

УДК 004.01

ББК 32.972

Р69

Коллектив авторов

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

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

ISBN 978-5-97060-850-0

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

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

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

УДК 004.01

ББК 32.972

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-97060-850-0

© Оформление, издание, ДМК Пресс, 2020.

Оглавление

Введение	0-1
Глава 1. Основы комбинационной логики. Маршрут разработки цифровых схем	1-1
Глава 2. Основы последовательностной логики. Управление энергопотреблением цифровой схемы	2-1
Глава 3. Шифраторы и дешифраторы. Скорость работы комбинационных блоков	3-1
Глава 4. Мультиплексор, демультиплексор и селектор. Построение иерархических модулей	4-1
Глава 5. Сумматор, компаратор, устройство сдвига и АЛУ. Повышение скорости арифметических операций	5-1
Глава 6. Последовательная логика. Счетчики и сдвиговые регистры	6-1
Глава 7. Память: регистровый файл и стек	7-1
Глава 8. Конечные автоматы: основы	8-1
Глава 9. Использование конечных автоматов для связи с периферийными устройствами	9-1
Глава 10. Конвейерная обработка данных	10-1
Глава 11. Софт-процессор: основы микроархитектуры	11-1
Приложение А. Путь вперед: от устройств на базе FPGA к массовому рынку ASIC для популярных гаджетов	A-1
Приложение Б. История успеха победы российской команды на международном конкурсе Innovate FPGA от Intel	B-1

Юрий Панчул, Александр Романов

Цифровой синтез: практический курс

Введение

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

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

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

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

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

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

Обоснования для создания этой книги

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

К таким проектам относятся, например:

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

Предтечами создания практикума стали три проекта:

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

¹ <https://mipt-ilab.github.io/mipt-mips/>.

² <https://github.com/MIPSfpga/schoolMIPS>.

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

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

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

История создания данной книги

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

Чем эта книга отличается от других?

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

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

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

Содержание книги

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

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

Рассмотрим кратко по главам содержание книги:

Глава 1. Основы комбинационной логики. Маршрут разработки цифровых схем

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

Глава 2. Основы последовательностной логики. Управление энергопотреблением цифровой схемы

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

Глава 3. Шифраторы и дешифраторы. Скорость работы комбинационных блоков

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

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

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

Глава 5. Сумматор, компаратор, устройство сдвига и АЛУ. Повышение скорости арифметических операций

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

Глава 6. Последовательностная логика. Счетчики и сдвиговые регистры

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

Глава 7. Память: регистровый файл и стек

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

Глава 8. Конечные автоматы: основы

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

Глава 9. Использование конечных автоматов для связи с периферийными устройствами

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

Глава 10. Конвейерная обработка данных

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

блока; описываются дополнительные приемы повышения эффективности конвейерных схем.

Глава 11. Софт-процессор: основы микроархитектуры

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

Приложение А. Путь вперед: от FPGA-устройств к массовому рынку ASIC для популярных гаджетов

Приложение дает общее представление об этапах проектирования чипов **ASIC**, начиная с идеи специализированного чипа и заканчивая его конечной реализацией на кристалле. Это первый шаг в подготовке инженеров, прошедших обучение проектированию на **ПЛИС**, к применению имеющихся у них навыков в разработке **ASIC** для конкретных приложений.

Приложение Б. История успеха победы российской команды на международном конкурсе InnovateFPGA от Intel

Это приложение – небольшое интервью с участником российской команды, занявшей второе место на одном из основных международных конкурсов по проектированию на **ПЛИС** – **InnovateFPGA** от Intel.

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

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

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

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

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

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

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

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

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

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

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

Антон Клотц,
University Program Manager
Cadence Design Systems

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

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

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

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

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

Различные лабораторные работы по проектированию и синтезу цифровых схем на языках регистровых передач для описания аппаратуры стали необходимой частью подготовки электронных инженеров – проектировщиков микросхем для массовых электронных изделий. Через подобный практикум проходят будущие создатели смартфонов, разработчики автомобильной электроники и процессоров для различных встроенных применений, включая электронику для космических зондов. В США известным примером такого практического курса является 6.111 из MIT¹, а также различные варианты лабораторных заданий с **Verilog/VHDL** и **ПЛИС/FPGA** предусмотрены учебными программами практически для всех, кому преподают электронику, включая студентов местных университетов в небольших штатах.

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

¹ <http://web.mit.edu/6.111/volume2/www/f2018/index.html>.

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

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

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

С пожеланиями успехов авторам и пользователям данного учебного пособия,

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

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

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

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

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

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

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

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

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

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

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

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

Сергей Иванец, Александр Романов

Цифровой синтез: практический курс

**Глава 1. Основы комбинационной логики.
Маршрут разработки цифровых схем**

Содержание

1.1. Краткие теоретические сведения	1-4
1.2 Использование схемотехнического редактора	1-7
1.2.1 Установка пакета Quartus Prime	1-7
1.2.2 Создание проекта в схемотехническом редакторе	1-11
1.2.3 Создание файла в схемотехническом редакторе	1-16
1.2.4 Использование схемотехнического редактора (Schematic editor)	1-18
1.2.5 Разработка более сложной схемы	1-22
1.3 Компиляция проекта	1-24
1.3.1 Использование RTL Viewer	1-25
1.4 Назначение выводов. Компиляция проекта	1-25
1.5 Конфигурирование ПЛИС	1-27
1.6 Разработка схемы с использованием языка описания аппаратуры.	
Симуляция	1-30
1.6.1 Загрузка и установка ModelSim Starter Edition	1-31
1.6.2 Загрузка и установка пакетов Icarus Verilog и GTK Wave	1-33
1.7 HDL-модуль и его описание	1-34
1.8 Тестбенч и его описание	1-36
1.8.1 Симуляция с использованием ModelSim	1-38
1.8.2 Симуляция с использованием Icarus Verilog и GTK Wave	1-39
1.9 Создание описания схемы на языке Verilog HDL.	
Синтез схемы	1-40
1.10 Упражнения	1-43
1.10.1 Основное задание	1-43
1.10.2 Контрольные вопросы	1-44

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

Требования к аппаратным и программным средствам

Для выполнения практических работ понадобится следующее программное обеспечение:

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

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

Также в данном практикуме используется отладочная плата компании **Terasic DE10Lite**³. Она содержит микросхему **ПЛИС** компании **Intel FPGA MAX10⁴ (10M50DAF484C7G)**. В папке doc дополнительных материалов к настоящей главе (https://github.com/RomeoMe5/DDLM, lab_01/doc) размещены инструкция к данной отладочной плате и ее электрическая схема (эти же документы могут быть бесплатно загружены с сайта компании **Terasic**).

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

¹ <http://dl.altera.com/?edition=lite>.

² Инсталлятор можно найти в папке **pkg** материалов к данной главе.

³ <http://de10-lite.terasic.com/>.



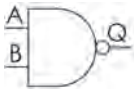
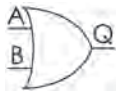
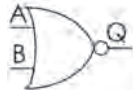
⁴ <https://www.altera.com/products/fpga/max-series/max-10/overview.html>.

⁵ <https://www.xilinx.com/support/answers/53764.htm>.

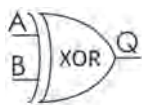
1.1. Краткие теоретические сведения

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

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

Вентиль	Символ	Уравнение	Таблица истинности															
НЕ (NOT)		$Q = \overline{A}$	<table><tr><th>A</th><th>Q</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	Q	0	1	1	0									
A	Q																	
0	1																	
1	0																	
И (AND)		$Q = A \cdot B = A \& B$	<table><tr><th>A</th><th>B</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Q	0	0	0	0	1	0	1	0	0	1	1	1
A	B	Q																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
И-НЕ (NAND)		$Q = \overline{A \cdot B} = \overline{A \& B}$	<table><tr><th>A</th><th>B</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Q	0	0	1	0	1	1	1	0	1	1	1	0
A	B	Q																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
ИЛИ (OR)		$Q = A + B$	<table><tr><th>A</th><th>B</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	A	B	Q	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Q																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
ИЛИ-НЕ (NOR)		$Q = \overline{A + B}$	<table><tr><th>A</th><th>B</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	A	B	Q	0	0	1	0	1	0	1	0	0	1	1	0
A	B	Q																
0	0	1																
0	1	0																
1	0	0																
1	1	0																

Исключающее
ИЛИ (XOR)



$$Q = A \oplus B$$

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

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

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

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

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

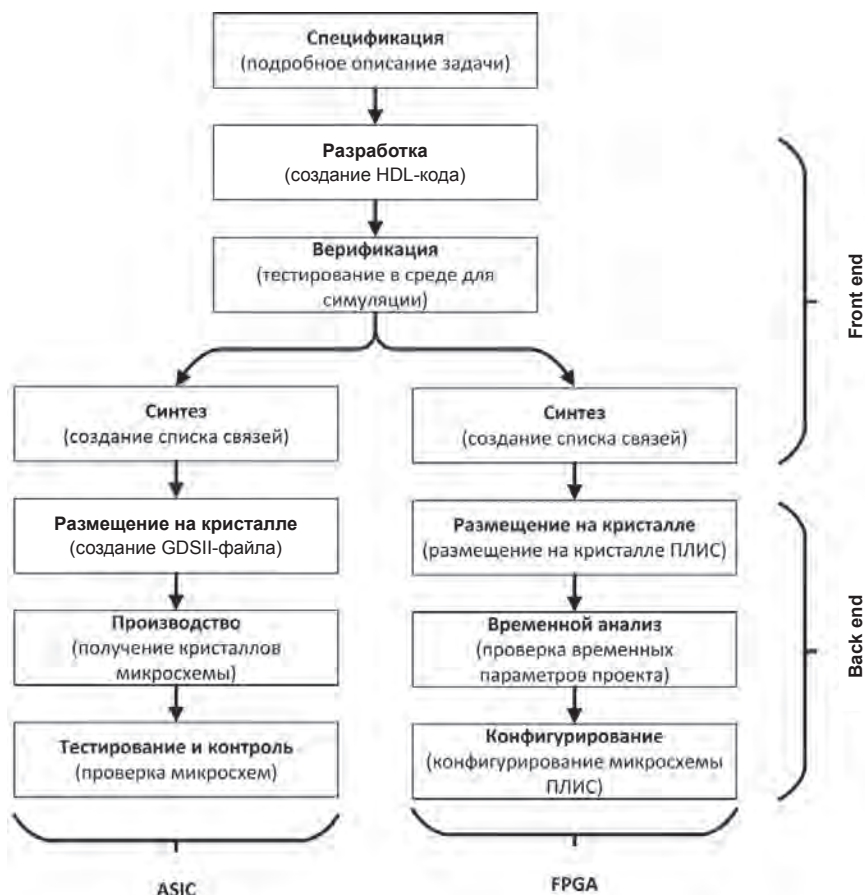


Рис. 1.1 Порядок разработки проектов для ASIC и FPGA

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

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

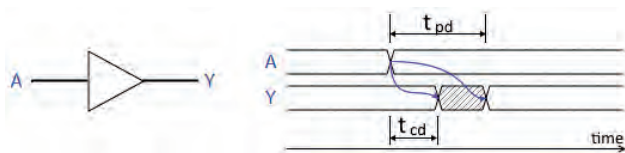


Рис. 1.2 Минимальная и максимальная задержки

1.2 Использование схемотехнического редактора

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

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

1.2.1 Установка пакета Quartus Prime

Существует несколько версий пакета Quartus Prime:

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

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

¹ <https://fpgasoftware.intel.com/?edition=lite>.

² https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/po/ss-quartus-comparison.pdf.

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

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

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

Основные этапы установки приведены на [рис. 1.3](#), [1.4](#), [1.5](#), [1.6](#), [1.7](#).



Рис. 1.3 Начало установки Quartus Prime Lite Edition

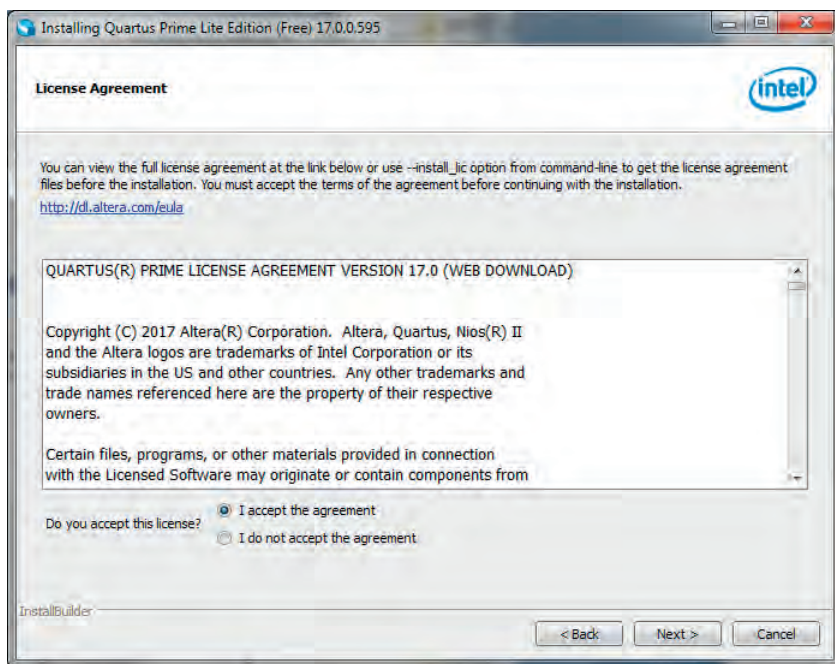


Рис. 1.4 Соглашение о лицензии в Quartus Prime Lite Edition

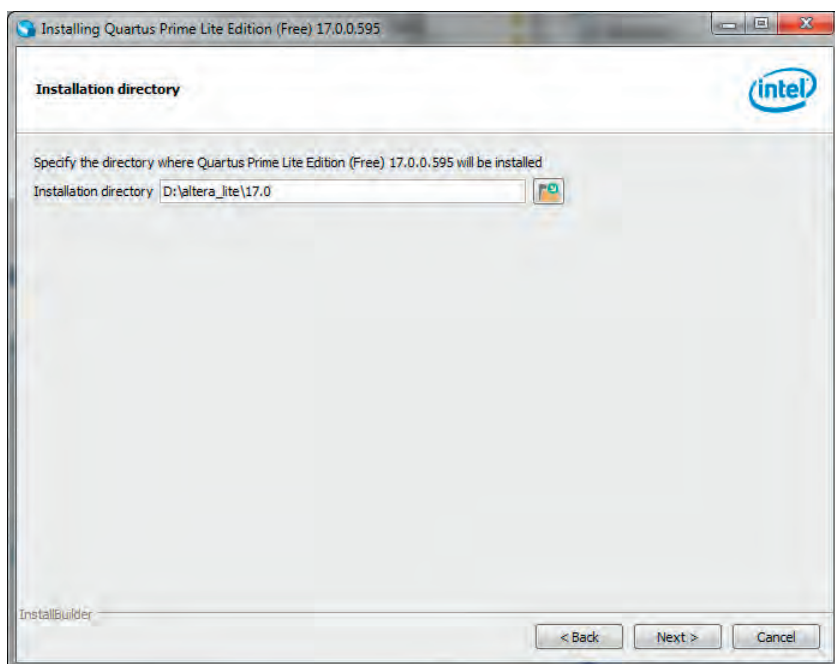


Рис. 1.5 Назначение папки, в которую будет установлен Quartus Prime Lite Edition

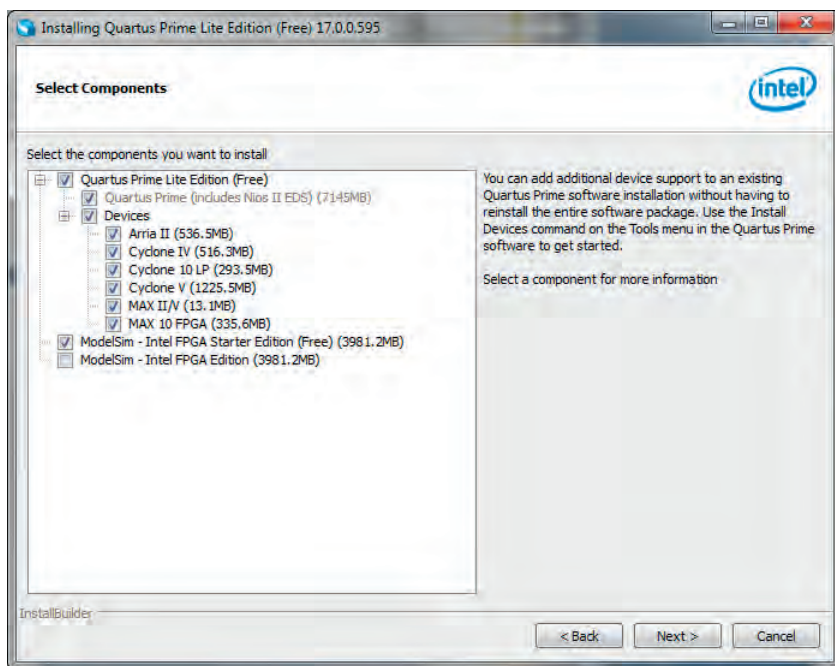


Рис. 1.6 Выбор компонентов Quartus Prime Lite Edition и поддерживаемых семейств микросхем

