от предыдущего примера разделено на семь сегментов равной длины (по 8 КБ), и самый верхний восьмой сегмент разделен на четыре части с размером в 2 КБ. Используя Verilog, опишите дешифратор адреса, который распознает сегменты по

16-битовому адресу. Опишите дешифратор, используя блок **always** и различные процедурные утверждения. Сделайте, по крайней мере, два различных описания дешифратора.

5. Опишите J-K, и T триггеры, используя блок **always** в Verilog.

6. Используя шаблоны для FSM-машин типа Mealy и Moore, опишите триггеры из предыдущей задачи.

7. Примените шаблоны для FSM-машин типа Mealy и Moore к примеру системы, которая описывает движение автомобиля с четырьмя состояниями: остановка, медленое, среднее, и высокое движения, и с двумя входами, представляющими ускорение и торможение. Выход представлен отдельным индикатором для каждого из состояний. Состояния закодировать, используя схему шифрования с одним активным состоянием (one-hot). Каким будет различие, если Вы примените различные схемы шифрования состояний (сделайте это для кодировки состояний их последовательным двоичным кодом, кодами Джонсона и Грея).

8. Опишите на языке Verilog общий синхронный n-разрядный суммирующий/вычитающий счетчик, что считает до p, когда суммирует, и считает до q, когда вычитает. Используя эту модель, породите экземпляр 8-битового счетчика, доходящего до единицы при суммировании, до двух при вычитании.

9. Опишите асинхронный счётчик со сквозным переносом (ripple counter), который делит входные такты на 32. Для каскадов сквозного переноса (ripple stages) счетчик использует триггер D-типа, выход которого связан обратной связью с его D входом так, что каждый каскад (stage) делит его входные такты на два. Для описания используйте моделирование поведенческого стиля. Как вы будете изменять счетчик для деления входных тактов на число, которое находится между 17 и 31, и оно не может быть выражено как 2^k (k - целое число)?

10. Спроектируйте параметризованный делитель частоты, который делит частоту входных тактов на N, и обеспечивает коэффициент заполнения произведенных тактов длительностью M (M<N-1) входных тактов.

11. Нарисуйте схему в графическом схеморедакторе и напишите программу на языке Verilog, реализующей схему дешифратора для семисегментного индикатора.

12. Нарисуйте схему в графическом схеморедакторе и напишите программу на языке Verilog, реализующей схему контроля четности.

13. Нарисуйте схему в графическом схеморедакторе и напишите программу на языке Verilog, реализующей схему определения числа дней в месяце по номеру месяца.

14. Создайте модуль преобразования 8-разрядного числа в соответствующее двоично-десятичное (BCD - binary-decimal code) представление.

15. Создайте модуль преобразования пары двоично-десятичных чисел (BCD) в соответствующий байт.

16. Создайте модуль преобразования 4-битного набора в коде Грея в эквивалентное двоичное число.

17. Создайте модуль преобразования 8-битного набора в коде Грея в эквивалентное двоичное число.

18. Создайте модуль вычисления 13-й цифры штрих-кода EAN-13 по L и G кодировке первых 6 цифр. Согласно таблице

0 LLLLLL

1 LLGLGG

2 LLGGLG

3 LLGGGL

4 LGLLGG

5 LGGLLG

6 LGGGLL

7 LGLGLG

8 LGLGGL

9 LGGLGL

19. Создайте модуль преобразования 4-битного набора в семисегментный код.

X g f e d c b a

0 0 1 1 1 1 1 1

1 0 0 0 0 1 1 0

2 1 0 1 1 0 1 1

3 1 0 0 1 1 1 1

4 1 1 0 0 1 1 0

5 1 1 0 1 1 0 1

6 1 1 1 1 1 0 1

7 0 0 0 0 1 1 1

8 1 1 1 1 1 1 1

9 1 1 0 1 1 1 1

A 1 1 1 0 1 1 1

B 1 1 1 1 1 0 0

C 0 1 1 1 0 0 1

D 1 0 1 1 1 1 0

F 1 1 1 1 0 0 1

G 1 1 1 0 0 0 1

20. Создайте модуль преобразования 4-битного числа в его дополнение до единицы (ones' complement) и в дополнение до двух (two's complement).

21. Создайте модуль для сложения двух двоично-десятичных чисел.

22. Создайте модуль компаратора двух 4-разрядных чисел a и b. Модуль компаратора будет генерировать два выходных бита f и q. Один бит f равен 0, если a > b, и равен 1, если a <b. Второй бит q равен 1, если a = b, и равен 0 в противном случае.

23. Создайте 2-битный модуль АЛУ (ALU). Входами модуля являются два 2-битных операнда a и b, 3-битный вход f вида операции и 1-битовый вход переноса из младших разрядов ci. Выходами являются 2-битовый результат D и 1-битовый выход переноса в следующие разряды. Алгоритм работы задается значение f: если f = 1, d = a + b; если f = 2, d = a + b + ci; если f = 3, d = a - b; Если f = 4, d = a - b - ci; если f = 5, d = a OR b; Если f = 6, d = a AND b; если f = 7, d = a XOR b.

