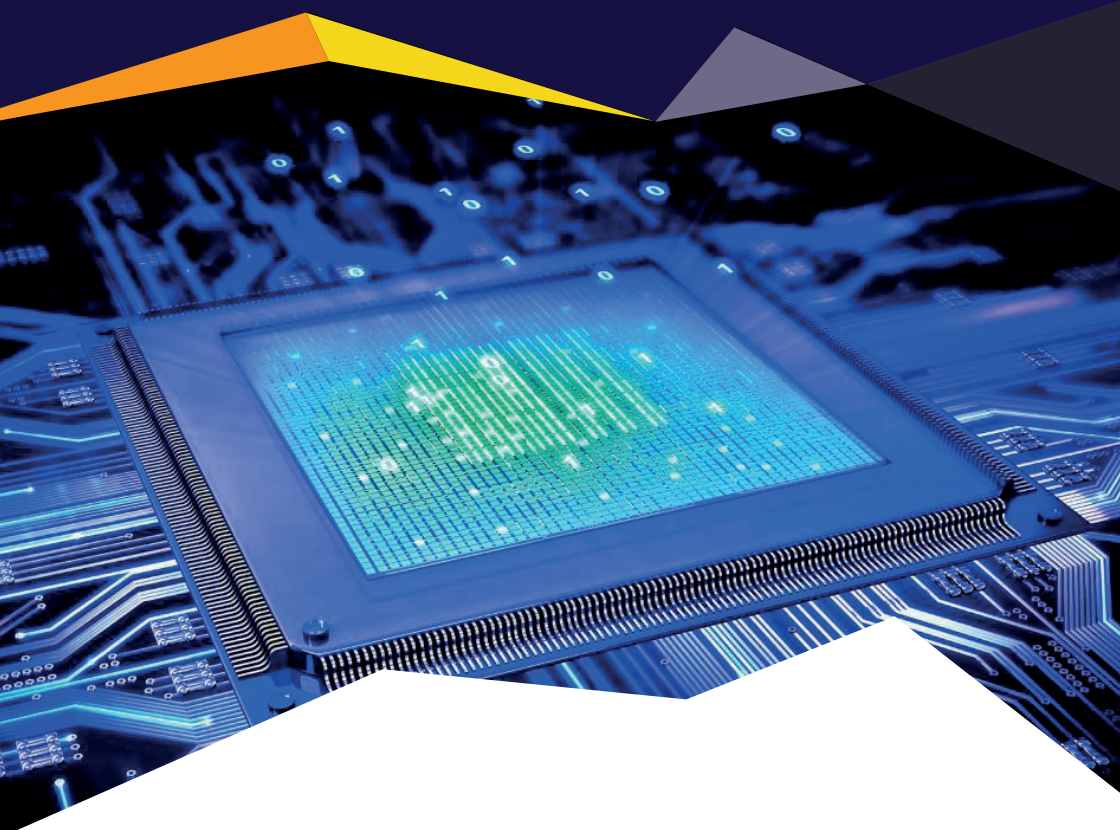




СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
SIBERIAN FEDERAL UNIVERSITY

ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ

УЧЕБНОЕ ПОСОБИЕ



**ИНСТИТУТ КОСМИЧЕСКИХ
И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**

Министерство науки и высшего образования Российской Федерации
Сибирский федеральный университет

ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ

Учебное пособие

Красноярск
СФУ
2020

УДК 621.3.049.77:004.42(07)

ББК 32.844.15я73

П784

Рецензенты:

В. А. Хабаров, кандидат технических наук, ведущий инженер-конструктор отдела систем управления АО «НПП “Радиосвязь”»;

В. В. Прудков, кандидат технических наук, начальник группы отдела проектирования и испытания бортовой РЭА систем управления КА АО «ИСС» им. акад. М. Ф. Решетнева

П784

Программируемые логические интегральные схемы :
учеб. пособие / Н. Ю. Сиротинина, О. В. Непомнящий,
А. И. Постников, Д. А. Недорезов. – Красноярск : Сиб. федер.
ун-т, 2020. – 224 с.

ISBN 978-5-7638-4244-9

Рассмотрены архитектуры программируемых логических интегральных схем, маршруты проектирования на их основе, средства разработки ПЛИС-проектов на языке описания аппаратуры Verilog. Представлены практические работы, позволяющие освоить разработку ПЛИС-проектов.

Предназначено для студентов бакалавриата всех форм обучения по направлению подготовки 09.03.01 «Информатика и вычислительная техника» в качестве основной литературы по дисциплине «Программируемые логические интегральные схемы». Может использоваться в качестве основной и дополнительной литературы при изучении дисциплин «Программируемые логические интегральные схемы. Дополнительные главы», «Прикладная теория цифровых автоматов», «Схемотехника ЭВМ» и др.

Электронный вариант издания см.:

<http://catalog.sfu-kras.ru>

УДК 621.3.049.77:004.42(07)

ББК 32.844.15я73

Разработано в рамках проекта Erasmus+ 573545-EPP-1-2016-1-DE-EPPKA2-CBHE-JP Applied curricula in space exploration and intelligent robotic systems / Прикладные образовательные программы в области освоения космоса и интеллектуальных робототехнических систем.

Co-funded by the Erasmus+ Programme of the European Union: Joint project Capacity Building in the field of Higher Education 573545-EPP-1-2016-1-DE-EPPKA2-CBHE-JP Applied curricula in space exploration and intelligent robotic systems.

The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

ISBN 978-5-7638-4244-9

© Сибирский федеральный университет, 2020

ОГЛАВЛЕНИЕ

Введение.....	6
1. Общие сведения о ПЛИС.....	9
1.1. История вычислительной техники и предпосылки появления ПЛИС	9
1.2. Классификация ПЛИС.....	15
1.3. Области применения ПЛИС	25
1.4. Архитектура ПЛИС класса FPGA	26
1.4.1. Программируемая коммутирующая среда FPGA.....	31
1.4.2. Способ хранения конфигурации FPGA.....	34
1.4.3. Специализированные блоки современных FPGA.....	35
1.4.4. Аппаратные и программные встроенные микропроцессорные ядра	38
1.4.5. Дерево синхронизации и диспетчер синхронизации	40
1.5. Основные производители ПЛИС и их продукция	42
1.5.1. ПЛИС Altera (Intel FPGA)	43
1.5.2. ПЛИС фирмы Xilinx	48
1.5.3. ПЛИС фирмы Lattice	49
Вопросы и задания.....	51
2. Разработка ПЛИС-проектов. Синтезируемые описания на языке Verilog	53
2.1. Маршруты и технологии проектирования изделий на базе ПЛИС	53
2.2. Языки описания аппаратуры: общие сведения	61
2.3. Язык описания аппаратуры Verilog	66
2.4. Verilog: основные конструкции	69
2.4.1. Имена и комментарии.....	69
2.4.2. Типы переменных	70
2.4.3. Особые типы данных: цепи и регистры.....	72
2.4.4. Многоразрядные переменные – шины	74
2.5. Структура ПЛИС-проекта. Модуль	75
2.5.1. Порты в описании модуля.....	77
2.5.2. Параметры в описании модуля	80
2.5.3. Создание экземпляра модуля – инстанса.....	81
2.6. Структурное описание модулей.....	82

2.7. Функциональное описание модулей	92
2.7.1. Операции языка Verilog	92
2.7.2. Приоритет операций	104
Вопросы и задания	105
3. Синтезируемое описание асинхронных (комбинационных) схем	107
3.1. Оператор непрерывного присваивания (назначения) assign	108
3.2. Поведенческий блок always в описаниях комбинационных схем	114
3.3. Синтезируемая конструкция цикла for	116
3.4. Описание условной логики в комбинационных схемах	118
3.4.1. Тернарная операция ?:	119
3.4.2. Условная конструкция if	119
3.4.3. Конструкция множественного выбора case	124
3.5. Пример разработки асинхронной схемы: арифметико-логическое устройство	127
Вопросы и задания	130
4. Синтезируемое описание синхронных схем	131
4.1. Поведенческий блок always при описании синхронных схем	131
4.2. Синхронные и асинхронные управляющие сигналы	134
4.2.1. Синхронный сброс	134
4.2.2. Асинхронный сброс	136
4.3. Блокирующее и неблокирующее присваивание	138
4.4. Синтезируемое описание типовых синхронных схем, используемых в ПЛИС-проектах	142
4.4.1. Модули памяти	143
4.4.2. Регистры с расширенной функциональностью	147
4.4.3. Таймеры-счетчики	148
4.5. Пример разработки модуля памяти: регистровый файл для АЛУ	152
4.6. Пример разработки иерархического проекта: процессор	155
4.7. Синтезируемое описание цифровых автоматов	160
4.7.1. Формы задания конечных автоматов	163
4.7.2. Кодирование состояний	164
4.7.3. Описание конечного автомата Мура	167
4.7.4. Описание конечного автомата Мили	170

4.7.5. Сравнительный анализ выходных сигналов автоматов Мура и Мили	173
4.8. Пример разработки цифрового автомата: устройство управления процессором.....	173
Вопросы и задания.....	177
5. Библиотеки модулей. IP-ядра	179
5.1. Применение IP-ядер в ПЛИС-проектах	179
5.1.1. Виды IP-ядер.....	179
5.1.2. Требования к IP-ядрам.....	181
5.2. Soft-процессоры	182
Вопросы и задания.....	186
6. Практикум по курсу «Программируемые логические интегральные схемы»	187
6.1. Аппаратное обеспечение для выполнения работ.....	187
6.2. Программное обеспечение для выполнения работ.....	187
6.3. Общие требования к выполнению практических работ.....	189
6.4. Тематические практические работы	189
6.5. Итоговый ПЛИС-проект по разделу «Синтезируемые описания на языке Verilog»	216
Заключение.....	218
Список литературы	219

ВВЕДЕНИЕ

Цель преподавания дисциплины «Программируемые логические интегральные схемы» (ПЛИС) – обучение студентов использованию методов и средств проектирования цифровых устройств на базе программируемых интегральных схем.

Разработка и создание цифровых устройств различного назначения на базе ПЛИС является одним из наиболее активно развивающихся направлений развития цифровой элементной базы. Основными областями применения ПЛИС в настоящее время являются:

- цифровая обработка сигналов;
- высокопроизводительные вычисления;
- криптографические системы, системы защиты информации;
- мосты (коммутаторы) между системами с различной логикой, напряжением питания, с несовместимыми или нестандартными интерфейсами;
- нейронные сети и прочие разработки из области искусственного интеллекта;
- системы с перестраиваемой (реконфигурируемой) архитектурой;
- прототипирование заказных интегральных схем.

Обширный список областей применения ПЛИС позволяет предположить, что специалисты в области разработки ПЛИС-проектов будут востребованы на рынке труда.

Пособие содержит базовый объем информации, достаточный для того, чтобы студент получил представление об особенностях организации современных ПЛИС с различными архитектурами, их преимуществах и недостатках, применимости для решения различных типов прикладных задач, проблемах эффективного использования. Практическая часть пособия позволяет освоить базовые навыки применения средств разработки и отладки ПЛИС-проектов.

После освоения курса студент должен уметь:

- выполнять декомпозицию проектируемого цифрового устройства на модули;
- описывать отдельные модули и ПЛИС-проект в целом средствами схемного ввода и с использованием языка описания аппаратуры Verilog;

- разрабатывать тестовое окружение ПЛИС-проекта;
- выполнять моделирование, тестирование и отладку ПЛИС-проектов средствами среды разработки Quartus и ModelSim;
- программировать целевую ПЛИС;
- тестировать полученное изделие.

Пособие состоит из пяти глав и практической части.

Первая глава содержит общие сведения о ПЛИС, предпосылки появления ПЛИС в контексте истории развития вычислительной техники. Приведена классификация ПЛИС, и охарактеризованы основные области их применения. Особое внимание уделено архитектурам ПЛИС класса FPGA. Перечислены основные производители ПЛИС, и дана краткая характеристика их продукции. Дано представление о маршруте проектирования цифровых устройств на базе ПЛИС. Введено понятие языков описания аппаратуры.

Во второй главе рассмотрены общие вопросы разработки ПЛИС-проектов и создания синтезируемых описаний на языке Verilog. Рассмотрены основные конструкции Verilog, соглашения о порядке именования. Описаны основные типы переменных, как универсальные, так и специфические для языков описания аппаратуры, такие как цепи и регистры. Перечислены основные операции Verilog. Рассмотрена структура ПЛИС-проекта. Введены понятия модуля и экземпляра модуля (инстанса), даны основные способы и правила описания модулей.

Третья глава посвящена вопросам синтезируемого описания асинхронных (комбинационных) схем. Рассмотрены операция непрерывного присваивания (назначения) `assign` и особенности ее применения в синтезируемых ПЛИС-проектах, способы описания условной логики в комбинационных схемах.

Четвертая глава посвящена синтезируемым описаниям синхронных схем. Дано понятие поведенческого блока `always`, рассмотрены особенности описания синхронных и асинхронных управляющих сигналов. Особое внимание уделено специфическим операциям блокирующего и неблокирующего присваивания. Рассмотрены синтезируемые описания типовых синхронных схем, используемых в ПЛИС-проектах: модулей памяти, регистров с расширенной функциональностью, таймер-счетчиков. Описаны подходы к проектированию цифровых автоматов Мура и Мили.

В последней главе дано общее представление о возможности повторного использования разработанных компонентов, о библиотечных и IP-ядрах различной сложности, в частности, о soft-процессорах.

Курс практических работ состоит из двух видов заданий: тематических практических работ и заключительного проекта.

Цель тематических работ – изучение особенностей организации ПЛИС, средств и инструментов разработки ПЛИС-проектов, подходов к описанию модулей различных типов.

Цель итогового проекта – получение навыков применения инструментов и приемов разработки ПЛИС-проекта в комплексе.

Для того чтобы успешно усвоить материал курса, необходимо иметь навыки программирования на языке высокого уровня, представление о математической логике, прикладной теории цифровых автоматов, схемотехнике ЭВМ.

Материал курса далее будет востребован в дисциплинах «Микропроцессорные системы», может оказаться полезным при выполнении выпускной квалификационной работы и при дальнейшем обучении в магистратуре. Пособие может использоваться в качестве основной и дополнительной литературы для таких дисциплин, как «Программируемые логические интегральные схемы. Дополнительные главы» (магистратура), «Прикладная теория цифровых автоматов», «Схемотехника ЭВМ» и др.

1. ОБЩИЕ СВЕДЕНИЯ О ПЛИС

1.1. История вычислительной техники и предпосылки появления ПЛИС

В 1937 году американский математик, специалист в области теории шифрования Клод Шеннон (Claude Elwood Shannon) показал, что существует соответствие «один к одному» между конструкциями булевой логики и некоторыми электронными схемами, которые получили название «логические вентили». Он продемонстрировал, что комбинационные схемы – логические вентили, объединенные связями, – эквивалентны выражениям булевой алгебры. Своей работой “A Symbolic Analysis of Relay and Switching Circuits” он создал основу для практического проектирования цифровых схем, которые могли осуществлять вычисления [1, 2].

Одним из основных кандидатов на роль первой электронной вычислительной машины, воплотившей принципы Шеннона, считается программируемый электронный калькулятор общего назначения ENIAC (Electronic Numerical Integrator and Computer – электронный цифровой интегратор и вычислитель), который был создан в период с 1942 по 1946 год в университете Пенсильвании математиками и инженерами Джоном Мочли (John J. Mauchly) и Преспером Эккертом (J. Presper Eckert). Важной особенностью этой вычислительной системы являлось то, что порядок выполнения вычислений задавался схемой коммутации компонентов на 40 наборных полях. Таким образом, для каждой прикладной задачи вычислительная система фактически конфигурировалась заново. Процесс коммутации занимал длительное время – от нескольких часов до нескольких суток [3]. В проекте принимал активное участие американский физик и математик Джон фон Нейман (John von Neumann). Фон Нейман в своей работе представил научному миру свои идеи по организации компьютера [4]. Именно эти идеи, получившие название «принципы фон Неймана», в основном определили развитие вычислительной техники на длительный период. Одним из принципов фон Неймана является принцип программного управления.

Принцип программного управления гласит, что все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов – команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, т. е. в порядке их положения в программе. При необходимости эта последовательность может быть изменена с помощью специальных команд. Была сформирована архитектура вычислительной машины, выполненной согласно принципам фон Неймана. Позднее в архитектуре ЭВМ был выделен процессор (рис. 1).

Процессор (центральный процессор, ЦП; от англ. central processing unit, CPU) – электронный блок, исполняющий машинные команды, главная часть аппаратного обеспечения компьютера.

В то же время Аланом Тьюрингом было доказано, что программируемая вычислительная машина является универсальным инструментом, т. е. способна реализовать любой алгоритм обработки информации. Далее история развития цифровой техники стала прежде всего историей развития программируемых электронных вычислительных машин. Эту историю принято разбивать на поколения [3].

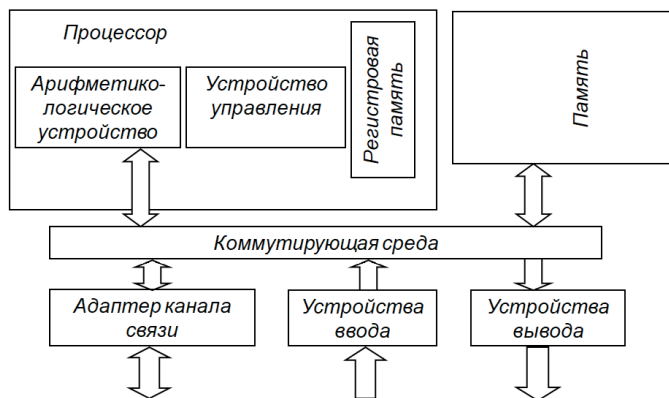


Рис. 1. Архитектура фон Неймана

Первое поколение (1945–1954), представителем которого был упомянутый ENIAC, было реализовано на электронных лампах.

Во втором поколении (1955–1964) вместо электронных ламп использовались транзисторы, а в качестве устройств памяти стали применяться магнитные сердечники и магнитные барабаны.

В третьем поколении (1965–1974) впервые стали использоваться интегральные схемы, выполняющие как функции обработки информации, так и функции ее хранения.

Интегральная схема (микросхема) – микрoeлектронное изделие, выполняющее определенную функцию преобразования, обработки сигнала, накопления информации и имеющее высокую плотность электрически соединенных элементов (или элементов и компонентов), которые с точки зрения требований к испытаниям, приемке, поставке и эксплуатации рассматриваются как единое целое.

Для оценки сложности цифровых СБИС используется единица «эквивалентный вентиль» (например, 2-входовый элемент «И-НЕ»), который соответствует четырем эквивалентным транзисторам (рис. 2).

На первых этапах для реализации процессоров использовались микросхемы низкой и средней степени интеграции и сам процессор представлял собой сложный многокомпонентный модуль. По мере развития технологии и роста степени интеграции микросхем количество элементов и соединений между ними, реализуемых на одном кристалле, постоянно росло. В 1970-е годы интегральные схемы содержали уже тысячи транзисторов, что позволило реализовать в одной микросхеме отдельные элементы цифровой схемотехники: логические элементы, затем более сложные модули – регистры, счётчики, сумматоры. Позднее появились микросхемы, содержа-

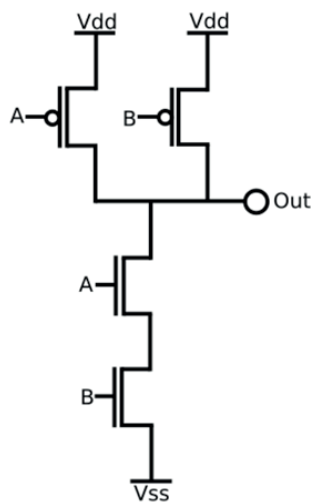


Рис. 2. Пример эквивалентного вентиля 2-входового И-НЕ

щие функциональные блоки процессора: микропрограммное устройство, арифметико-логическое устройство, устройства работы с шинами данных и команд [2, 3].

Наконец, в 1971 году фирма Intel выпустила первый микропроцессор, который положил начало четвертому поколению ЭВМ.

Микропроцессор – это программно-управляемое устройство, которое отвечает за выполнение арифметических, логических операций и операций управления, записанных в машинном коде, конструктивно выполненное в виде одной большой интегральной схемы (БИС) [5].

Многие исследователи считают, что на этом смена поколений ЭВМ прекратилась, прогресс идет в основном по пути развития того, что уже изобретено. Основным направлением развития долгое время оставалось уменьшение размеров электронных компонент, увеличение их числа на отдельном кристалле. К настоящему моменту это направление развития вплотную приблизилось к физическому пределу, а каждый следующий шаг становится все более затруднительным. В то же время запросы потребителей и рынка к вычислительной технике постоянно растут. В основном это касается таких характеристик, как производительность, автономность и энергонезависимость, компактность вычислительных устройств. Ведется поиск новых архитектур, в основном в направлении параллельной организации вычислительных систем, но это не снимает остроты проблемы. Ситуация требует поиска резервов для повышения перечисленных характеристик и принципиально новых решений.