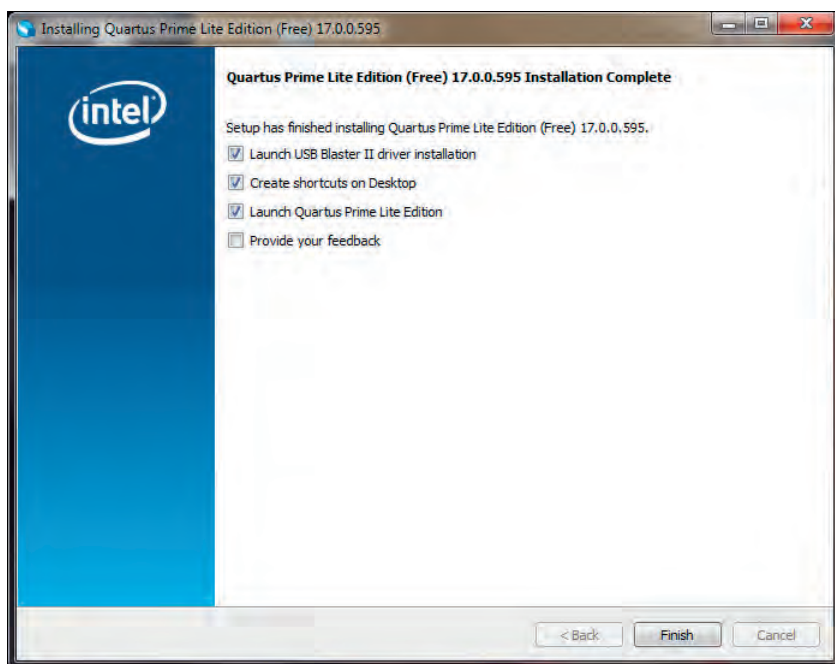


Рис. 1.7 Окончание установки Quartus Prime Lite Edition

1.2.2 Создание проекта в схемотехническом редакторе

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

Перед созданием нового проекта необходимо определить некоторые условия:

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

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

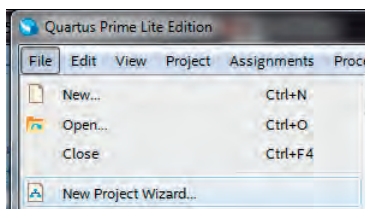



Рис. 1.8 Запуск мастера нового проекта New Project Wizard

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

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

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

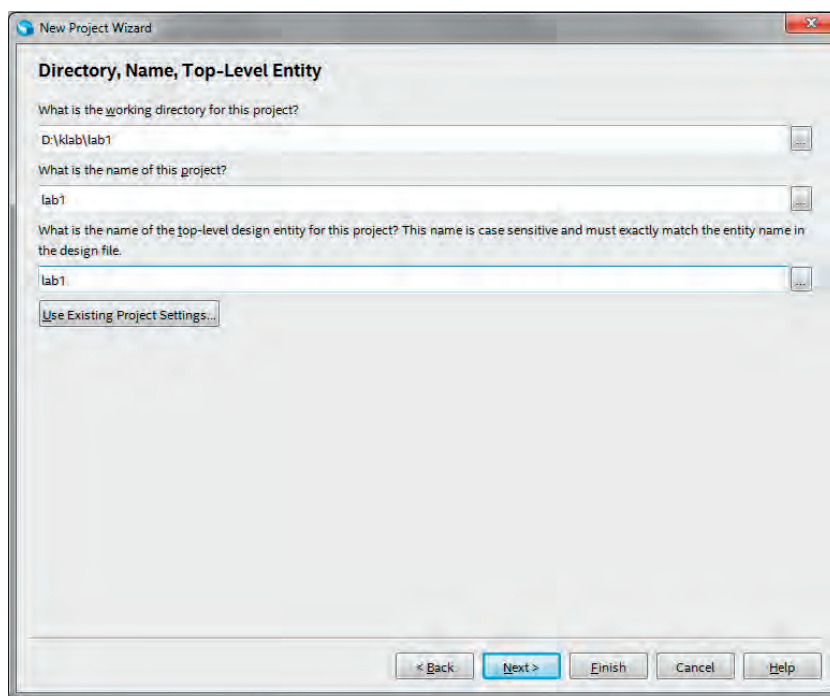


Рис. 1.9 New Project Wizard. Шаг 1

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

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

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

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

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

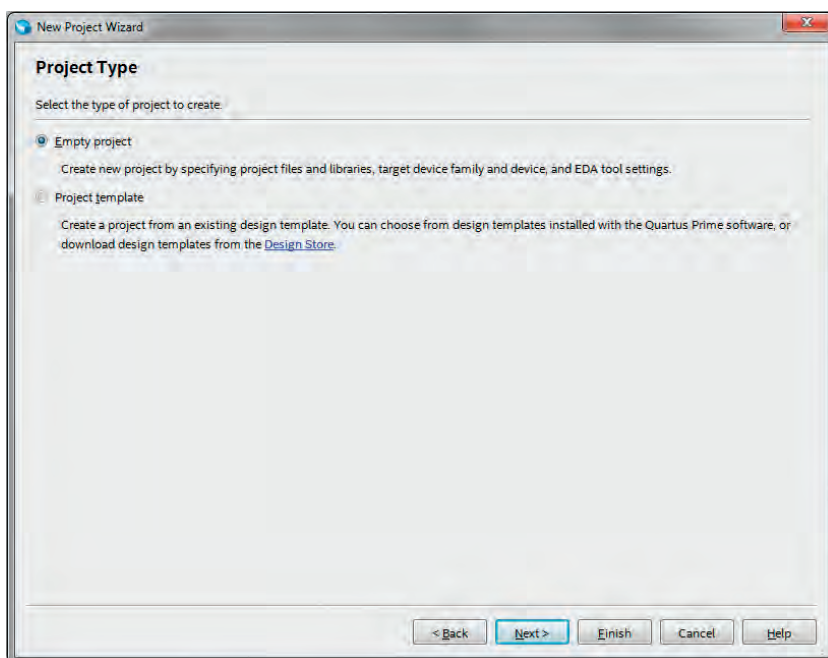


Рис. 1.10 New Project Wizard. Шаг 2

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

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

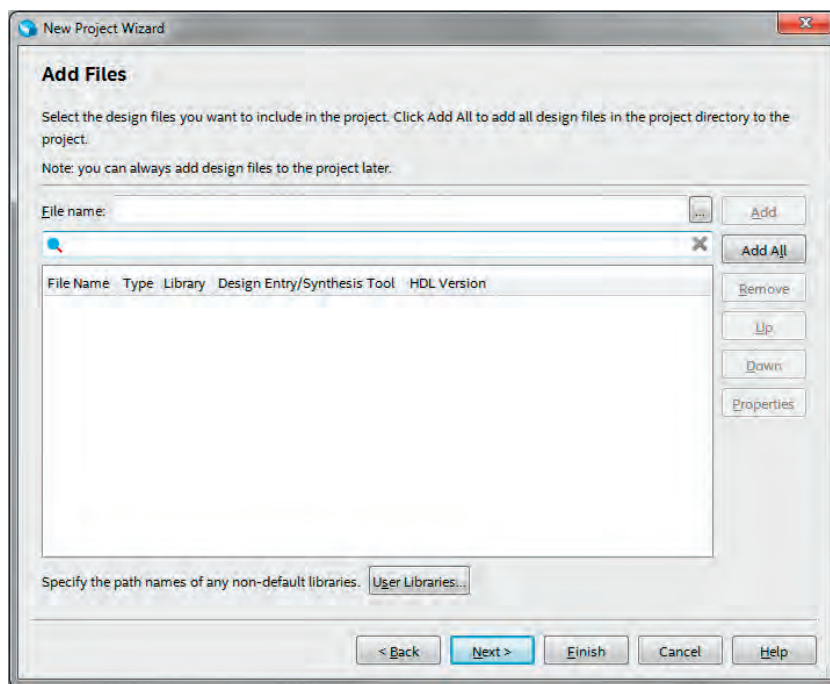


Рис. 1.11 New Project Wizard. Шаг 3

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

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

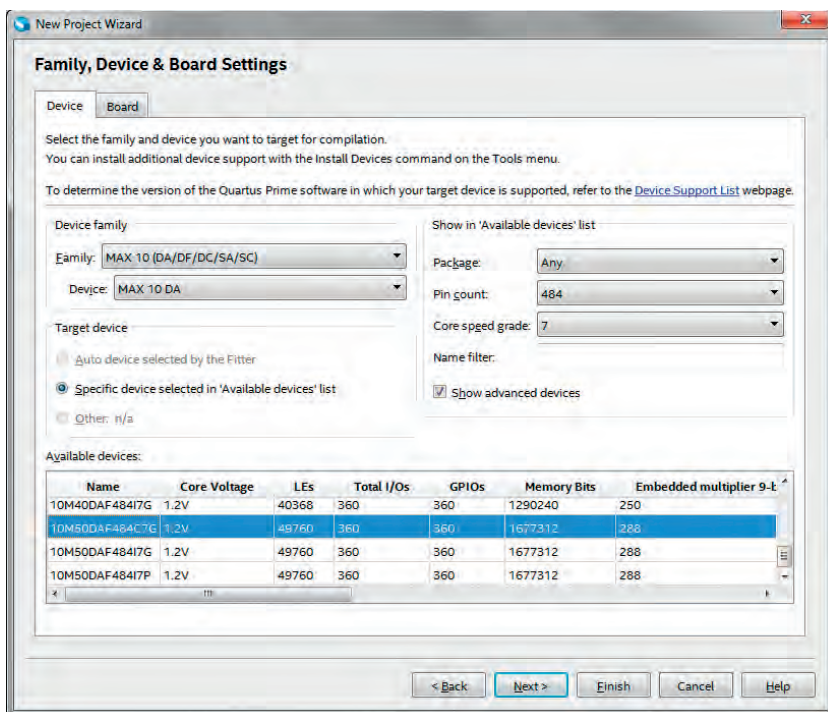


Рис. 1.12 New Project Wizard. Шаг 4

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

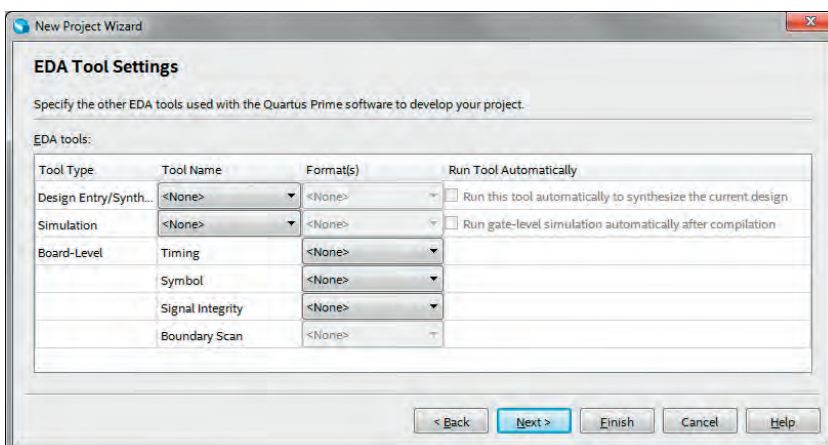


Рис. 1.13 New Project Wizard. Шаг 5

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

Илья Кудрявцев, Ирина Романова,
Александр Романов, Юрий Панчул

Цифровой синтез: практический курс

**Глава 2. Основы последовательностной логики.
Управление энергопотреблением цифровой схемы**

Содержание

2.1	Последовательностные элементы	2-3
2.1.1	Асинхронный RS-триггер	2-3
2.1.2	D-защелка	2-8
2.1.3	D-триггер	2-11
2.1.4	JK-триггер	2-15
2.1.5	T-триггер	2-16
2.2	Реализация последовательностной логики на языке Verilog	2-16
2.2.1	Реализация D-защелки с использованием языка Verilog	2-21
2.2.2	Реализация D-триггера с использованием языка Verilog	2-23
2.2.3	Реализация JK-триггера с помощью языка Verilog	2-24
2.3	Синхронный и асинхронный сбросы	2-25
2.3.1	D-триггер с синхронным сбросом	2-25
2.3.2	D-триггер с асинхронным сбросом	2-28
2.3.3	Преимущества и недостатки синхронного и асинхронного сбросов	2-29
2.3.4	Активируемый триггер	2-30
2.3.5	Параметризированный триггер	2-32
2.4	Энергопотребление цифровых устройств	2-35
2.4.1	Общие сведения	2-35
2.4.2	Разработка структуры устройства с учетом энергопотребления	2-37
2.5	Упражнения	2-40
2.5.1	Основное задание	2-40
2.5.2	Контрольные вопросы	2-40

Александр Барабанов, Александр Романов

Цифровой синтез: практический курс

**Глава 3. Шифраторы и дешифраторы.
Скорость работы комбинационных блоков**

Содержание

3.1	Шифраторы	3-3
3.2	Описание шифраторов на языке Verilog	3-4
3.2.1	Неприоритетный шифратор на основе оператора непрерывного присваивания assign	3-4
3.2.2	Неприоритетный шифратор, реализованный с помощью операторов if	3-6
3.2.3	Неприоритетный шифратор, реализованный с помощью оператора case	3-8
3.2.4	Приоритетный шифратор, реализованный с помощью операторов if	3-10
3.2.5	Приоритетный шифратор с использованием оператора assign	3-11
3.2.6	Параметрический шифратор	3-12
3.2.7	Изучение временных характеристик цифровых устройств	3-13
3.3	Дешифраторы	3-22
3.3.1	Дешифратор с использованием оператора case	3-22
3.3.2	Дешифратор с использованием оператора сдвига	3-24
3.3.3	Дешифратор с использованием оператора непрерывного присваивания assign	3-24
3.3.4	Дешифратор для учебного MIPS-совместимого процессора	3-26
3.3.5	Параметрический дешифратор	3-28
3.3.6	Сравнение дешифраторов	3-30
3.4	Оптимизация при синтезе	3-31
3.4.1	Режимы оптимизации	3-32
3.5	Упражнения	3-39
3.5.1	Основное задание	3-39
3.5.2	Задания для самостоятельной работы	3-39
3.5.3	Контрольные вопросы	3-40

Александр Романов, Юрий Панчул

Цифровой синтез: практический курс

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

Содержание

4.1	Проектирование мультиплексоров	4-3
4.1.1	Однобитный мультиплексор 2в1	4-3
4.1.2	Двухбитный мультиплексор 2в1	4-8
4.1.3	Двухбитный мультиплексор 4в1	4-11
4.1.4	Иерархический подход при проектировании цифровых систем	4-12
4.1.5	Неполный мультиплексор 3в1	4-14
4.1.6	Разработка логических функций на мультиплексорах	4-17
4.2	Демультиплексор	4-18
4.3	Селектор	4-23
4.4	Полностью параметризованный мультиплексор и селектор. Конструкция generate	4-24
4.5	Некоторые хитрости при создании селекторов и мультиплексоров	4-27
4.5.1	Селектор, созданный на логических элементах И и ИЛИ	4-27
4.5.2	Распределенный селектор	4-28
4.5.3	Пример использования распределенного мультиплексора «из жизни»	4-31
4.6	Упражнения	4-32
4.6.1	Основное задание	4-32
4.6.2	Задания для самостоятельной работы	4-33
4.6.3	Контрольные вопросы	4-34

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

Требования к аппаратным и программным средствам

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

- персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;
- пакет **Quartus Prime** (есть студенческая версия);
- пакет **ModelSim Altera Edition** или программы **Icarus Verilog** и **GTKWave**;
- отладочная плата компании **Terasic DE10Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах¹ к данной книге).

4.1 Проектирование мультиплексоров

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

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

4.1.1 Однобитный мультиплексор 2в1

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

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

¹ <https://github.com/RomeoMe5/DDLM>.