24. Создайте счетчик, однократно проходящий все свои состояния от нуля до модуля счета по внешнему сигналу (положительный цифровой зуб пилы), несколькими способами с помощью: цифрового автомата (FSM-машины), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

25. Создайте счетчик, однократно проходящий все свои состояния от нуля до модуля счета и обратно до нуля по внешнему сигналу (положительный и отрицательный цифровой зуб пилы), несколькими способами с помощью: цифрового автомата (FSM-машины), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

26. Создайте счетчик, заданное количество раз проходящий все свои состояния от нуля до модуля счета (несколько положительных цифровых зубов пилы) по внешнему сигналу, несколькими способами с помощью: цифрового автомата (FSM-машины), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

27. Создайте счетчик, заданное количество раз проходящий все свои состояния от нуля до модуля счета и обратно до нуля (несколько положительных и отрицательных цифровых зубов пилы) по внешнему сигналу, несколькими способами с помощью: цифрового

автомата (FSM-машины), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

28. Создайте счетчик, заданное количество раз проходящий все свои состояния от нуля до модуля счета, после чего он должен пройти заданное раз вниз от модуля счета до нуля (сначала несколько положительных цифровых зубов пилы, после которых должны проследовать несколько отрицательных цифровых зубов пилы) по внешнему сигналу, несколькими способами с помощью: цифрового автомата (FSM-машины), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

29. Создайте счетчик, заданное количество раз проходящий все свои состояния от нуля до модуля счета, но разной скоростью роста (с разным инкрементом счета). При первом проходе инкремент равен 1 (обычный), при втором проходе инкремент равен 2 и так далее (несколько положительных цифровых зубов пилы с разным наклоном) по внешнему сигналу. Реализовать несколькими способами с помощью: цифрового автомата (FSM-машины), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

30. Создайте счетчик, заданное количество раз проходящий все свои состояния от нуля до модуля счета, но разной скоростью роста (с разным инкрементом счета). В интервале от нуля до четверти модуля счета инкремент равен 1 (обычный), в интервале от четверти модуля счета до половины модуля счета равен 2, в интервале от половины модуля счета до трех четвертей модуля счета равен 3 и на последнем участке до модуля счета равен 4 (несколько положительных экспоненциальных цифровых зубов пилы) по внешнему сигналу. Реализовать несколькими способами с помощью: цифрового автомата (state machine), конструкции **`if else`**, а также с помощью различных примитивов защелок и триггеров (latch, flip-flop). Сравните полученные RTL-схемы.

31. Создайте иллюминацию "бегущие огни" в одном направлении на светодиодах (т.е. выглядящую как светящая точка, всегда смещающаяся в одном направлении) двумя способами с помощью сдвигового регистра и счетчика с дешифратором. Сравните полученные RTL-схемы.

32. Создайте иллюминацию "бегущие огни" со сменяющимся направлением движения на светодиодах (т.е. выглядящую как светящая точка, сначала смещающаяся в одном направлении, а затем в обратном) двумя способами с помощью сдвигового регистра и счетчика с дешифратором. Сравните полученные RTL-схемы.

33. Создайте иллюминацию "бегущие огни" в одном направлении на светодиодах, но с растущей скоростью пробега (т.е. выглядящую как светящая точка, всегда смещающаяся в одном направлении, но после каждого прохода наращивающая скорость пробега) двумя способами с помощью сдвигового регистра и счетчика с дешифратором. Сравните полученные RTL-схемы.

34. Создайте иллюминацию "бегущие огни" со сменяющимся направлением движения на светодиодах и с растущей скоростью пробега (т.е. выглядящую как светящая точка, сначала смещающаяся в одном направлении, а затем в обратном и после каждого прохода

наращивающая скорость пробега) двумя способами с помощью сдвигового регистра и счетчика с дешифратором. Сравните полученные RTL-схемы.

35. Напишите программы на языке Verilog, реализующие схему делителя с дробным коэффициентом деления 3/2, 5/2, 7/2.

36. Напишите программу на языке Verilog, реализующую схему счетчика на основе сдвигового регистра.

37. Напишите программу на языке Verilog, реализующую схему генератора случайных чисел на основе сдвигового регистра с обратными связями.

Литература

Амосов В.В. Схемотехника и средства проектирования цифровых устройств. Учебное пособие. – СПб.:БХВ-Петербург, 2007. – 542 с.

Грушвицкий Р.И., Мурсаев А.Х., Угрюмов Е.П. Проектирование систем на микросхемах с программируемой структурой. СПб.:БХВ-Петербург, 2006. – 736 с.

Максфилд К. Проектирование на ПЛИС. Курс молодого бойца. — М.: Издательский дом «Додэка-XX1», 2007. — 408 с.

Соловьев В. В. Основы языка проектирования цифровой аппаратуры Verilog. – М.: Горячая линия – Телеком, 2014. – 208 с.

Харрис Д., Харрис С. Цифровая схемотехника и архитектура компьютера. – Burlington:Morgan-Kaufman, 2013 – 1621 с.

Salcic Z., Smailagic A. Digital systems design and prototyping (2nd ed.): using field programmable logic and hardware description languages, Kluwer Academic Publishers Norwell, 2000. – 596 p.

Приложение

Обозначение, структура и протокол обращения к жидкокристаллическому индикатору (ЖКИ) (LCD module), расположенному на оценочных комплектах DE2 и MAXII по освоению ПЛИС фирмы Altera

1. Система обозначений ЖКИ-индикаторов фирм Crystalfontz America и Winstar (не полная)