Программно-управляемые устройства универсальны, т. е. способны решить практически любую задачу за счет реализации принципа программного управления, описанного выше. Программное управление позволяет изменить алгоритм работы и выполняемые функции за счет замены программного обеспечения, не затрагивая аппаратную часть. Однако выполнение программы требует значительных затрат времени на обработку машинных команд, обмен с памятью команд и данных и прочие этапы командного цикла, а сами программно-управляемые системы имеют достаточно сложную архитектуру. Большинство специальных задач может быть успешно решено с использованием программно-управляемых устройств (ЭВМ, микроконтроллеров) за счет разработки и оптимизации соответствующего программного обеспечения. Но если

накладываются жесткие критические ограничения на объем аппаратуры и время выполнения вычислений, программно-управляемые устройства часто не обеспечивают нужные характеристики. Аппаратная реализация функций в подавляющем большинстве случаев является более быстродействующей, хотя и лишена свойства универсальности. По этой причине перед разработчиками вычислительной техники, наряду с задачей проектирования универсальных вычислительных машин, всегда стояли проблемы создания специализированной аппаратуры, предназначенной для выполнения конкретных, специфических операций. Долгое время для подобного применения разрабатывались специализированные многокомпонентные устройства из отдельных элементов и микросхем малой и средней степени интеграции, а в настоящее время с этой целью применяются специализированные интегральные микросхемы.

Специализированные (заказные) интегральные микросхемы (Application Specific Integrated Circuit, ASIC) – микросхемы, которые имеют жестко заданную на этапе разработки структуру, проектируются и изготавливаются под заказ и предназначены для выполнения конкретной задачи или круга задач [6]. Специализированные интегральные схемы выполняют строго ограниченные функции, характерные только для данного устройства, причем эффективность реализации этих функций обычно очень высока.

Микросхемы класса ASIC получили широкое распространение, поскольку они являются практически единственным приемлемым решением при реализации сложных изделий микроэлектроники для портативной и носимой аппаратуры. ASIC широко используются в различных отраслях микроэлектроники, включая телеметрию, коммуникационные системы, глобальные навигационные системы, мультимедиа, автомобильную и бытовую электронику и т. д. Основное преимущество таких микросхем – быстродействие. При этом современные ASIC настолько сложны, что их проектирование даже с применением современных инструментов САПР является серьезной проблемой и часто требует значительных трудозатрат и недопустимо много времени. При мелкосерийном и единичном производстве ASIC стоят значительно дороже массово изготавливаемых микросхем, а в случае даже незначительных изменений в алгоритме функционирования требуют повторной разработки в полном объеме. Поэтому применение ASIC становится эконо-

мически оправданным в случае достаточно большого тиража продукции из-за высоких затрат на разработку и отладку проекта [6, 7].

Сравнительно новым решением, которое может во многих случаях заменить ASIC, являются программируемые логические интегральные схемы.

Программируемая логическая интегральная схема (ПЛИС) – электронный компонент, логика работы которого не определяется при изготовлении, а задаётся посредством программирования связей между элементами [7, 8].

Важный момент, на который стоит еще раз обратить внимание: при программировании ПЛИС не пишется программа, определяющая последовательность операций, а задаются (программируются) связи между элементами. Можно сказать, что ПЛИС представляет собой конструктор, с помощью которого фактически собирается аппаратная реализация требуемой функции.

В зависимости от сложности ПЛИС реализуемые функции могут варьироваться от довольно простых, соответствующих одной или нескольким микросхемам низкого уровня интеграции, до весьма сложных устройств. На ПЛИС можно реализовать алгоритмы цифровой обработки сигналов (ЦОС) и другие сложные задачи обработки данных и управления, включая и возможность сборки своего собственного процессора (soft-процессора) с уникальной архитектурой.

История ПЛИС пишется прямо сейчас. Хотя первые реализации микросхем, которые можно отнести к ПЛИС, относятся к середине 70-х годов прошлого века, заметное влияние на современную вычислительную технику они начали оказывать только последние 10–15 лет. В настоящее время ПЛИС – современная, активно развивающаяся, удобная в освоении и применении элементная база. Развитие ПЛИС связано прежде всего с увеличением количества элементов на кристалле. Общепринятой оценкой логической емкости ПЛИС является число эквивалентных вентилях, определяемое как среднее число вентилях «И-НЕ» на два входа, необходимых для реализации эквивалентного проекта. Эта оценка очень приближительна, однако для проведения сравнительного анализа различных проектов ее можно использовать. Широко выпускаемые ПЛИС с эквивалентной емкостью более 1 миллиона логических вентилях предоставляют разработчику достаточно ресурсов для реали-

зации сложных и интересных проектов. Цены производителей на ПЛИС неуклонно падают. Существуют и также постоянно совершенствуются инструменты и системы автоматизированного проектирования на базе ПЛИС. Все это вызвало потребность в подготовке специалистов, способных проводить разработку аппаратуры на базе ПЛИС, владеющих основными методами и средствами проектирования, ориентирующихся в современной элементной базе и программном обеспечении.

1.2. Классификация ПЛИС

Отметим, что в области ПЛИС, которая возникла сравнительно недавно и активно развивается, терминология не является устоявшейся. Разные авторы предпочитают разные варианты перевода англоязычных терминов или заимствований; как правило, это не создает особых проблем. Другая сложность состоит в том, что иногда нет однозначного соответствия между русско- и англоязычными терминами.

Наиболее точно термину «ПЛИС» соответствует Programmable Logic Device, PLD. Но нужно иметь в виду, что часто в русскоязычной литературе термину «ПЛИС» перевод как FPGA – Field Programmable Gate Array, дословно – программируемый пользователем массив логических вентилей. Это соответствие, строго говоря, не вполне точно. FPGA представляют собой один из типов ПЛИС, наиболее распространенный в настоящее время.

Основными критериями такой классификации являются наличие, вид и способы коммутации элементов логических матриц. По этим признакам можно выделить следующие классы ПЛИС (рис. 3).

Рассмотрим приведенные на рис. 3 типы ПЛИС с указанием исторической последовательности их появления.

Согласно приведенной классификации на верхнем уровне ПЛИС подразделяются на логические устройства, программируемые в заводских условиях (factory programmable), и устройства, программируемые пользователем (field programmable) [9].

К первым можно отнести микросхемы энергонезависимой памяти, обычно используемые для хранения неизменяемых программ и данных – ROM (read-only memory), а также устройства с масочным

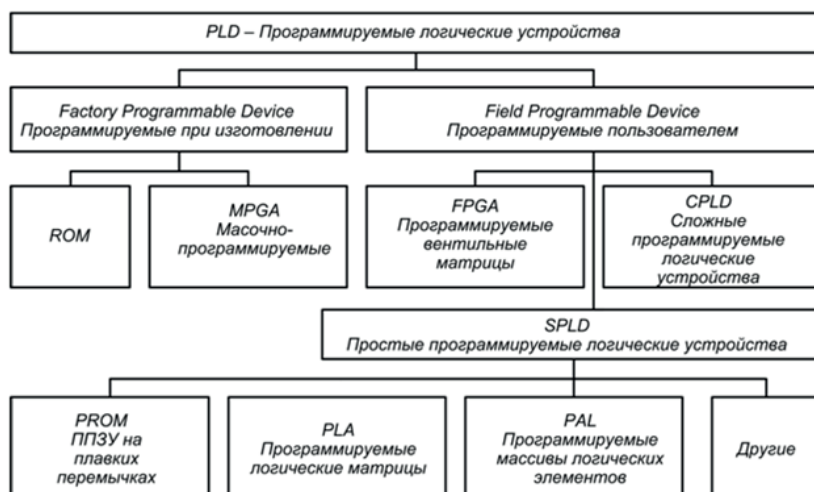


Рис. 3. Семейства программируемых логических интегральных схем

программированием (mask programmable gate array). Программирование масочных схем происходит в процессе изготовления БИС. Обычно на кристалле полупроводника вначале создаются все запоминающие элементы (ЗЭ), а затем, на заключительных технологических операциях, с помощью фотошаблона слоя коммутации реализуются связи между линиями адреса, данных и собственно запоминающим элементом. Этот шаблон (маска) выполняется в соответствии с техническим заданием заказчика.

Класс устройств, программируемых при изготовлении, является достаточно обширным; к нему могут быть, в частности, отнесены некоторые разновидности специализированных заказных микросхем. Но устройства этого типа в данном курсе не рассматриваются подробно.

Первые из появившихся программируемых пользователем логических устройств относятся к классу простых программируемых логических устройств (ППЛУ, Simple Programmable Logic Devices, SPLD) [9, 10].

История развития ПЛИС начинается с появления в начале 70-х годов прошлого века программируемых постоянных запоминающих устройств (ППЗУ, Programming Read-Only Memory, PROM), входящих

в класс программируемых логических устройств (ПЛУ, Programmable Logic Device, PLD). Изначально основной функцией ППЗУ являлось хранение программ, констант, таблиц знакогенераторов и т. п. При этом на вход схемы поступает адрес ячейки памяти, а на выход – хранимые данные.

В то же время если предположить, что входные сигналы схемы – это значения аргументов логической функции, а в ячейки памяти занесены соответствующие им значения этой функции, то мы получаем фактически ее табличную реализацию. ППЗУ реализует функции в виде комбинационной схемы, соответствующей совершенной дизъюнктивной нормальной форме (СДНФ). Поскольку слово памяти имеет обычно несколько разрядов, а схема ППЗУ соответственно несколько выходов, то может быть реализовано сразу несколько логических функций.

Конструктивно такая микросхема представляет собой фиксированную матрицу элементов «И», в которой реализуются все возможные конъюнкции входных сигналов, подсоединенную к программируемому набору логических функций «ИЛИ». Незапрограммированное ППЗУ имеет полные соединения между шинами входных и выходных переменных (на рис. 4 отмечены косыми крестиками) на всех пересечениях матриц «И» и «ИЛИ». Программирование ППЗУ, т. е. занесение в него данных, осуществляется путем электрического разрушения ненужных соединений. Таким образом, на выходе можно реализовать любую функцию от трех переменных в виде СДНФ.

В качестве примера рассмотрим ППЗУ с тремя входами (a , b и c) и тремя выходами (w , x и y) (рис. 4). На программируемом массиве элементов «ИЛИ» в процессе программирования оставляются связи, отмеченные кружками, остальные разрушаются.

Схема, приведенная на рис. 4, реализует следующие функции:

$$\text{выход } w = a \& b \& \bar{c} \vee a \& b \& c = a \& b;$$

$$\begin{aligned} \text{выход } x &= \bar{a} \& \bar{b} \& \bar{c} \vee \bar{a} \& \bar{b} \& c \vee \bar{a} \& b \& \bar{c} \vee \bar{a} \& b \& c \vee a \& \bar{c} \& \bar{b} \& c = \\ &= \bar{a} \& \bar{b} \vee \bar{a} \& b \vee a \& \bar{b} = \bar{a} \vee \bar{b}; \end{aligned}$$

$$\text{выход } y = \bar{a} \& \bar{b} \& c \vee \bar{a} \& b \& c \vee a \& \bar{b} \& c \vee a \& b \& \bar{c} = \bar{a} \& c \vee \bar{b} \& c \vee a \& b \& \bar{c}.$$

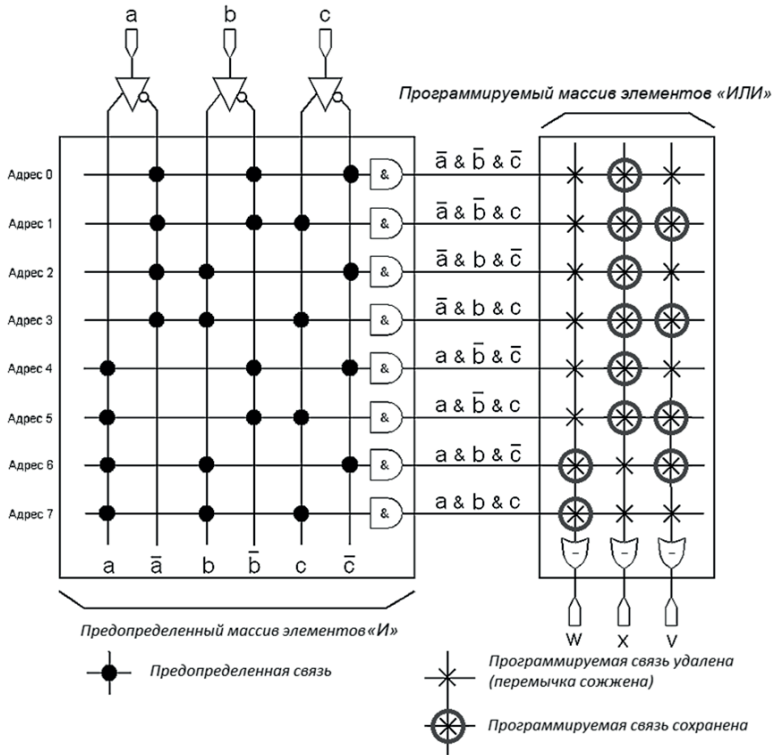


Рис. 4. Пример использования ППЗУ для реализации простых логических функций

Такой способ программирования позволяет использовать микросхемы ППЗУ в качестве простых программируемых устройств для реализации простых логических функций. Для более сложных устройств он не подходит, поскольку сложность схемы с увеличением числа входных сигналов растет экспоненциально: если добавить один входной сигнал (аргумент функции), сложность схемы увеличится примерно в два раза. Время прохождения сигналов через схему также возрастет, а значит, снизится ее быстродействие. Такой способ реализации не оптимизируется. По этим причинам использование ППЗУ в качестве ПЛИС оправдано для реализации устройств с небольшим числом входов (аргументов функции) и большим числом реализуемых функций.

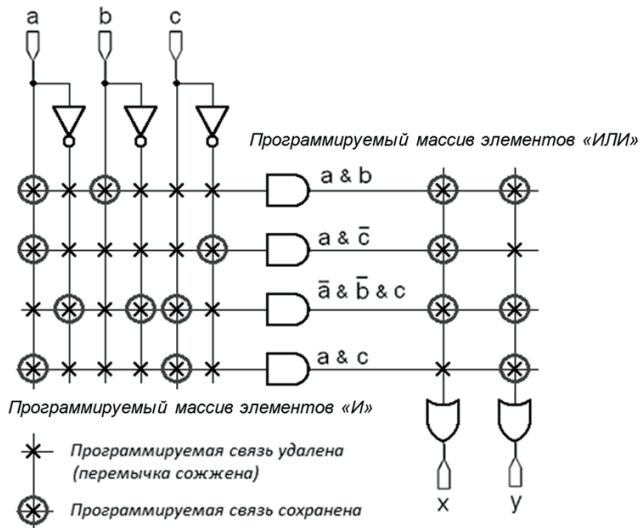


Рис. 5. Пример использования ПЛМ (PLA) для реализации простой логической функции

Исторически следующими появились программируемые логические матрицы (ПЛМ), созданные в 1975 году [9, 10].

Программируемые логические матрицы (ПЛМ, Programmable Logic Array, PLA) являются универсальными логическими схемами, предназначенными для реализации систем логических функций, заданных в дизъюнктивной нормальной форме [11, 12].

ПЛМ состоит из двух программируемых логических матриц (матрицы «И» и матрицы «ИЛИ») и вспомогательных схем, обеспечивающих сопряжение и программирование микросхемы. Термин «матрица» здесь используется в смысле «регулярная структура», набор однородных элементов и связей между ними. Кроме того, схемные элементы подобных изделий удобнее всего располагать по строкам и столбцам, обеспечивая тем самым регулярность структуры БИС [12].

Схема, приведенная на рис. 5, реализует следующие функции:

$$\text{выход } x = a \& b \vee a \& \bar{c} \vee \bar{a} \& \bar{b} \& c;$$

$$\text{выход } y = a \& b \vee \bar{a} \& \bar{b} \& c \vee a \& c.$$

Такие микросхемы обеспечивали более широкие возможности re-конфигурирования, но в то же время имели более высокую цену. Недостаточно использовались ресурсы программируемой матрицы «ИЛИ». Решением проблемы явились устройства, основанные на архитектуре программируемой матричной логики (ПМЛ, Programmable Array Logic, PAL), которые появились в конце 70-х годов XX века и были выделены в отдельный класс. В ПМЛ программируемой является логическая матрица элементов «И», а матрица элементов «ИЛИ» является фиксированной (рис. 6) [13, 14].

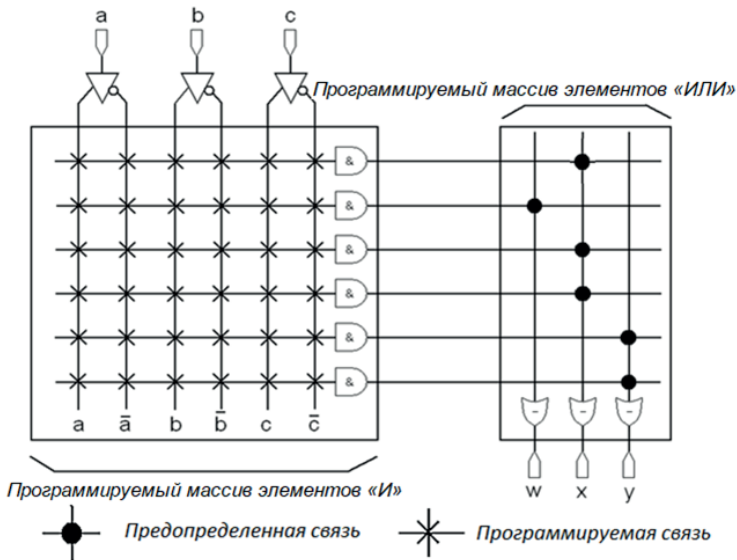


Рис. 6. Незапрограммированное устройство PAL

Эти устройства сочетают достаточно высокую гибкость, сравнимую с характеристиками, свойственную ПЛМ, и быстродействие, свойственное ППЗУ. При этом матрицы логических компонентов имеют значительно меньший размер. Но функциональность программируемого массива логики ограничена, так как его архитектура позволяет выполнять операцию логического сложения только над ограниченным набором логических произведений.

В ПМЛ выработанные матрицей «И» термы поступают на фиксированную (непрограммируемую) матрицу элементов «ИЛИ». Это означает жесткое, заранее заданное распределение имеющихся термов между отдельными элементами «ИЛИ». Каждому элементу «ИЛИ» придаются свои собственные термы, и если для разных функций понадобятся одинаковые термы, придется вырабатывать их в матрице «И» несколько раз. Однако то, что матрица «ИЛИ» не программируется, для многих задач упрощает схему ПМЛ в сравнении со схемой ПЛМ [14, 15].

ПЛМ обладают большей функциональной гибкостью, все воспроизводимые ими функции могут быть комбинациями любого числа термов, формируемых матрицей «И». Это полезно при реализации наборов функций, имеющих большие взаимные пересечения по термам. Такие наборы свойственны, например, задачам формирования сигналов управления машинными циклами процессоров.

Для широко распространенных в практике задач построения «произвольной логики» большое пересечение функций по термам не типично. Для них программируемость матрицы «ИЛИ» используется мало и лишь дополнительно усложняет микросхему. Поэтому ПМЛ распространены больше, чем ПЛМ, и к их числу относится большинство программируемых логических устройств [12, 13].

Устройства ПМЛ и ПЛМ – однократно программируемые: их конфигурация не может быть изменена. В дальнейшем с целью уменьшения энергопотребления и расширения функциональных возможностей логических устройств стали использовать PAL-микросхемы, основанные на КМОП-транзисторах. Этот класс устройств на базе PAL получил собственное имя – «общий логический массив» (Generic Array Logic, GAL). Микросхемы GAL основаны на электрически стираемом перепрограммируемом ПЗУ (ЭСППЗУ, Electrically Erasable Programmable Read-Only Memory, EEPROM), одном из видов энергонезависимой памяти, и могут реконфигурироваться. Микросхемы GAL были впервые разработаны фирмой Lattice Semiconductor [16]. Аналогичные устройства PEEL (programmable electrically erasable logic) были предложены International CMOS Technology (ICT).

Рассмотренные выше способы реализации относятся к простым программируемым логическим устройствам, являющимся основой

большого класса микросхем ПЛУ. Дальнейшее развитие этого класса привело к появлению более сложных устройств.

Сложные программируемые логические интегральные схемы (СПЛИС, Complex Programmable Logic Devices, CPLD) содержат несколько блоков, подобных ПМЛ, объединенных средствами программируемой коммутационной матрицы.

В СПЛИС могут входить сотни блоков и десятки тысяч эквивалентных вентилях. Воздействуя на программируемые соединения коммутационной матрицы и ПМЛ, входящих в состав СПЛИС, можно реализовать требуемую схему. Сложные программируемые логические устройства являются промежуточным, переходным, звеном между программируемыми массивами логических элементов и программируемым пользователем базовыми матричными кристаллами (БМК) [14, 15].

Каждый матричный логический блок (МЛБ), входящий в СПЛИС, представляет собой структуру типа ПМЛ, т. е. программируемую матрицу «И», фиксированную матрицу «ИЛИ» и макроячейки (macrocells), соединённые с внешними выводами и внутренними шинами. История СПЛИС началась в 1984 году, когда только что образованная компания Altera представила сложное ПЛУ, основанное на сочетании КМОП- и СППЗУ-технологий. Основой для программирования этих устройств были ячейки СППЗУ, представляющие собой конструктивно простую и быстродействующую энергонезависимую память с электрическим стиранием, предоставляющие возможность для многократной перезаписи хранимой информации. Именно это сделало их идеальными для использования при разработке и создании прототипов оборудования. Тем не менее в скорости обработки информации микросхемы, базирующиеся на ПЛУ, уступали специализированным заказным интегральным микросхемам ASIC [15, 17].