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

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

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

Листинг 4.1 Комбинационный мультиплексор 2в1

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

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

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

Листинг 4.2 Мультиплексор 2в1 на основе тернарного оператора

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

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



```

        y = d0;
    end
endmodule

```

Листинг 4.3 Мультиплексор 2в1 на основе условного оператора

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

```

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

```

Листинг 4.4 Мультиплексор 2в1 на основе оператора множественного выбора

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

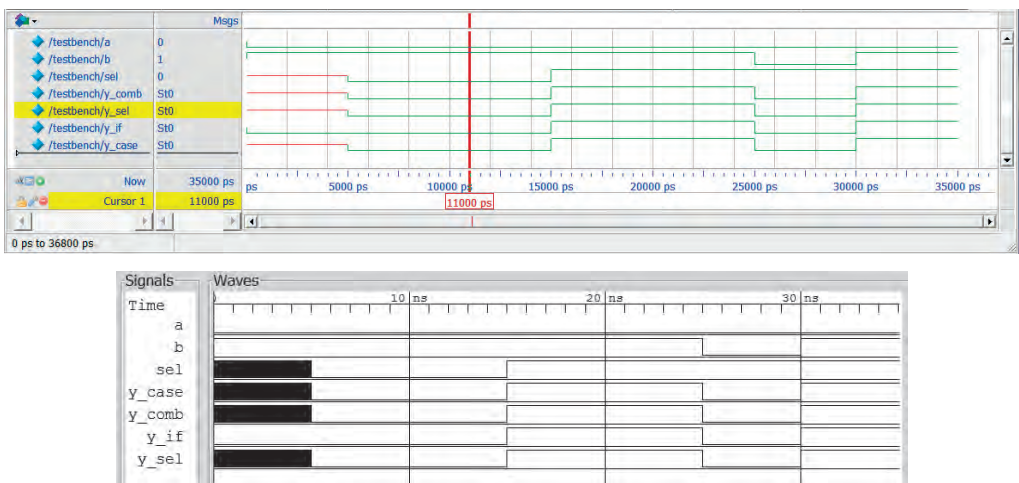


Рис. 4.1 Результаты моделирования различных реализаций однобитных мультиплексоров 2в1



```

Transcript
# testbench
# End time: 11:21:44 on Nov 12,2017, Elapsed time: 0:00:00
# Errors: 0, Warnings: 0
# vsim work.testbench
# Start time: 11:21:44 on Nov 12,2017
# Loading work.testbench
# Loading work.b1_mux_2_1_comb
# Loading work.b1_mux_2_1_sel
# Loading work.b1_mux_2_1_if
# Loading work.b1_mux_2_1_case
# a=0 b=1 sel=x y_comb=x y_sel=x y_if=0 y_case=x
# a=0 b=1 sel=0 y_comb=0 y_sel=0 y_if=0 y_case=0
# a=0 b=1 sel=1 y_comb=1 y_sel=1 y_if=1 y_case=1
# a=0 b=0 sel=1 y_comb=0 y_sel=0 y_if=0 y_case=0
# a=0 b=1 sel=1 y_comb=1 y_sel=1 y_if=1 y_case=1
# 0 ps
# 37 ps

```

Рис. 4.1 (Продолжение)

Содержимое тестбенча (тестового окружения) приведено ниже:

```

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

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

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

```

```

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

```

Листинг 4.5 Тестбенч для моделирования различных реализаций однобитных мультиплексоров 2в1

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

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

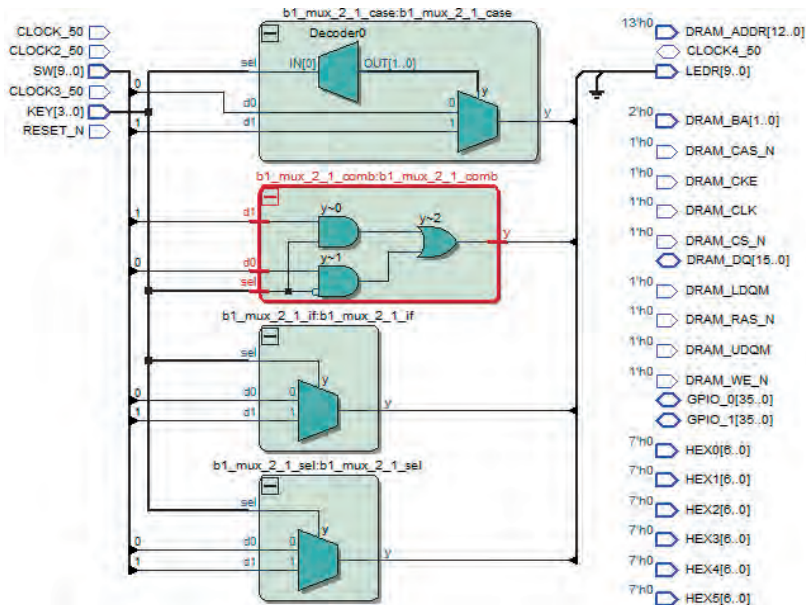


Рис. 4.2 RTL-представление различных реализаций однобитных мультиплексоров 2в1

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

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

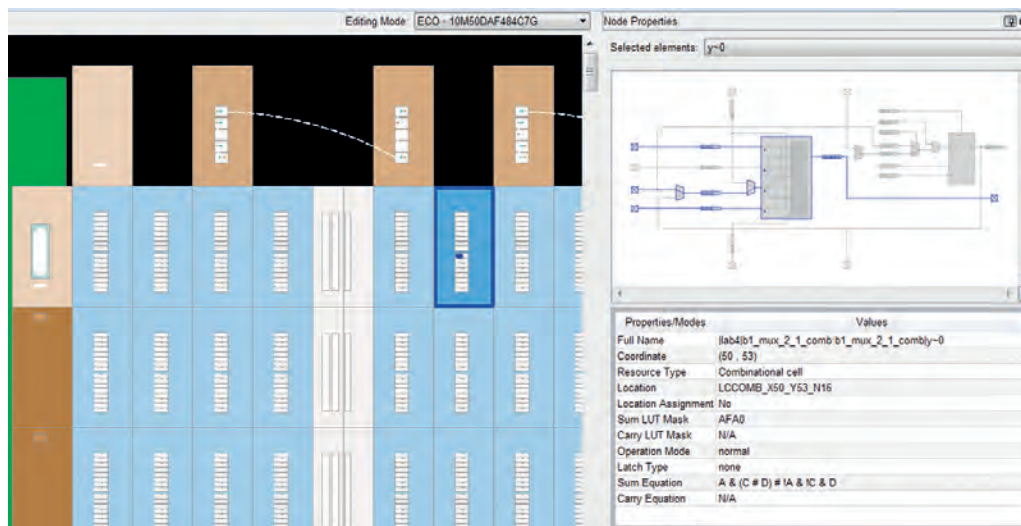


Рис. 4.3 Мультиплексор 2в1 в Chip Planner Quartus Prime

Дополнительное задание для самостоятельной работы

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

```
module b1_mux_2_1_concat
(
    input d0,
    input d1,
    input sel,
    output y
);
    wire [1:0] dataIn;
    assign dataIn = {d1,d0};
    assign y = dataIn[sel];
endmodule
```

Листинг 4.6 Мультиплексор 2в1, реализованный с помощью операции конкатенации

4.1.2 Двухбитный мультиплексор 2в1

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

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

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

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

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

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

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

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

Листинг 4.8 Некорректный комбинационный двухбитный мультиплексор 2в1

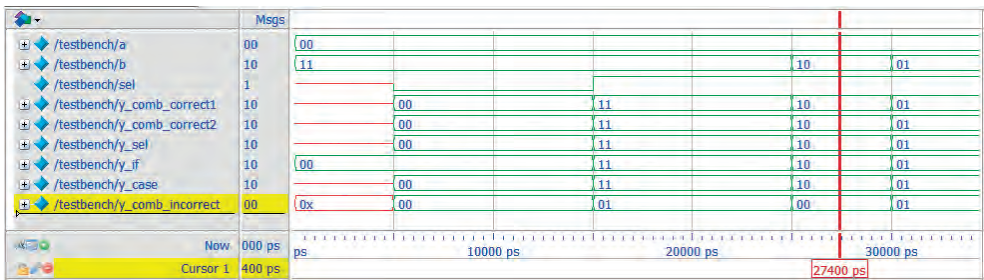


Рис. 4.4 Результаты моделирования двухбитного мультиплексора 2в1

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

Примеры правильной реализации комбинационного мультиплексора приведены ниже:

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

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

```

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

```

Листинг 4.9 Правильный комбинационный двухбитный мультиплексор 2в1

Дополнительное задание для самостоятельной работы

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

4.1.3 Двухбитный мультиплексор 4в1

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

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

```

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

```

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

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

```

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

```

```

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

```

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

4.1.4 Иерархический подход при проектировании цифровых систем

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

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

```

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

```

Листинг 4.12 Двухбитный модульный мультиплексор 4в1

Результаты моделирования всех представленных вариантов мультиплексоров совпадают:

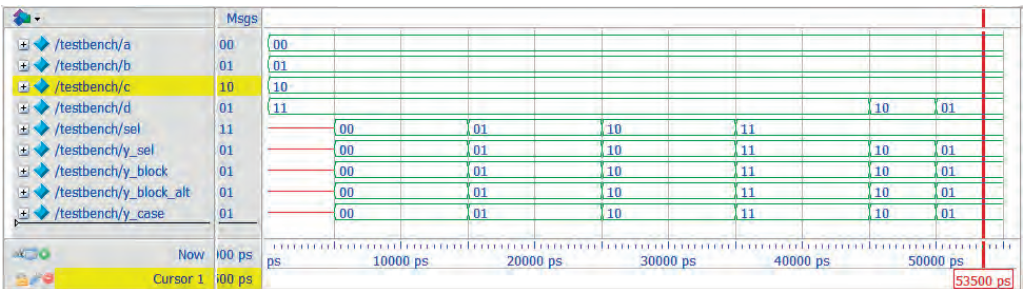


Рис. 4.5 Результаты моделирования различных реализаций мультиплексоров 4в1

Результаты синтеза различных реализаций мультиплексоров существенно различаются:

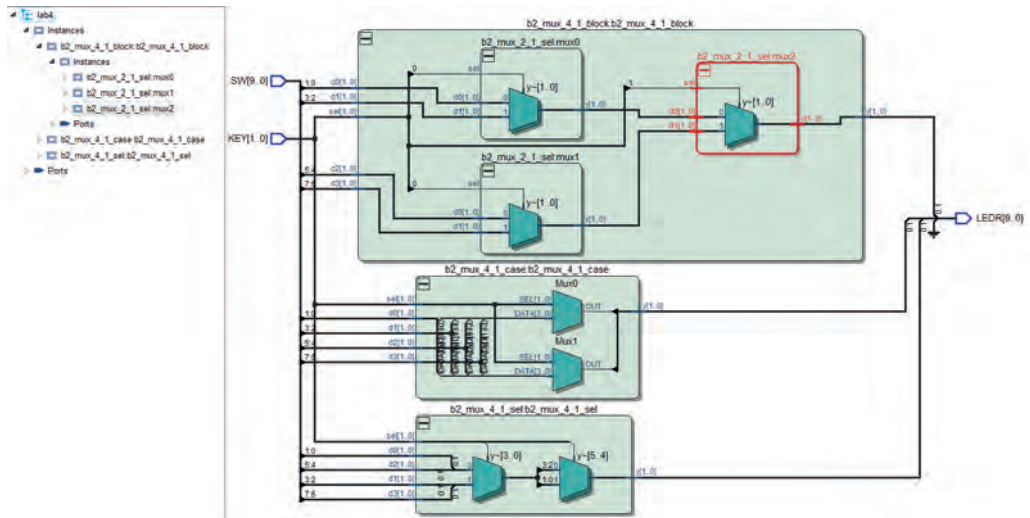


Рис. 4.6 RTL-схема различных реализаций мультиплексоров 4в1

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

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

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

```

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

```

endmodule

Листинг 4.13 Альтернативная реализация двухбитного модульного мультиплексора 4в1

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

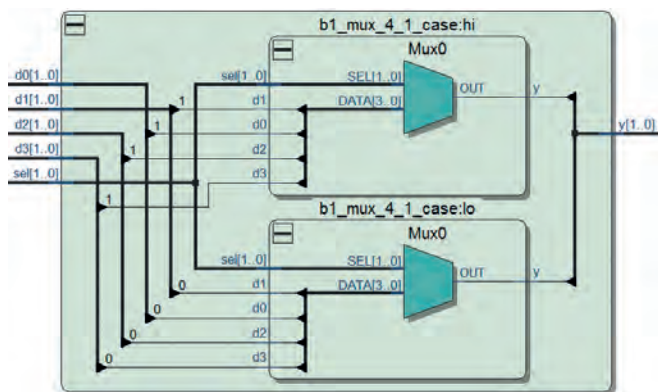


Рис. 4.7 RTL-схема модульного мультиплексора 4в1

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

Дополнительное задание для самостоятельной работы

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

4.1.5 Неполный мультиплексор 3в1

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

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

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

Листинг 4.14 Мультиплексор 3в1 с защелкой

Результаты моделирования мультиплексора 3в1:

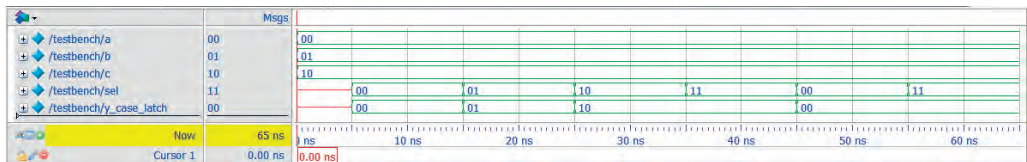


Рис. 4.8 Результаты моделирования мультиплексора 3в1

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

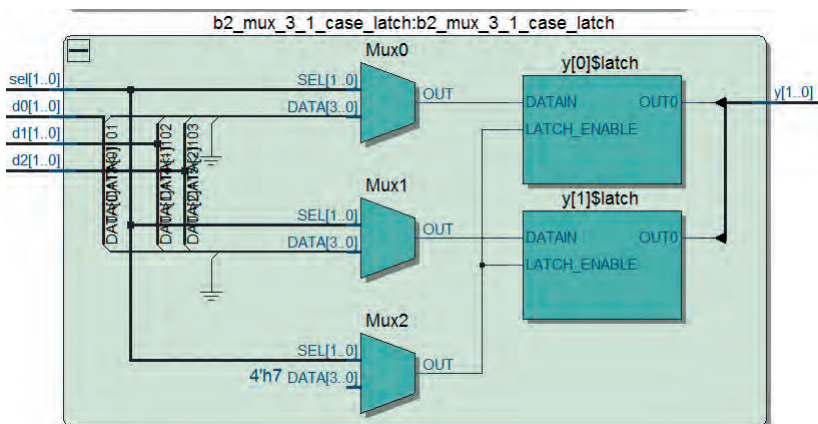


Рис. 4.9 Мультиплексор 3в1 с защелкой в RTL-схеме

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

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

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

Листинг 4.15 Мультиплексор 3в1 без защелки

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

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

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

Листинг 4.16 Мультиплексор 3в1 с x-состоянием

Результаты моделирования мультиплексора 3в1:

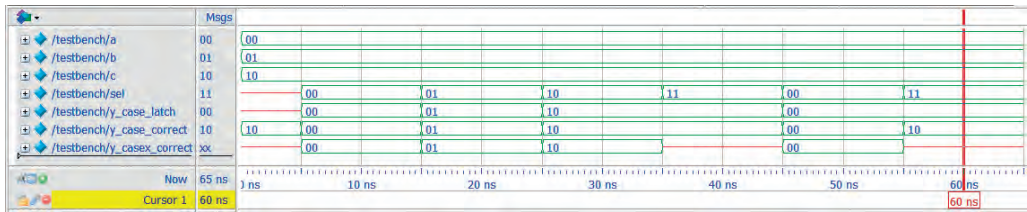


Рис. 4.10 Результаты моделирования мультиплексора 3в1

Схематически такие мультиплексоры синтезируются без регистра защелки:

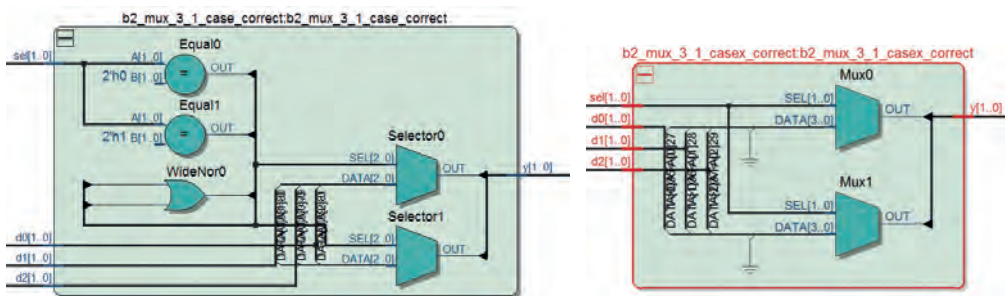


Рис. 4.11 Мультиплексоры 3в1 в RTL-схеме

Часто разработчики используют директиву

```
//synopsys full_case parallel_case
```

Однако ее применения правильнее избегать из-за возможных неприятных последствий, описанных в статье С. Е. Cummings¹.

Дополнительное задание для самостоятельной работы

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

4.1.6 Разработка логических функций на мультиплексорах

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

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

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

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

Идея создания логических элементов из мультиплексоров широко распространена в сообществе разработчиков цифровых систем, а задача «Реализуйте такой-то логический элемент на двухвходовом мультиплексоре» является типичным вопросом в интервью для соискателей работы в этой сфере². И это неудивительно, поскольку мультиплексоры находят широчайшее применение в цифровом синтезе и являются важными элементами в структуре арифметических блоков ([глава 5](#)), в цифровых автоматах ([главы 8, 9](#)), при реализации софт-процессоров ([глава 11](#)) и т. д. Особая роль отводится мультиплексорам для реализации совместного использования и распределения ресурсов в арифметических операциях при разработке систем цифровой обработки сигналов (данный подход хорошо продемонстрирован в книге «**A Verilog HDL Primer**»³ в главе 4 «Оптимизация моделей»).

Дополнительное задание для самостоятельной работы

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

4.2 Демультиплексор

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

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

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

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

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

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

Листинг 4.17 Демультиплексор 1в4 с одноразрядным сдвигом

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

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



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

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

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

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

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

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

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



```

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

```

Листинг 4.20 Пример модуля с конкретизацией параметров и директивами условной компиляции

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

```

module b2_demux_1_4_block
(
    input [1:0] din,
    input [1:0] sel,
    output [1:0] dout0,
    output [1:0] dout1,
    output [1:0] dout2,
    output [1:0] dout3
);
    b1_demux_1_4_case dmux0 (.din(din[0]), .sel(sel), .dout0(dout0[0]),
        .dout1(dout1[0]), .dout2(dout2[0]), .dout3(dout3[0]));
    b1_demux_1_4_shift dmux1 (.din(din[1]), .sel(sel), .dout0(dout0[1]),
        .dout1(dout1[1]), .dout2(dout2[1]), .dout3(dout3[1]));
endmodule

```

Листинг 4.21 Двухбитный модульный демультиплексор 1в4

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

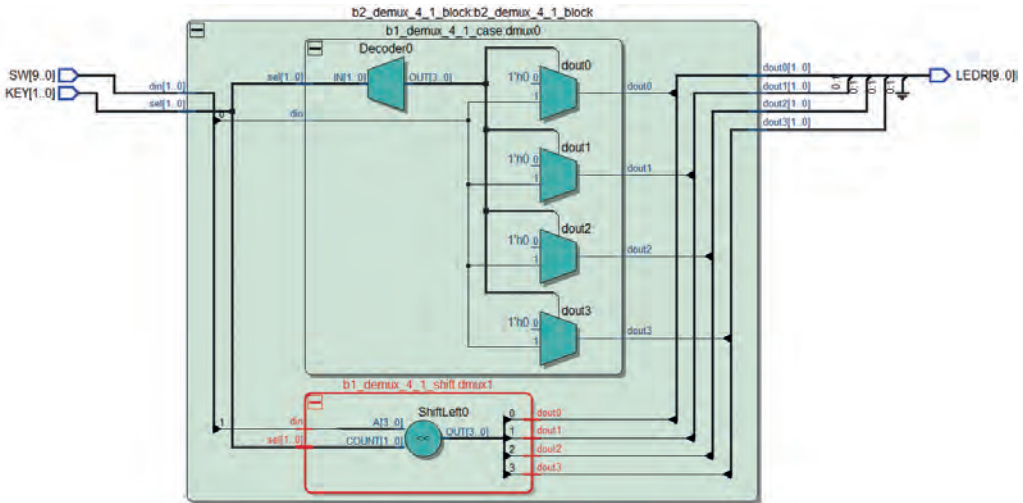


Рис. 4.12 Двухбитный модульный демультиплексор 1в4 в виде RTL-схемы

Результаты моделирования реализаций восьмибитного параметризованного демультиплексора и модульного демультиплексора, иллюстрирующие однотипную логику их поведения, приведены на [рис. 4.13](#).