CFA H 1602 B - N Y A - JP **

1 2 3 4 5 6 7 8 9

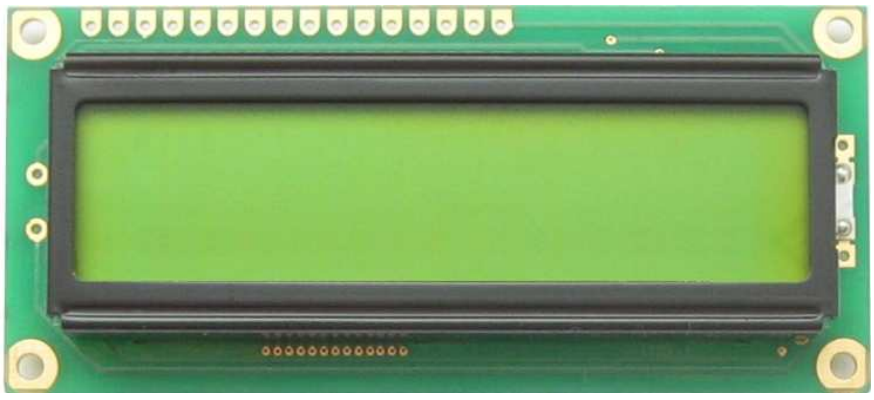
№	Пояснение	Значение	Описание
1	Производитель	CFA W	продукт Crystalfontz America продукт Winstar
2	Тип модуля	H	символьный
		G	графический
		X	TAB
3	Число символов, строк или точек	08,16,20,24,40...	количество символов в строке (для символьных ЖКИ)
		120,122,128...	количество точек в строке (для графических ЖКИ)
		01,02,03,04...	количество строк (для символьных ЖКИ)
		08,16,32...240...	количество точек в столбце (для графических ЖКИ)
4	Модель индикатора	A - Z	Серийный номер модели
5	Тип и цвет подсветки	N	без подсветки
		A	светодиодная янтарная
		B	электролюминисцентная синяя

		D	электролюминисцентная зелёная
		F	CCFL белая
		G	светодиодная зелёная
		P	светодиодная синяя
		R	светодиодная красная
		T	светодиодная белая
		W	электролюминисцентная белая
		Y	светодиодная желто-зеленая
6	Тип и цвет ЖКИ	B	TN позитивный, серый
		F	FSTN позитивный
		G	STN позитивный, серый
		M	STN негативный, синий
		N	TN негативный
		T	FSTN негативный
		Y	STN позитивный, желто-зелёный
7	Тип поляризации, диапазон температур, угол зрения	A	на отражение, 0 - +50°C, 6:00
		B	на просвет и отражение, 0 - +50°C, 6:00
		C	на просвет, 0 - +50°C, 6:00
		D	на отражение, 0 - +50°C, 12:00
		E	на просвет и отражение, 0 - +50°C, 12:00
		F	на просвет, 0 - +50°C, 12:00
		G	на отражение, -20 - +70°C, 6:00
		H	на просвет и отражение, -20 - +70°C, 6:00
		I	на просвет, 0 - +50°C, 6:00

		J	на отражение, -20 - +70°C, 12:00
		K	на просвет и отражение, -20 - +70°C, 12:00
		L	на просвет, -20 - +70°C, 12:00
8	Знакогенератор	JS / JP	английский/японский
		EE / EU / EP / ES / EC	английский/европейский
		CP	английский/русский
9	Специальный код	V	встроенный источник отрицательного напряжения
		T	встроенный источник отрицательного напряжения и температурная компенсация
		TS	Touch Screen
		E	подсветка в торец

Пример: **CFAH1602B-NYA-JP** - алфавитно-цифровой модуль серии B без подсветки, STN позитивный серый, на стандартный диапазон температур с углом наблюдения, смещенным к нижней строке от нормали к индикатору, с англо/японским знакогенератором.

1 2 16



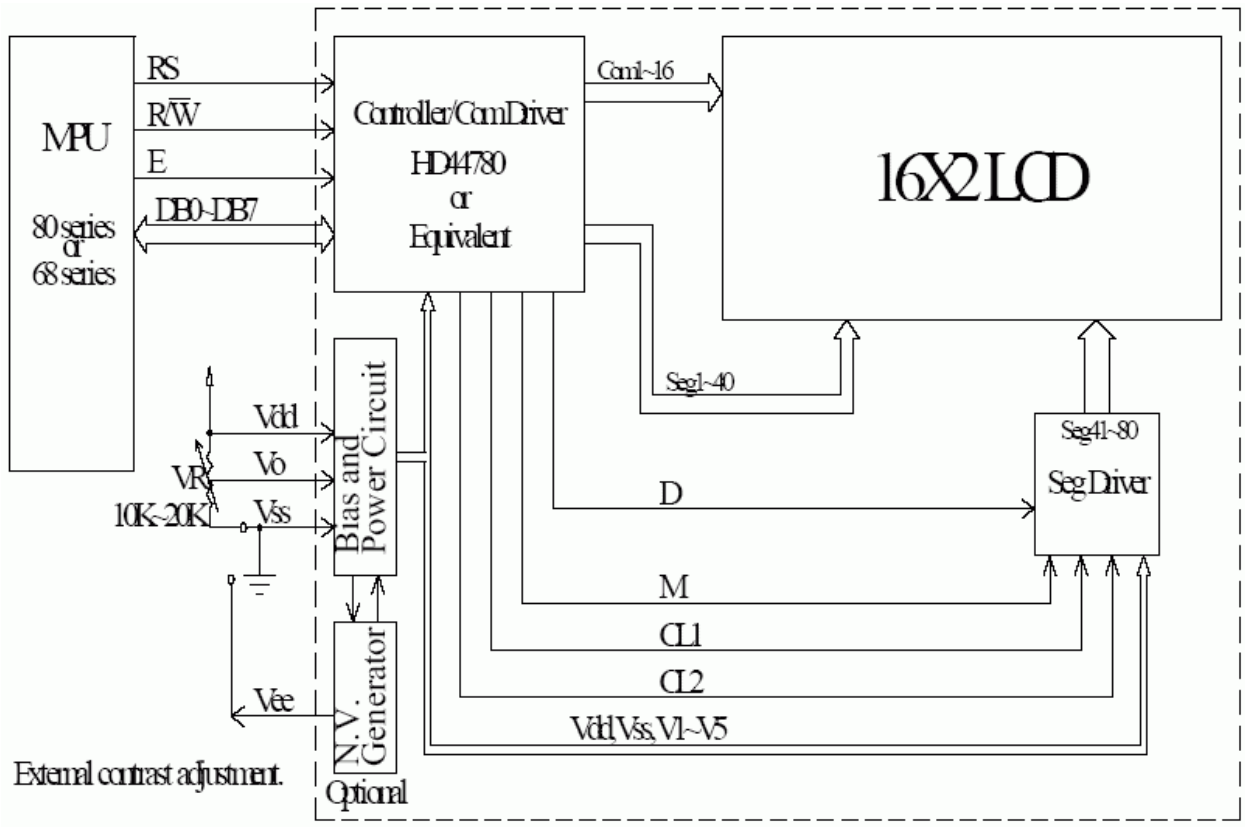
Индикатор имеет следующее расположение выводов (в последних двух столбцах выводы ПЛИС на соответствующих отладочных платах):