Современные СПЛИС (рис. 7), как правило, являются электрически перепрограммируемыми и энергонезависимыми, т. е. сохраняют логическую структуру после отключения питания. Наличие внутренней энергонезависимой конфигурационной памяти, наряду с такой важной характеристикой, как устойчивость показателей, делает СПЛИС незаменимыми для современных цифровых схем в качестве устройства для запуска и инициализации цифровых устройств. Учитывая развитие СПЛИС и их возможностей, такие их архитектурные преимущества, как цена, энергонезависимая конфигурация, макроячейки с предсказуемыми харак-

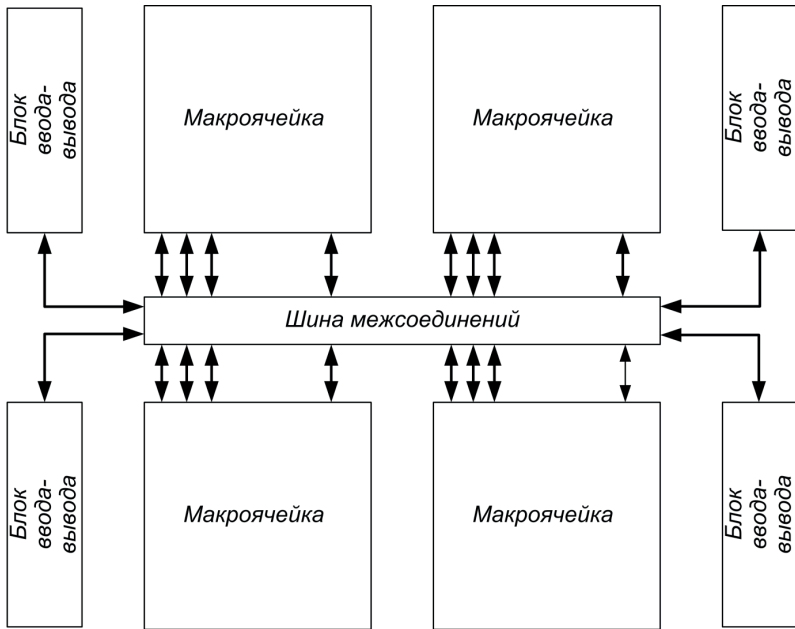


Рис. 7. Внутренняя структура СПЛИС [15]

теристиками параметров, меньшее энергопотребление, можно предположить, что СПЛИС будут иметь в обозримом будущем устойчивую нишу применения. Их можно использовать для решения таких задач, как специализация цифровых схем, мобильные технологии, расширение числа входов/выходов для более сложных микросхем, предобработка сигналов (например контроллер COM-порта, USB, VGA), а также в других целях. В настоящее время ПЛИС класса СПЛИС разрабатываются фирмами Altera, Atmel, Lattice Semiconductor, Cypress Semiconductor, Xilinx и др.

Программируемые пользователем вентиляльные матрицы (ППВМ, Field-Programmable Gate Array, FPGA) – разновидность ПЛИС, содержащая логические элементы (как правило, на базе таблиц истинности) и блоки их коммутации. Для получения требуемой логической функции программируются оба компонента устройства [11, 15].

В русскоязычной литературе для обозначения программируемых пользователем вентиляльных матриц сокращение ППВМ употребляется

крайне редко. Этот тип программируемых логических интегральных схем обычно обозначается английской аббревиатурой FPGA. Нужно также иметь в виду, что во многих источниках термины «ПЛИС» и «FPGA» используются как синонимы; поэтому во избежание путаницы необходимо точно выяснить, что именно автор имеет в виду. В данном пособии далее мы будем придерживаться аббревиатуры ПЛИС в тех случаях, когда речь идет о программируемых логических устройствах в целом, и FPGA – для обозначения программируемых пользователем вентильных матриц.

Структура программируемой пользователем вентильной матрицы содержит множество простых программируемых логических блоков, объединенных в массив. Между блоками имеются «программируемые коммутаторы», при помощи которых можно коммутировать соединения между логическими блоками. Каждый логический блок программируется на выполнение определенной логической функции, затем программируются коммутаторы, и таким образом реализуется вся логическая цепочка.

Конфигурация FPGA хранится в распределённой памяти, которая может быть выполнена на основе энергозависимых ячеек статического ОЗУ. Подобные микросхемы производят, например, фирмы Xilinx и Altera. В этом случае при каждом включении питания микросхемы необходимо заново конфигурировать её при помощи начального загрузчика, который может быть встроен и в саму FPGA. Если распределенная память выполнена с использованием энергонезависимых ячеек flash-памяти или перемычек antifuse (такие микросхемы производят фирмы Actel и Lattice Semiconductor), то в этих случаях конфигурация сохраняется при исчезновении электропитания.

FPGA могут быть модифицированы практически в любой момент процесса их использования. Функции, исполняемые FPGA, могут неоднократно перепрограммироваться за счет изменения топологии внутренних соединений и функций, реализуемых отдельными блоками. Однако такая гибкость требует существенного увеличения количества транзисторов микросхемы.

Именно программируемые логические устройства этого типа являются основным объектом изучения в курсе «ПЛИС»; поэтому внутренняя структура и организация FPGA будут подробно рассмотрены в следующей главе.

1.3. Области применения ПЛИС

Первые ПЛИС использовались преимущественно для создания связующей логики, для реализации конечных автоматов средней сложности и для решения некоторых задач обработки данных. По мере усложнения и увеличения размеров ПЛИС начинают пользоваться большим спросом. В начале 90-х годов прошлого века самый большой объем закупок ПЛИС отмечался в области сетей и телекоммуникаций, в которых предусматривались обработка и передача больших потоков информации. К концу 90-х ПЛИС использовались для создания прототипов заказных микросхем (ASIC) или для создания испытательных стендов, на которых проверяется физическая реализуемость новых алгоритмов. К началу второго тысячелетия появились высокопроизводительные ПЛИС, которые содержат миллионы вентилях. Некоторые из них содержат встроенные микропроцессорные ядра, высокоскоростные интерфейсы ввода/вывода и другие устройства. В последнее время программируемые логические интегральные схемы становятся все более распространенной и привычной элементной базой для разработчиков цифровых устройств. В настоящее время ПЛИС заполняют несколько крупных сегментов рынка.

Основные отрасли науки и техники, в которых применяются ПЛИС:

- авиакосмическая;
- автомобильная;
- военно-промышленный комплекс;
- телекоммуникации, сетевые технологии;
- хранение данных;
- медицинское и научное оборудование;
- высокопроизводительные вычисления, включая майнинг.

Системы на кристалле на базе ПЛИС

Под системой на кристалле (СнК, System-on-a-Chip, SoC) в микроэлектронике понимают электронную схему, выполняющую функции целого устройства и размещенную на одной интегральной схеме [18]. Как правило, SoC применяются в портативных и встраиваемых системах.

Типичная СнК содержит:

- одно или несколько универсальных или специализированных процессорных ядер, или ядер цифровой обработки сигналов (Digital Signal Processing, DSP);

- банк памяти, состоящий из модулей ПЗУ, ОЗУ, ППЗУ или Flash;
- источники опорной частоты, например кварцевые резонаторы и фильтр умножителя частоты (phase-locked loops);
- стандартные интерфейсы для внешних устройств: USB, FireWire, Ethernet, USART, SPI;
- входы и выходы цифро-аналоговых и аналого-цифровых преобразователей;
- регуляторы напряжения и стабилизаторы питания.

Системы на кристалле потребляют меньше энергии, стоят дешевле и работают надёжнее, чем многокорпусное изделие с той же функциональностью. Но создание СнК для конкретной задачи может оказаться чрезмерно дорогим процессом из-за сложности разработки и отладки, перенастройки процесса изготовления и низкого процента годных изделий. Первоначально ПЛИС использовались для макетирования и совместной отладки программной и аппаратной части СнК [19, 20]. Возможности и характеристики современных ПЛИС являются достаточными для реализации СнК в виде ПЛИС-проекта. В настоящее время ПЛИС становятся все более привлекательными устройствами для реализации СнК.

1.4. Архитектура ПЛИС класса FPGA

ПЛИС класса программируемых пользователем логических матриц (FPGA) являются в настоящее время самыми распространенными. Как уже указывалось ранее, в данном пособии далее для обозначения программируемых пользователем логических матриц (конкретного типа ПЛИС) мы будем придерживаться обозначения FPGA; термин «ПЛИС» будет относиться к программируемым логическим интегральным схемам в целом.

Как правило, человек, который начинает изучать ПЛИС, уже имеет опыт программирования. ПЛИС-проекты разрабатываются с использованием специализированных языков описания аппаратуры, синтаксис которых напоминает языки программирования высокого уровня. При этом часто используется словосочетание «программирование ПЛИС». Все это может привести к подмене понятий и ошибочному представлению о принципах работы ПЛИС. Поэтому на первом этапе знакомства

с программируемыми логическими устройствами необходимо осознать следующие важные моменты.

ПЛИС не является устройством, управляемым программой. Описание ПЛИС-проекта в процессе подготовки не переводится в программу в исполняемом процессором коде; в ПЛИС нет как такового процессора, нет команд и нет программы, которая хранится в памяти. В процессе работы ПЛИС не выполняет стандартный машинный цикл исполнения команды «чтение команды – дешифрация команды – выборка операндов – исполнение команды – завершение команды, подготовка к выборке следующей».

Описание ПЛИС-проекта переводится в так называемый файл прошивки, который задает, какие внутренние блоки ПЛИС будут задействованы, какие логические функции они будут выполнять и как будут связаны друг с другом. Процесс программирования ПЛИС заключается в конфигурировании внутренних модулей и связей между ними в соответствии с файлом прошивки. Этот процесс также иногда называется прошивкой ПЛИС (чтобы еще больше запутать непосвященных).

После того, как ПЛИС сконфигурирована (прошита), мы фактически получаем новое цифровое логическое устройство, которое функционирует в соответствии со сформированной внутренней структурой и в зависимости от подаваемых на ее вход сигналов. При этом все процессы в ПЛИС протекают непрерывно, одновременно и параллельно, как в любой электронной схеме.

ПЛИС, выполненная по технологии FPGA [21] (рис. 8), состоит из прямоугольной матрицы программируемых (конфигурируемых) логических блоков (ПЛБ, configurable logic block, CLB), окруженных блоками ввода-вывода (БВВ, Input/Output Block, IOB), которые непосредственно связаны с контактными площадками кристалла.

Между ПЛБ располагаются программируемые трассировочные линии, коммутируемые с использованием программируемых коммутаторов. В современных ПЛИС часто бывают встроены дополнительно блоки памяти, блоки цифровой обработки сигналов или умножители, модули фазовой автоподстройки частоты и другие компоненты.

FPGA включают в себя три главных программируемых элемента:

- программируемые логические блоки, которые являются функциональными элементами для построения логики пользователя;

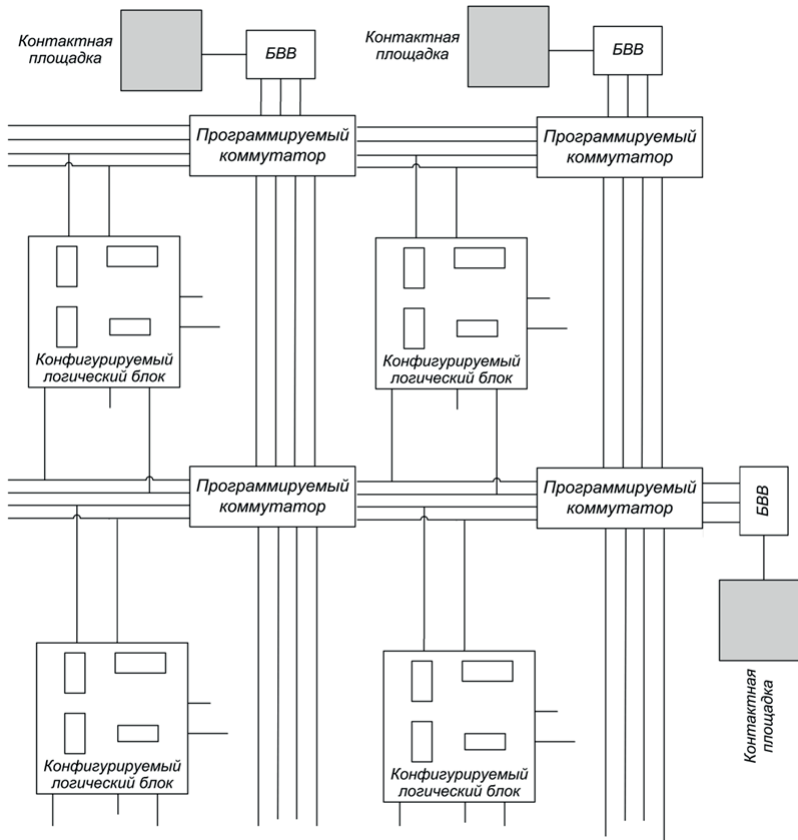


Рис. 8. Внутренняя структура FPGA

- внутренние программируемые связи, обеспечивающие управление путями соединения входов и выходов ПЛБ и блоков ввода-вывода на соответствующие сети;
- блоки ввода-вывода, которые обеспечивают связь между контактами корпуса и внутренними сигнальными линиями.

Программируемый (конфигурируемый) логический блок – это базовый элемент FPGA, реализующий простую логическую функцию (возможно, с сохранением результата) [23]. Термин CLB чаще

встречается в документации FPGA компании Xilinx; в компании Altera (Intel-FPGA) используется название Logic Array Block (LAB) – массив логики.

Сложность и структура ПЛБ определяются производителем. Если ПЛБ очень простой, потребуется огромное число программируемых связей, чтобы собрать требуемую схему. Если он очень сложный, теряется гибкость проектирования. Выбор структуры ПЛБ производителем FPGA – это всегда поиск компромисса по площади кристалла, по быстродействию, энергопотреблению и т. д. Обычно ПЛБ представляет собой что-то среднее: он достаточно сложен, чтобы в нем можно было реализовать логическую функцию нескольких переменных, но и довольно мал, чтобы была возможность разместить множество таких блоков внутри ПЛИС и гибко связать их в единую схему. ПЛБ может состоять из одного или нескольких базовых логических элементов (Basic Logic Element, BLE или просто Logic Element, LE). В FPGA обычно используются логические элементы на базе LUT [22, 24].

Lookup Table, LUT – справочная таблица, или таблица соответствия, хранится в виде набора масок (LUT-mask) в ячейках статической памяти с произвольным доступом (SRAM, static random access memory), где адрес – это аргумент, а содержимое ячейки – значение [10, 11, 24]. Произвольный доступ к памяти означает, что для записи/чтения есть возможность выбирать любой из битов и в произвольном порядке. Статическая память в отличие от динамической позволяет сохранять данные без регенерации, но является энергозависимой, т. е. она может сохранять данные без перезаписи только до того времени, пока есть питание. Подобную организацию мы уже рассматривали на примере простой ПЛИС на базе ППЗУ, но там вся логическая схема реализовывалась в виде таблицы, а в FPGA она реализована на уровне базового логического элемента, из которого будет строиться более сложный проект.

В первых FPGA использовались трехвходовые LUT (рис. 9).

Для построения LUT на k входов (k -LUT), которая реализует любую логическую функцию от k переменных, требуется SRAM емкостью 2^k бит и $2^k - 1$ мультиплексоров [24].

Представленная схема LUT позволяет реализовать функцию алгебры логики (ФАЛ), но является асинхронной и не выполняет функции хранения, а для реализации большинства логических устройств

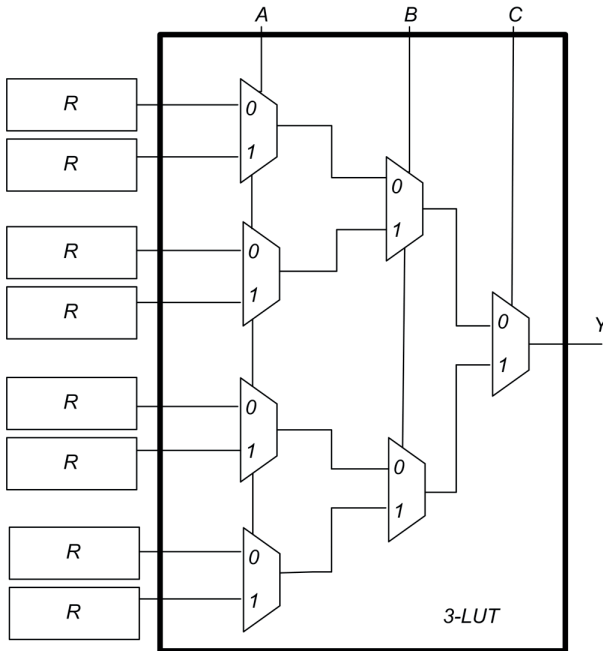


Рис. 9. Организация LUT

требуется синхронные схемы. Поэтому базовые логические элементы, помимо LUT, содержат триггер, управляемый синхросигналом, поступающим на соответствующий вход, и выходной мультиплексор. Управляющий сигнал выходного мультиплексора определяет, в каком режиме будет работать элемент: синхронном или асинхронном. Если требуется работа в качестве только комбинационного устройства, то выходной мультиплексор коммутирует выход элемента LUT на выход всей ячейки; если выход должен быть регистровым, то сигнал с LUT защелкивается по сигналу синхронизации в D-триггер, выход которого через мультиплексор соединяется с выходом элемента (рис. 10). Управляющий вход мультиплексора (на рис. 10 не показан) подключен к соответствующему биту конфигурационной памяти [23].

Важно, что при таком подходе можно достаточно точно спрогнозировать время прохождения сигнала и оно не будет зависеть от реа-

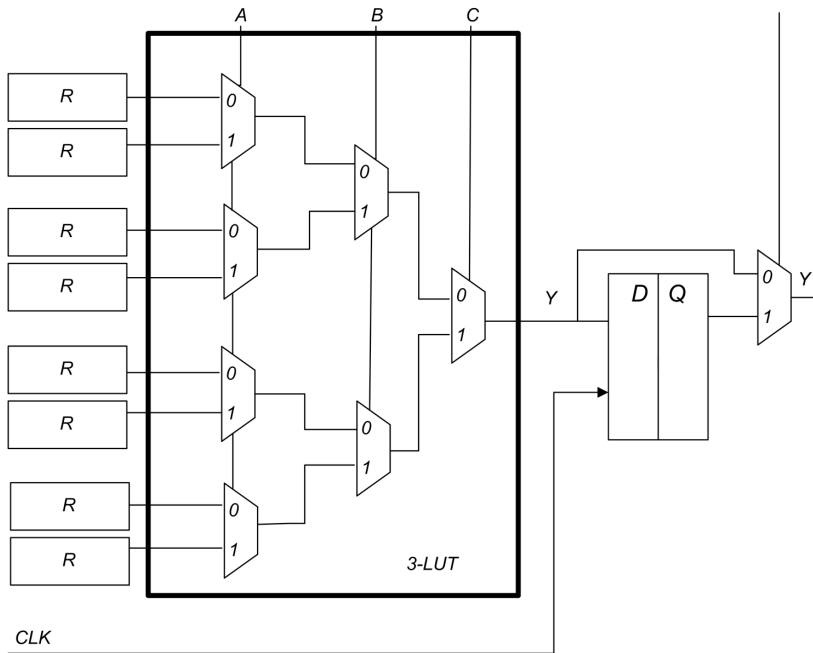


Рис. 10. Обобщенная структура логического элемента (LE) FPGA

лизуемой логической функции. Эта особенность позволяет достаточно точно выполнять временной анализ схемы.

В дальнейшем структура базовых логических элементов усложнялась. Сейчас реализация этих модулей отличается у различных фирм-производителей. Часть из них мы рассмотрим в следующих разделах. Но, несмотря на вариации, основная идея FPGA осталась без изменений.

1.4.1. Программируемая коммутирующая среда FPGA

Известны два способа организации внутренних связей FPGA: островной и иерархический.

Островная FPGA (island-style FPGA или mesh-based FPGA) называется так потому, что конфигурируемые блоки равны между собой и нахо-

дятся, как острова в океане, между узлами коммутации и линиями связи. Структура связей ПЛИС с островной организацией приведена на рис. 8. Типичный пример таких микросхем – это серии Altera Cyclone и Stratix.

Островная FPGA представляет собой регулярную структуру. Все каналы трассировки имеют одинаковую ширину (одинаковое количество проводников).

Входы расположены на отдельных сторонах логического блока, выходной контакт может трассироваться в двух каналах: либо справа от блока, либо снизу. Выходные контакты каждого логического блока могут соединяться с трассировочными сегментами в смежных каналах. Аналогично контактная площадка блока ввода-вывода может соединяться с трассировочным элементом в любом смежном канале. Например, верхняя контактная площадка чипа может соединяться с любым из W -проводников (где W – ширина канала) в горизонтальном канале, расположенном непосредственно под ним.

Как правило, трассировка FPGA несегментирована, т. е. каждый сегмент проводника соединяет только один логический блок с переключательным блоком. Из-за огибания программируемых переключателей в переключательном блоке трассировка получается более длинной.