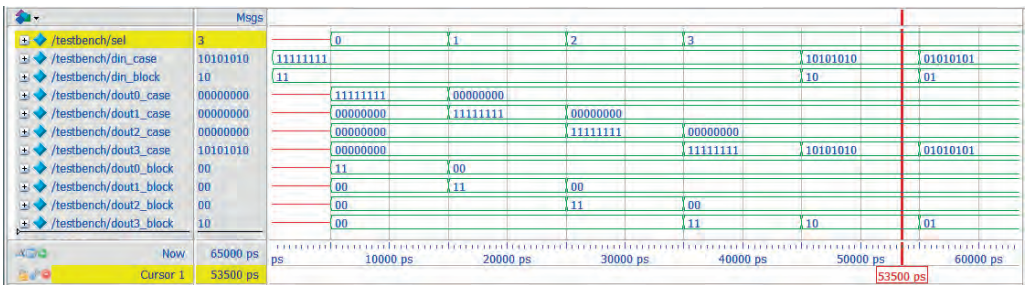


Рис. 4.13 Результаты моделирования демультиплексоров 1в4

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

Дополнительное задание для самостоятельной работы

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

4.3 Селектор

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

Пример реализации селектора и результаты его моделирования приведены ниже:

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

always @(*)
    case (sel)
        8'b00000001: y=d0;
        8'b00000010: y=d1;
        8'b00000100: y=d2;
        8'b00001000: y=d3;
        8'b00010000: y=d4;
        8'b00100000: y=d5;
        8'b01000000: y=d6;
        8'b10000000: y=d7;
        default: y={DATA_WIDTH{1'b0}};
    endcase
endmodule
```

Листинг 4.22 N-битный селектор 8в1

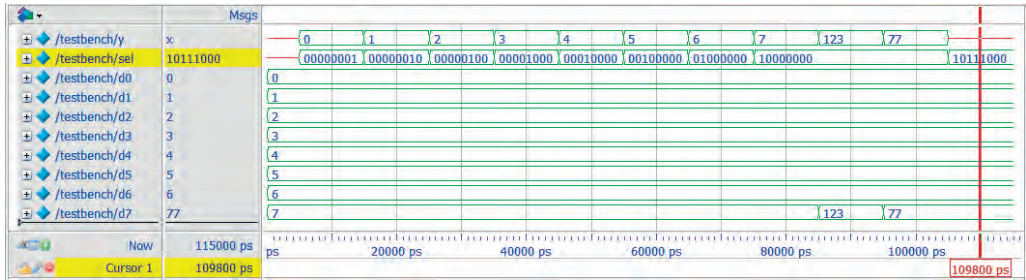


Рис. 4.14 Результаты моделирования N-битного селектора 8в1

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

Дополнительное задание для самостоятельной работы

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

4.4 Полностью параметризованный мультиплексор и селектор. Конструкция generate

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

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

Листинг 4.23 Полностью параметризованный мультиплексор

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

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

Здесь необходимо также отметить следующий момент код:

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

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

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

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

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

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

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

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

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

1 Sutherland S. Verilog-2001: A Guide to the New Features of the Verilog® Hardware Description Language (Vol. 652). Springer Science & Business Media, 2012.

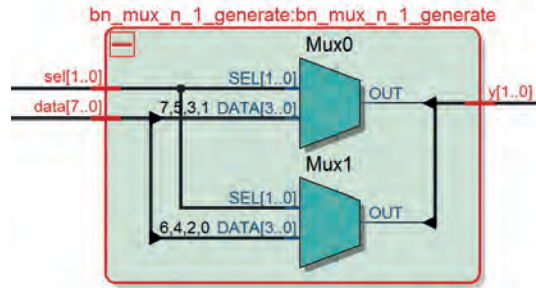


Рис. 4.15 Полностью параметризованный мультиплексор в виде RTL-схемы

Еще более красивый пример использования конструкции **generate** можно продемонстрировать на примере полностью параметризованного селектора:

```
module bn_selector_n_1_generate
#( parameter DATA_WIDTH = 8,
  parameter INPUT_CHANNELS = 2 )
(
  input [(INPUT_CHANNELS*DATA_WIDTH)-1:0] data,
  input [INPUT_CHANNELS-1:0] sel,
  output [DATA_WIDTH-1:0] y
);
genvar i;
generate
  for(i=0;i<INPUT_CHANNELS;i=i+1)
  begin: gen_array
    assign y = sel[i] ? data[((i+1)*DATA_WIDTH)-1:(i*DATA_WIDTH)] :
      {DATA_WIDTH{1'bx}};
  end
endgenerate
endmodule
```

Листинг 4.24 Полностью параметризованный селектор

Следует обратить внимание на то, что в данном модуле предусмотрена обработка ситуации, когда вход **sel** некорректный (тогда на выходе будет состояние 'x'). Здесь также следует оговориться, что множество параллельных присваиваний 'x' может не поддерживаться синтезирующим САПР'ом. В такой ситуации каждый случай следует рассматривать индивидуально.

Используя конструкцию **generate**, можно создавать множество повторяющихся блоков и даже размножать модули, об этом более подробно рассказано в главе 10.

Дополнительное задание для самостоятельной работы

Ответьте на вопрос, почему нельзя модуль полностью параметризованного мультиплексора (листинг 4.23) реализовать по аналогии с селектором (листинг 4.24) следующим образом:

```
assign y = data[((sel+1)*DATA_WIDTH)-1:(sel*DATA_WIDTH)];
```

4.5 Некоторые хитрости при создании селекторов и мультиплексоров

Рассмотрим еще несколько примеров хитростей, которые можно использовать при создании мультиплексоров и селекторов.

4.5.1 Селектор, созданный на логических элементах И и ИЛИ

Как уже было сказано, селектор – это **one-hot** версия мультиплексора, удобная для применения в проектах, где требуется высокая частота функционирования модуля. Обычно селекторы создают на основе оператора **full case**, но существует также подход с использованием логических операций **И** и **ИЛИ**. Рассмотрим пример такого селектора, состоящего из **DATA_WIDTH*4**-битного информационного входного сигнала, одного 4-битного адресного входного сигнала и одного **DATA_WIDTH**-битного выходного сигнала:

```
module bn_selector_4_1_and_or
#( parameter DATA_WIDTH = 32)
(
    input [(DATA_WIDTH*4)-1:0] data,
    input [3:0] sel,
    output [DATA_WIDTH-1:0] y
);
    wire [(DATA_WIDTH*4)-1:0] mask;
    wire [(DATA_WIDTH*4)-1:0] masked_data;