Pin	Symbol	Level	Description	MAX	DE2

№					
1	Vss	0V	Ground (*земля*)		
2	Vdd	5.0V (3V)	Supply Voltage **for** logic (3V option) *питание*	B3	
3	Vo	(Variable)	Operating voltage **for** LCD (contrast) (*настройка контрастности*)		
4	RS register selection	H/L	H: Data, L: Instruction code (*данные/команды*)	C11	K1
5	R/W	H/L	H: Read(MPU→Module) L: Write(MPU←Module) (*чтение/запись*)	D11	K4
6	E	H,H->L *	Chip enable signal	A10	K3
7	DB0	H/L	Data bit 0	B10	J1
8	DB1	H/L	Data bit 1	C10	J2
9	DB2	H/L	Data bit 2	D10	H1
10	DB3	H/L	Data bit 3	A9	H2
11	DB4	H/L	Data bit 4	B9	J4
12	DB5	H/L	Data bit 5	C9	J3
13	DB6	H/L	Data bit 6	D9	H4
14	DB7	H/L	Data bit 7	A8	H3
15	A		LED+ (анод подсветки)	NC	NC
16	K		LED- (катод подсветки)	NC	

Модуль может работать в 4-битном режиме, что уменьшает число необходимых выводов - нужно подключить только Vss(Gnd), Vdd, Vo, RS, RW, E, DB4-DB7 и при необходимости подсветку. Также, поскольку обычно обмен данными идет только в одну сторону - от микроконтроллера к ЖК дисплею, то вывод RW может не управляться ПЛИС, а просто быть подключенным к земле внутри нее.

Структурная схема ЖКИ



Vo – напряжение, регулирующее контрастность ЖК индикатора, с помощью резистора 10-20 ком (Типичное значение Vo = +0,8В при напряжении питания Vdd = 5В)

Контроллер ЖК индикатора построен на контроллере HD44780, который стал промышленным стандартом де-факто на рынке цифро-буквенных дисплеев.

ЛОГИЧЕСКАЯ СТРУКТУРА КОНТРОЛЛЕРА HD44780 ДЛЯ ЖКИ (LCD)

Контроллер имеет свой блок управления, который обрабатывает команды, и память, что делится на три вида:

DDRAM (Display Data RAM) — память дисплея. Эта память используется, чтобы хранить отображаемые данные (символы), представленные в 8-разрядной кодировке. Ее расширенная вместимость – 80×8 бит или 80 символов. Все, что запишется в **DDRAM**, будет выведено на экран. То есть, например, записали мы туда код **0x31** — на экране появится символ **«1»** т.к. **0x31** это ASCII код цифры **1.** Но есть тут одна особенность — **DDRAM** память гораздо больше, чем видимая область экрана. Как правило, **DDRAM** содержит **80 ячеек**: 40 ячеек в первой строке и 40 – во второй. Дисплей может двигаться по этой строке как окошко на логарифмической линейке, высвечивая видимую область длиною в 16 ячеек (см. таблицу П1). То есть, например, можно загрузить в **DDRAM** сразу пять пунктов вашего меню, а потом просто гонять дисплей туда-сюда, показывая по одному пункту. Для перемещения дисплея имеется специальная команда. Также имеется

понятие курсора, то есть такого знакоместа на дисплее, в которое будет записан следующий символ. Такое знакоместо именуется как текущее значение счетчика адреса. Курсор не обязательно может быть на экране, он может располагаться и за экраном или быть отключен вовсе.

Таблица П1. Размещение (Address) кодов символов в памяти DDRAM, отображаемых на двухстрочечном 16-символьном дисплее (стартовое состояние)

Позиция символа	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
1-я строка	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2-я строка	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

CGROM — несменяемая таблица символов. Когда мы записываем байт в ячейку памяти **DDRAM**, то из таблицы **CGROM** берется растровая картинка символа и рисуется на экране. Объем **CGROM** таков, что содержит растровые картинки 256–8 байт, примерно совпадающих с символами из ASCII-таблицы без первых восьми символов. **CGROM** нельзя изменить, поэтому важно, чтобы имелась дополнительная таблица **CGRAM**, содержащая растровые картинки русских букв (если, конечно, планируется русскоязычный интерфейс).

CGRAM — перезагружаемая таблица символов, заполненная по включению контроллера HD44780 заводскими значениями (по умолчанию). Однако мы можем ее менять, создавая свои символы. Адресуется она линейно, то есть вначале идет 8 байт одного символа, построчно, снизу вверх (один бит равен одной точке на экране). Потом второй символ аналогично. Поскольку знакоместо у нас – это 5 на 8 точек, то **старшие три бита роли не играют**. Всего в **CGRAM** может быть 8 символов, соответственно **CGRAM** имеет **64** байта памяти. Эти программируемые символы имеют коды от 0x00 до 0x07. Так что, поместив, например, в первые **8 байт CGRAM** (первый символ с кодом 00) какую-нибудь растровую комбинацию, и записав затем ноль в **DDRAM** (код первого символа в таблице **CGRAM**), мы увидим на экране выдуманный нами символ.