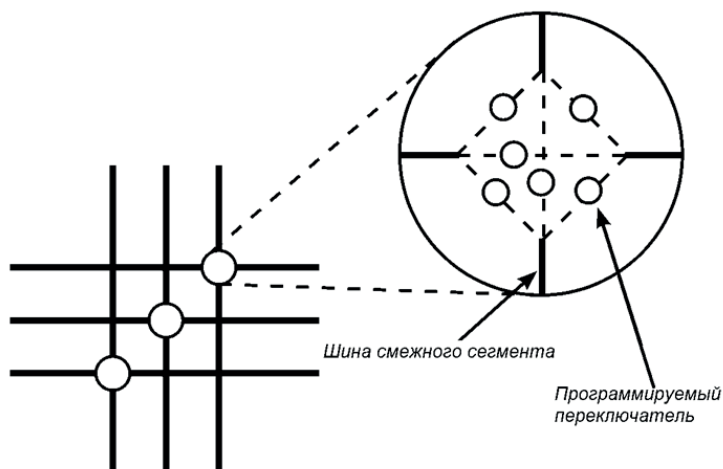


Рис. 11. Топология переключательного блока

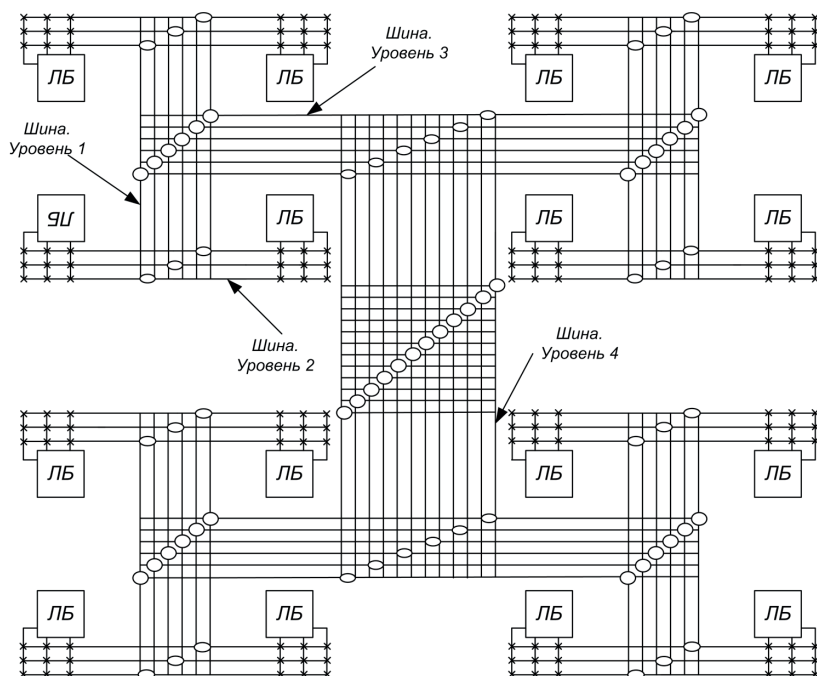


Рис. 12. Иерархическая система связей в ПЛИС

В месте пересечения вертикальных и горизонтальных каналов создаются переключательные блоки. На рис. 11 показаны соединения в переключательном блоке.

При такой архитектуре для каждого проводника, входящего в переключательный блок, существуют три программируемых переключателя, которые позволяют ему подключаться к трём другим проводникам в смежных сегментах канала. Модель, или топология, выключателей, используемая в этой архитектуре, называется планарной, или доменной, топологией переключательных блоков. В этой топологии проводник трассы № 1 подключается только к проводнику трассы № 1 в смежных каналах, проводник трассы № 2 подключается только к проводникам трассы № 2 и т. д.

Второй известный тип организации связей в FPGA – иерархический (рис. 12). В ПЛИС-проекте всегда есть фрагменты, которые взаимодействуют друг с другом более тесно, чем с отдаленными модулями

проекта. В FPGA с иерархической организацией обеспечить связь близлежащих CLB просто: для этого требуется немного коммутаторов, что обеспечивает высокое быстродействие. Но если нужен более крупный блок вычислителей, то сигнал должен выйти на более высокий уровень иерархии и потом зайти вглубь в соседний домен. Типичные представители иерархических ПЛИС – это микросхемы компании Altera серии Flex10K, APEX.

Нельзя сказать однозначно, какой из способов организации ПЛИС лучше, оба имеют свои плюсы и минусы. В настоящее время островная организация является более распространенной.

1.4.2. Способ хранения конфигурации FPGA

По способу хранения конфигурации FPGA можно разделить на следующие типы.

SRAM-Based. Это одна из самых распространенных разновидностей FPGA [24, 25]. Конфигурация FPGA хранится в ячейках статической памяти, изготовленной по стандартной технологии КМОП. Достоинство этой технологии – возможность многократного перепрограммирования FPGA. Недостатки – не самое высокое быстродействие, использование для хранения конфигурации (прошивки) энергозависимой памяти, из-за чего после включения питания прошивку нужно вновь загружать. Значит, на плате должен находиться программатор, специальная микросхема энергонезависимой памяти, обычно электрически стираемое перепрограммируемое ПЗУ (Electrically Erasable Programmable Read-Only Memory, EEPROM), чаще всего используется flash-память, или микроконтроллер, – все это удорожает конечное изделие.

Flash-based. В таких микросхемах хранение конфигурации происходит во внутренней памяти на базе технологии flash. Преимущество этого типа FPGA в том, что при выключении питания прошивка не пропадает. После подачи питания микросхема опять готова к работе. Однако у этого типа FPGA есть и свои недостатки. Для реализации flash-памяти внутри КМОП-микросхемы требуется совместить два разных технологических процесса, поэтому они значительно дороже. Кроме того, такие микросхемы, как правило, имеют ограниченное количество циклов перепрограммирования.

Antifuse. Специальная технология, по которой выполняются однократно программируемые FPGA [25]. Программирование такой FPGA заключается в расплавлении в нужных местах чипа специальных перемычек для образования нужной схемы. Недостаток – собственно программировать/прошивать чип можно только один раз. После этого исправить уже ничего нельзя. Процесс прошивки длительный и требует специального оборудования. Преимущества: FPGA этого типа быстродействующие, могут работать на больших частотах, меньше подвержены сбоям при воздействии радиации.

1.4.3. Специализированные блоки современных FPGA

Используя программируемые логические блоки ПЛИС и программируемые связи, можно создать практически любое цифровое устройство. Но при этом оно заведомо будет реализовано не оптимальным способом. При этом существуют функции и модули, которые требуются при реализации большинства ПЛИС-проектов: память, сумматоры, умножители, регистры с расширенной функциональностью (сдвиговые), стандартная логика ввода-вывода и пр. Можно выделить операции (и, соответственно, аппаратные модули для их реализации), востребованные в наиболее распространенных применениях ПЛИС.

Блоки ввода-вывода сигналов

FPGA находят широкое применение также благодаря тому, что имеют большое число блоков ввода-вывода сигналов, настраиваемых под различные стандарты электрического соединения микросхем. Блоки ввода-вывода (БВВ) обеспечивают интерфейс между выводами корпуса ПЛИС FPGA и ее внутренними логическими схемами. Каждому выводу корпуса придается блок ввода-вывода, который может быть конфигурирован как вход, выход или двунаправленный вывод [25, 27].

Встроенные блоки ОЗУ

В большинстве ПЛИС-проектов возникает необходимость использовать память для хранения обрабатываемых данных или состояний цифрового автомата. Поскольку реализации LUT фактически содержат ячейки памяти, их можно использовать и в этих целях, но это не будет оптимальным решением. Такая память будет иметь низкое быстродействие и будет занимать ресурсы ПЛИС, которые

нужны для обработки данных. Поэтому современные ПЛИС содержат блоки встроенной памяти (встроенное ОЗУ). В зависимости от архитектуры микросхемы эти блоки могут быть расположены по периметру кристалла, распределены по его площади, относительно изолированы друг от друга или организованы в столбцы, как показано на рис. 13 [9, 10, 27].

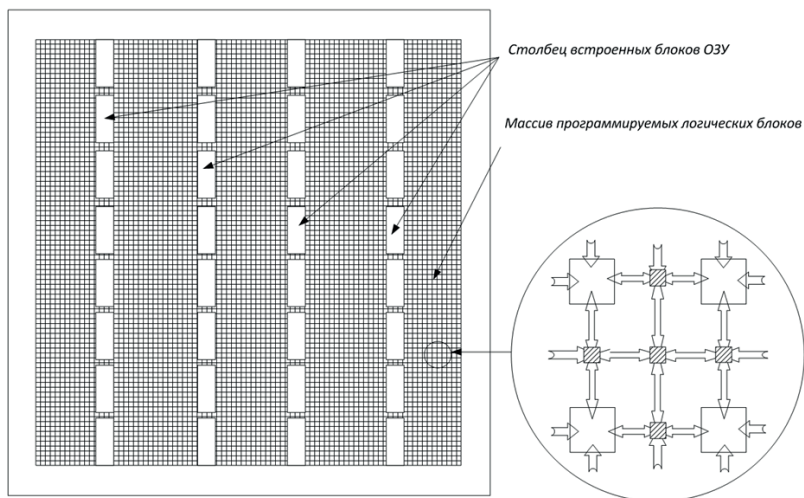


Рис. 13. Расположение встраиваемых блоков ОЗУ

В зависимости от устройства размер блоков ОЗУ может меняться от нескольких тысяч до нескольких десятков тысяч бит. Каждая микросхема может содержать от нескольких десятков до нескольких сотен таких блоков. Таким образом, полная ёмкость простирается от нескольких сотен тысяч бит до нескольких миллионов бит.

Каждый блок ОЗУ может либо использоваться как независимое запоминающее устройство, либо находиться в связке с несколькими блоками для реализации массивов памяти большого объёма. Блоки могут использоваться для различных целей, например как стандартные одно- и двухпортовые ОЗУ, очереди FIFO (first-in first-out – первый пришёл, первый вышел), конечные автоматы и т. д. [10, 27].

Встроенные умножители, сумматоры и блоки умножения с накоплением

Для цифровой обработки сигналов (ЦОС), некоторых моделей нейросетей и ряда других применений наиболее часто используемыми функциями являются операции многоразрядного сложения, умножения, совмещенная операция умножения-сложения (умножение с накоплением). Все эти функции можно реализовать, пользуясь стандартными возможностями ПЛИС. Но современные семейства FPGA включают уже готовые встроенные аппаратные модули, реализующие функциональность более высокого уровня.

Специализированные блоки обычно расположены в непосредственной близости от блоков встроенного ОЗУ, так как они часто используются вместе (рис. 14).

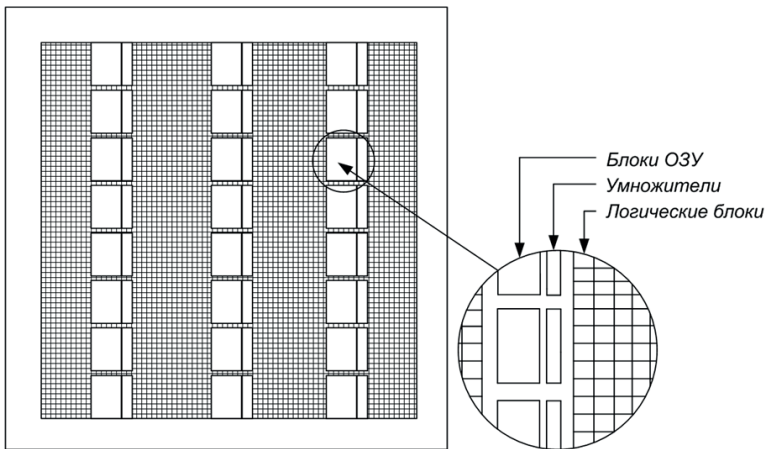


Рис. 14. Вид на кристалл со столбцами встроенных умножителей и блоков ОЗУ

Использование готовой аппаратной реализации этих функций позволяет сократить площадь кристалла и повысить быстродействие проекта по сравнению с реализацией их на базе логических блоков [10, 27].

Схемы ускоренного переноса

Разрядность информации, обрабатываемой в ПЛИС-проектах, может быть разной. Имея в распоряжении модули определенной разрядно-

сти, реализующие многие стандартные функции, например сумматоры или счетчики, можно строить схемы большей разрядности, соединяя их каскадом с использованием сигнала переноса. Но при этом быстродействие падает из-за того, что окончательный результат будет получен только после того, как отработают все ступени каскада и будут учтены все сигналы переноса. Современные ПЛИС содержат специальную логику и внутренние соединения, необходимые для реализации схем ускоренного переноса. Эта логика дополняется специальными внутренними соединениями между логическими ячейками в рамках каждого ПЛБ и между ПЛБ. Специальная логика быстрого переноса и выделенная маршрутизация повышают быстродействие при выполнении таких логических функций, как счётчики, сумматоры и т. п. [10, 22, 27].

1.4.4. Аппаратные и программные встроенные микропроцессорные ядра

При реализации сложной обработки данных ее все-таки лучше реализовывать на программно-управляемом устройстве, т. е. микропроцессоре. Но часто возникает ситуация, когда универсальный микропроцессор не обеспечивает нужную производительность из-за того, что многие сложные математические функции реализуются программно. В этом случае можно выбрать одно из следующих решений.

Можно разработать специализированный процессор, реализующий требуемую «тяжелую» функцию аппаратно. Но в результате будет получена специализированная микросхема, т. е. ASIC, со всеми вытекающими последствиями: долгим временем проектирования и высокой стоимостью. Для научных исследований или изготовления единичных экземпляров опытной аппаратуры этот вариант не подходит.

Ресурсов современных ПЛИС в большинстве случаев оказывается достаточно для того, чтобы реализовать процессор непосредственно с использованием программируемой логики. Такой процессор называется soft-процессором, или программным процессорным ядром. Это позволит решить описанную проблему, но при этом реализация процессора займет значительную часть ресурсов ПЛИС, а его быстродействие будет не слишком высоким, так как в качестве компонентов процессора в этом случае будут выступать программируемые ресурсы ПЛИС, гораз-

до более сложные и медленные, чем обычные вентили. Однако у этого решения есть преимущества. Например, можно реализовать ядро в точности требуемой разрядности и с требуемым набором функций, а также максимально использовать ресурсы ПЛИС, применяя, где это возможно, параллельную обработку информации. Для этого можно разместить в ПЛИС столько процессорных ядер, сколько позволят все совокупные ресурсы программируемых логических блоков.

Помимо этого в настоящее время выпускаются ПЛИС, которые содержат одно или несколько встроенных микропроцессорных ядер. Аппаратные микропроцессорные ядра изготавливаются как отдельные аппаратные блоки. Система команд и архитектура такого ядра могут быть как универсальными, так и ориентированными на определенную прикладную область (чаще всего ЦОС), а недостающие в системе команд «тяжелые» функции реализуются аппаратно на ресурсах программируемой логики ПЛИС.

Подобные ядра могут быть расположены в виде полосы вдоль одной из сторон главной структуры ПЛИС (рис. 15). При таком подходе

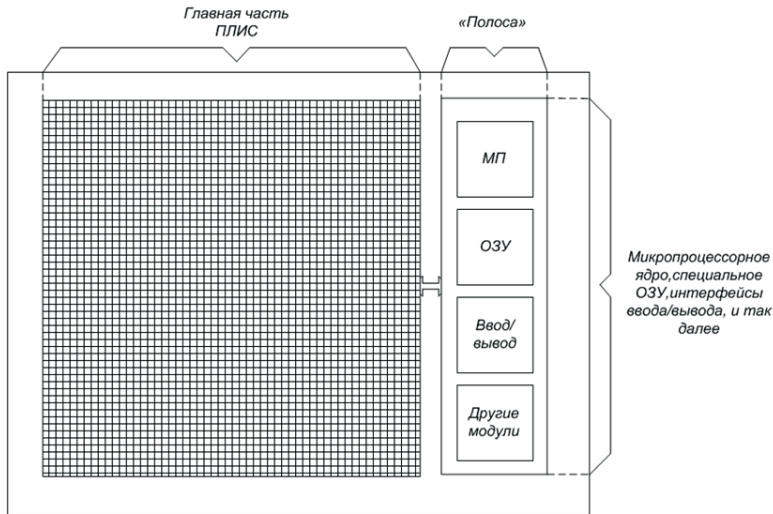


Рис. 15. Вид на кристалл со встроенным ядром, находящимся за пределами главной части

все компоненты обычно формируются на одном кремниевом кристалле [10, 17, 27].

Преимущество такой реализации проявляется в главной части ПЛИС, которая получается идентичной для устройств со встроенным и без встроенного микропроцессорного ядра. Это может существенно упростить работу инженеров со средствами разработки. Другое преимущество заключается в том, что поставщики ПЛИС могут связать все дополнительные функции, такие как память, устройства ввода-вывода и другие, в одну полосу для дополнения микропроцессорного ядра. Другим вариантом является размещение одного или более микропроцессорных ядер прямо в основной части ПЛИС.

Разработка и отладка проектов с использованием встроенного аппаратного или soft-процессорного ядра является задачей высокой сложности, поскольку требует совместной отладки и оптимизации аппаратной реализации и программного обеспечения с учетом жестких требований к быстродействию и задействованным ресурсам.

1.4.5. Дерево синхронизации и диспетчер синхронизации

Все синхронные элементы внутри ПЛИС, такие как триггеры ПЛБ, синхронизируются с помощью тактового сигнала. Тактовые сигналы обычно вырабатываются за пределами микросхемы и поступают на вход синхронизации, а затем распределяются через специальные устройства и подаются на соответствующие элементы. Для того чтобы разрабатываемое на базе ПЛИС устройство работало правильно и надежно, необходимо, чтобы сигнал синхронизации поступал на элементы схемы одновременно.

Дерево синхронизации

Если бы тактовые сигналы распространялись внутри ПЛИС по одному длинному проводнику, синхронизируя все триггеры поочередно, то за счет задержки распространения сигнала триггер, расположенный ближе к входу синхронизации микросхемы, получил бы синхроимпульс немного раньше, чем последние триггеры в этой цепочке. Подобная ситуация называется фазовым сдвигом, и она порождает целый ряд проблем [10, 28, 29].

Чтобы свести разницу во времени поступления синхросигнала в разные области ПЛИС, для передачи синхросигнала используют спе-

циальные линии передачи, отделенные от линий общего назначения, называемые «дерево синхронизации». Это название возникло потому, что главный синхросигнал разветвляется, подобно ветвям дерева; при этом триггеры могут рассматриваться как «листья» на концах «веток». Такая структура обеспечивает меньшую временную разницу в получении тактовых сигналов [28]. Упрощённое изображение дерева синхронизации приведено на рис. 16, программируемые логические блоки не показаны.

Принцип действия, рассмотренный выше, на самом деле сильно упрощён. На практике в микросхемах существует множество выводов синхронизации (неиспользуемые выводы синхронизации могут быть использованы как выходы общего назначения), а внутри устройства имеются множественные домены синхронизации (деревья синхронизации). И даже при использовании дерева синхронизации может возникнуть некоторый сдвиг фаз между регистрами, находящимися на одной «ветви», а также между «ветвями» [10, 28].

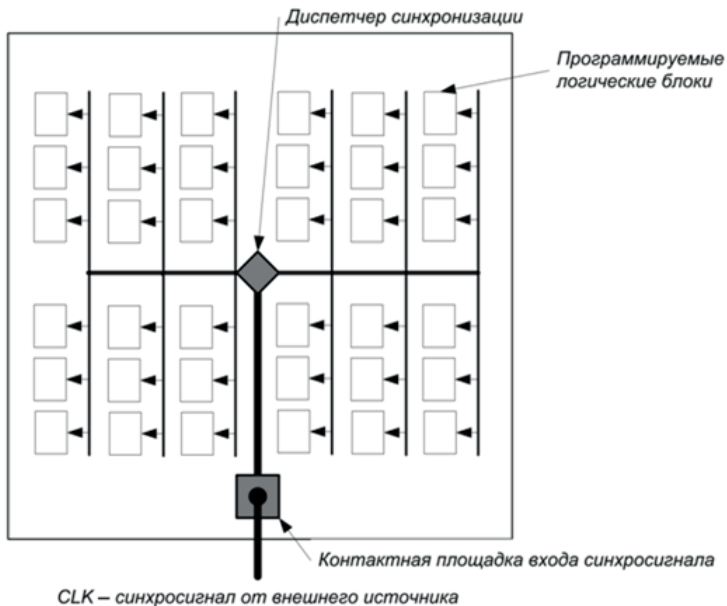


Рис. 16. Простое дерево синхронизации

Диспетчер синхронизации

Вывод синхронизации микросхемы может быть подключен к дереву синхронизации. Однако, как правило, этот вывод подключают не напрямую к дереву синхронизации, а к входу устройства (или блока) управления синхронизацией, называемого диспетчером синхронизации, который генерирует дочерние тактовые сигналы (рис. 17).

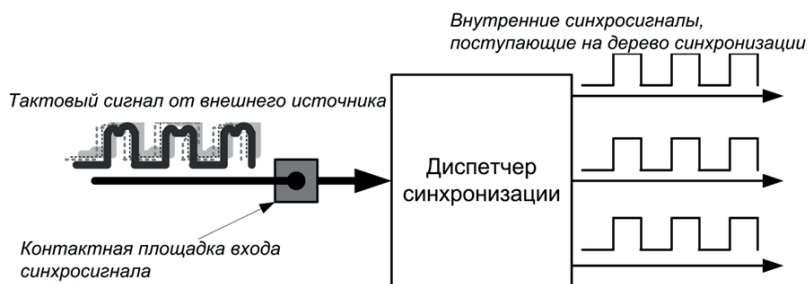


Рис. 17. Диспетчер синхронизации, генерирующий дочерние тактовые сигналы

Дочерние тактовые сигналы могут использоваться для управления внутренними деревьями (доменами) синхронизации или для выдачи сигналов на внешние выводы микросхемы, которые, в свою очередь, можно использовать для синхронизации других устройств, расположенных на печатной плате. Каждое семейство микросхем ПЛИС располагает собственным типом диспетчера синхронизации, и в одном устройстве может находиться множество модулей диспетчера синхронизации. Различные диспетчеры синхронизации могут поддерживать все или только некоторые из следующих функций: устранение флуктуаций, частотный синтез, фазовый сдвиг, автокоррекция сдвига фаз [10, 28].