    assign mask = {{DATA_WIDTH{sel[3]}},
                  {DATA_WIDTH{sel[2]}},
                  {DATA_WIDTH{sel[1]}},
                  {DATA_WIDTH{sel[0]}}};
    assign masked_data = data & mask;
    assign y = masked_data[DATA_WIDTH*4-1:DATA_WIDTH*3] |
              masked_data[DATA_WIDTH*3-1:DATA_WIDTH*2] |
              masked_data[DATA_WIDTH*2-1:DATA_WIDTH] |
              masked_data[DATA_WIDTH-1:0];
endmodule
```

Листинг 4.25 И-ИЛИ селектор

Получившаяся **RTL**-схема селектора представлена ниже. Этот селектор полностью реализован с помощью логических элементов:

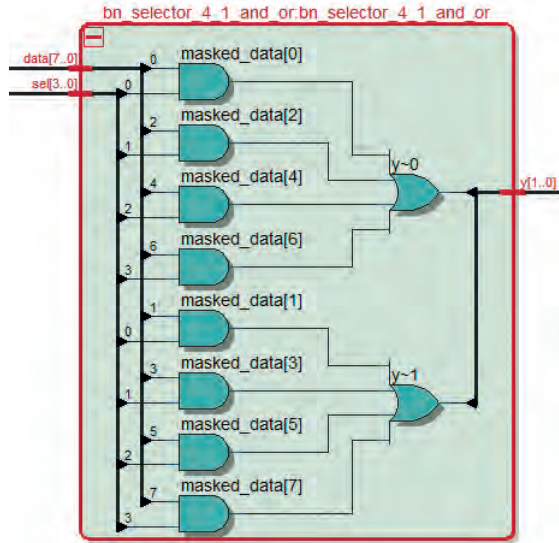


Рис. 4.16 И-ИЛИ селектор в виде RTL-схемы

4.5.2 Распределенный селектор

В некоторых задачах данные, поступающие на входные порты селектора, могут приходить с различной задержкой, в случае когда на различных путях распространения данных (**data flow**) находятся разные комбинационные блоки с различной задержкой (например, в АЛУ (рис. 4.17)). Другой пример – цепочка из комбинационных блоков, данные с которой снимаются с одного из ее блоков, в зависимости от управляющего сигнала (рис. 4.18). Обычно на выходе таких блоков стоит селектор, который коммутирует нужное значение на выход.

Задержки блоков могут значительно отличаться между собой. Предположим, все однотипные блоки проиндексированы в порядке возрастания их задержки $dN > dI > d3 > d2 > d1$. Селектор в зависимости от разрядности данных и количества входов также будет иметь определенную задержку, обозначенную $s1$.

В этом случае рассматриваемые блоки будут иметь различную задержку сигналов в зависимости от того, по какому пути они проходят. Данное явление называется гонками (состязаниями) сигналов и устраняется введением синхронной логики путем защелкивания результирующего сигнала на триггере на выходе блока или с помощью логических методов путем добавления избыточной логики, как, например, соседнее кодирование.

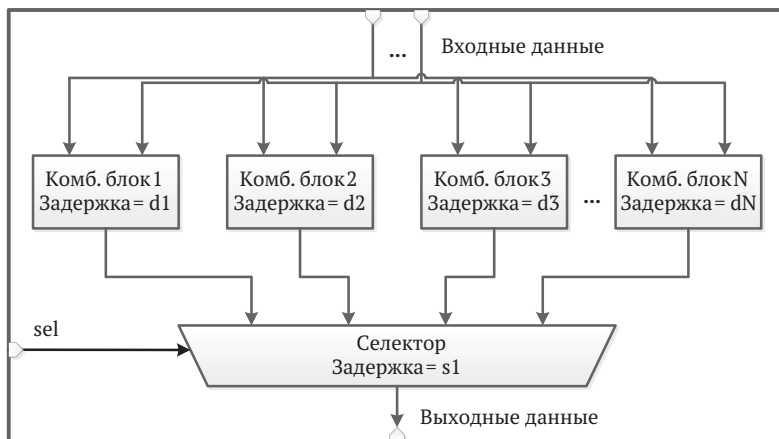


Рис. 4.17 Пример АЛУ

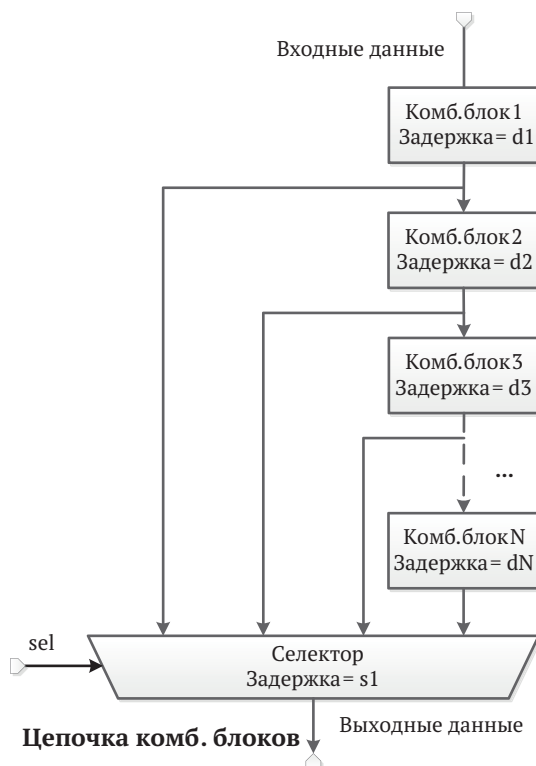


Рис. 4.18 Пример конвейера комбинационных блоков

Наибольшая задержка схемы **АЛУ** равна сумме задержки на селекторе и блоке с самой большой задержкой:

$$dALU_1 = s1 + dN,$$

а в случае с цепочкой комбинационных блоков наибольшая задержка будет равна сумме задержек всех блоков и селектора:

$$dPipeline_1 = s1 + \sum_{i=1}^N di.$$

Существует прием по улучшению характеристик таких модулей и уменьшению их наибольшей задержки за счет разбиения селектора на селекторы меньших размеров путем размещения их параллельно комбинационным модулям, как приведено на рисунках ниже.

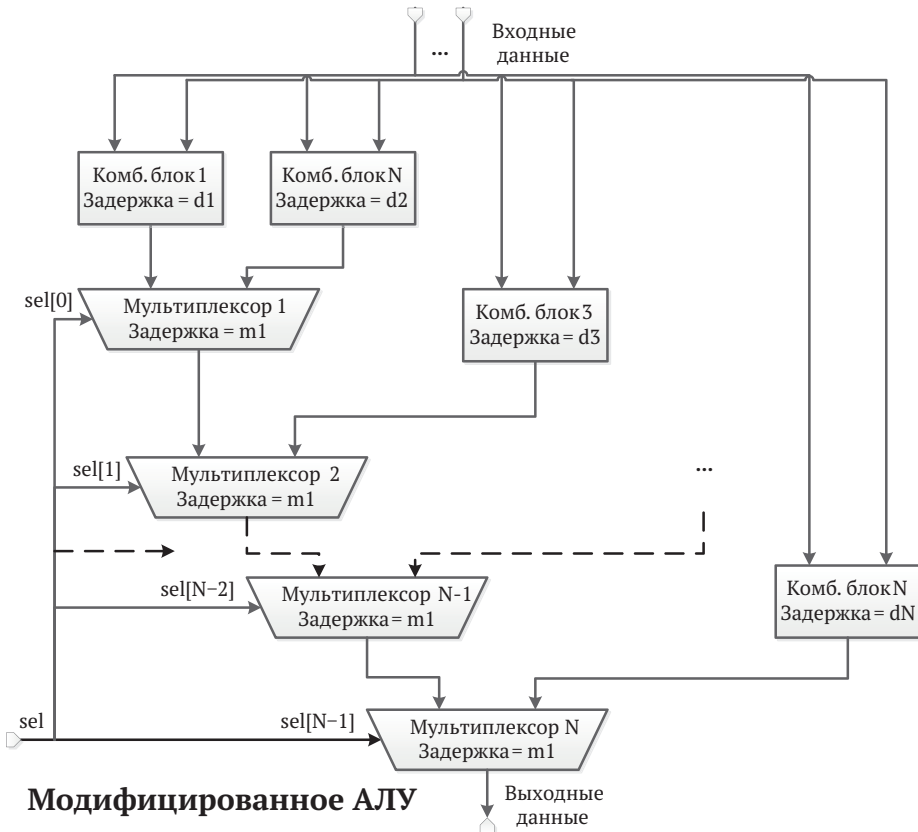


Рис. 4.19 Пример модифицированного АЛУ



Рис. 4.20 Пример улучшенной цепочки комбинационных блоков

Поскольку размеры этих селекторов меньше, чем у исходного селектора, обозначим их задержку как $s0$, и при этом она будет меньше ($s0 < s1$).

В таком случае если $s0 \ll d1$, то наибольшая задержка АЛУ будет:

$$dALU_0 = s0 + dN < dALU_1,$$

а в случае с цепочкой комбинационных блоков:

$$dPipeline_0 = s0 + \sum_{i=1}^N di < dPipeline_1.$$

То есть часть задержки на большом селекторе распределяется на этапы выше параллельно с комбинационными блоками.

4.5.3 Пример использования распределенного мультиплексора «из жизни»

Мультиплексоры широко распространены в различных применениях. В том числе без них не обходятся процессоры. В настоящее время активно развивается направление по разработке суперскалярных процессоров, где несколько арифметических операций могут выполняться параллельно.

Рассмотрим пример: имеется **N функциональных узлов**, у которых входные данные кладутся в регистры (**операнды**). Сами устройства содержат свои конвейеры и много комбинационной логики на выходе (**результаты**). При обычном подходе **результаты** должны поступать в **регистровый файл**, откуда вместе с внешними данными на следующем такте снова поступать на входы **функциональных узлов**.

Существует подход, когда добавляется быстрый обходной путь (**hot bypass**), с помощью которого данные с выходов **функциональных узлов** сразу поступают к ним на входы. При этом такой путь, скорее всего, будет критическим за счет комбинационной логики на выходе **функциональных узлов**. Чтобы уменьшить данный путь, мультиплексор разделяют на **две** стадии: верхняя коммутирует данные с **регистрового файла** и **внешних входов** (где задержка небольшая и критический путь не возникает), а нижняя – данные с верхней стадии мультиплексора и данные, пришедшие по обходному пути.

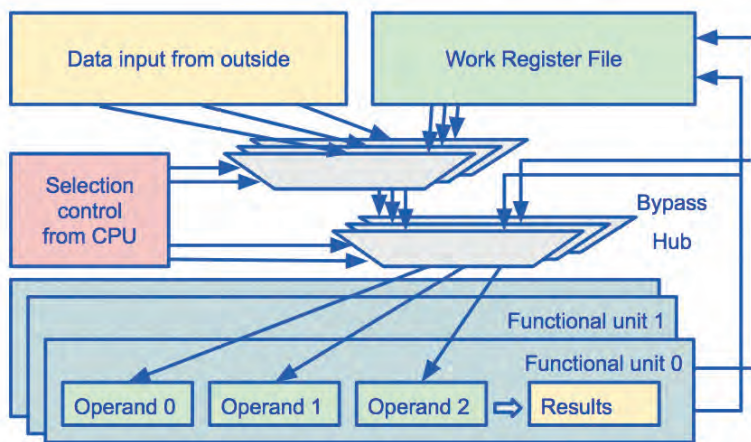


Рис. 4.21 Пример разделения мультиплексора для сокращения критического пути

Дополнительное задание для самостоятельной работы

Создайте полностью параметризованный **И-ИЛИ** селектор, используя оператор **generate**.

Реализуйте примеры **АЛУ** и конвейер на комбинационных блоках на рис. 4.17 и 4.18 на **Verilog**, исследуйте их **RTL**-схему и задержку прохождения сигналов.

Реализуйте примеры **АЛУ** и конвейер на комбинационных блоках на рис. 4.19 и 4.20 на **Verilog**, исследуйте их **RTL**-схему и задержку прохождения сигналов, сравните с предыдущим примером.

4.6 Упражнения

4.6.1 Основное задание

Выполните дополнительные задания для разделов 4.1, 4.2, 4.3, 4.4 и 4.5.

4.6.2 Задания для самостоятельной работы

Используя примеры кода из данной главы, реализуйте следующие мультиплексоры:

1. Мультиплексор 8в1, используя оператор `?:`.
2. Мультиплексор 8в1, используя оператор `if`.
3. Мультиплексор 8в1, используя оператор `case`.
4. Мультиплексор 8в1, создав два мультиплексора 4в1 и один мультиплексор 2в1. Реализовать модули мультиплексоров 4в1 и 2в1 при помощи оператора `?:`.
5. Мультиплексор 8в1, создав два мультиплексора 4в1 и один мультиплексор 2в1. Реализовать модули мультиплексоров 4в1 и 2в1 при помощи оператора `if`.
6. Мультиплексор 8в1, создав два мультиплексора 4в1 и один мультиплексор 2в1. Реализовать модули мультиплексоров 4в1 и 2в1 при помощи оператора `case`.
7. Мультиплексор 8в1, создав семь мультиплексоров 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора `?:`.
8. Мультиплексор 8в1, создав семь мультиплексоров 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора `if`.
9. Мультиплексор 8в1, создав семь мультиплексоров 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора `case`.
10. 3-битный мультиплексор 3в1, используя оператор `?:`.
11. 3-битный мультиплексор 3в1, используя оператор `if`.
12. 3-битный мультиплексор 3в1, используя оператор `case`.
13. 3-битный мультиплексор 3в1, используя два 3-битных мультиплексора 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора `?:`.
14. 3-битный мультиплексор 3в1, используя два 3-битных мультиплексора 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора `if`.
15. 3-битный мультиплексор 3в1, используя два 3-битных мультиплексора 2в1. Реализовать модуль мультиплексора 2в1 с помощью оператора `case`.
16. 3-битный мультиплексор 3в1, используя три 1-битных мультиплексора 3в1. Реализовать модуль мультиплексора 3в1 с помощью оператора `case`.
17. 3-битный мультиплексор 3в1, используя три 1-битных мультиплексора 3в1. Реализовать модуль мультиплексора 3в1 с помощью оператора `if`.
18. 3-битный мультиплексор 3в1, используя три 1-битных мультиплексора 3в1. Реализовать модули мультиплексора 3в1 с помощью оператора `?:`.

Разработайте таблицу истинности и изобразите условную схему подключения:

1. Мультиплексора 4в1, который реализует логический элемент **ИЛИ-НЕ**.
2. Мультиплексора 4в1, который реализует логический элемент **ИЛИ**.
3. Мультиплексора 4в1, который реализует логический элемент **И-НЕ**.
4. Мультиплексора 4в1, который реализует логический элемент **И**.
5. Мультиплексора 2в1, который реализует инвертор.

С помощью мультиплексора 8в1 опишите логическую функцию в виде условной схемы подключения:

1. $y = \overline{A} \wedge \overline{B} \wedge C \vee A \wedge \overline{B} \wedge \overline{C}$
2. $y = \overline{A} \wedge B \wedge C \vee A \wedge \overline{B} \wedge \overline{C}$
3. $y = A \wedge B \wedge C \vee A \wedge B \wedge \overline{C}$
4. $y = \overline{A} \wedge B \wedge C \vee A \wedge \overline{B} \wedge \overline{C} \vee \overline{A} \wedge B \wedge \overline{C}$
5. $y = (\overline{A} \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee B \vee \overline{C})$
6. $y = (A \vee B \vee C) \wedge (A \vee \overline{B} \vee \overline{C}) \wedge (\overline{A} \vee \overline{B} \vee \overline{C})$
7. $y = \overline{A} \vee A \wedge \overline{B} \wedge \overline{C}$
8. $y = \overline{A} \oplus (B \wedge C \vee A \wedge \overline{B} \wedge \overline{C})$
9. $y = \overline{A} \oplus (B \wedge C)$
10. $y = A \oplus (B \vee C)$

4.6.3 Контрольные вопросы

1. Каким логическим устройством (комбинационным/последовательностным) является мультиплексор? Обоснуйте свой ответ.
2. Приведите три примера реализации мультиплексора 4в1 на **Verilog**. Какая реализация лучше?
3. Что из себя представляет тестбенч при разработке на **Verilog**? Для чего требуется разрабатывать тестбенчи?
4. В чем преимущества иерархического подхода при проектировании цифровых систем?
5. Каковы преимущества подхода к проектированию мультиплексоров с использованием оператора **case**?
6. Для чего используется ключевое слово **default** совместно с оператором **case**? Какие могут возникнуть проблемы при применении директивы `// synopsys full_case parallel_case`?
7. Как можно создавать логические схемы на мультиплексорах? Реализуйте логический элемент **2И** с использованием мультиплексора **2в1**.

8. Какую функцию выполняет демультиплексор? Приведите пример параметризуемого демультиплексора на **Verilog**.
9. Какова особенность селектора и в чем его отличие от мультиплексора?
10. Для чего используется конструкция **generate**? Приведите пример.
11. Для чего применяется оператор частичного выбора (**part select**)? Приведите примеры.
12. Что такое распределенный селектор, и в каких задачах он может применяться? Приведите примеры.

Михаил Шуплецов, Александр Романов

Цифровой синтез: практический курс

**Глава 5. Сумматор, компаратор, устройство сдвига и АЛУ.
Повышение скорости арифметических операций**

Содержание

5.1	Арифметические комбинационные блоки: сумматор, компаратор, устройство сдвига и АЛУ	5-3
5.1.1	Сумматор	5-3
5.1.2	Сумматор с синхронизированными входами и выходами	5-10
5.1.3	Компаратор	5-16
5.1.4	Устройство сдвига	5-17
5.1.5	Арифметико-логическое устройство	5-20
5.2	Полусумматор, полный сумматор, перенос, каскадное соединение сумматоров	5-25
5.2.1	Полусумматор	5-26
5.2.2	Полный сумматор	5-27
5.2.3	Перенос, каскадное соединение сумматоров	5-28
5.3	Повышение скорости арифметических операций на примере сумматоров разных типов	5-32
5.3.1	Построение сумматора с ускоренным групповым переносом	5-33
5.3.2	Последовательное соединение сумматоров с ускоренным групповым переносом	5-41
5.3.3	Двухуровневая реализация сумматора с ускоренным групповым переносом	5-42
5.3.4	Префиксный сумматор	5-44
5.4	Упражнения	5-48
5.4.1	Основное задание	5-48
5.4.2	Задания для самостоятельной работы	5-48
5.4.3	Контрольные вопросы	5-49

Александр Телятников, Александр Романов

Цифровой синтез: практический курс

**Глава 6. Последовательная логика.
Счетчики и сдвиговые регистры**

Содержание

6.1	Последовательностные устройства	6-3
6.1.1	Модель Хаффмана для последовательностных устройств	6-4
6.2	Последовательностные присвоения	6-5
6.2.1	Блокирующие и неблокирующие присвоения	6-5
6.2.2	Порядок присвоений	6-6
6.2.3	Сравнение двух типов присвоения	6-7
6.3	Циклы симуляции	6-9
6.4	Порядок правильного использования блокирующих и неблокирующих присвоений	6-10
6.4.1	Защелки (latch), и почему их рекомендуется избегать	6-11
6.4.2	Комбинационная и последовательностная логика	6-13
6.5	Счетчики	6-14
6.5.1	Простой счетчик с асинхронным сбросом	6-15
6.5.2	Счетчик с предустановкой	6-18
6.5.3	Счетчик с управляемым направлением счета	6-19
6.5.4	Делитель частоты	6-19
6.5.5	Широтно-импульсная модуляция	6-20
6.5.6	Счетчик Грея	6-21
6.6	Сдвиговые регистры	6-22
6.6.1	Сдвиговый регистр с сигналом разрешения	6-23
6.6.2	Регистр сдвига с линейной обратной связью	6-24
6.6.3	Циклический избыточный код	6-25
6.7	Примеры организации взаимодействия цифровых систем с простыми периферийными модулями	6-26
6.7.1	Матрица светодиодов	6-26
6.7.2	Пьезоэлектрический излучатель	6-27
6.7.3	Ультразвуковой дальномер	6-28
6.8	Упражнения	6-28
6.8.1	Основное задание	6-28
6.8.2	Задания для самостоятельной работы	6-29
6.8.3	Контрольные вопросы	6-30
6.8.4	Приложение к главе	6-31

Сергей Иванец, Александр Романов

Цифровой синтез: практический курс

Глава 7. Память: регистровый файл и стек

Содержание

7.1	Полупроводниковые устройства памяти	7-3
7.2	Двухпортовая память	7-7
7.3	Иерархия памяти в компьютере. Память в микросхемах ПЛИС	7-12
7.4	Регистровый файл	7-14
7.5	Реализация ПЗУ с помощью таблицы перекодировки	7-19
7.6	Однопортовая память	7-22
7.7	Однопортовое ПЗУ	7-30
7.8	Двухпортовая память	7-35
7.9	Простой стек (LIFO)	7-49
7.9.1	Краткие сведения об организации стека	7-49
7.9.2	Реализация стека с помощью сдвигового регистра	7-50
7.9.3	Реализация стека с помощью сдвига указателя	7-55
7.10	Очередь (FIFO)	7-60
7.11	Встроенная память микросхем ASIC	7-61
7.12	Упражнения	7-63
7.12.1	Основное задание	7-63
7.12.2	Задания для самостоятельной работы	7-63
7.12.3	Контрольные вопросы	7-64
7.12.4	Темы для индивидуальной работы	7-65

Сергей Иванец, Александр Романов

Цифровой синтез: практический курс

Глава 8. Конечные автоматы: основы

Содержание

8.1	Конечные автоматы	8-3
8.2	Описание конечного автомата на языке Verilog	8-8
8.2.1	Описание выходной логики конечного автомата	8-11
8.3	Описание конечного автомата Мура на языке Verilog	8-13
8.3.1	Создание и компиляция проекта	8-15
8.3.2	Моделирование конечного автомата Мура в ModelSim	8-17
8.3.3	Автомат Мура с регистровыми выходами	8-19
8.3.4	Создание и компиляция проекта	8-22
8.4	Описание конечного автомата Мили на языке Verilog	8-23
8.4.1	Создание и компиляция проекта	8-25
8.4.2	Моделирование конечного автомата Мили в ModelSim	8-26
8.5	Упражнения	8-27
8.5.1	Основное задание	8-27
8.5.2	Задания для самостоятельной работы	8-33
8.5.3	Контрольные вопросы	8-39
8.5.4	Темы для индивидуальной работы	8-40

Станислав Жельнио, Александр Романов

Цифровой синтез: практический курс

**Глава 9. Использование конечных автоматов
для связи с периферийными устройствами**

Содержание

9.1	Конечные автоматы для связи с периферийными устройствами	9-3
9.1.1	Конечный автомат	9-3
9.1.2	Последовательный периферийный интерфейс SPI	9-4
9.2	Оборудование и исходный код	9-5
9.2.1	Датчик освещенности	9-5
9.2.2	Подключение периферийного устройства	9-7
9.2.3	Подключение датчика	9-7
9.2.4	Симуляционная модель периферийного устройства	9-8
9.2.5	Порядок проведения симуляции, синтеза и запуска проекта	9-10
9.3	Проектирование конечного автомата	9-10
9.3.1	Академический подход	9-10
9.3.2	Реализация конечного автомата	9-15
9.3.3	Задержка выходных сигналов автомата	9-22
9.3.4	Автомат на основе счетчика	9-24
9.4	Упражнения	9-29
9.4.1	Основное задание	9-29
9.4.2	Задания для самостоятельной работы	9-29
9.4.3	Контрольные вопросы	9-31

Юрий Панчул, Александр Антонов, Александр Романов

Цифровой синтез: практический курс

Глава 10. Конвейерная обработка данных

Содержание

10.1 Концепция конвейерной обработки как основы проектирования высокопроизводительных цифровых схем	10-3
10.1.1 Конвейерные идеализмы	10-5
10.1.2 Одинаковая задержка стадий	10-5
10.1.3 Унифицированные типы запросов	10-6
10.1.4 Независимость запросов	10-6
10.1.5 Планирование вычислительной нагрузки	10-6
10.2 Практическое сравнение комбинационной, многотактной и конвейерной реализаций арифметического блока	10-7
10.2.1 Функциональное моделирование	10-8
10.2.2 Комбинационная реализация	10-9
10.2.3 Многотактная реализация	10-13
10.2.4 Конвейерная реализация	10-14
10.2.5 Сравнение результатов	10-17
10.3 Дополнительные приемы эффективного проектирования конвейерных схем	10-19
10.3.1 Поведенческий и структурный стили кодирования	10-19
10.3.2 Реализация регулярных структур	10-21
10.3.3 Конструкция generate в рамках поведенческого стиля описания аппаратуры	10-23
10.3.4 Использование сигнала enable для разрешения работы модуля	10-25
10.3.5 Синхронизация стадий конвейера с переменными задержками	10-27
10.4 Упражнения	10-30
10.4.1 Основное задание	10-30
10.4.2 Задания для самостоятельной работы	10-30
10.4.3 Контрольные вопросы	10-36

Глава знакомит с концепцией конвейеризации в проектировании цифровых схем. Основной целью практической работы является знакомство с основами конвейеризации для улучшения характеристик цифровых схем по сравнению с комбинационной и неконвейеризированной последовательностной реализациями. Дается описание свойств вычислительной нагрузки и деталей организации цифровых схем, которые следует принимать во внимание для достижения максимальной пользы от внедрения конвейеризации. Также рассматривается вопрос обеспечения эффективности кодирования конвейерных структур, для чего демонстрируются: разница между поведенческим и структурным стилями; использование циклов и выражений **generate** для реализации регулярных схем; использование сигнала **enable** для управления запуском вычислений и уменьшения энергопотребления; синхронизация конвейера с операциями с переменной задержкой.

Требования к аппаратным и программным средствам

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

- персональный компьютер с установленной операционной системой Windows (виртуальная машина с ОС Windows не подойдет), x64, 8GB RAM, USB port;
- пакет **Quartus Prime** (есть студенческая версия);
- пакет **ModelSim Altera Edition** или программы **Icarus Verilog** и **GTKWave**;
- отладочная плата компании **Terasic DE10-Lite** или другая отладочная плата на основе **ПЛИС Intel FPGA** или **Xilinx** (может потребоваться миграция проектов, если она еще не сделана в дополнительных материалах¹ к данной книге).

Для запуска функциональных моделей требуется установленная среда программирования на языке **Python** с подключенными библиотеками **NumPy**, **SciPy**, **Matplotlib**.

10.1 Концепция конвейерной обработки как основы проектирования высокопроизводительных цифровых схем

Конвейеризация (pipelining) – это прием, который широко используется в проектировании цифровых схем с 1960-х гг. и позволяет в разы повысить пропускную способность цифровой схемы. При этом конвейеризация позволяет избежать кратного увеличения количества элементов цифровой логики. Данный подход полезен для проектирования вычислительных блоков различного назначения, включая процессоры, коммуникационные блоки и блоки подсистем памяти.

Основная идея конвейеризации заключается в применении следующих решений (рис. 10.1):

¹ <https://github.com/RomeoMe5/DDLM>.

- 1) тракт данных разбивается на последовательность стадий (ступеней конвейера) таким образом, что задержка прохождения данных в каждой из стадий меньше задержки всего исходного тракта данных;
- 2) запуск обработки новой порции данных (запроса), поступивших на вход конвейера, производится по мере освобождения первой стадии. Таким образом, обработка нового запроса начинается параллельно с продолжением обработки предыдущих запросов на более поздних стадиях конвейера.

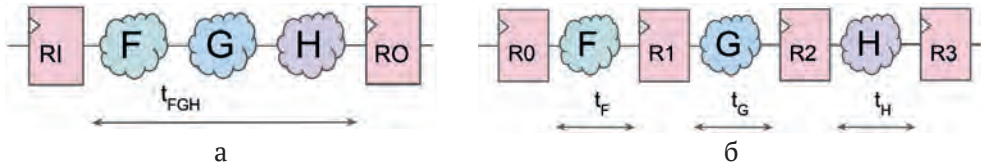


Рис. 10.1 Сравнение комбинационной (а) и конвейерной (б) реализаций цифровой схемы

Основой повышения производительности является распараллеливание обработки данных во времени: каждая стадия обрабатывает свой запрос параллельно с тем, как другие стадии обрабатывают другие запросы. Обе схемы могут принять к выполнению и завершить по одному запросу в течение одного цикла синхросигнала. Поскольку каждая из задержек t_F , t_G и t_H меньше общей задержки t_{FGH} (до трех раз), конвейерная схема может работать на более высокой тактовой частоте (до трех раз соответственно), что, в свою очередь, обеспечивает большую пропускную способность (тоже до трех раз). При этом с точки зрения объема цифровой логики конвейеризированная схема добавляет только буферные регистры для промежуточных результатов вычисления. Данные закономерности определяются следующими формулами:

$$BW_{peak(a)} = Fmax_a = \frac{1}{t_{FGH}},$$

$$BW_{peak(b)} = Fmax_b = \frac{1}{\max(t_F, t_G, t_H)},$$

$$\max(t_F, t_G, t_H) < t_{FGH} \Rightarrow BW_{peak(b)} > BW_{peak(a)},$$

где $BW_{peak(a)}$ и $BW_{peak(b)}$ – пиковая пропускная способность (в запросах за с); $Fmax_a$ и $Fmax_b$ – максимальная тактовая частота для схем (а) и (б) соответственно.

Конвейеризация широко применяется в современных цифровых аппаратных блоках. Примером такого блока является процессорное ядро **MIPS M5150** с конвейерной организацией (рис. 10.2). Данный конвейер обрабатывает следующие машинные инструкции: арифметические операции, операции с памятью, управление ветвлением и т. д. Подобные процессоры в настоящее время широко используются для встроенных приложений и приложений реального времени.

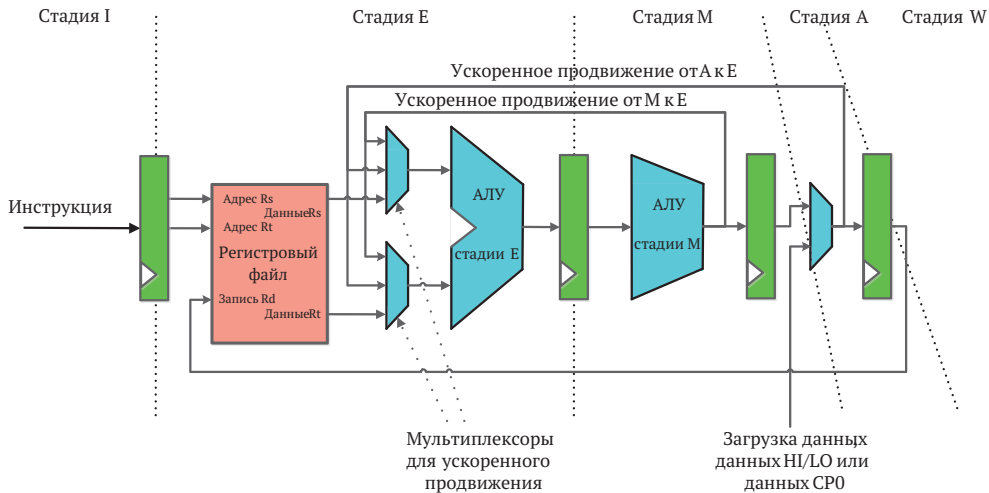


Рис. 10.2 Пример процессорного ядра с конвейерной организацией (MIPS M5150)

10.1.1 Конвейерные идеализмы

Эффективность применения конвейеризации ограничена определенными внутренними свойствами реализованных вычислений. К сожалению, не каждый вычислительный процесс может быть эффективно реализован в конвейерном стиле. Для максимизации пользы от применения конвейеризации разработчик должен адаптировать свою схему под эти свойства. В литературе¹ эти свойства носят название «**конвейерные идеализмы**». Выделяют три таких свойства:

- одинаковая задержка стадий;
- унифицированные типы запросов;
- независимость запросов.

Рассмотрим эти концепции более подробно.

10.1.2 Одинаковая задержка стадий

В идеальном случае задержка конвейера должна быть распределена по стадиям конвейера в равных пропорциях. В этом случае введение конвейера с k стадиями позволяет пропорционально повысить тактовую частоту и, соответственно, пропускную способность, в k раз (без учета временных затрат на предустановку и удержание сигналов на регистрах). Если задержка одной из стадий больше задержек остальных стадий, она становится **критической** для всего конвейера, т. е. ограничивает возможность повышения тактовой частоты конвейера в целом.

Неравенство задержек на стадиях конвейера называется **внутренней фрагментацией**. Устранение этого неравенства за счет перемещения вычислений между

¹ Shen J. P., Lipasti M. H. Modern processor design: fundamentals of superscalar processors. Waveland Press, 2013.

стадиями называется **балансировкой конвейера**. В том случае, если перемещение не имеет видимых побочных эффектов, инструменты синтеза могут выполнять его автоматически (этот процесс называется **retiming**). При этом отметим, что в общем случае правильное распределение вычислений по стадиям является обязанностью разработчика.

10.1.3 Унифицированные типы запросов

В простых конвейерных структурах тракт данных одинаков для всех типов запросов. В том случае, если определенный вычислительный ресурс какой-либо стадии не востребован для обработки какого-либо типа запросов (либо, в наихудшем случае, стадия конвейера для этих запросов сводится исключительно к копированию промежуточных результатов вычислений на другую стадию), поступление такого запроса приведет к простаиванию вычислительного ресурса и снижению потенциальной эффективности конвейера. Это простаивание называется **внешней фрагментацией**, и оно минимизируется разработчиками за счет унификации типов запросов и их обработки конвейерной структурой.

10.1.4 Независимость запросов

Поскольку конвейеризация подразумевает распараллеливание процесса обработки запросов, достижение пиковой эффективности возможно только в случае взаимной независимости запросов. Такая вычислительная нагрузка носит название **поточковой** – отдельные запросы в ней не имеют общих данных. В случае невозможности обеспечения независимости запросов может потребоваться введение дополнительных блокировок, приостанавливающих вычисления до разрешения зависимостей. Например, для выполнения одного запроса потребуются ожидание результатов выполнения другого.

В рамках данной практической работы все примеры основаны на потоковых арифметических вычислениях, поэтому такая проблема не возникает. При этом следует отметить, что более сложные блоки (например, современные высокопроизводительные процессоры) реализуют агрессивное распараллеливание обработки инструкций, сохраняя для программиста видимость последовательного выполнения инструкций. Это ведет к появлению отдельного класса проблем (таких как, например, **конвейерные конфликты**, вызванные конвейерной обработкой последовательных инструкций). Более подробно данные вопросы освещаются при изучении примера реализации микроархитектуры простого MIPS-процессора в [главе 11](#).

10.1.5 Планирование вычислительной нагрузки

Потенциальная эффективность конвейеризированной цифровой схемы ограничена степенью присутствия конвейерных идеализмов. В случае если достичь заданных параметров не удастся, решением может стать изменение самой вычислительной нагрузки с целью большего ее согласования с конвейерными идеализмами и ее лучшей адаптации к конвейерной реализации. Например, при проектировании системы команд процессора следует иметь в виду, что миними-

зация взаимосвязей между ними и унификация их обработки сделает аппаратуру эффективнее, а ее проектирование – менее трудоемким.

10.2 Практическое сравнение комбинационной, многотактной и конвейерной реализаций арифметического блока

В реальных цифровых схемах комбинационный, многотактный и конвейерный подходы применяются совместно. Выбор конкретного решения зависит от множества факторов: свойств вычислительной нагрузки, требований к тактовой частоте, характеристик компонентов технологической библиотеки и стратегии оптимизации. В данной главе будет проводиться сравнение следующих характеристик:

- 1) максимальной тактовой частоты;
- 2) максимальной пропускной способности в форме интервала инициации (минимального количества тактов синхросигнала между запросами) и количества запросов, обрабатываемых за секунду;
- 3) минимальной задержки обработки запроса (в тактах синхросигнала и наносекундах);
- 4) потребления ресурсов (логических ресурсов, регистров, арифметических блоков);
- 5) потребления энергии.

В данной главе проектирование конвейерных структур рассматривается на примере арифметических блоков. Эти блоки имеют единственный тип запроса, поскольку один и тот же алгоритм применяется к каждому входному набору данных. Кроме того, запросы обрабатываются независимо. Такие блоки могут быть особенно эффективно реализованы в конвейерном стиле: второй и третий идеализмы удовлетворяются автоматически ([раздел 10.1.1](#)). Первый же идеализм все еще остается актуальным.

В качестве демонстрационного примера рассмотрим 8-разрядный блок возведения числа в **пятью** степень. Поскольку алгоритм предполагает повторение одной и той же операции (умножения), он может быть естественным путем разбит на стадии с одинаковой задержкой. Таким образом, первый идеализм также легко удовлетворяется, если количество операций является кратным количеству стадий конвейера. Как результат данный пример в целом демонстрирует идеализированную ситуацию, при которой конвейеризация дает наибольший прирост эффективности схемы. Проектирование реальной аппаратуры, как правило, существенно сложнее и может потребовать от разработчика дополнительного анализа и принятия компромиссных решений.

10.2.1 Функциональное моделирование

Перед проектированием непосредственно RTL-модели цифрового блока полезно создать его **эталонную модель (Golden model)**. **Эталонная модель** предоставляет набор выходных реакций на тестовые векторы для верификации функциональности проектируемой аппаратуры. Данная модель будет общей для различных реализаций (комбинационной, многотактной, конвейеризированной) аппаратного блока. Такие модели также называют функциональными. В этой практической работе для описания функциональной модели используется язык программирования **Python**. Исходный код функциональной модели представлен ниже ([листинг 10.1](#)). Следует обратить внимание, что модель аппаратуры должна включать в себя исключительно «синтезируемые» операции (те, которые могут быть реализованы аппаратно). Корректность самой модели верифицируется с помощью встроенной математики Python.