Доступ к памяти

Установкой специальной команды (Set CGRAM Address или Set DDRAM Address) на шине контроллера HD44780 осуществляется выбор той памяти, куда будет производиться запись и с какого адреса. А потом просто посылаем цуг байтов. Если выполнить запись в **DDRAM**, то на экране (или в скрытую область для ячеек памяти с номером выше 16) начнут поочередно появляться символы. Если выполнить запись в **CGRAM**, то байты начнут заполнять память знакогенератора. Чтобы эти вновь рождённые символы увидеть, надо потом не забыть переключится обратно на область **DDRAM**.

Обменные регистры

Контроллер HD44780 имеет два 8-битовых обменных регистра: **IR** *регистр инструкций* или *команд* (instruction register) и **DR** *регистр данных* (data register). Эти два регистра могут быть выбраны при помощи сигнала RS *выбор регистра* (register selection).

Регистр **IR** хранит коды команд, таких как очистка дисплея, установка курсора в первую позицию, сдвиг курсора и установка адреса данных для памяти дисплея (**DDRAM**) и для памяти символьного генератора (**CGRAM**). Регистр **IR** может быть записан только извне, например, с помощью микропроцессора MPU (microprocessor unit) или ПЛИС.

Регистр **DR** временно хранит данные, которые будут записаны или прочтены в/из **DDRAM** или **CGRAM**. После того, как адресная информация записана в **IR** с помощью команды Set DDRAM/CGRAM Address (в зависимости от бита DB7), данные, направленные в **DR**, будут сохранены в **DDRAM** или **CGRAM** (соответственно) с помощью команды Write Data to RAM. Данная запись в **DR** (и затем последующие записи) приводит к автоматической инкрементации адреса с помощью Адресного Счетчика AC, так что нет необходимости постоянном применении команды Set DDRAM/CGRAM Address.

Общий алгоритм записи в ЖКИ индикатор (чтения из него):

Передний фронт **E** фиксирует моменты достоверных значений сигналов **RS** и **R/$\overline{W}$**, а задний – шину данных **DB**. Сам импульс **E** должен быть длиннее 0,3 мкс (3 Мгц). Вид операции, выполняемой ЖКИ индикатором (контроллером HD44780), определяется состоянием двухбитной шиной управления из **RS** и **R/$\overline{W}$** (подробности в таблице П2).

Таблица П2. Упрощенная таблица видов операций, задаваемых на шине контроллера HD44780, с помощью сигнальных проводов **RS** и **R/$\overline{W}$**.

Состояние контактов		**Вид операции (operation)**
RS	**R/$\overline{W}$**	
0	0	Запись в регистр **IR** для выполнения внутренних операций (очистка экрана, и др.)
0	1	Чтение флага ***Busy*** (DB7) и адресного счетчика ***Address Counter*** (DB0 to DB7)
1	0	Запись в память **DDRAM** или **CGRAM** (через регистр **DR**)
1	1	Чтение данных из памяти **DDRAM** или **CGRAM** (через регистр **DR**)

Комментарий к таблице

Флаг **Busy** (BF)

Когда флаг busy = 1, контроллер в БИС находится в режиме выполнения внутренних операций, и следующая инструкция не будет принята. Когда RS=0 и R/W=1, флаг busy – подается на DB7. Следующая инструкция должна быть написана после обеспечения, что флаг busy ушел на 0.

Address Counter (AC)

Адресный Счетчик (AC address counter) назначает адрес и для **DDRAM**, и для **CGRAM**

Таблица П3. Полная таблица команд (Instruction Codes) со временем выполнения из описания производителя

Instruction	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution time (f_{osc}=270 kHz)
Clear Display	0	0	0	0	0	0	0	0	0	1	Write “00H” to DDRAM and set DDRAM address to “00H” from AC	1,53ms
Return Home	0	0	0	0	0	0	0	0	1	-	Set DDRAM address to “00H” from AC and return cursor to its original position if shifted. The contents of DDRAM are not changed.	1,53ms
Entry Mode Set Разрешение скроллинга экрана с указанием направления	0	0	0	0	0	0	0	1	I/D	SH	Assign display moving direction (increment/decrement) and enable the shift (SH) of entire display (scrolling).	39µs
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B	Set display (D), cursor (C), and blinking of cursor (B) on/off control bit.	39 µs
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	-	-	Set cursor moving and display shift control bit, and the direction, without changing of DDRAM data.	39 µs
Function Set	0	0	0	0	1	DL	N	F	-	-	Set interface data length (DL:8-bit/4-bit), numbers of display	39 µs

Установка базовых режимов – число строк и фонт											line (N:2-line/1-line)and, display font type (F:5×11 dots/5×8 dots)	
Set CGRAM Address	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0	Set CGRAM address in address counter.	39 μs
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Set DDRAM address in address counter.	39 μs
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 μs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43 μs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43 μs

* ”-”:don’t care

Таблица П4. Укороченная таблица команд с переводом на русский язык

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Значение
0	0	0	0	0	0	0	1	Очистка экрана. Счетчик адреса на 0 позицию DDRAM
0	0	0	0	0	0	1	-	Адресация на DDRAM сброс сдвигов, Счетчик адреса на 0
0	0	0	0	0	1	I/D	S	Настройка сдвига экрана и курсора
0	0	0	0	1	D	C	B	Настройка режима отображения
0	0	0	1	S/C	R/L	-	-	Сдвиг курсора или экрана в зависимости от битов
0	0	1	DL	N	F	-	-	Выбор числа линий, ширины шины и размера символа
0	1	AG	AG	AG	AG	AG	AG	Переключить адресацию на

								SGRAM и задать адрес в SGRAM
1	AD	AD	AD	AD	AD	AD	AD	Переключить адресацию на DDRAM и задать адрес в DDRAM

Обозначения в таблице

- **I/D** – инкремент или декремент счетчика адреса. По умолчанию стоит 0 — Декремент. Т.е. каждый следующий байт будет записан в n-1 ячейку. Если поставить 1 — будет Инкремент.