1.5. Основные производители ПЛИС и их продукция

В настоящее время на рынке микросхем программируемой логики присутствует множество производителей, основными из которых долгое время являлись: Xilinx, Altera (Intel-FPGA), Lattice Semiconductor, Actel и Atmel. Продукция других фирм также находит свое применение,

но их доля невелика. Такая ситуация связана прежде всего с тем, что продукты Xilinx и Altera первыми появились на российском рынке и их применение сегодня во многом обусловлено наличием опыта работы как с самими микросхемами, так и со средствами разработки, традициями применения ПЛИС одного производителя в рамках одного предприятия и действительными и мнимыми рисками, связанными с переходом на новую элементную базу.

1.5.1. ПЛИС Altera (Intel FPGA)

Altera, одна из крупнейших фирм-разработчиц ASIC и программируемых логических интегральных схем (ПЛИС), была основана в 1983 году. Как предприятие без собственных производственных мощностей, Altera в первую очередь концентрировалась на разработке схем и модулей на основе таких языков описания аппаратуры, как VHDL, Verilog и собственный AHDL. Основные направления работы – программируемые микросхемы, а также услуги по преобразованию ПЛИС-проектов в базис ASIC для массового производства. В июне 2015 года было объявлено о приобретении корпорацией Intel компании Altera за \$16,7 млрд, и теперь эти изделия носят название Intel-FPGA.

Поскольку практическая часть курса будет выполняться на ПЛИС данной фирмы, рассмотрим их организацию более подробно.

В ПЛИС Intel-FPGA для хранения конфигурации используется конфигурационная память (Configuration RAM, CRAM). Эта память распределена по всему кристаллу; значения, записанные в нее, управляют внутренним коммутационным полем, определяя структуру синтезируемого цифрового устройства [11]. В большинстве микросхем Intel-FPGA эта память энергозависимая, и перед началом работы, а также в случае отключения питания требуется ее загрузка из какого-либо внешнего по отношению к ПЛИС носителя. Для этих целей используется устройство-программатор, которое может находиться на одной плате с ПЛИС либо быть реализованным в виде отдельного устройства.

Используя логический элемент, показанный на рис. 18, можно получить любую логическую функцию трех переменных и ее синхронный вариант на базе D-триггера [24]. А как быть, если нам понадобится реализовать какой-либо триггер, отличный от D-триггера? Для реализации некоторых

типов достаточно имеющегося комбинационного устройства на базе LUT на входе D-триггера, но для реализации всех типов триггеров в базовую схему вносят изменения. Во-первых, необходимо ввести обратную связь: для этого с выхода D-триггера подадим сигнал на один из входов LUT. Для того чтобы достигнутый ранее функционал не пострадал, мы не имеем права занимать уже имеющиеся входы; заменим LUT на четырехвходовый. Во-вторых, увеличим функциональность самого D-триггера, добавим дополнительные линии управления: вход разрешения ENA (от англ. “enable” – «разрешить») и вход асинхронного сброса aclr (от англ. “asynchronous clear” – «асинхронная очистка»). В результате получим схему, пригодную для синтеза любых триггеров (рис. 18).

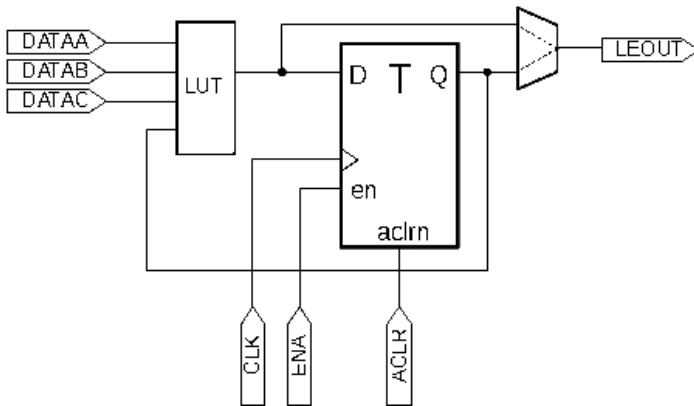


Рис. 18. Структура программируемого логического блока с обратной связью [24]

Существует обширный класс функций, использующих сигнал переноса для объединения малоразрядных реализаций с целью получения требуемой разрядности. Так как эти функции часто реализуются на ПЛИС, для работы с переносом в базовый логический элемент добавлена схема работы с переносом. Фактически при этом в ПЛБ добавляются еще одна LUT, ответственная за формирование сигнала переноса, и дополнительные входы для получения входного переноса из предыдущего блока и выхода для его передачи в другие логические блоки (рис. 19) [24].

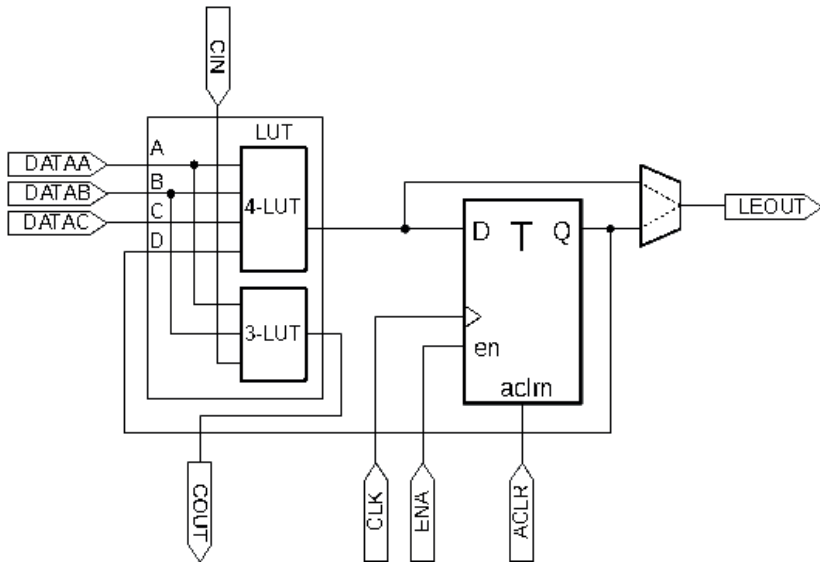


Рис. 19. Структура программируемого логического блока с каналом переноса [24]

Используя сигнал переноса, легко объединить ячейки для получения многоразрядных схем. Например, на рис. 20 показана схема 4-разрядного сумматора с последовательным переносом, построенная на четырех базовых логических элементах.

При разработке базовой логической ячейки решали две задачи: во-первых, синтезируемые устройства должны обладать максимальным быстродействием; во-вторых, использование ресурсов должно быть как можно более полным. В предыдущем примере, если требуется работа сумматора исключительно как комбинационной схемы, выходные мультиплексоры получают сигнал с выходов LUT, а вся цепочка триггеров окажется неиспользованной. В противоположность этому при синтезе последовательного или параллельного регистра все LUT будут по сути выполнять функцию проводника: соединять вход LE со входом D-триггера. Небольшое дополнение к схеме даст возможность при необходимости использовать комбинационное устройство и триггер элемента раздельно для синтеза независимых модулей (рис. 21) [24].

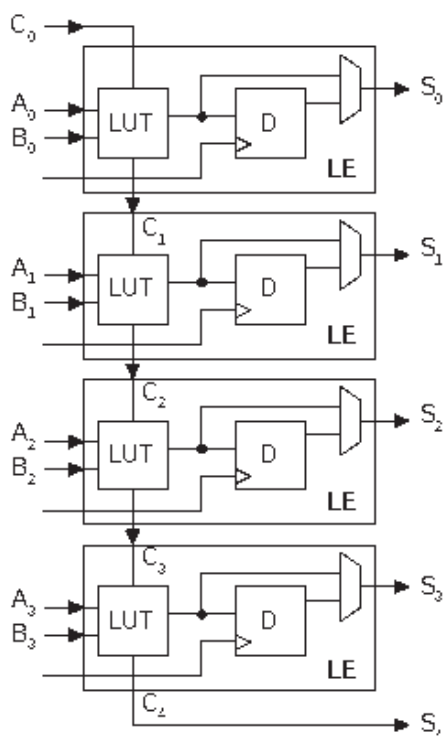


Рис. 20. Четырехразрядный сумматор [24]

Мультиплексор на входе триггера позволит выбирать источник сигнала: либо с входа DATAС, либо с выхода LUT. Кроме того, появляется возможность организовать дополнительный канал для соединения триггеров соседних LE с целью увеличения быстродействия при построении последовательных регистров. На рис. 21 показана схема получившегося логического элемента. Вход REGIN и выход REGOUT образуют выделенный канал для соединения триггеров; вход SLOAD (от англ. “synchronous loading” – «синхронная загрузка») управляет выбором источника сигнала для входа триггера [24].

Рассмотренный элемент LE в том или ином виде присутствует в различных семействах ПЛИС; в нем могут быть увеличены количе-

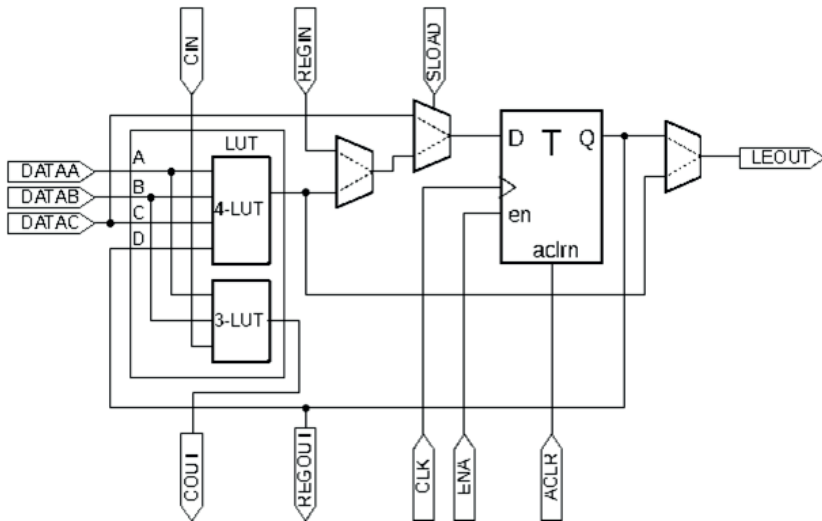


Рис. 21. Структура логического элемента (LE) с возможностью разделения LUT и триггера [24]

ство входов и сложность LUT, добавлены дополнительные соединения внутри LE и порты для интеграции в глобальную коммутационную сеть ПЛИС. Так, например, в семействе Stratix IV в один элемент входят две шестивходовые LUT, два выделенных полных сумматора и четыре триггера. На рис. 22 показана схема логического элемента ПЛИС Cyclone IV.

Так же, как и в базовом логическом элементе, каждый триггер имеет вход данных, вход тактовых импульсов CLK, вход разрешения ENA и вход асинхронного сброса (очистки) ACLR. Добавился сигнал синхронной очистки SCLR (Synchronous Clear): если на этом входе появится логическая единица, то следующим тактовым импульсом в триггер будет записан логический ноль. Все это позволяет настроить программируемый триггер каждой LE на работу в режимах D, T, JK или RS триггера [11, 24].

Комбинационное устройство (по сравнению с базовой схемой) получило некоторое усложнение. На входе C LUT мультиплексор выбирает источник сигнала, благодаря чему LUT может реализовывать логическую функцию четырех переменных; кроме того, в качестве переменной может быть использован флаг переноса или выход собственного триггера.

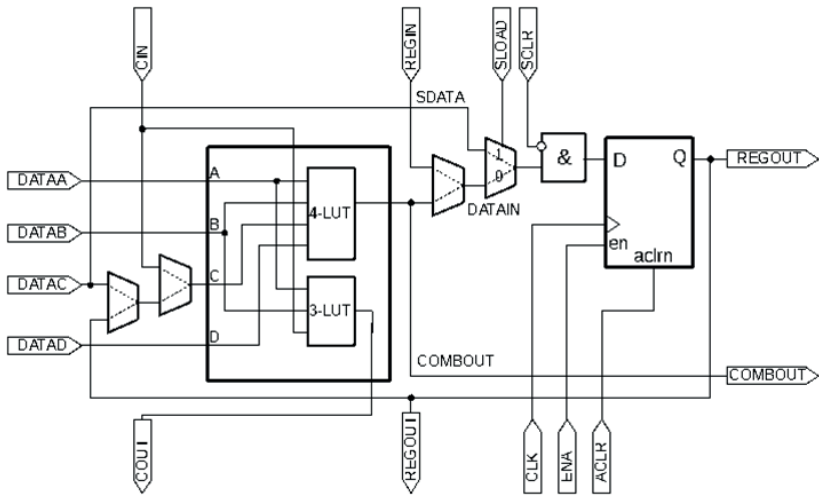


Рис. 22. Логический элемент (LE) Cyclone IV [24]

В Cyclone IV LUT может работать в двух режимах: нормальном и арифметическом. Средство синтеза при компиляции проекта автоматически выберет оптимальный режим для реализации требуемой функции. Нормальный режим предназначен для реализации основной логики и различных комбинационных функций. В этом режиме четыре входа LE (DATAA, DATAB, DATAC, DATAD) поступают на четыре входа LUT. Компилятор автоматически выбирает вход переноса CIN, вход DATAC или выход триггера (цепь обратной связи) в качестве одного из входов LUT. Арифметический режим подходит для синтеза сумматоров, счетчиков, аккумуляторов и компараторов (цепей сравнения). В этом режиме LUT представляет собой полный одноразрядный сумматор, включающий обработчик логики флага переполнения. Компилятор сам создает цепи переноса во время синтеза многоразрядных арифметических устройств [24, 29, 30].

1.5.2. ПЛИС фирмы Xilinx

Компания Xilinx с 1985 года выпускает ПЛИС, представляющие основные архитектуры, имеющиеся в микроэлектронике (CPLD

и FPGA). Xilinx, как и большинство производителей ПЛИС, является fabless-компанией, т. е. не имеет собственного производства, а заказывает изготовление микросхем по разработанным проектам сторонним компаниям [31]. Соответственно по мере освоения новых технологических процессов мировыми производителями Xilinx имеет возможность выпустить новое поколение ПЛИС с использованием освоенного технологического процесса. Это и наблюдалось на протяжении существования компании.

В конце 90-х годов XX века основной линейкой FPGA среди продукции Xilinx была серия XC4000. Архитектура XC4000 оказалась довольно удачной, но в целом ряде проектов возможности стандартной логической ячейки ПЛИС используются далеко не полностью. Возникла естественная альтернатива: использовать дорогие устройства с максимальной производительностью и полным набором функциональных возможностей или, напротив, дешевые ПЛИС, обеспечивающие базовые функции, требующиеся для большинства типовых проектов. В результате в номенклатуре изделий Xilinx появились различные серии FPGA, отвечающие упомянутым альтернативам. Стандартное недорогое решение представляла серия Spartan, на смену которой пришли серии Artix (минимальная стоимость) и Kintex (умеренная стоимость при высокой производительности в задачах цифровой обработки сигналов), тогда как наилучшие технические характеристики достигаются в серии Virtex. На данный момент наиболее современными являются семейства серии 7 – Artix-7 (технологический процесс 28 нм), Kintex-UltraScale и Virtex-UltraScale (технологический процесс 20 нм), UltraScale+ (технологический процесс 16 нм).

Кроме того, Xilinx выпускает микросхемы с архитектурой СПЛИС и средства загрузки конфигурации – микросхемы флэш-памяти и контроллер CF-карт [32].

1.5.3. ПЛИС фирмы Lattice

Компания Lattice Semiconductor была образована в 1983 году как разработчик без собственных производственных мощностей. Головной офис компании находится в г. Хиллсборо, штат Орегона, США. Ввиду того, что Lattice Semiconductor не располагает собственными производ-

ственными мощностями, микросхемы изготавливаются на заводах и по технологическим нормам Fujitsu [16, 33]. В сравнении с конкурентами это является большим преимуществом. Например 90-нанометровая технология Fujitsu позволяет сделать на один слой металлизации больше, чем технологии заводов TSMC или UMC, на которых изготавливаются микросхемы Altera и Xilinx соответственно [33].

В 1986 году Lattice Semiconductor вышла на рынок PLD малой плотности. В 1991 году компания Lattice Semiconductor совершила революцию на рынке ПЛИС, представив первую СПЛИС большой плотности с возможностью внутрисхемного программирования. Сегодня все продукты компании, базирующиеся на технологии E2 или Flash, могут программироваться и перепрограммироваться в системе. В 1999 году Lattice Semiconductor приобрела компанию Vantis, до этого входившую в состав AMD и специализировавшуюся на разработке программируемой логики для коммуникационных приложений, а в 2002 году – отделение компании Agere Systems, занимавшееся разработкой FPGA. Это позволило компании Lattice Semiconductor в том же 2002 году представить новые продукты на рынке FPGA. В настоящее время Lattice Semiconductor – это серьезный, набирающий мощь конкурент на рынке для таких лидирующих фирм, как Intel и Xilinx.

Lattice Semiconductor Corporation в настоящее время является одним из мировых лидеров в производстве микросхем программируемой логики: устройств CPLD и FPGA и сопутствующего программного обеспечения. Отличием FPGA фирмы Lattice Semiconductor является то, что конфигурационная память встроена внутрь кристалла FPGA, что позволяет сократить число связей на плате микросхемы, снизить уровень помех и увеличить надежность изделия. При этом микросхемы обладают свойством внутрисхемной программируемости и являются энергонезависимыми. Число циклов перепрограммирования неограниченно. Также стоит отметить низкое электропотребление микросхем. В мае 2019 года компания анонсировала новую серию ПЛИС на базе MachXO3 – MachXO3D с интегрированным криптомодулем и двойным блоком внутренней flash-памяти до 2700 кбит. Чип представлен в исполнении на 4300 и 9400 логических элементов в QFN- и BGA-корпусах. FPGA способна обеспечить безопасность и защиту от любого рода угроз на протяжении всего жизненного цикла продукта [33].

Для разработки приложений на базе микросхем Lattice Semiconductor существует программный пакет ispLEVER. Бесплатные версии ispLEVER Starter и ispLEVER Classic можно получить на сайте Lattice Semiconductor (www.latticesemi.com). Следует отметить, что программный пакет ispLEVER позволяет использовать любые современные языки и средства разработки устройств на базе ПЛИС начиная от визуальной схематической разработки и заканчивая языками описания аппаратуры высокого уровня; ispLEVER позволяет выполнить весь цикл разработки устройств на базе ПЛИС Lattice Semiconductor, включая отладку и моделирование работы системы (в полные версии входит пакет ModelSim от Mentor Graphics, являющийся стандартом де-факто для моделирования работы сложных систем). Для FPGA Lattice Semiconductor существует большой набор IP-ядер и от Lattice Semiconductor, и от других производителей [16, 33].

Вопросы и задания

1. В чем состоит принцип программного управления, описанный фон Нейманом?
2. Какая единица служит для оценки сложности цифровых СБИС?
3. В чем состоит избыточность программно-управляемых устройств?
4. Из каких этапов состоит выполнение команды процессора в программно-управляемых устройствах?
5. Дайте определение специализированной (заказной) интегральной микросхемы (Application Specific Integrated Circuit, ASIC).
6. Дайте определение программируемой логической интегральной микросхемы.
7. В чем отличие программирования микропроцессора и программирования ПЛИС?
8. За счет чего ПЛИС обеспечивают более высокое быстродействие, чем программно-управляемые устройства?
9. Перечислите основные области применения ПЛИС.
10. Перечислите основные классы ПЛИС.

11. Опишите особенности архитектур ПЛИС различных классов, их преимущества и недостатки.
12. Опишите архитектуру и перечислите основные блоки ПЛИС класса FPGA.
13. В чем состоят функции программируемого логического блока, LUT, программируемого коммутатора?
14. Опишите обобщенную структуру логического элемента (LE) FPGA.
15. Перечислите способы организации коммутирующей среды FPGA.
16. Перечислите и охарактеризуйте основные способы хранения конфигурации FPGA.
17. Какие специализированные блоки могут входить в состав современных ПЛИС / FPGA?
18. Что такое дерево синхронизации FPGA? Каковы требования к его организации?
19. Перечислите основных производителей ПЛИС, дайте краткую характеристику их продукции.
20. Опишите структуру логического элемента (LE) Cyclone IV (Intel FPGA), особенности его организации и назначение отдельных компонентов.

2. РАЗРАБОТКА ПЛИС-ПРОЕКТОВ. СИНТЕЗИРУЕМЫЕ ОПИСАНИЯ НА ЯЗЫКЕ VERILOG

2.1. Маршруты и технологии проектирования изделий на базе ПЛИС

Проектирование интегральных схем и создание ПЛИС-проектов имеют ряд общих черт, поэтому дадим краткую характеристику проектированию ИС как более общему процессу.

Проектирование – это комплекс работ, выполняемых с целью получения описаний нового или модернизируемого технического объекта, достаточных для реализации или изготовления объекта в заданных условиях. Комплекс проектных работ включает в себя теоретические и экспериментальные исследования, расчеты, конструирование [34].