```
# alg model - python math can be used here
def pow5_alg(x):
    return pow(x, 5) & 0xff

# hw model - synthesizable operations only
def pow5_hw(x):
    y = 1;
    for mul in range(0, 5):
        y = (y * x) & 0xff
    return y

# generating stimulus
for x in range(0, 10):
    alg_val = pow5_alg(x)
    hw_val = pow5_hw(x)
    if (alg_val == hw_val):
        print("Correct! x: ", hex(x).ljust(6), "; y: ", hex(hw_val).ljust(6))
    else:
        print("ERROR! x: ", hex(x).ljust(6), "; y(model): ", hex(alg_val).
              ljust(6), "; y(hw): ", hex(hw_val).ljust(6))
```

Листинг 10.1 Функциональная модель модуля возведения числа в степень

Тестовый вывод модели:

```
Correct! x: 0x0 ; y: 0x0
Correct! x: 0x1 ; y: 0x1
Correct! x: 0x2 ; y: 0x20
Correct! x: 0x3 ; y: 0xf3
Correct! x: 0x4 ; y: 0x0
Correct! x: 0x5 ; y: 0x35
Correct! x: 0x6 ; y: 0x60
Correct! x: 0x7 ; y: 0xa7
Correct! x: 0x8 ; y: 0x0
```

Correct! x: 0x9 ; y: 0xa9

Листинг 10.2 Тестовый вывод функциональной модели модуля возведения числа в степень

10.2.2 Комбинационная реализация

Комбинационная реализация – это простейшая реализация аппаратного блока обработки. С одной стороны, по сравнению с конвейерной реализацией, комбинационная реализация имеет меньшую задержку обработки запроса благодаря отсутствию внутренней фрагментации и необходимости буферизировать промежуточные результаты вычислений. Но в случае поступления нескольких различных типов запросов может возникнуть проблема внешней фрагментации. Тогда критическая задержка будет оставаться постоянной для всех типов запросов. Аналогично конвейерной реализации, комбинационная реализация требует отдельных вычислительных ресурсов для каждой операции и функционирует на относительно низкой тактовой частоте. Технически такие схемы могут работать в высокочастотном домене в виде так называемых **многотактных путей** (когда итерация вычислений производится раз в несколько тактов), но из-за необходимости ожидать эти несколько тактов пропускная способность всего тракта данных остается низкой.

Ниже приведен пример комбинационного модуля, вычисляющего пятую степень числа (файл `lab_10_1_pow_5\02_syn_pow_5_single_cycle_struct`):

```
module pow_5_single_cycle_struct
# (
    parameter w = 8
)
(
    input clk,
    input rst_n,
    input arg_vld,
    input [w - 1:0] arg,
    output res_vld,
    output [w - 1:0] res
);