- **S** – сдвиг экрана, если поставить 1, то с каждым новым символом будет сдвигаться окно экрана, пока не достигнет конца DDRAM.

- **D** – включить дисплей. Если поставить туда 0, то изображение исчезнет. В это время можем вносить информацию в видеопамять, и она не будет видна. А чтобы картинка появилась, в эту позицию надо записать 1.

- **C** – включить курсор в виде прочерка. При записи сюда 1 включится курсор.

- **B** – сделать курсор в виде мигающего черного квадрата.

- **S/C –** сдвиг курсора или экрана. Если стоит 0, то сдвигается курсор. Если 1, то экран. По одному разу за команду.

- **R/L** – определяет направление сдвига курсора и экрана. 0 – влево, 1 – вправо.

- **D/L** – бит, определяющий ширину шины данных. 1 – 8 бит, 0 – 4 бита

- **N** – число строк. 0 – одна строка, 1 – две строки.

- **F** – размер символа 0 – 5×8 точек. 1 – 5×10 точек (встречается крайне редко)

- **AG** – адрес в памяти **CGRAM**.

- **AD** – адрес в памяти **DDRAM**.

Подробное описание отдельных команд (по материалам сайта http://makesystem.net)

Clear Display

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	0	1

Очистить дисплей.

После отправки этой команды дисплею он начинает записывать во все ячейки DDRAM памяти символ “Space”, т.е. пустоту, после чего в **Address Counter** записывает 0×00. Поскольку ячеек относительно много, а дисплей относительно медленно работает, то Очистка дисплея (Clear Display) самая долго выполняемая команда ~2..5 мс.

Return Home

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	1	x

Возврат каретки.

Записывает в **Address Counter** значение 0×00 без изменения DDRAM памяти.

Entry Mode Set

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	1	I/D	SH

Режим ввода.

Выбираем, каким образом будут отображаться (записываться в DDRAM и CGRAM память) введённые нами символы : слева направо или справа налево, плюс активировать сдвиг экрана после заполнения видимой части строки (напоминает старую, пишущую машинку).

I/D = 1 – слева направо (инкремент)
I/D = 0 – справа налево (декремент)
SH = 1 – активируем сдвиг в выбранном ранее направлении заполнения строки
SH = 0 – дезактивируем сдвиг

Display ON/OFF CONTROL

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	1	D	C	B

Управление дисплеем.

D = 1 – включить дисплей
D = 0 – отключить дисплей
C = 1 – включить курсор (символ подчеркивания)
C = 0 – отключить курсор
B = 1 – включить курсор (черный квадрат)
B = 0 – отключить курсор

Cursor or Display Shift

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	1	S/C	R/L	X	X

Сдвиг курсора или всего экрана. При помощи этой команды можно получить эффект бегущей строки.

S/C = 1 – сдвигать будем дисплей целиком
S/C = 0 – сдвигать будем курсор
R/L = 1 – сдвиг вправо
R/L = 0 – сдвиг влево

Function Set

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	DL	N	F	X	X

Настройка рабочих параметров дисплея. При помощи этой команды выбираем разрядность шины данных/команд, число рабочих строк в дисплее и размер шрифта.

DL = 1 (Data Length) – используем 8-битную шину данных/адресов : DB7 .. DB0.
DL = 0 – используем 4-битную шину данных/адресов: DB7 .. DB4, остальные соединить на землю.
N = 1 – используем обе строки.
N = 0 – работаем только с верхней строкой.
F = 1 – шрифт размером 5×7 пикселей.
F = 0 – шрифт размером 5×10 пикселей. При использовании обеих строк шрифт автоматически устанавливается на 5×7 пикселей независимо от F-бита.

Set CGRAM Address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

Указываем адрес CGRAM-ячейки, в которую будем записывать байт, т.е. рисовать. Один символ занимает 8 или 10 байт подряд (в зависимости от размера шрифта), всего 64 байта памяти.

Set DDRAM Address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

Указываем адрес DDRAM-ячейки, в которую будет записан символ. Если адрес находится в видимой области DDRAM памяти, то символ тут же будет отображен. Если используем только верхнюю строку, то для нее резервируется DDRAM память от 0×00 до 0x4F. Если используем обе строки, то для каждой резервируется DDRAM память от 0×00 до 0×27 для верхней строки и от 0×40 до 0×67 для нижней, т.е. по 40 байт на строку.

Read Busy Flag & Address

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0

Для того чтобы выполнить эту команду, надо установить вывод “**R/W** = 1″, тем самым сообщая дисплею, что идет команда чтения. MSB прочитанного байта представляет собой флаг занятости– **Busy Flag** (BF). Если “**BF** = 1″, то значит дисплей еще не закончил выполнение предыдущей инструкции, и любая другая инструкция, посланная в этот момент, будет проигнорирована. Остальные 7 бит, представляют собой содержимое **Address Counter**.