Объектами проектирования могут быть изделия (например, обрабатывающий центр, двигатель внутреннего сгорания, ЭВМ) или процессы (например технологические, вычислительные).

Проектное решение – описание объекта или его составной части, достаточное для рассмотрения и принятия заключения об окончании проектирования или путях его продолжения.

Выделяют структурное и функциональное описание частей проекта.

Под структурным описанием объекта понимают перечень его элементов и способов их связи друг с другом.

Функциональное (поведенческое) описание объекта определяет процессы, протекающие в его частях (функции), и их логическую связь.

Важное значение имеет дополнение описаний параметрическими характеристиками объекта.

Параметр объекта – величина, характеризующая некоторое свойство объекта или режим его функционирования.

В описании проекта выделяют иерархические уровни (уровни абстрагирования). На верхнем иерархическом уровне рассматривается весь сложный объект как совокупность взаимодействующих подсистем.

На следующем уровне подсистем они рассматриваются отдельно как системы, состоящие из некоторых составных частей, и имеют большую подробность описаний.

Проектная процедура (процедура синтеза) – часть проектирования, заканчивающаяся получением проектного решения.

Примерами проектных процедур служат синтез функциональной схемы устройства, оптимизация параметров функционального узла, трассировка межсоединений на печатной плате и т. п.

Процесс проектирования разделяется на этапы.

Этап проектирования – это условно выделенная часть проектирования, сводящаяся к выполнению одной или нескольких проектных процедур, в результате чего вырабатывается проектное решение определенного иерархического уровня.

Различают нисходящее (сверху вниз) и восходящее (снизу вверх) проектирование.

При нисходящем проектировании задачи высоких иерархических уровней решаются прежде, чем задачи более низких иерархических уровней. Процесс разделения описания на составляющие блоки (декомпозиция) и поблочного их рассмотрения можно продолжить вплоть до получения описаний блоков, состоящих из базовых элементов. При восходящем проектировании последовательность противоположная.

Процедуры анализа заключаются в исследовании проектируемого объекта или его описания, направленном на получение полезной информации о свойствах объекта.

Часто задача анализа формулируется как задача установления соответствия двух различных описаний одного и того же объекта. При этом одно из описаний считается первичным, и его корректность предполагается установленной. Другое описание относится к более подробному уровню иерархии или к другому аспекту, и его правильность нужно установить сопоставлением с первичным описанием. Такое сопоставление называется верификацией [34].

Маршрутом проектирования (design flow) называется последовательность проектных процедур, ведущая к получению требуемых проектных решений. Это понятие стихийно возникло среди проектировщиков, а позже стало общепринятым [11, 34, 35].

Принципы построения маршрутов проектирования:

- разделение (декомпозиция) сложной задачи на более простые задачи синтеза промежуточных проектных решений;
- чередование процедур синтеза и верификации;
- неоднократный возврат на предыдущие этапы для коррекции принятых решений и устранения ошибок (итерационность процесса);
- повышение тщательности тестирования по мере приближения к окончательному проектному решению.

Декомпозиция сложной задачи синтеза на ряд простых выполняется в соответствии с блочно-иерархическим подходом к проектированию. Это позволяет организовать параллельно-последовательное выполнение проектных процедур коллективом разработчиков.

Чередование процедур синтеза и верификации необходимо потому, что для сложных задач проектирования нет методов, обеспечивающих сразу безошибочное получение проектных решений. Основные решения принимает человек на основе эвристических приемов, и полностью избежать ошибок невозможно. Результаты предложенных при синтезе проектных решений обязательно контролируются верификацией.

Поскольку на любой стадии или этапе проектирования может быть обнаружена ошибочность или неоптимальность ранее принятых решений, может возникнуть необходимость возврата на более ранние этапы проектирования для их пересмотра. Подобные возвраты характерны для проектирования и обуславливают его итерационный характер, т. е. повторяются неоднократно [35].

Усиление тщательности анализа по мере приближения к окончательному решению выражается также в том, что проверка производится по все большему числу показателей, оговариваемых в ТЗ, зачастую с учетом статистического характера параметров и нестабильности внешних условий. Это не означает, что на начальных этапах проектирования не следует проводить тщательное тестирование, напротив: цена исправления ошибки возрастает по мере приближения к финалу проектирования, и чем раньше она будет обнаружена, тем лучше. Но многие тесты невозможно выполнить до того, как будут приняты проектные решения нижнего уровня.

Известны следующие основные маршруты проектирования интегральных схем: традиционный маршрут нисходящего проектирования,

маршрут с предварительной абстракцией проекта на системном уровне и узкоспециализированные технологии восходящего проектирования. В данном курсе мы будем рассматривать традиционный маршрут проектирования как наиболее распространенный.

В традиционном маршруте проектирования интегральных схем выделяют следующие уровни представления проекта (рис. 23) [34, 35, 36]:

- системный уровень;
- автоматный (поведенческий) уровень;
- логический (схемный) уровень;
- транзисторный уровень;
- уровень топологии кристалла.

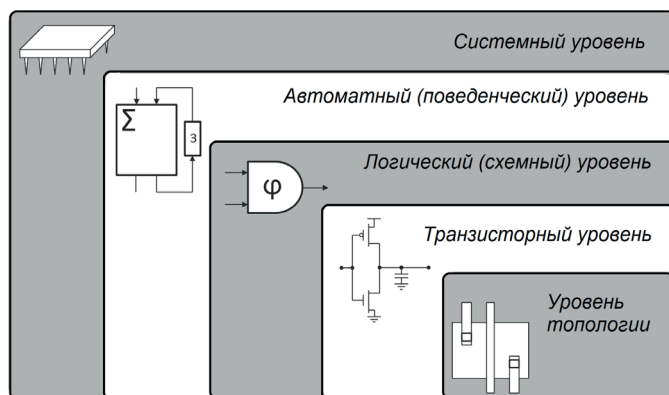


Рис. 23. Уровни представления проекта при проектировании ИС

При таком подходе изначально разрабатывается техническое задание на проектирование, в том числе формируется описание (спецификация) проекта.

Как правило, ТЗ является смесью словесного и технического описания. Задача первого этапа проектирования – переход от технического задания к формализованному описанию проектируемого устройства. В результате определяются перечень основных блоков устройства (или алгоритма) и их связи и/или взаимодействия. Возможным вариантом для достаточно сложных систем будет разумное совмещение и поведенческого, и структурного разбиения проекта. На системном (архи-

тектурном) этапе проектирования СБИС определяются требования, предъявляемые к проекту в целом и к его отдельным модулям, различные ограничения (по потребляемой энергии, временным параметрам, массо-габаритные и т. д.), создается системная спецификация проекта, разрабатываются алгоритмы, реализуемые в системе. Еще одна важная задача этапа – определение интерфейсных функций проекта и протоколов внешнего обмена. Именно требуемые временные характеристики и правила взаимодействия с внешними приборами определяют допустимую организацию и структуру внутренних узлов проекта.

В завершение системного этапа проектирования выполняется блочный синтез, при котором составляется общая архитектура построения СБИС, выделяются обрабатывающие и управляющие блоки. На основании анализа требований ТЗ принимаются решения по распараллеливанию и/или конвейеризации обработки данных. Далее происходит переход к структуре будущей системы посредством компоновки и верификации модулей в единый комплекс.

На автоматном (функциональном) уровне, называемом также уровнем регистровых передач (Register Transfer Level, RTL), схема описывается в виде регистров и цепей межрегистровых передач, включающих комбинационные схемы. С помощью моделей уровня RTL проверяется соответствие полученных укрупненных функциональных схем выбранным ранее принципам действия и структурным схемам с учетом временных параметров, а также выполняется верификация принятых алгоритмов функционирования.

На логическом (схемном, вентиляльном) уровне (Gate Level) детализация описаний схем доходит до отдельных логических элементов, таких как элементы «И», «ИЛИ», «ИЛИ-НЕ», триггеры. С помощью моделей логического уровня проверяется корректность разрабатываемых схем, т. е. их соответствие описаниям цифрового устройства, полученным на предыдущих иерархических уровнях, как по функциям, так и по временным параметрам. Моделям логического уровня отводится важная роль в процедурах синтеза тестов, где с их помощью моделируются неисправности аппаратуры [34, 35]. В современных системах структурного синтеза на функционально-логическом уровне стремятся получить не просто работоспособное решение, но решение с оптимальным использованием площади кристалла, быстродействием, рассеиваемой мощностью.

Следующий уровень представляет переход от логической схемы к схеме электрической принципиальной, когда проект описывается на уровне полупроводниковых компонент и соединений между ними.

Уровень топологии представляет собой топологическую схему размещения компонентов разрабатываемой интегральной схемы на кристалле и завершается созданием фотошаблонов и технологической цепочки изготовления разработанного изделия.

Корректность результатов синтеза проверяется с помощью программ верификации функциональных и логических схем. Верификация требуется после основных проектных операций синтеза и реализуется программами, ориентированными соответственно на уровни системный (архитектурный), регистровых передач или вентильный. В большинстве случаев верификация выполняется путем моделирования. Поскольку в итерационном цикле проектирования моделирование должно выполняться многократно, требования к быстродействию программ моделирования предъявляются весьма жесткие.

После этапа топологического проектирования моделирование повторяется уже с учетом уточненных задержек, обусловленных паразитными параметрами межсоединений. Для определения значений параметров схемы, получившихся после топологического проектирования, используют специальные программы уточнения задержек (экстракция параметров). Для этих целей возможно применение и программ схемотехнического моделирования.

Общие принципы создания ПЛИС-проекта в основном совпадают с принципами проектирования интегральных схем. Отличие состоит в том, что ПЛИС уже содержит электронные компоненты, реализованные на транзисторном и топологическом уровне, и нет необходимости описывать их при проектировании. Поэтому ПЛИС-проект описывается в соответствии с трехуровневым представлением (рис. 24) на следующие уровни:

- системный уровень;
- автоматный (поведенческий) уровень;
- логический (схемный) уровень.

Основные задачи системного уровня разработки ПЛИС-проекта – выбор допустимых для реализации проекта элементов, определение связей между ними, настройка параметров (временных, представления и разрядности передаваемых/храняемых данных и т. д.). Примером мо-

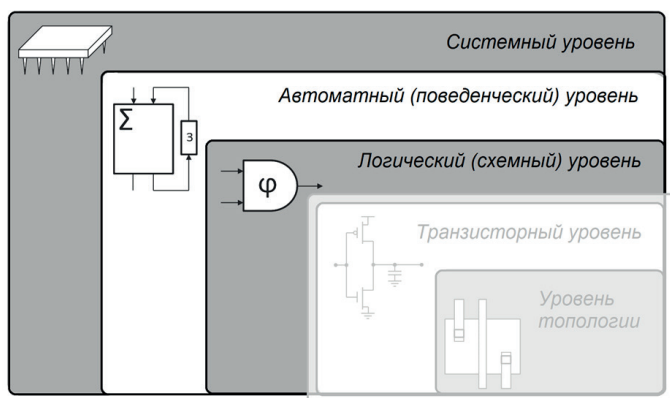


Рис. 24. Уровни представления ПЛИС-проекта

жет служить САПР Quartus (Intel / Altera), включающая в свой состав специальное средство, редактирующее проект на уровне блоков (block-level editing) [14, 18].

После формирования спецификаций для отдельных модулей и выбора семейства ПЛИС разрабатывается синтезируемое описание проектируемого устройства на уровне регистровых передач (RTL-описание). Оно может быть представлено либо в графическом виде, либо полностью на языке описания аппаратуры (HDL).

Главные достоинства графического способа – его наглядность и традиционность, так как большинство разработчиков воспринимают схемы именно в виде изображений. Конечно, эти преимущества проявляются только при правильном иерархическом и структурном разбиении проекта, когда число уровней иерархии, число блоков на каждом уровне иерархии не слишком велико (рекомендуется от 3 до 9 уровней и не более 7 блоков на каждом уровне). В большинстве случаев графическое представление не заменяет текстового представления, а только предваряет его.

Современные языки описания аппаратуры позволяют описать как структуру (перечень блоков и соединений между ними), так и поведение (алгоритм работы) отдельных блоков проектируемого устройства. Достоинства текстового способа описания проекта – более привычный вид для программистов, компактность, возможность автоматического пре-

образования в представление более низкого уровня (компиляция и синтез) [35–37].

Графическое представление проекта может быть преобразовано в описание на HDL. Корректно составленное описание на HDL в большинстве случаев также может быть преобразовано в графическое представление. На практике часто применяют следующий подход: на системном уровне разрабатываемую схему представляют в графическом виде как набор модулей и связей между ними, а входящие в него модули описывают на HDL. Такая организация проекта делает более наглядным и упрощает понимание, с одной стороны, взаимосвязей модулей всей иерархической структуры проекта, а с другой – алгоритмов функционирования отдельных модулей и устройства в целом.

Переход от регистрового (RTL) представления ПЛИС-проекта к логическому обычно осуществляется автоматически средствами САПР, подобно тому как программа на языке высокого уровня компилируется в исполняемый код. Часть модулей может быть описана на уровне логического представления вручную с целью оптимизации, так же как в программы на языке высокого уровня могут быть добавлены вставки на языке ассемблера; но так как этот процесс очень трудоемкий, ресурсы, доступные разработчику ПЛИС-проектов, постоянно растут, а средства компиляции и синтеза активно развиваются, потребность в этом постепенно сокращается.

Итоговым этапом проектирования на ПЛИС является подготовка загрузочного файла (прошивки), содержащего конфигурационную информацию для выбранной микросхемы ПЛИС. При этом обычно используется САПР фирм-производителей ПЛИС. Размещение и трассировка проекта могут быть выполнены как в автоматическом, так и в полуавтоматическом режимах.

Разработка современных ИС и ПЛИС-проектов, содержащих миллионы логических вентилях, возможна только с применением систем автоматизированного проектирования.

Система автоматизированного проектирования – автоматизированная организационно-техническая система, реализующая информационную технологию выполнения функций проектирования, состоящая из комплекса технических, программных и других средств автоматизации деятельности проектировщика.

САПР в области радиоэлектроники имеют примерно сорокалетнюю историю. Первые программы анализа электронных схем и проектирования печатных плат появились в начале 60-х годов прошлого века. Значительным стимулом для прогресса автоматизации проектирования в электронике (Electronics Computer Aided Design, ECAD) стали разработка и развитие технологии, схемотехники и системотехники интегральных схем [34]. В настоящее время на рынке интеллектуальных технологий имеется большое число программ ECAD, причем процесс обновления состава программного обеспечения в САПР ведущих фирм происходит весьма динамично. Современные интегрированные САПР СБИС состоят из большого числа программ, различающихся ориентацией на различные проектные процедуры и разные типы схем. Наиболее известными разработчиками интегрированных САПР являются фирмы Synopsys, Cadence Design Systems, Mentor Graphics. Наряду с ними отдельные программы или группы программ предлагают многие фирмы, работающие в области ECAD. Правильный выбор САПР предопределяет эффективность и успешность всей процедуры проектирования, сокращение промежутка времени от спецификации проекта до выпуска первых продажных образцов (Time-to-Market). Используемые методы и средства проектирования тесно связаны с выбором САПР, и, наоборот, выбор САПР определяет допустимые и целесообразные методы и средства проектирования, так что эти вопросы нельзя рассматривать в отрыве друг от друга [14].

Взаимосвязь между САПР и выбранным типом проектируемой ИС или базовой ПЛИС значительно более тесная, чем, например, между средствами проектирования микропроцессорных систем и используемым типом микропроцессора. Выбор элементной базы для проекта в значительной мере предопределяет требуемые для реализации проекта САПР [34, 38].

2.2. Языки описания аппаратуры: общие сведения

В традиционном маршруте проектирования ИС и ПЛИС-проектов могут применяться алгоритмические языки, например C/C++ или язык платформы MatLab, но они обычно используются на верхнем уровне

представления проектов. При этом задача эффективной компиляции программы на алгоритмическом языке в описание аппаратуры до настоящего времени не решена. Поэтому на более низком уровне, где система имеет привязку к конкретной архитектуре, используются специальные языки описания аппаратуры, которые уже упоминались ранее.

Язык описания аппаратуры (hardware description language, HDL) – специализированный формальный язык, используемый для описания структуры и поведения электронных (чаще всего цифровых) логических схем.

Языки описания аппаратуры по синтаксису могут быть похожи на языки программирования, например на С. Они включают выражения, операторы, управляющие структуры. HDL являются неотъемлемой частью САПР, и ИС, и ПЛИС. Для описания моделей аппаратуры, как правило, используется один из трех наиболее распространенных HDL: VHDL, Verilog и SystemC. Первые два языка известны с конца 80-х годов предыдущего столетия и сейчас являются наиболее популярными и распространенными среди разработчиков аппаратуры; SystemC появился сравнительно недавно.

На верхнем уровне иерархии все эти языки достаточно схожи: здесь модель аппаратуры представляется в виде набора взаимодействующих модулей, каждый из них имеет интерфейс и реализацию. При помощи интерфейсов описываются порты обмена данными и управляющими сигналами. Реализация модуля определяет его функционирование. В процессе компиляции и синтеза средствами САПР описание на HDL преобразуется во внутреннюю структуру модуля.

Помимо синтеза структуры модуля описание на HDL используется для моделирования работы аппаратуры. Поэтому важнейшим отличием HDL от обычных языков программирования является явное использование концепции времени. Поскольку временные характеристики разрабатываемой аппаратуры имеют, как правило, большое значение, при описании аппаратуры с использованием языков HDL одним из важнейших понятий является понятие времени (временных интервалов).

Язык VHDL (Very high speed integrated circuit Hardware Description Language – язык описания сверхскоростных БИС) был разработан международной группой по заданию Министерства обороны США

в 1983 году для формального описания логических схем на всех этапах разработки электронных систем: от модулей микросхем до крупных вычислительных систем. Стандарт IEEE 1076 на версию этого языка был утвержден в 1987 году [36, 37]. Работу над усовершенствованием стандарта ведет группа VASG (VHDL Analysis and Standardisation Group). До настоящего момента вышло 9 версий этого стандарта, закрепивших многие усовершенствования VHDL. Ведутся также работы по расширению VHDL в области описания аналоговой аппаратуры: VHDL-A, стандартизация внутренней формы представления VHDL-описаний в ЭВМ (группа VI-FASG), формы задания тестов для VHDL-моделей (группа WAVES), задания параметров задержек компонент (VITAL), алфавита представления значений сигналов в моделях и операции в этом алфавите (стандарт IEEE std_logic_1164) и т. д.

VHDL – это богатый строго типизованный и строго детерминистический язык. Проекты, написанные на VHDL, считаются самодокументированными. Однако его синтаксис сильно отличается от стиля языка C, а инженеры, работающие в VHDL, постоянно сталкиваются с необходимостью явного преобразования из одного типа данных в другой. VHDL имеет подчеркнуто однозначно недвусмысленную семантику и поэтому легче переносится между разными системами разработки (в том смысле, что точнее переносит все тонкости работы исходного проекта) [36–38].

Язык Verilog был разработан в 1985 году корпорацией США Gateway Design Automation и позднее утвержден институтом инженеров по электротехнике и радиоэлектронике (IEEE) – стандарт IEEE 1364 в 1994 году [10, 36, 39]. В отличие от VHDL, который строго типизирован, Verilog базируется на языке C, имеет меньше встроенных возможностей саморасширения, но зато более прост в реализации, использует более быстрый компилятор, имеет более развитый интерфейс с языком C и лаконичен, что позволяет уменьшить объем описаний схем примерно в полтора раза по сравнению с VHDL. Последней версией языка является Verilog 2005 и его надмножество System Verilog со многими новыми возможностями для верификации и моделирования.

Verilog синтаксически является довольно простым языком с небольшим количеством служебных слов. Именно его рекомендуют использовать в обучении, поскольку основные конструкции просты и до-

ступны. Verilog более лаконичный: в VHDL для описания какой-либо конструкции потребуется значительно больше символов, чем в Verilog. Именно поэтому в курсе ПЛИС в качестве основного языка описания аппаратуры выбран язык Verilog.

Основные достоинства этих языков следующие [10, 37–39]:

Стандартность. VHDL и Verilog официально признаны стандартом описания цифровой аппаратуры, который поддерживается военно-промышленным комплексом и радиоэлектронной промышленностью западных стран. Этот стандарт облегчает обмен документацией между отдельными группами разработчиков и специалистов по эксплуатации аппаратуры, различными системами автоматизации проектирования (САПР).

Многоаспектность и многоуровневость. Универсальное средство заменяет несколько специализированных. Языки VHDL и Verilog пригодны для описания как схем аппаратуры, так и функциональных тестов и алгоритмов функционирования. Они покрывают широкий диапазон уровней структурной детализации описаний цифровой аппаратуры:

- от описаний архитектуры ЭВМ на уровне устройств типа «процессор-память» до описаний узлов типа «триггер на уровне вентилях и МОП-ключей»;
- алгоритмов ЭВМ на уровне команд до описаний алгоритмов устройств на уровне межрегистровых передач и булевских функций;
- функциональных тестов до тестов проверки схем. На высших уровнях абстракции VHDL- и Verilog-описания можно рассматривать как средства спецификации требований к проекту.