    wire arg_vld_q;
    wire [w - 1:0] arg_q;
    reg_rst_n i_arg_vld (clk, rst_n, arg_vld, arg_vld_q);
    reg_no_rst # (w) i_arg (clk, arg, arg_q);
    wire res_vld_d = arg_vld_q;
    wire [w - 1:0] res_d = arg_q * arg_q * arg_q * arg_q * arg_q;
    reg_rst_n i_res_vld (clk, rst_n, res_vld_d, res_vld);
    reg_no_rst # (w) i_res (clk, res_d, res);
endmodule
```

Листинг 10.3 Исходный код комбинационной реализации модуля возведения числа в степень

Для моделирования схемы в среде **Modelsim** и получения временных диаграмм необходимо выполнить следующие действия:

- 1) запустить скрипт моделирования, который произведет моделирование всех модулей в проекте: **lab_10\src\lab_10_1_pow_5\01_sim_pow_5\ 01_simulate_with_modelsim.bat**;
- 2) открыть вкладку **Wave**. Результат должен быть таким же, как и на временных диаграммах:

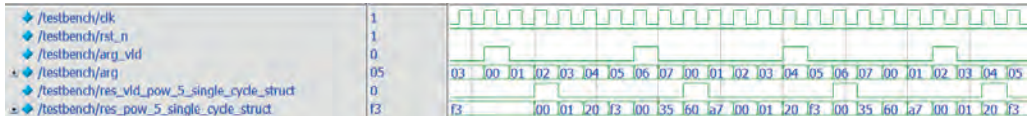


Рис. 10.3 Результаты моделирования комбинационной реализации модуля возведения числа в степень

Следует обратить внимание, что в результате моделирования получен вектор входных и выходных пар значений (**0x00/0x00, 0x06/0x60, 0x04/0x00, 0x02/0x20** и т. д.), совпадающих с выходами функциональной модели ([раздел 10.2.1](#)). Это подтверждает корректность работы спроектированного блока.

Для сборки проекта и его прототипирования на ПЛИС необходимо выполнить следующие действия:

- 1) запустить скрипт **make_project.bat** для создания папки для синтеза и для хранения файлов проекта: **lab_10\src\lab_10_1_pow_5\02_syn_pow_5_single_cycle_struct\ de10_lite\make_project.bat**;
- 2) открыть файл проекта в среде **Quartus Prime**: **lab_10\src\lab_10_1_pow_5\ 02_syn_pow_5_single_cycle_struct\ de10_lite\project\de10_lite.qpf**;
- 3) запустить сборку проекта (**Processing → Start Compilation**);
- 4) после завершения компиляции можно открыть просмотр RTL (**Tools → Netlist Viewers → RTL Viewer**) и оценить количество функциональных блоков и их схему их соединения. RTL-представление комбинационной реализации показано на [рис. 10.4](#);
- 5) сгенерировать файл прошивки (**bitstream**) и загрузить его в плату с ПЛИС.

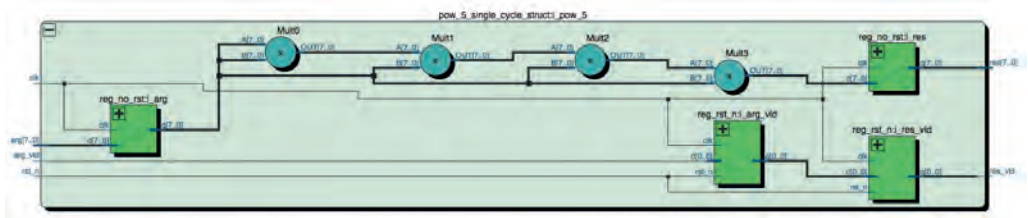


Рис. 10.4 RTL-представление комбинационной реализации модуля возведения числа в степень

Для выполнения прототипирования на плате с ПЛИС необходимо перевести переключатель **SW[9]** в активное положение, а затем вернуть в выключенное положение. Это перезапустит работу системы. Затем требуется задать входное значение переключателями **SW[7:0]** и нажать **KEY[0]**, для того чтобы отправить его на обработку. Индикация на светодиодах обозначит момент готовности выходных данных. Выходной байт данных будет выведен на два последних семисегментных индикатора.

Для оценки максимальной тактовой частоты **Fmax** (рис. 10.5) требуется запустить **TimeQuest Timing Analyzer → Slow 1200mV 85C Model → Fmax Summary**.

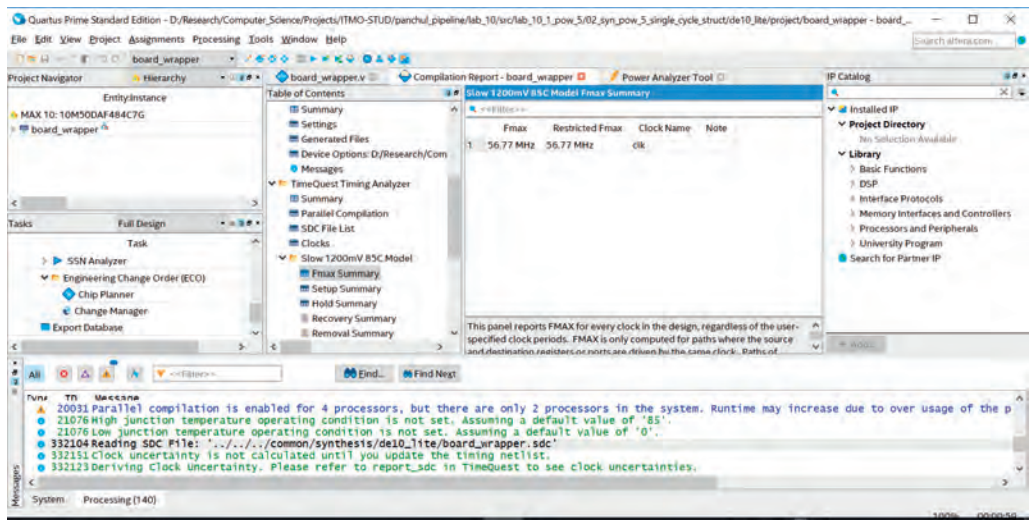


Рис. 10.5 Максимальная тактовая частота для комбинационной реализации модуля возведения числа в степень

В получившемся примере максимальная тактовая частота составляет **56,77 МГц**. Интервал инициации комбинационной реализации занимает один такт. Соответственно, пропускная способность для вычисленной частоты составляет **56,77 млн запросов в секунду**.

Задержка комбинационной реализации также равна одному такту. Соответственно, задержка для заданной тактовой частоты составляет **1/Fmax (17,61 нс)**.

Для оценки потребления ресурсов кристалла (рис. 10.6) необходимо открыть раздел **Flow Summary**:

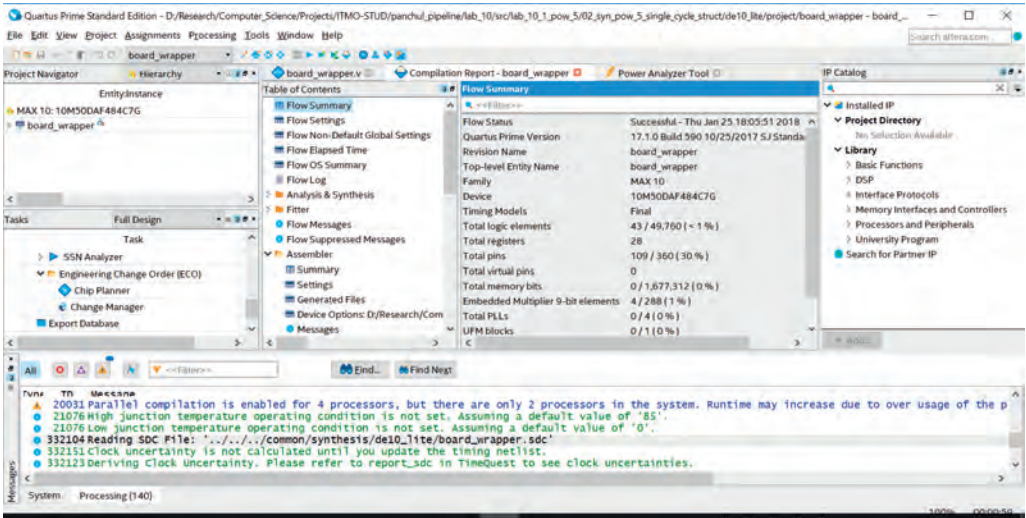


Рис. 10.6 Оценка потребления ресурсов ПЛИС

Для приведенного примера получены следующие оценки потребления ресурсов: 43 логических элемента, 28 регистров, 4 умножителя. Зафиксируем данные значения для последующего сравнения.

Для оценки энергопотребления (рис. 10.7) требуется запустить меню **Processing** → **Power Analyzer Tool**. Здесь можно выставить произвольную частоту переключения ввода-вывода (**I/O toggle rate**). Оставив значение по умолчанию – 12,5 %, – нужно нажать **Start**. По завершении анализа нажать **Report**.

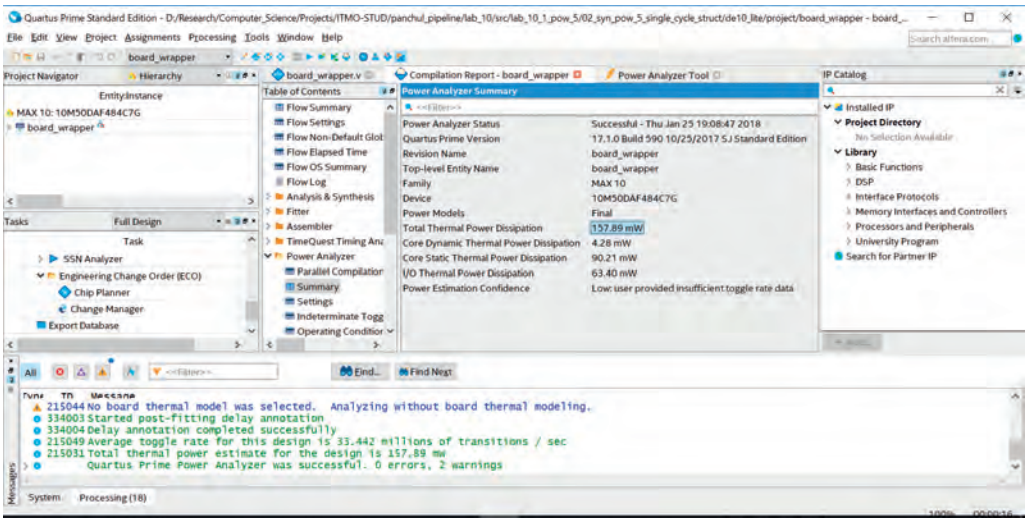


Рис. 10.7 Оценка энергопотребления

Для приведенного примера получены следующие оценки энергопотребления: **90,21 мВт** (статическое) и **4,28 мВт** (динамическое). Зафиксируем данные значения для последующего сравнения.

Сбор данных о разработанном модуле завершен.

10.2.3 Многотактная реализация

Многотактная неконвейеризированная реализация, по сравнению с комбинационной и конвейерной реализациями, позволяет сэкономить аппаратные ресурсы за счет их повторного использования на различных этапах обработки запроса. Здесь, как и в конвейерной реализации, требуется буферизация для хранения результатов промежуточных вычислений. Кроме того, координация передачи данных между вычислительными устройствами требует добавления дополнительной логики управления.

По сравнению с конвейерной реализацией, в многотактной внешняя фрагментация приводит к меньшим потерям эффективности (неиспользуемые стадии вычислений могут быть пропущены), при этом проблема внутренней фрагментации все равно сохраняется. В случае нескольких различных типов запросов пропускная способность многотактной реализации, как правило, выше, чем у комбинационной, но хуже, чем у конвейерной. Это объясняется тем, что запуск обработки нового запроса не начинается, пока не завершится предыдущий.

Исходный код примера многотактной реализации модуля возведения числа в пятую степень приведен ниже:

```
module pow_5_multi_cycle_struct
# ( parameter w = 8 )
(
    input clk,
    input rst_n,
    input arg_vld,
    input [w - 1:0] arg,
    output res_vld,
    output [w - 1:0] res
);
    wire arg_vld_q;
    wire [w - 1:0] arg_q;

    reg_rst_n i_arg_vld (clk, rst_n, arg_vld, arg_vld_q);
    reg_no_rst_en # (w) i_arg (clk, arg_vld, arg, arg_q);

    wire [3:0] shift_q;
    wire [3:0] shift_d = arg_vld_q ? 4'b1000 : shift_q >> 1;
    reg_rst_n # (4) i_shift (clk, rst_n, shift_d, shift_q);
    assign res_vld = shift_q [0];
    wire [w - 1:0] mul_q;
```

```

wire [w - 1:0] mul_d = (arg_vld_q ? arg_q : mul_q) * arg_q;
wire mul_en = arg_vld_q || shift_q [3:1] != 3'b0;
reg_no_rst_en # (w) i_mul (clk, mul_en, mul_d, mul_q);
assign res = mul_q;
endmodule

```

Листинг 10.4 Исходный код многотактной реализации модуля возведения числа в степень

Результаты моделирования многотактной реализации приведены ниже:



Рис. 10.8 Результаты моделирования многотактной реализации модуля возведения числа в степень

Следует обратить внимание на то, что пары входных и выходных значений идентичны для функциональной модели и комбинационной реализации, но поскольку цикл вычисления теперь занимает четыре такта, задержка формирования результата также составляет четыре такта.

RTL-представление многотактной реализации следующее:

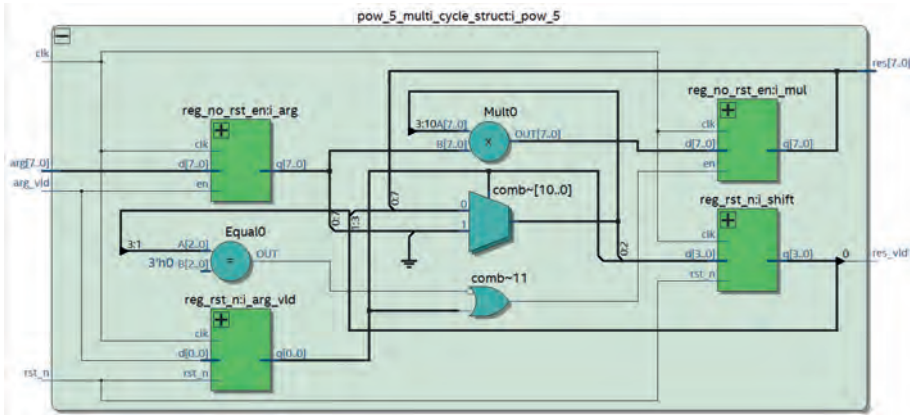


Рис. 10.9 RTL-представление многотактной реализации

Ход сбора данных аналогичен комбинационной реализации. Для этого необходимо повторить шаги, описанные в предыдущем разделе, для многотактной реализации. Проект расположен в дополнительных материалах¹ к данной главе в каталоге `lab_10\src\ lab_10_1_pow_5\ 05_syn_pow_5_multi_cycle_struct`.

10.2.4 Конвейерная реализация

В данном разделе описан пример конвейерной реализации проекта. Конвейерная реализация обеспечивает более высокую пропускную способность, требуя

¹ <https://github.com/RomeoMe5/DDLM>.

при этом больше аппаратных ресурсов (отдельные вычислительные ресурсы для каждой операции аналогично комбинационной реализации и аналогично много-тактной реализации, дополнительные ресурсы на буферизацию). Также конвейерная реализация может иметь увеличенную задержку обработки запроса из-за задержек на буферизацию и (при наличии) внутренней фрагментации.

В соответствии с первым конвейерным идеализмом задержку всей обработки целесообразно разбивать на одинаковые части. Это снижает внутреннюю фрагментацию и повышает максимальную тактовую частоту работы блока. Для рассматриваемого примера применен простой подход, который состоит в том, что алгоритм четырехкратного умножения разбит на четыре равные части. Таким образом, конвейер состоит из четырех стадий (по одному умножению на стадию).

Исходный код конвейерной реализации модуля возведения числа в **пятую** степень приведен ниже:

```
module pow_5_pipe_struct
# (
    parameter w = 8
)
(
    input clk,
    input rst_n,
    input arg_vld,
    input [ w - 1:0] arg,
    output [ 4:0] res_vld,
    output [5 * w - 1:0] res
);
    wire arg_vld_q_1;
    wire [w - 1:0] arg_q_1;

    reg_rst_n i0_arg_vld (clk, rst_n, arg_vld, arg_vld_q_1);
    reg_no_rst # (w) i0_arg (clk, arg, arg_q_1);

    assign res_vld [4 ] = arg_vld_q_1;
    assign res [4 * w +: w] = arg_q_1;

//-----
    wire [w - 1:0] mul_d_1 = arg_q_1 * arg_q_1;
    wire arg_vld_q_2;
    wire [w - 1:0] arg_q_2;
    wire [w - 1:0] mul_q_2;

    reg_rst_n i1_arg_vld ( clk , rst_n , arg_vld_q_1 , arg_vld_q_2 );
    reg_no_rst # (w) i1_arg ( clk , arg_q_1 , arg_q_2 );
    reg_no_rst # (w) i1_mul ( clk , mul_d_1 , mul_q_2 );

    assign res_vld [3 ] = arg_vld_q_2;
    assign res [3 * w +: w] = mul_q_2;
```



```

//-----
wire [w - 1:0] mul_d_2 = mul_q_2 * arg_q_2;
wire arg_vld_q_3;
wire [w - 1:0] arg_q_3;
wire [w - 1:0] mul_q_3;

reg_rst_n i2_arg_vld ( clk , rst_n , arg_vld_q_2 , arg_vld_q_3 );
reg_no_rst # (w) i2_arg ( clk , arg_q_2 , arg_q_3 );
reg_no_rst # (w) i2_mul ( clk , mul_d_2 , mul_q_3 );

assign res_vld [2 ] = arg_vld_q_3;
assign res [2 * w +: w] = mul_q_3;

//-----
wire [w - 1:0] mul_d_3 = mul_q_3 * arg_q_3;
wire arg_vld_q_4;
wire [w - 1:0] arg_q_4;
wire [w - 1:0] mul_q_4;

reg_rst_n i3_arg_vld ( clk , rst_n , arg_vld_q_3 , arg_vld_q_4 );
reg_no_rst # (w) i3_arg ( clk , arg_q_3 , arg_q_4 );
reg_no_rst # (w) i3_mul ( clk , mul_d_3 , mul_q_4 );

assign res_vld [1 ] = arg_vld_q_4;
assign res [1 * w +: w] = mul_q_4;

//-----
wire [w - 1:0] mul_d_4 = mul_q_4 * arg_q_4;
wire arg_vld_q_5;
wire [w - 1:0] arg_q_5;
wire [w - 1:0] mul_q_5;

reg_rst_n i4_arg_vld ( clk , rst_n , arg_vld_q_4 , arg_vld_q_5 );
reg_no_rst # (w) i4_arg ( clk , arg_q_4 , arg_q_5 );
reg_no_rst # (w) i4_mul ( clk , mul_d_4 , mul_q_5 );