Write data to RAM

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	D7	D6	D5	D4	D3	D2	D1	D0

Запись данных в **CGRAM**- или **DDRAM**-память. Прежде, чем начать запись данных, следует указать в какую именно память будет вестись запись. Это выполняется при помощи команды Set DDRAM Address или Set CGRAM Address. Сначала указываем стартовый адрес любой из двух видов памяти, а дальше посылаем цуг символов, при этом **Address Counter** автоматически инкрементируется или декрементируется (переходит на следующий адрес) в зависимости от настроек режима ввода.

Read data from RAM

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	D7	D6	D5	D4	D3	D2	D1	D0

Чтение данных из CGRAM или DDRAM память. Прежде чем выполнить эту команду, указываем память (начальный адрес), из которой собираемся читать. **Address Counter** автоматически инкрементируется или декрементируется в зависимости от настроек режима ввода.

Временная диаграмма чтения/записи

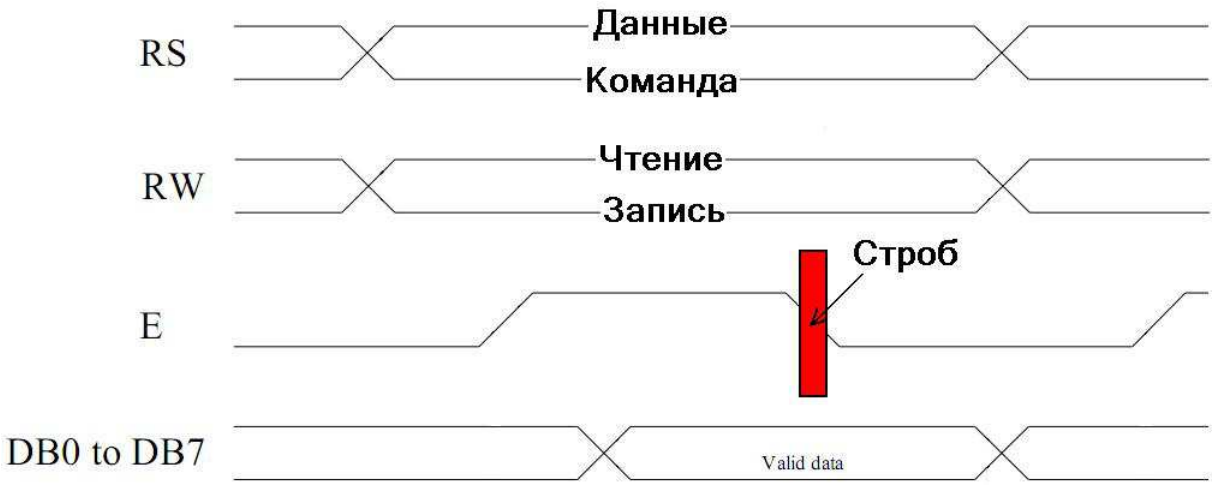
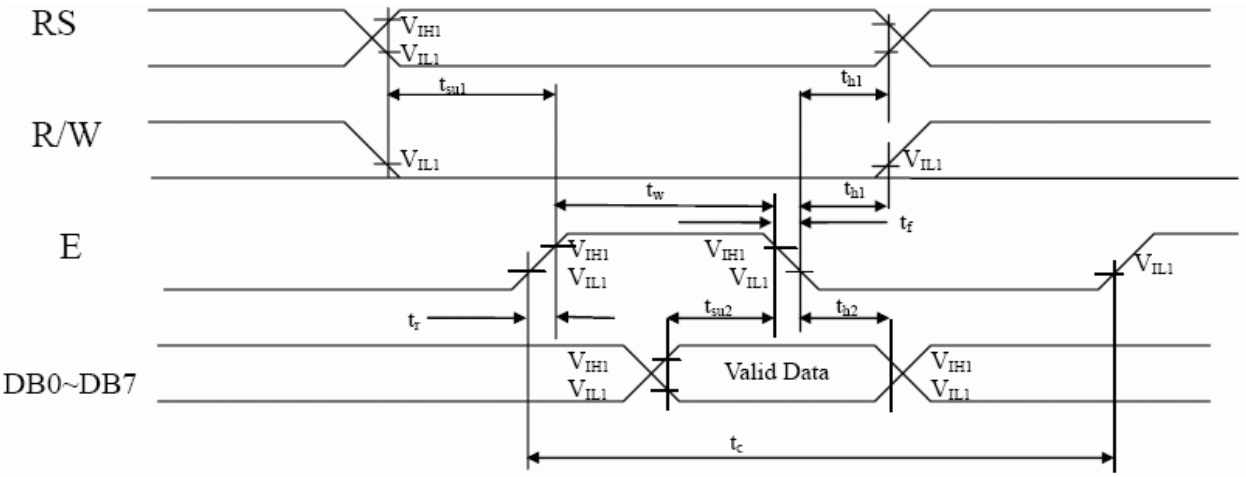