Человеко-машинность. Документация пишется один раз, а читается многократно. Создатели VHDL и Verilog нашли довольно удачный компромисс между требованиями к языку как к средству документирования, удобному для восприятия человеком, и как к формальному средству, удобному для ввода и обработки описаний систем в ЭВМ. Описания аппаратуры на языках VHDL и Verilog пригодны для обработки такими компонентами САПР, как:

- подсистемы моделирования;
- подсистемы формальной верификации;

- reuse-checker (программы проверки стиля HDL-описаний на синтезируемость и пригодность к повторному использованию в различных проектах);

- подсистемы логического синтеза;
- подсистемы синтеза с учетом тестопригодности;
- системы синтеза и анализа контрольных тестов;
- временные анализаторы;
- кремниевые компиляторы;
- подсистемы автоматизации конструкторского проектирования и др.

Имея в своем распоряжении выполненные с учетом требований многократного использования HDL-описания ранее спроектированных устройств, с помощью САПР несложно, включая эти описания в состав новых проектов, повторно реализовать их на более современной технологии и т. д.

Стандартизация входных языков и внутренних интерфейсов подсистем САПР, в том числе и на базе HDL, создает общую коммуникационную среду для САПР, позволяет упростить стыковку продуктов различных фирм, обмен библиотеками моделей компонентов и проектов, модернизацию отдельных подсистем САПР.

Как и все универсальные средства, по отдельным аспектам уступающие специализированным, VHDL и Verilog представляются весьма громоздкими и избыточными, что затрудняет их реализацию и изучение в полном объеме. Естественно выделить подмножества языка для отдельных компонент САПР и групп пользователей. Например, известны подмножества HDL для синтеза, подмножества для построения схем, пригодных к тестированию и верификации и т. д.

Языки VHDL и Verilog, наиболее часто используемые для описания аппаратуры, часто рассматриваются как конкуренты. Преимущества и выбор того или иного языка являются поводом для многочисленных диспутов и дискуссий. Наиболее конструктивным итогом этих споров представляется следующая рекомендация: разработчику ИС и ПЛИС-проектов нужно знать в той или иной степени оба языка и их особенности и использовать в каждом конкретном случае тот, который ему удобнее. Тем более, что современные САПР в большинстве случаев поддерживают оба языка и даже допускают их смешение в рамках одного проекта [10, 29, 38–40].

2.3. Язык описания аппаратуры Verilog

Verilog появился в 1984 году как проприетарный язык моделирования аппаратного обеспечения компании Gateway Design Automation Inc [10, 11, 36]. Для моделирования аппаратуры на языке Verilog выполнялось структурное или поведенческое описание моделируемого устройства. Для удобства выполнения моделирования в отдельном модуле могла быть описана последовательность тестовых воздействий. Программа-симулятор на основании этих описаний, а также библиотек элементов, в которых описывались характеристики компонентов интегральных схем, выполняла моделирование.

Позднее возникла идея использования Verilog для разработки синтезируемых проектов. Для этого необходимо описать разрабатываемое устройство на языке описания аппаратуры и затем выполнить его синтез, т. е. переход к аппаратной реализации. Эта идея была реализована в 1987 году [29, 39, 40].

Задача синтеза схемы возлагается на соответствующий компонент САПР. Она является достаточно сложной, особенно если синтез должен осуществляться с учетом ограничений или требований оптимизации, поэтому требует значительных вычислительных ресурсов и может занимать много времени. Эта задача может решаться неоднозначно, т. е. результат синтеза может быть разным для разных средств синтеза и условий его выполнения. Например, из одного и того же описания можно получить разные реализации, задавая условие минимизации временных задержек или занимаемых ресурсов ПЛИС. Это позволяет рассмотреть различные варианты реализации устройства, определить узкие места и в итоге повысить качество проектного решения. Наконец, задача синтеза она может оказаться нерешаемой, т. е. не все описания могут быть синтезированы. Поскольку для использования HDL в проектировании ИС и ПЛИС важно уметь описывать устройства для их последующего синтеза, в языках описания аппаратуры выделяют синтезируемое подмножество.

Синтезируемое подмножество Verilog – набор конструкций, которые могут быть преобразованы в список связей цифровых компонентов и далее в топологическое представление интегральной микросхемы или в конфигурационный файл для программирования ПЛИС [29, 40, 41].

Как видно на рис. 25, несинтезируемое подмножество языка включает в себя в том числе и синтезируемые конструкции.

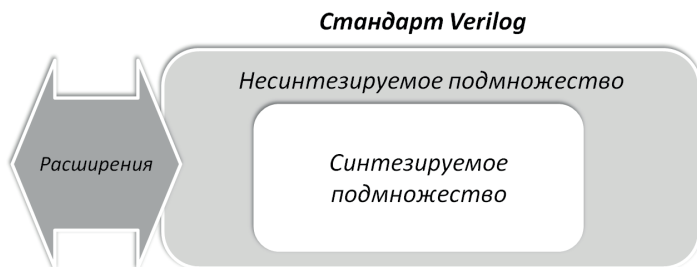


Рис. 25. Соотношение синтезируемого и несинтезируемого подмножеств языка Verilog

Такое смешение двух назначений одного инструмента (языка Verilog) вызывает значительные трудности при его освоении. В зависимости от контекста кода одни и те же конструкции могут рассматриваться в одном случае как синтезируемые, в другом – как несинтезируемые. Для упрощения понимания разницы между этими двумя трактовками в тексте будут регулярно встречаться повторяющиеся напоминания об особенностях использования HDL в целом и Verilog в частности. Это не невнимательность авторов, а стремление помочь читателю избежать досадных ошибок в освоении этого инструмента.

Прежде чем вести разговор о синтезируемом подмножестве языка, следует остановиться на отличиях синтеза от компиляторов поведенческих языков, например С. Компилятор заранее определенным образом переводит операторы языка высокого уровня в команды машинного языка. Синтез ПЛИС-проекта состоит в переводе описания в виде последовательных операторов языка HDL в схему. Поскольку разные ПЛИС имеют разную структуру, набор внутренних компонентов, временные характеристики, для выполнения синтеза нужно указать целевую ПЛИС и предоставить средству синтеза библиотеку элементов. Процесс синтеза больше похож на перебор вариантов и выбор наилучшего, удовлетворяющего временным ограничениям и требованиям к задействованным ресурсам ПЛИС.

Любой элемент языка может быть синтезирован, проигнорирован или вызвать ошибку [41].

Средствами синтеза поддерживаются:

- типы данных `wire` и `reg`, переменные типа `integer`;
- конструкция `assign`;
- процедурные блоки `always`;
- операции Verilog;
- оператор цикла `for`.

Средствами синтеза не поддерживаются:

- операции выборочного равенства и неравенства, так как в реальной схеме нет возможности сравнивать сигнал с неопределенным или третьим состоянием;
- операция возведения в степень, за исключением ситуации, когда степень равна двум;
- процедурный блок `initial`;
- операторы цикла `repeat`, `while`, `forever`;
- конструкции с такими ключевыми словами: `deassign`, `event`, `force`, `fork/join`, `primitive`, `real`, `release`, `time`, `wait`.

Средства синтеза игнорируют заданные явно временные задержки; поэтому разработчик должен использовать стиль описания, исключающий использование задержек, так как в противном случае результаты моделирования до и после синтеза могут отличаться.

К ошибкам может привести использование несинтезируемых конструкций (таких как блок `initial`, директивы компилятора и т. п.), нарушение стандарта языка Verilog, а также несоблюдение ряда рекомендаций, которые будут приведены в данном пособии.

Создавая описание синтезируемого ПЛИС-проекта, нужно иметь в виду еще одну важную особенность языков описания аппаратуры: в отличие от обычных алгоритмических языков, программы на которых исполняются последовательно в том порядке, как они были написаны программистом, HDL описывают процессы, протекающие в аппаратуре одновременно. Наиболее близки к HDL потоковые языки программирования. Изменения сигналов в цепях, переключение регистров, наступление тех или иных событий происходят постоянно, часто зависят не от того, в каком порядке они следуют в описании модуля, а от соединений модулей и последовательности подаваемых на схему входных воздействий.

2.4. Verilog: основные конструкции

2.4.1. Имена и комментарии

Приняты следующие соглашения об именах, применяемых пользователями в своих проектах:

- в именах нельзя использовать символ пробела;
- в качестве имен запрещено использовать зарезервированные слова языка Verilog;
- в именах можно использовать только цифры, буквы, символ подчеркивания « » и символ «\$»;
- имена могут начинаться с буквы или символа подчеркивания « »;
- Verilog отличает строчные и прописные буквы.

Других строгих требований к именам нет, но рекомендуется в рамках одного проекта, особенно при совместной разработке, обговорить и использовать одинаковый подход к назначению имен. В частности, желательно учесть следующие рекомендации:

- использовать одинаковые наименования для однотипных управляющих сигналов. Например, сигнал сброса может именоваться reset или CLR (от “clear” – очистка);
- если один из сигналов является инверсией другого, то рекомендуется давать им одинаковое имя с добавлением постфикса `_i` для инверсного сигнала. Например, data для прямого сигнала и data`_i` для инверсного;
- если в проекте происходят события, привязанные как к нарастающему, так и к падающему фронту сигнала, в именах рекомендуется использовать постфиксы, например `_p` (от “positive edge” – нарастающий фронт) и `_n` (от “negative edge” – падающий фронт), для отметки этой характеристики.

Для облегчения восприятия кода в его текст вводятся комментарии:

- два знака «`//`» используются для строчного комментария;
- набор знаков «`/*`» используется для начала блочного комментария, а «`*/`» – для конца комментария.

Пример 1. Имена переменных и комментарии

```
// это строчный комментарий,  
// он продолжается до конца строки  
/* это блочный комментарий */  
/* после него в строке*/  
/* могут быть другие конструкции*/ а = 1;  
  
// Примеры неверных имен переменных  
123q  
// начинается с цифры.  
abs@%11  
// содержит неразрешенные символы.  
assign  
// не может использоваться как имя, так как является  
// зарезервированным словом.  
  
// Примеры верных имен переменных  
q123  
// не начинается с цифры  
_abs$11  
// не содержит запрещенных символов  
a_assign  
// похоже, но не совпадает с зарезервированным словом  
/* три имени трех разных переменных */ abc, ABC, Abc
```

Эти соглашения действительны как для синтезируемого, так и для несинтезируемого подмножества языка.

2.4.2. Типы переменных

В языке Verilog используются переменные для хранения целых или действительных чисел, а также информации о временных отсчетах моделирования (модельном времени) [10, 11, 41].

Тип Integer используется для задания целых чисел, переменных цикла и т. д. Он является знаковым, т. е. подчиняющимся соответствующим правилам арифметики. Целочисленные переменные по умолчанию имеют разрядность 32.

Важный момент! Переменные этого типа – единственные, которые могут быть синтезированы. Все другие типы переменных являются несинтезируемыми; поэтому подробнее будут рассмотрены в соответствующем разделе.

Формат описания переменной типа `integer`:

```
integer <name> [= <value>];
```

где `value` – начальное значение переменной (может быть опущено).

Пример 2. Описание переменных типа `integer`

```
Integer Int;  
integer data = 255;
```

Для задания значений чисел в языке Verilog используется следующий формат:

```
[разрядность]'[s][основание]<значение> ,
```

где «разрядность» – разрядность числа в битах, представленная в десятичной системе счисления;

‘ – символ «апостроф»;

`s` – необязательный модификатор, указывающий на то, что число представлено в формате со знаком, для которого используется дополнительный код;

«основание» – модификатор основания системы счисления.

Допускаются следующие модификаторы:

`b` – двоичная система счисления;

`8` – восьмеричная система счисления;

`d` – десятичная система счисления (по умолчанию);

`h` – шестнадцатеричная система счисления;

«значение» – значение числа, записанное в системе счисления, установленной модификатором.

Пример 3. Различные способы записи чисел

```
8'b11110000 // двоичное 8-разрядное число 11110000  
16'hffff    // шестнадцатеричное 16-разрядное число ffff  
8'd123      // десятичное 8-разрядное число 123.
```

Явное указание разрядности позволяет правильно выбрать количество незначащих разрядов, заполняемых нулями (для беззнакового представления чисел), или позицию знакового разряда (для представления чисел со знаком). Например, десятичное число «123» формально может быть записано с использованием только 7 разрядов, поэтому модификатор 8' показывает, что в действительности следует использовать дополнительный, 8-й, разряд, поместив в него 0.

Если указанная в модификаторе разрядность числа больше, чем количество записанных разрядов, число автоматически расширяется до заданной разрядности незначащими нулями в старших разрядах:

```
6'b1111 // значение числа 001111
```

Если указанная в модификаторе разрядность числа меньше, чем количество записанных разрядов, соответствующее число старших разрядов отбрасывается:

```
4'b11000011 // значение числа 0011
```

Если переменная используется как синтезируемая, при синтезе разрядность определяется исходя из разрядности переменной или цепи, которой оно присваивается, и корректируется, о чем выдается предупреждение.

В записи числа можно использовать символы подчеркивания, которые игнорируются при анализе числа и служат для повышения читаемости текста.

2.4.3. Особые типы данных: цепи и регистры

В Verilog существуют два специфических для языков описания аппаратуры типа данных: цепи (net) и регистры (register). Они могут использоваться как в синтезируемых, так и в несинтезируемых описаниях. Ключевое различие между ними состоит в поведении [10, 21, 40, 41].

Цепи используются для представления реальных соединений в схеме (например, соединения между вентилями или подключенными модулями). Формат описания переменной типа «цепь»:

```
<net type> <name> [= <driver>];
```

где <net type> – вид цепи.

К цепям необходимо постоянно прилагать непрерывное воздействие, т. е. цепи не могут хранить присвоенную им величину. Цепь принимает значение выходной величины источника (драйвера), к которому обязательно должна быть подключена. Драйвером может быть как вычисляемое выражение, так и константное значение. Драйвер может быть назначен одним из двух способов:

- при объявлении переменной типа «цепь»;
- с использованием оператора непрерывного присваивания `assign` (будет рассмотрен позже).

В синтезируемых конструкциях используются два вида цепей: `wire` (провод) и `tri` (сеть с тремя состояниями).

По умолчанию переменная типа «цепь» имеет разрядность 1.

Основным типом цепей при проектировании цифровых систем на базе ПЛИС является `wire`.

Пример 4. Объявление цепи вида `wire` (провод)

```
wire reset = 1'b0; // назначение в качестве драйвера цепи
                  // константного значения
wire C=P & cin;    // назначение в качестве драйвера цепи
                  // выражения
```

Регистры используются для обозначения элементов, способных хранить данные, т. е. регистр сохраняет величину до тех пор, пока ей не будет присвоена другая величина.

Этот компонент объявляется с использованием ключевого слова `reg`. Это объект, который сохраняет записанное в него значение, и изменяет его только в явно определенные моменты.

Формат описания переменной типа «регистр»:

```
reg <name> [= <value>];
```

По умолчанию переменная типа «регистр» имеет разрядность 1.

Отличия переменных типа `integer` и `reg` состоят в описании их поведения. Значение целочисленной переменной может быть задано при объявлении, вычислено и изменено в выражениях. Значение регистра также может быть задано при объявлении и изменено в функциональных блоках `always`. Особенности применения конструкции `always` также

будут рассмотрены подробно далее, в разделе, посвященном описанию синхронной логики.

В синтезируемом проекте переменная типа `reg` не обязательно будет реализована с использованием триггера. Соответствующая синтезируемая схема может быть выполнена с использованием синхронной либо комбинаторной логики в зависимости от того, как именно описано присваивание новых значений такой переменной.

Основными значениями, которые могут быть присвоены цепям и регистрам в синтезируемых описаниях, являются следующие:

0 – логический ноль;

1 – логическая единица;

z – «третье состояние», высокий импеданс.

Может также применяться `tri` (для случаев, относящихся к аппаратно реализованным буферам с «третьим состоянием»). Начиная с технологических норм 90 нм, основные производители ПЛИС с архитектурой FPGA отказались от реализации аппаратных буферов с «третьим состоянием» для внутренних цепей. Использование буферов с тремя состояниями ограничивается выходными линиями FPGA. Сигналы с «третьим состоянием» могут быть объявлены для внутренних цепей ПЛИС и объединены с использованием схем разрешения одного из выходов, однако средства синтеза заменяют такие описания на эквивалентные узлы мультиплексируемой комбинаторной логики.

2.4.4. Многоразрядные переменные – шины

Обычное объявление переменной соответствует одноразрядному значению. Для хранения и передачи многоразрядных чисел или групп сигналов регистры и цепи объединяются в шины (`bus`) [10, 29, 40, 41]. Формат объявления многоразрядной переменной приведен в примере.

Пример 5. Объявление многоразрядной переменной

```
wire [63:0] address;  
// объявление 64-разрядной шины типа «цепь»  
reg [7:0] data = 255;  
// объявление 8-разрядной переменной типа «регистр»  
// и задание начального значения, равного 255
```

В этом примере объявляется 8-разрядная переменная `data`, разряды которой нумеруются от 7 до 0. При создании переменной присваивается десятичное значение 255.

Числа в скобках соответствуют индексу крайнего левого и крайнего правого разрядов шины. Поскольку обычно мы нумеруем что-либо по возрастанию слева направо, возникает вопрос: почему нумерация разрядов в примере идет в обратном порядке. Это объясняется тем, что в позиционном представлении числа старшие разряды располагаются слева. Соответственно в примере числа в скобках соответствуют старшему значащему разряду (Most Significant Bit, MSB) и младшему значащему разряду (Least Significant Bit, LSB). Допустима и форма записи, при которой первым указывается наименьший индекс:

```
reg [0:7] data = 8'b10000000;
```

Исходя из формата объявления присваивание константы $10\,000\,000_2$ будет производиться так, что единица в старшем, седьмом (крайнем левом в записи константы), разряде будет присвоена младшему нулевому (крайнему левому в объявлении индексов) разряду переменной `data`. Таким образом, хотя двоичная константа имеет значение 128_{10} , в результате инициализации переменная `data` примет значение 1.

Чтобы избежать подобной путаницы, приведенный в примере порядок нумерации рекомендуется в случае, когда шина используется для представления чисел. Если шина представляет собой группу сигналов, например управляющих, порядок нумерации может быть выбран произвольно.

2.5. Структура ПЛИС-проекта. Модуль

Модуль — это основная единица проекта Verilog независимо от того, является он синтезируемым или нет [29, 40].

В правильно структурированном проекте модуль реализует определенную функцию или подсистему разрабатываемой системы. Сложность модуля может быть различной.

Модуль должен быть описан, т. е. на используемом HDL должны быть представлены интерфейс модуля (перечень и характеристики портов ввода и вывода) и его функционал. Далее экземпляр описанного

модуля (англ. *instance*; в некоторых источниках на русском языке так же используются термины «инстанс» или «инстанция») может быть использован (установлен, инстанцирован) в проекте. Если по проекту требуется несколько модулей, выполняющих одну и ту же функцию и имеющих один и тот же перечень входных и выходных сигналов, то очевидно, что можно выполнить одно описание такого модуля и установить в проекте несколько его экземпляров.

Модуль обеспечивает требуемые функциональные возможности через интерфейс портов (вводы и выходы); при этом вся его «начинка» будет скрыта для модулей верхнего уровня. Это позволяет проектировщику гибко менять внутреннюю организацию модуля, не затрагивая остальную часть проекта.

Для тех, кто изучает данный курс, имея опыт программирования, полезной может оказаться аналогия между модульной организацией ПЛИС-проекта и объектно-ориентированным программированием. Если брать терминологию, принятую для языка C++, то модуль представляет собой конструкцию, во многом схожую с классом. Описание модуля по структуре совпадает с описанием класса. Входные и выходные порты модуля описывают интерфейс класса, а тело модуля – реализацию методов. Так же, как в программе возможно создание нескольких объектов одного класса, в ПЛИС-проекте создаются экземпляры модуля. Отчасти совпадает и иерархия этих конструкций. Однако модули не реализуют наследование и полиморфизм.

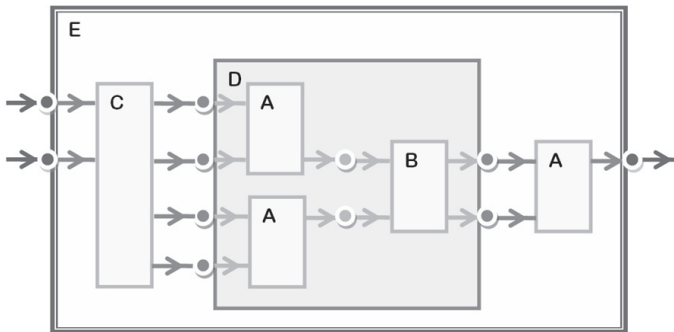
Проект же в целом представляет собой иерархическую (многоуровневую) структуру, т. е. модули могут входить в состав других модулей.

Базовые модули не включают в себя другие модули, т. е. находятся на нижнем уровне иерархии.

Субмодули находятся на промежуточном уровне, т. е. могут содержать модули нижних уровней иерархии и сами, в свою очередь, входить в модуль верхних уровней.

Головной модуль (*top level*) является модулем верхнего уровня, т. е. не входит ни в один модуль, а, напротив, включает в себя все модули проекта.

Структура проекта должна иметь только один модуль верхнего уровня. Рекомендуется, чтобы этот головной модуль содержал только перечень модулей и соединения между ними. Наиболее важной причиной нахождения логики вне корневого модуля является эффективный



A, B, C – базовые модули; D – submodule; E – головной (top-level) модуль

Рис. 26. Многоуровневая структура ПЛИС-проекта

синтез. Большинство средств САПР не сможет оптимизировать логику в submodule, если в головном модуле есть промежуточная логика. Исключение логики из головного модуля облегчает восприятие структуры проекта. Еще одной причиной исключения логики из корневого модуля является необходимость дать возможность программе-трассировщику высокого уровня создать список межсоединений.