assign res_vld [0 ] = arg_vld_q_5;
assign res [0 * w +: w] = mul_q_5;
endmodule

```

Листинг 10.5 Исходный код конвейерной реализации модуля
возведения числа в степень

Результаты моделирования конвейерной реализации следующие:



Рис. 10.10 Результаты моделирования конвейерной реализации

Станислав Жельнио, Александр Романов

Цифровой синтез: практический курс

Глава 11. Софт-процессор: основы микроархитектуры

Содержание

11.1 Теоретические основы процессорного ядра schoolMIPS	11-4
11.1.1 Микроархитектура	11-4
11.1.2 Проектирование. Тракт данных	11-5
11.1.3 Проектирование. Устройство управления	11-14
11.1.4 Реализация. Тракт данных	11-20
11.1.5 Реализация. Устройство управления	11-26
11.2 Использование процессорного ядра schoolMIPS на практике	11-27
11.2.1 Элементы управления и индикации	11-28
11.2.2 Сборка и запуск проекта	11-29
11.2.3 Структура каталога программы	11-31
11.2.4 Порядок запуска программы	11-32
11.3 Упражнения	11-38
11.3.1 Основное задание	11-38
11.3.2 Задания для самостоятельной работы	11-38
11.3.3 Контрольные вопросы	11-39

Чарльз Данчек, Александр Романов

Цифровой синтез: практический курс

**Приложение А. Путь вперед: от устройств на базе
FPGA к массовому рынку ASIC для популярных
гаджетов**

Содержание

A.1	Прототипирование на FPGA как один из этапов проектирования ASIC	A-3
A.1.1	Стратегия прототипирования	A-3
A.1.2	Предостережения относительно прототипирования	A-5
A.2	Последовательность этапов проектирования ASIC	A-7
A.2.1	Схема разработки ASIC	A-8
A.2.2	Усовершенствование проекта (design refinement)	A-10
A.3	Этап логического синтеза	A-11
A.3.1	Переработка RTL-кода	A-12
A.3.2	Настройка библиотеки	A-15
A.3.3	Ограничения по проектированию (design constraints)	A-17
A.3.4	Синтез иерархии проекта	A-19
A.3.5	Оценка результатов компиляции	A-21
A.3.6	Список соединений на Verilog, передаваемый на уровень физического синтеза (hand-off Verilog netlist)	A-23
A.4	Этап физического синтеза	A-24
A.4.1	Настройка библиотек	A-25
A.4.2	Разработки плана размещения	A-29
A.4.3	Обеспечение низкого энергопотребления	A-33
A.4.4	Этап размещения (placement)	A-34
A.4.5	Тактовое дерево (clock tree)	A-37
A.4.6	Маршрутизация (routing)	A-40
A.4.7	Маршрутизация в соответствии с указаниями о технических изменениях (ECO)	A-48
A.4.8	Доводка кристалла (chip finishing)	A-50
A.5	Этап передачи в производство	A-53
A.5.1	Проверка временных параметров	A-54
A.5.2	Проверка соответствия проектным нормам (DRC/ERC check)	A-56
A.5.3	Проверка соответствия топологии схеме (LVS)	A-60
A.5.4	Геометрическая модель в формате GDSII (stream out GDSII)	A-62
A.6	Производство ASIC	A-63
A.7	Заключение	A-64

*Данное приложение является дорожной картой, в которой поэтапно описывается процесс разработки чипов **ASIC**. Его цель – дать инженерам, прошедшим обучение проектированию на **FPGA**, возможность с успехом применять и развивать свои навыки по разработке чипов **ASIC** для конкретных применений.*

A.1 Прототипирование на FPGA как один из этапов проектирования ASIC

Чипы **ASIC** адаптированы к конкретному приложению – например, как элемент цифровой камеры, мобильного телефона или **GPS**-трекера. В отличие от **FPGA**, чип **ASIC** не имеет накладных логических или проводных ресурсов. Проекты **ASIC** начинаются с голой кремниевой матрицы, на которой размещают только логику и память, необходимые для приложения. Такие микросхемы для конкретных приложений работают быстрее, потребляют меньше энергии и имеют меньшую площадь, чем соответствующие устройства на **FPGA**. Этим обеспечиваются главные преимущества при производстве гаджетов массового спроса для потребительского рынка, а также других конкурентоспособных по стоимости коммерческих продуктов.

Замечание 1: согласно обширному компаративному исследованию университета Торонто, при сравнении **90 нм FPGA** с **ASIC** оказалось, что при миграции с устройства **FPGA** на чип **ASIC** в среднем обеспечивается уменьшение площади в **21** раз, уменьшение задержки в критических путях до **2,1** раза и динамическое снижение мощности в **9** раз. Точные результаты варьируются в зависимости от технологии проектирования, скорости, а также смешивания логических элементов, памяти и блоков **DSP**. Тем не менее можно сделать вывод, что в целом реализация **ASIC** имеет значительные преимущества в скорости, по занимаемой площади и потреблению энергии. Следует также учитывать и экономические факторы. **FPGA** характеризуются относительно высокой стоимостью каждого устройства, но нулевыми затратами на проектирование. **ASIC**, как объясняется в следующем разделе, имеют значительную одноразовую стоимость проектирования, но более низкую цену отдельного устройства. Для больших объемов производства привлекательна низкая себестоимость на единицу продукции с более высокой скоростью и меньшим энергопотреблением.

A.1.1 Стратегия прототипирования

Успешные компании-разработчики чипов часто создают новый продукт, начиная с создания устройства на базе **FPGA**. Этот ранний прототип служит удобной платформой для проверки функционирования оборудования, а также корректности работы соответствующей программы и прикладного программного обеспечения в целевой системе. Проверка реального аппаратного прототипа устройства более реалистична, чем использование программного обеспечения для имитационного моделирования. Хотя прототип на базе **FPGA** работает не так быстро, как целевой **ASIC**, он все же намного быстрее, чем симуляция. Вместо того чтобы ожидать завершения прогона симуляции (он может занять не одну неделю), команда

разработчиков может запустить прототип и начать исправлять ошибки уже через несколько часов.

Применение раннего прототипирования также уменьшает время выхода на рынок нового гаджета. Компания, которая своевременно выпускает новый продукт для критической ниши рынка (даже если это всего лишь ранний прототип), может занять на нем значительную долю. Затем компания может продолжить длительный цикл разработки **ASIC**, превращая проверенное устройство **FPGA** в высокопроизводительный чип **ASIC**. Прототип является также удобным средством для кодирования и тонкой настройки встроенного программного обеспечения или кода, которые тесно взаимодействуют с оборудованием.

Ранние прототипы также снижают затраты в долгосрочной перспективе. Миниатюризация технологий приводит к тому, что затраты на проектирование для изготовления **ASIC** быстро растут. В стоимости проектирования львиную долю затрат составляют единовременные издержки на создание набора от **30** до **50 CMOS**-масок. Для **90 нм ASIC** только этот набор масок может стоить до **1** млн долларов США. И эта стоимость продолжает расти с каждым технологическим шагом. При **32 нм** набор масок может уже стоить **3** млн долларов США.

При отладке прототипа **FPGA** функциональные ошибки обнаруживаются на ранней стадии. В этом случае **ASIC** с большей вероятностью будет работать уже после первого цикла изготовления. Это позволяет избежать дорогостоящего повторного цикла, связанного с изменением одной или нескольких **CMOS**-масок.

Эта стратегия прототипирования **FPGA** настолько распространена, что упоминается даже в объявлениях о вакансиях для инженеров-конструкторов. Типичное описание должностных требований, недавно опубликованное для большой компании в Кремниевой долине, выглядело примерно так:

Job Requirments:

- Write RTL code for on chip logic modules, based on microarchitecture specs.
- Perform early chip *prototyping*, using an Altera or Xilinx FPGA device.
- Synthesize all modules, meet speed and power goals, and reach timing closure.
- Support the Design Verification (DV) team for functional verification of modules.
- Hand off and integrate all logic modules into the target ASIC environment.

Рис. А.1 Типичное объявление о работе в Кремниевой долине

Должностные требования:

1. Разработка **RTL**-кода для логических модулей на чипе на основе спецификаций микроархитектуры.
2. Выполнение раннего прототипирования микросхем с использованием **FPGA Altera** или **Xilinx**.

3. Синтез модулей с соблюдением требований по скорости, энергопотреблению, а также временным ограничениям.
4. Поддержка команды верификации для функциональной проверки модулей.
5. Трансфер и интеграция логических модулей в целевую среду **ASIC**.

Хотя прототипирование **FPGA** является проверенной стратегией, в нем есть свои риски. Как бы то ни было, следует избегать наихудшего сценария, при котором прототипирование **FPGA** все еще находится на стадии устранения проблем проектирования, в то время как первые корпусированные чипы **ASIC** уже поступают с завода. Чтобы избежать проскальзывания графика, необходимо понимать риски.

А.1.2 Предостережения относительно прототипирования

Потенциальные пользователи прототипа **FPGA** стремятся получить его как можно скорее. В частности, команда разработчиков программного обеспечения может захотеть запустить тяжелое программное обеспечение (например, встроенный **Linux**) на прототипе, чтобы как можно раньше идентифицировать любые функциональные ошибки. Но прототипирование, начатое слишком рано (или включающее слишком много аппаратных средств), может привести к ряду проблем, на которых акцентировано внимание ниже.

Дождитесь зрелого RTL-кода

Прототипирование занимает значительное время, требует наличия умений и разработки отладочной платы. Эти усилия не должны быть потрачены впустую в попытках отладить незрелый, нестабильный или неполный **RTL**-код. Эффективнее выполнять отладку раннего **RTL**-кода, используя традиционные методы моделирования. **RTL**-код готов к прототипированию только тогда, когда отдельные логические модули проверены на уровне блоков, объединены вместе и способны передавать все тестовые векторы в базовом наборе тестбенчей. Следует дождаться зрелого кода, даже если разработчики программного обеспечения хотят получить прототип пораньше.

Не размещайте все модули

Некоторые проекты **ASIC** легче размещаются на устройствах **FPGA**. В качестве простого примера можно указать на то, что проект с высоким соотношением триггеров и логических элементов может легче уместиться на одном **FPGA**, архитектура которого, как правило, содержит много триггеров.

Не каждый аппаратный модуль на целевой **ASIC** всегда может отображаться на платформе **FPGA**; некоторые из них лучше всего прототипируются с использованием аппаратного обеспечения, внешнего по отношению к **FPGA**. Одним из примеров является встроенный макрос **RF CMOS**, который реализует малошумный усилитель или квадратурный модулятор, используемый в беспроводной связи. Другим примером служат физические (топологические) **IP**-ядра без **RTL**-кода или списка соединений для **FPGA**, такие как, например, блок канального уров-

ня (**PHY**) внутри трансивера **Ethernet**, являющийся схемой самого низкого уровня, которая управляет сериализованными данными на витой паре или оптоволоконном кабеле.

Подобные модули требуют дополнительных аппаратных устройств (коммерческий чип или отладочная плата от поставщика оборудования). Задача сопряжения этого внешнего оборудования с модулями ввода-вывода **FPGA** должна выполняться с осторожностью, поскольку **FPGA** содержат настраиваемые буферы ввода-вывода, позволяющие управлять уровнями напряжения и тока на пинах чипа.

Некоторые **ASIC**-блоки, если у них имеются кольца контактных площадок ввода-вывода, нельзя прототипировать на **FPGA**. Например, **ПО Quartus Prime** от **Intel FPGA (Altera)** по умолчанию обнаруживает только встроенные входы/выходы и не воспринимает инструкций по инстанцированию для ячеек ввода-вывода, имеющих в библиотеке **ASIC**. Это также относится и к ячейкам библиотеки **ASIC**, созданным в **RTL**-коде, например к регистру, предназначенному для синхронизации с асинхронным источником данных.

Логика, предназначенная только для тестирования изготовленной и корпусированной **ASIC** на наличие производственных дефектов, также должна быть исключена из прототипа, поскольку он предназначен лишь для функционального анализа **RTL**-кода и встроенного программного обеспечения. Существуют более эффективные методы для проверки логики тестирования **ASIC** и для изготовления тестовых шаблонов с использованием логического моделирования.

Минимизируйте фрагментацию проекта

Предположим, что платформой для **FPGA**-прототипирования является чип **Intel FPGA MAX10 (10M50DAF484C7G)**. Он содержит **50** тыс. логических элементов (**LE**). **LE** – самый маленький модуль логики в этом семействе устройств. Каждый **LE** имеет четыре основных входа данных, достаточную логику для реализации любой функции этих входов и одного выходного триггера. Таким образом, один **LE** может выполнять работу десяти стандартных ячеек на целевой **ASIC**. Емкость чипа рассматриваемого **FPGA** эквивалентна приблизительно **500 000** логическим вентилям **ASIC**. Если целевой **ASIC** требуется **1** млн логических элементов, то для прототипирования понадобится два устройства **MAX10** (при предполагаемом полном использовании чипов, которое достигается редко). Это означает, что модули, описанные в **RTL**-коде, требуется распределить на два (или более) устройства. Каждое устройство должно иметь достаточное количество ресурсов (выводов, буферов тактовых сигналов, блоков **RAM**) для выполнения возложенных на него функций. Связь между двумя или более устройствами, имеющими много интерфейсов, может быть узким местом; следовательно, для решения задачи разбиения требуются значительные усилия, а полученный прототип может работать медленнее, чем хотелось бы.

Если весь проект осуществляется в рамках одной **FPGA**, задача прототипирования становится значительно проще.

Замечание 2: типичное использование ресурсов **FPGA** при проектировании составляет **75 %**. Для прототипирования лучше не превышать уровень использования ресурсов более **50–60 %**. Это оставляет место для изменений, облегчает задачу размещения и маршрутизации, а также сокращает время запуска проекта.

Поставщики инструментов **EDA** предлагают высококачественные, готовые к запуску прототипы **FPGA**, известные как эмуляторы. Эмулятор **Cadence** – это **Palladium**. Эмулятор для **ПОО Synopsys** под названием **ZeBu** использует коммерческие **FPGA Xilinx**. **Mentor Graphics** предлагает применять эмулятор **Veloce**, который позволяет обрабатывать проекты с миллиардами логических элементов без разбиения на части, но он предназначен только для верификации и отладки проектов.

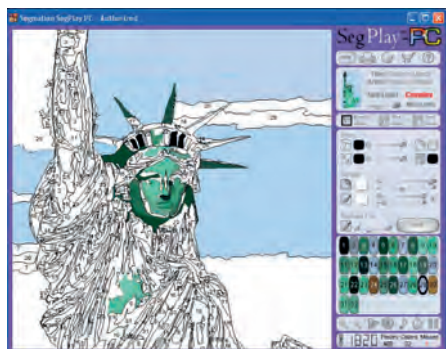
Поскольку эмуляция может сократить срок внедрения **ASIC**, часто нет необходимости создавать ранний прототип на **FPGA**.

А.2 Последовательность этапов проектирования ASIC

Как было указано ранее, устройство **FPGA** (например, **MAX10**) собирают из готовых элементов. В устройстве содержится до 50 тысяч логических элементов (**LE**); каждый – со своим триггером, а также имеется много килобайтов оперативной и флеш-памяти, множителей **18×18** и аналого-цифровых преобразователей (**ADC**). Необходимые ресурсы маршрутизации для соединения этих элементов встроены в устройство. Благодаря этому устройство может быть легко сконфигурировано для использования требуемых ресурсов на чипе.

Типичная **ASIC** на базе стандартных ячеек начинается с пустой матрицы. Она не имеет заранее созданной логики и соединительных линий. Неизвестен даже размер матрицы. Все аппаратные элементы должны быть размещены в чипе, включая **IP**-ядра (например, **MIPS M5150 CPU**). Таким образом, проходя через этапы проектирования **ASIC**, необходимо помнить о многих сопутствующих рисках:

1. Проект на **FPGA** (даже большой и сложный) намного более предсказуем, легче отлаживается и модифицируется.
2. **FPGA** с большей вероятностью будет работать правильно на первом же цикле разработки. В качестве упрощенной описательной аналогии процесс конфигурации **FPGA** может быть представлен в виде нумерованной раскрески на [рис. А.2\(а\)](#). Разработка **ASIC** – это словно рисование на пустом холсте ([рис. А.2\(а\)](#)). Команда разработчиков **ASIC** должна начинать с нуля, с оптимальной компоновки встроенных ресурсов – плана размещения. Этот план может потребовать много подробных итераций, прежде чем будет достигнута оптимальная схема маршрутизации на чипе.



а)



б)

Рис. А.2 Сравнение FPGA и ASIC

3. На кристалле **ASIC** нет встроенной схемы подключения. Разработчики должны решить, какие металлические слои использовать для распределения электропитания, синхронизации, прокладки соединений. После маршрутизации разработчики могут столкнуться с проблемами синхронизации. Исправление одного нарушения синхронизации часто влечет за собой другое. Разработка **ASIC** менее предсказуема, более итеративна, и при этом возникает больше сложностей при отладке. Используемые инструменты электронной автоматизации проектирования (**EDA**) более сложны, чем инструменты **Quartus Prime** или **Icarus**. Несмотря на более длительный цикл проектирования, новая **ASIC** с меньшей вероятностью будет работать после первого цикла разработки.

А.2.1 Схема разработки ASIC

На рисунке ниже приведены основные этапы проектирования **ASIC**, которое начинается с переработки файлов **RTL**-кода, написанных для прототипа **FPGA**. Для соответствия **FPGA MAX10** на данном рисунке указано, что разработка ведется для **65-нм CMOS**-технологии. Эта технология, как и ранее, применяется для создания многих чипов, таких как микроконтроллеры. Основные принципы логического и физического проектирования, рассмотренные в следующих разделах, за некоторыми исключениями, по-прежнему применяются к **32-, 14- и 7-нм** технологиям.

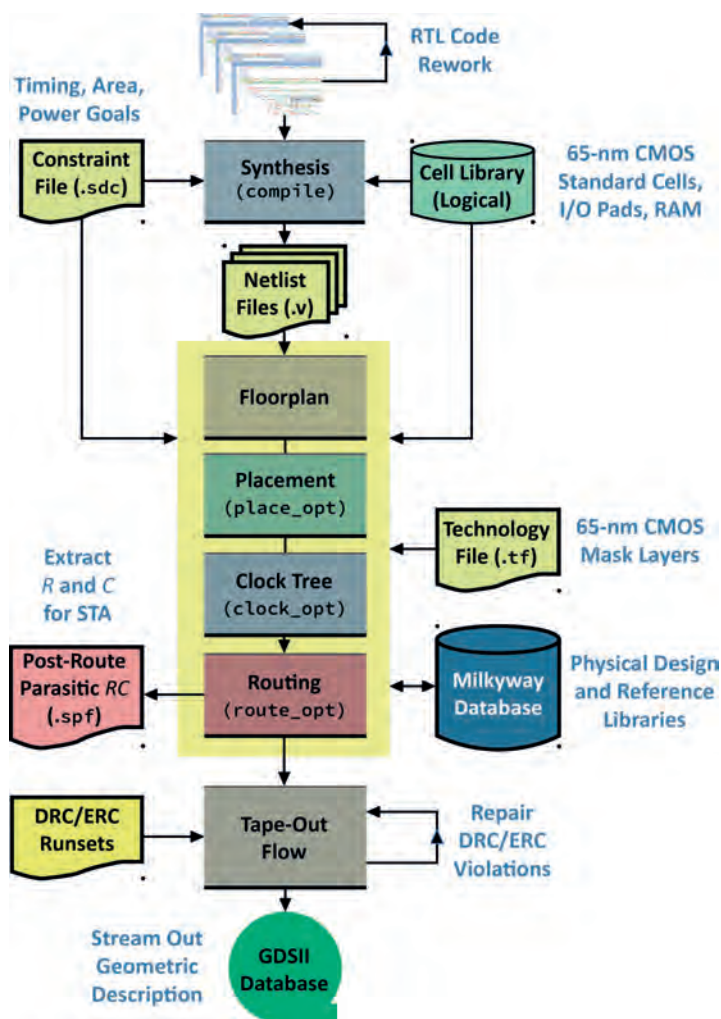


Рис. А.3 Последовательность этапов проектирования ASIC

Прямоугольники на рис. А.3 соответствуют этапам синтеза, размещения или маршрутизации с использованием коммерческих инструментов EDA. Файлы, считанные или записанные инструментами синтеза, изображены в виде значков документов. Библиотеки изображены в цилиндрических контейнерах.

Некоторые прямоугольники аннотируются круглыми скобками с командами ключевых инструментов, используемых при выполнении последовательной компиляции для синтеза. Эти TCL-команды относятся к инструментам Synopsys, таким как Design Compiler. Они отличаются в других средствах проектирования: например, в Genius Cadence команде compile соответствует команда synthesizer. Далее рассмотрена вся последовательность этапов проектирования ASIC. При этом внимание обращено на возможные «подводные камни» и на то, как их можно обойти.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.a-planet.ru.**

Оптовые закупки: тел. +7 (499) 782-38-89

Электронный адрес: **books@alians-kniga.ru.**

Под общей редакцией А. Ю. Романова, Ю. В. Панчула
Цифровой синтез: практический курс

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Корректор *Синяева Г. И.*

Верстка *Орлов И. Ю.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.

Гарнитуры «PT Serif», «PT Sans», «PT Mono». Печать цифровая.

Отпечатано в ПАО «Т8 Издательские Технологии»

109316, Москва, Волгоградский проспект, д. 42, корпус 5.

Усл. печ. л. 45,18. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**