Рис. Временная диаграмма команд чтения/записи

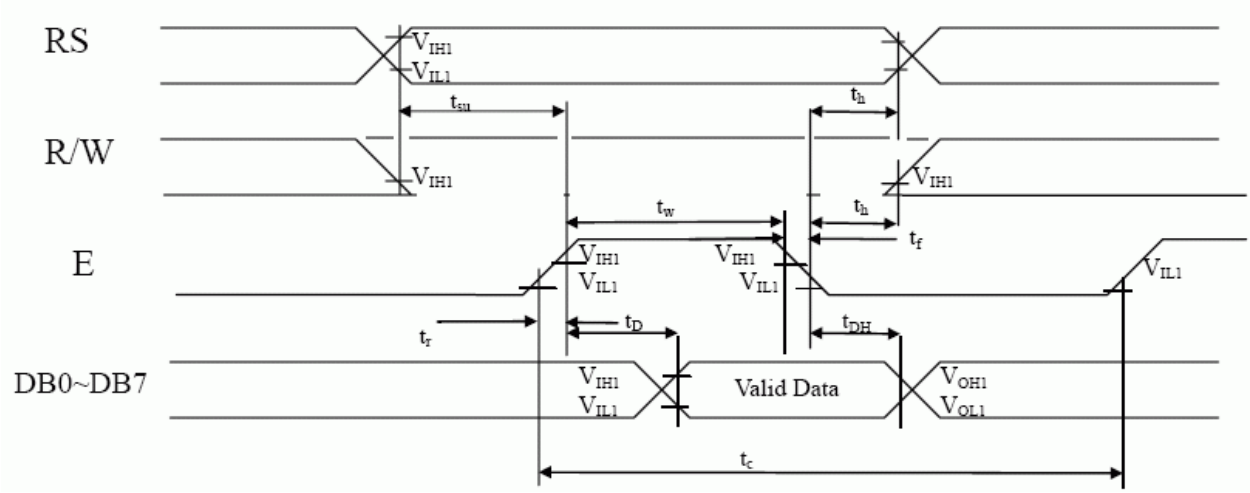
Диаграмма записи с предельными временными ограничениями



(V_{DD} = 4,5V-5,5V , Ta= –30 - +85°C)

Mode	Characteristic	Symbol	Min.	Typ.	Max.	Unit
Write Mode	E cycle Time	tc	500	-	-	ns
	E Rise/Fall Time	t_R, t_F	-	-	20	ns
	E Pulse Width (High, Low)	tw	230	-	-	ns
	R/W and RS Setup Time	tsu1	40	-	-	ns
	R/W and RS Hold Time	t_{H1}	10	-	-	ns
	Data Setup Time	tsu2	80	-	-	ns
	Data Hold Time	t_{H2}	10	-	-	ns

Диаграмма чтения



(Vdd = 4.5 - 5.5V , Ta= –30 - +85°C)

Mode	Characteristic	Symbol	Min.	Typ.	Max.	Unit
Read Mode	E cycle Time	tc	500			ns
	E Rise/Fall Time	t_R, t_F	–		20	ns
	E Pulse Width (High, Low)	tw	230			ns
	R/W and RS Setup Time	tsu	40			ns
	R/W and RS Hold Time	t_H	10			ns
	Data Output Delay Time	t_D	–		120	ns
	Data Hold Time	t_{DH}	5			ns

Инициализация ЖКИ

Первое, что нужно сделать после включения ЖКИ - это провести инициализацию, без которой большая часть дисплеев на HD44780 просто откажется работать.

Инициализация заключается в посылке нескольких команд в определенной последовательности. Количество команд инициализации может несколько отличаться у разных контроллеров, но все же базовый набор команд для восьми- и четырехбитного интерфейсов, подходящий для большинства контроллеров, приведен ниже.

Во время инициализации лучше не анализировать флаг **Busy**, а просто ждать положенное время перед посылкой следующей команды, так как флаг начинает выставляться не сразу,

а после какой-то команды (подробности смотрите в технической документации в файле CFAH1602*.pdf).

Инициализация для восьмибитного интерфейса (f_0=270 кГц)

1) включение питания

2) пауза >30 мс

3) FUNCTION SET

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	1	N	F	X	X

N=0 - однострочный дисплей, N=1 - двустрочный дисплей

F=0 - шрифт 5x8, F=1 - шрифт 5x11

4) пауза >39 мкс

5) DISPLAY ON/OFF CONTROL

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	1	D	C	B

D=0 - дисплей выключен, D=1 - дисплей включен

C=0 - курсор выключен, C=1 - курсор включен

B=0 - мерцание выключено, B=1 - мерцание включено

6) пауза >39 мкс

7) DISPLAY CLEAR

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	0	0	1

8) пауза >1.53 мс

Инициализация для четырехбитного интерфейса (f_0=270 кГц)

1) включение питания

2) пауза >30 мс

3) FUNCTION SET

RS	R/W	DB7	DB6	DB5	DB4
0	0	0	0	1	0
0	0	0	0	1	0
0	0	N	F	X	X

N=0 - однострочный дисплей, N=1 - двустрочный дисплей

F=0 - шрифт 5x8, F=1 - шрифт 5x11

4) пауза >39 мкс

5) DISPLAY ON/OFF CONTROL

RS	R/W	DB7	DB6	DB5	DB4
0	0	0	0	0	0
0	0	1	D	C	B

D=0 - дисплей выключен, D=1 - дисплей включен

C=0 - курсор выключен, C=1 - курсор включен

B=0 - мерцание выключено, B=1 - мерцание включено

9) ENTRY MODE SET

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	1	I/D	SH

I/D=0 - уменьшение указателя при операции с памятью, I/D=1 - увеличение указателя при операции с памятью

SH=0 - сдвигание дисплея выключено, SH=1 - сдвигание дисплея включено

6) пауза >39 мкс

7) DISPLAY CLEAR

RS	R/W	DB7	DB6	DB5	DB4
0	0	0	0	0	0
0	0	0	0	0	1

8) Пауза >1.53 мс

9) ENTRY MODE SET

RS	R/W	DB7	DB6	DB5	DB4
0	0	0	0	0	0
0	0	0	1	I/D	SH

I/D=0 - уменьшение указателя при операции с памятью, I/D=1 - увеличение указателя при операции с памятью

SH=0 - сдвигание дисплея выключено, SH=1 - сдвигание дисплея включено