В схеме на рис. 26 приведены три экземпляра модуля A. Два из них входят в состав submodule D, а один – в состав модуля верхнего уровня.

Структура описания модуля:

```
module <имя> ([параметры] <список портов>)
    <описание выходов и входов>
    <тело модуля>
endmodule
```

2.5.1. Порты в описании модуля

Порты обеспечивают взаимодействие модуля с другими компонентами проекта.

Существуют три вида портов:

- вход (input);
- выход (output);
- двунаправленный вывод (inout).

Порты могут иметь тип регистра или цепи; по умолчанию принят тип порта «цепь».

При выборе типов портов рекомендуется соблюдать следующие правила [10, 21, 40, 43]:

1) входы должны иметь тип `wire`. Это связано с тем, что они транслируют сигналы от внешних источников в данный порт и значение для входов должно быть известно и актуально в любой момент времени, что обеспечивается именно типом `wire`;

2) выходные сигналы могут иметь как тип `wire`, если они не предусматривают функцию хранения, так и `reg`. В последнем случае их значение может не изменяться при изменении входных сигналов, а выход будет удерживать состояние, присвоенное ему ранее, что возможно при использовании сигнала типа `reg`. По умолчанию все порты получают тип `wire`. При необходимости изменить тип на `reg` каждый сигнал должен быть упомянут отдельно;

3) двунаправленные порты (`inout`) внутри модуля могут быть только цепью (`wire`).

Для создания иерархических проектов, состоящих из нескольких модулей, требуется описать, каким образом они соединены друг с другом. Когда мы устанавливаем в проект экземпляр модуля, мы должны задать подключение его входных и выходных портов к источникам сигнала.

При подключении модулей рекомендуется соблюдать следующие правила:

1) сигналы на входы модуля могут поступать от источников любого типа: цепей, регистров, целочисленных переменных;

2) выходные сигналы должны подключаться к цепям, т. е. переменным типа `wire`;

3) двунаправленные порты (`inout`) подключаться извне должны только к цепям.

Если для соединения модулей требуются дополнительные цепи или регистры, они должны быть объявлены.

Пример 6. Описание модуля с выходными портами разного типа

```
module fig(  
    output reg [7:0] r_sum,  
    output [7:0] w_sum
```

```

input [7:0] a,b,
input clk,en,
);
// далее описано тело модуля
assign w_sum = a + b;

always @ (posedge clk)
    if (en == 1) r_sum = a + b;
endmodule

```

В теле модуля могут присутствовать следующие компоненты:

- объявления переменных;
- операторы непрерывного присваивания assign;
- установленные компоненты модулей (инстансы) более низкого уровня;
- блоки с поведенческим описанием always;
- задачи и функции.

Эти компоненты могут быть расположены в любом порядке и в любом месте в описании тела модуля. Если описываемый модуль предполагается в дальнейшем использовать для синтеза ПЛИС-проекта, то он должен содержать только синтезируемые конструкции.

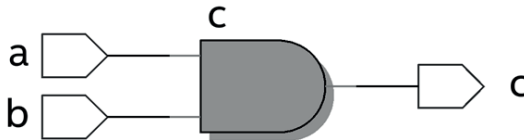


Рис. 27. Результат синтеза описания модуля А

При использовании в проекте модулей нижнего уровня следует помнить, что имена портов и переменных, объявленные внутри модуля нижнего уровня, работают только внутри этого модуля и не входят в пространство имен модулей верхнего уровня или других модулей нижнего уровня (см. пример на рис. 27).

Пример 7. Описания модуля А

```

module A (a, b, c); // начало описания модуля А
// описание типов портов
output c;

```

```
// выходной порт по умолчанию имеет тип wire
input a, b;
// два входных порта по умолчанию имеют тип wire

assign c = a & b;
// тело модуля, описывающее реализуемую функцию
endmodule
```

Уже на данном этапе, до изучения некоторых специальных конструкций языка, интуитивно понятно, что описанный модуль имеет три порта: a, b, c. Порты a и b являются входными, порт c – выходной, а модуль реализует функцию логического «И» над сигналами, поступающими на входы, и выдает результат в выходной порт.

2.5.2. Параметры в описании модуля

Параметры модуля используются для задания характеристик модуля, которые известны на момент компиляции, но могут отличаться для разных экземпляров модуля. Эти характеристики могут описывать характерные задержки, ширину шин, разрядность обрабатываемой информации или любой другой параметр, не изменяющийся во время симулирования модели [10, 29, 40, 41].

Использование параметров позволяет однократно описать модуль с характеристиками, определяемыми посредством параметров, и изменить значения параметров при установке модуля, тем самым получив экземпляры модуля (инстансы) с различными параметрами, например разрядностью обрабатываемых данных.

Назначения параметров для модулей большого проекта могут быть заданы для всего проекта в целом и сохранены в отдельном файле. Это позволит избежать ошибок, связанных с несогласованным заданием параметров.

Пример 8. Использование параметров при описании модуля

```
module nop #(parameter BIT = 4)
// описываем параметр и задаем его значение
(
    output [BIT-1:0] c,
```

```
// определяем через параметр все зависимые величины
output flag,
input [BIT-1:0] a,
input [BIT-1:0] b
);
assign c = 0;
assign flag = 0;
endmodule
```

В примерах параметры чаще всего будут использоваться для задания разрядности обрабатываемых данных и, соответственно, разрядности шин для их передачи и регистров для их хранения.

Далее будут рассмотрены возможные варианты описания модуля и используемые для этих целей конструкции.

2.5.3. Создание экземпляра модуля – инстанса

Создание экземпляра описанного модуля в модуле более высокого уровня имеет следующий синтаксис:

```
<имя модуля> ([ #( <список назначения параметров> ] ) <имя
экземпляра модуля>
(<список подключения портов модуля>);
```

Для создания иерархических проектов, состоящих из нескольких модулей, требуется описать, каким образом они соединены друг с другом. Когда мы устанавливаем в проект экземпляр модуля, мы должны задать подключение его входных и выходных портов к источникам сигнала.

При подключении модулей рекомендуется соблюдать следующие правила:

- 1) сигналы на входы модуля могут поступать от источников любого типа: цепей, регистров, целочисленных переменных;
- 2) выходные сигналы должны подключаться к цепям, т. е. переменным типа wire;
- 3) двунаправленные порты (inout) подключаться извне должны только к цепям.

2.6. Структурное описание модулей

Verilog позволяет выполнять как структурное, так и функциональное (поведенческое) описание модулей.

Структурное описание представляет собой набор компонентов (примитивов, библиотечных модулей, модулей, определенных пользователем) и связей между ними.

Оно фактически является еще одним (языковым) способом описания схемы устройства и может рассматриваться как список соединений (netlist), в котором представлены установленные в проекте модули, порты которых между собой связаны проводами. Такой способ описания системы практически не отличается от обычного (графического) и не предоставляет дополнительных возможностей для проектирования.

На самом нижнем уровне структурное описание может включать логические примитивы языка Verilog. Это встроенные в язык модули, реализующие булевы функции (табл. 1). При использовании одного или нескольких экземпляров примитива в модуле разработчика каждый экземпляр должен быть установлен (инстанцирован). При этом указываются имя примитива, имя экземпляра и список имен внешних цепей, подключаемых к портам этого экземпляра. Положение выходных и входных портов определяется порядком их следования в списке портов (табл. 1). Если для соединения примитивов или модулей требуются дополнительные цепи или регистры, они должны быть объявлены.

Таблица 1

Примитивы логических функций языка Verilog

Имя примитива	Функция	Порядок и количество портов
not	Инверсия	(1 – output, 1 – input)
and	И (конъюнкция)	(1 – output, 2 или более – input)
nand	И-НЕ	(1 – output, 2 или более – input)
or	ИЛИ (дизъюнкция)	(1 – output, 2 или более – input)
nor	ИЛИ-НЕ	(1 – output, 2 или более – input)
xor	Исключающее ИЛИ	(1 – output, 2 или более – input)
xnor	Исключающее ИЛИ-НЕ	(1 – output, 2 или более – input)

Для описания соединений субмодулей используются переменные типа `wire`. В приведенных выше примерах их соединение было описано непосредственно при объявлении экземпляров (инстансов) субмодулей, т. е. они подключались прямо к их портам. При этом интуитивно понятно, что провод должен быть подсоединен к источнику сигнала и передавать его одному или нескольким приемникам. Источником сигнала на проводе (драйвером) служит либо входной порт модуля верхнего уровня (в этом случае сигнал поступает извне проектируемого устройства), либо выходной порт какого-либо из субмодулей, причем этот источник должен быть один. В качестве получателя сигнала с провода могут быть указаны порты нескольких модулей [10, 21, 29, 40, 41].

Пример 9. Использование примитивов логических функций

```
module ex01_primitives
// Структурное описание одноразрядного сумматора
// на базе логических примитивов
(
    output sum, c_out,
// выходы типа «цепь» – сумма и выходной перенос

    input a, b, c_in
// входы типа «цепь» – слагаемые a, b и входной перенос
);

// тело модуля
    wire w1, w2, w3;
// описание цепей для соединений внутри модуля
    and g1(w1, a, b);
// инстанс g1 примитивы and – логическое «и»
    xor g2(w2, a, b);
// инстанс g2 примитивы xor – логическое «исключающее или»
    and g3(w3, w2, c_in);
// инстанс g3 примитивы and
    or g4(c_out, w1, w3);
// инстанс g4 примитивы or – логическое «или»
    xor g5(sum, w2, c_in);
// инстанс g5 примитивы xor
endmodule
```

Результат синтеза проекта приведен на рис. 28.

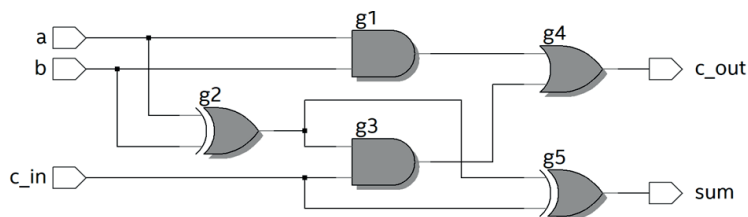


Рис. 28. Результат синтеза проекта одноразрядного сумматора

Наряду с возможностью применения многовходовых логических примитивов язык Verilog представляет еще один вариант реализации логических функций нескольких переменных – унарные побитовые операции, которые выполняются над всеми битами одной многоразрядной переменной и формируют одноразрядный результат (табл. 5). Этот способ описания является функциональным и будет рассмотрен далее.

Пример 10. Структурное описание модуля

```
module ex02_structure
#(parameter N = 4)
// Структурное описание 4-разрядного сумматора на базе
// модуля одноразрядного сумматора
(
    output cout,
    // выход типа «цепь» – выходной перенос
    output [N-1:0] S,
    // выход типа «цепь – шина» – сумма, 4 разряда
    input cin,
    // вход типа «цепь» – входной перенос
    input wire [N-1:0] A,
    // вход типа «цепь – шина» – слагаемое A, 4 разряда
    input wire [N-1:0] B
    // вход типа «цепь – шина» – слагаемое B, 4 разряда
);

wire [2:0] C;
// описание цепей для соединений внутри модуля
ex01_primitives sum0(S[0],C[0],A[0],B[0],cin);
// инстанс sum0 модуля ex01_primitives
ex01_primitives sum1(S[1],C[1],A[1],B[1],C[0]);
```

```

// инстанс sum1 модуля ex01_primitives
ex01_primitives sum2(S[2],C[2],A[2],B[2],C[1]);
// инстанс sum2 модуля ex01_primitives

ex01_primitives sum3(S[3],cout,A[3],B[3],C[2]);
// инстанс sum3 модуля ex01_primitives
endmodule

```

Результат синтеза примера приведен на рис. 29.

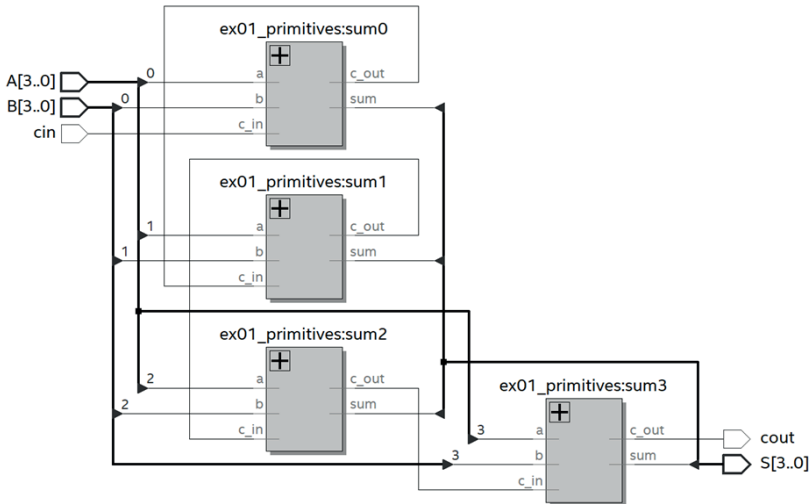


Рис. 29. Результат синтеза проекта 4-разрядного сумматора

В приведенном примере использовано позиционное назначение портов. При позиционном назначении (ordered mapping) портов следующий за именем экземпляра модуля список подключаемых к портам внешних переменных (в круглых скобках) должен строго соответствовать количеству и порядку расположения подключаемых портов в описании этого модуля. В случае если к какому-либо порту не подключается никакая внешняя цепь, в этом месте оставляется пропуск между запятыми.

Другим способом описания является непосредственное (именованное, поименное) (named mapping) подключение внешних цепей

к портам модуля нижнего уровня, при котором они могут следовать в произвольном порядке. Пара «порт модуля – подключаемая цепь» оформляется следующим способом:

.<имя порта> (<имя подключаемой цепи>)

Пример 11. Структурное описание модуля с именованным подключением субмодулей

```
module ex03_named_mapping
#(parameter N = 4)
// Структурное описание 4-разрядного сумматора на базе
модуля одноразрядного сумматора
// С именованным подключением портов
module ex02_structure
(
    output cout,
    output [N-1:0] S,
    input cin,
    input wire [N-1:0] A,
    input wire [N-1:0] B
);
wire [2:0] C;
ex01_primitives
sum0(.c_out(C[0]),.sum(S[0]),.a(A[0]),.b(B[0]),.c_
in(cin));
ex01_primitives
sum1(.a(A[1]),.b(B[1]),.c_in(C[0]),.sum(S[1]),.c_
out(C[1]));
ex01_primitives
sum2(.sum(S[2]),.a(A[2]),.b(B[2]),.c_out(C[2]),.c_
in(C[1]));
ex01_primitives
sum3(.sum(S[3]),.c_out(cout),.a(A[3]),.b(B[3]),.c_
in(C[2]));
endmodule
```

Результат синтеза этого описания ничем не отличается от приведенного на рис. 28. Из текста примера можно видеть, что позиционное назначение несколько компактнее, поскольку не включает в себя имена портов компонента. Тем не менее именованное назначение позволяет

избежать неявных ошибок, например в случае, когда разработчик перепутал порядок следования имен. Сравним разные способы именования и подключения портов.

Пример 12. Сравнение различных способов подключения входных сигналов

```
module ex03_compare
(
    input clk, // тактовый сигнал
    input data, // сигнал данных
    input reset, // сигнал сброса
    input en, // сигнал разрешения работы
    ...
    <прочие сигналы>
endmodule

...
example inst0(net1, net2, net3, net4);
example inst1(clk_net, data_net, en_net, reset_net);
example inst2(clk_net, data_net, reset_net, en_net);
example inst3(.clk(clk_net), .data(data_net), .en(en_net), .reset(reset_net));
```

Первый вариант объявления (inst0) не дает возможности быстро убедиться, что подключение выполнено правильно, поскольку имена цепей отличаются только номером.

Второй вариант (inst1) использует «содержательные» имена, которые включают в себя фрагменты имен портов: например, `clk_net` дает понятие о том, что этот сигнал следует подключить к порту `clk`. Однако требуется отдельно сопоставить порядок объявления портов в описании модуля с порядком упоминания сигналов, чтобы убедиться, что для `inst1` перепутан порядок следования сигналов `en_net` и `reset_net`. Позиционное назначение не позволит САПР выявить или исправить эту логическую ошибку, поскольку никаких формальных запретов на подключение сигнала `en_net` к входу `reset` не существует.

Сравним это с именованным назначением, реализованным для экземпляра `inst3`. В этом примере порядок упоминания сигналов `en` и `reset` также не совпадает с порядком объявления сигналов в описании модуля. Однако явное упоминание имен сигналов заставит САПР подключить

цепь `en_net` именно ко входу `en`, несмотря на то, что на третьем месте в объявлении компонента находится `reset`.

Таким образом, именованное назначение представляется более безопасным с точки зрения устранения логических ошибок при соединении компонентов [10, 21]. В сложных проектах разработчики, как правило, предпочитают именованное описание подключения.

В целом структурное описание применяется чаще всего в двух противоположных ситуациях: на нижнем уровне проектирования и на уровне сборки системы.

В первом случае этот метод выбирают, когда разработчик желает получить больший контроль над реализацией модулей. Действительно, описание структуры модулей нижнего уровня в ряде случаев позволяет получить преимущество по быстродействию или задействованным ресурсам по сравнению с синтезируемым из функционального описания решением. В то же время при структурном описании вся ответственность за реализацию функции возлагается на разработчика, а ручное описание всегда чревато ошибками, которые в таком случае особенно сложно выявляются. Процедуры автоматического синтеза, как правило, гораздо более надежны и занимают меньше времени.

Во втором случае описание системы в целом в виде соединения специализированных, библиотечных и реализованных на нижнем уровне модулей является часто единственно возможным вариантом сборки системы. При этом ручное выполнение этого описания может служить источником ошибок [21].

Для удобства описания и применения такие элементы, как шины, регистры, логические примитивы и модули нижнего уровня, могут быть собраны в массивы однородных элементов.

Пример 13. Объявление массивов шин

```
wire [3:0] a;           // 4-разрядная шина a

//Массив - НЕСКОЛЬКО элементов для переменных типа wire
wire [3:0] w [0:7]; // шина, состоящая из восьми
                   // 4-разрядных элементов
wire _fl [0:7];      //шина, состоящая из восьми
                   //1-разрядных элементов
```

Обратите внимание, что в зависимости от расположения квадратных скобок справа или слева от имени объявляемой переменной конструкция, в них заключенная, может являться описанием разрядности шины (слева) или размера массива (справа). Между этими вариантами описания есть существенная разница.

В виде массивов компонентов удобно рассматривать память. Verilog предоставляет возможность описывать массивы с помощью конструкции вида:

Пример 14. Объявление массива регистров – аналог модуля памяти

```
reg [7:0] memory [0:1];
```

В данном случае `memory` представляет собой имя (идентификатор), назначенное разработчиком; объявляется массив 8-разрядных ячеек, к которым можно обращаться по номеру (адресу): `memory [0]`, `memory [1]` и т. д.

Допустимо объявление многомерных массивов. Например:

```
reg [7:0] memory [0:7] [0:7];
```

Обращение к отдельным ячейкам будет производиться, например, как `memory [1; 5]`.

Технически многомерные массивы представляют собой линейную последовательность ячеек, а несколько индексов, задающих «координаты» ячейки, используются лишь для удобства иллюстрирования обращения к многомерным структурам данных. Аналогично можно считать, что 8-разрядная ячейка данных является в действительности одномерным массивом отдельных разрядов. Для разрядов действительно допустимы индивидуальные операции:

```
a[7] = b[4]
```

В этом примере 7-му разряду переменной `a` присваивается значение 4-го разряда переменной `b`.

Возможны операции с группами разрядов, однако количество разрядов в группах с левой и правой стороны от знака присваивания должно совпадать:

```
a[7:5] = b[3:1]
```

В этом примере указаны по три разряда обеих переменных.

В процессе конструирования цифровых устройств иногда возникает необходимость установки в модуле нескольких примитивов или модулей нижнего уровня, выполняющих одинаковые функции. Например, в устройстве, обрабатывающем многоразрядные данные, блоки, выполняющие необходимые операции для каждого двоичного разряда, могут быть идентичными. В таких случаях язык Verilog предоставляет возможность использования массивов примитивов или модулей [10, 21, 29, 41].

Пример 15. Использование массива примитивов

```
module primitive_array
#(parameter N = 4)
(
    input[N-1:0] IN1,
    input[N-1:0] IN2,
    input[N-1:0] IN3,
    output[N-1:0] OUT
);

wire [N-1:0] w_1;
wire [N-1:0] w_2;
not N[N-1:0] (w_1, IN1);
and A[N-1:0] (w_2, IN2, w_1);
xor X[N-1:0] (OUT, IN3, w_2);
endmodule
```

Результат синтеза примера приведен на рис. 30.

Пример 16. Использование массива модулей в описании 4-разрядного сумматора

```
module ex16_structure
#(parameter N = 4)
(cout, S, cin, A, B);
output cout=P[3];
output [N-1:0] S;
input cin;
input wire [N-1:0] A;
input wire [N-1:0] B;

wire [N-1:0] P;
```

```

wire [N-1:0] C={P[2:0],cin};
ex01_primitives sum[N-1:0]
(.sum(S),.c_out(P),.a(A),.b(B),.c_in(C));
endmodule

```

Результат синтеза примера приведен на рис. 31.

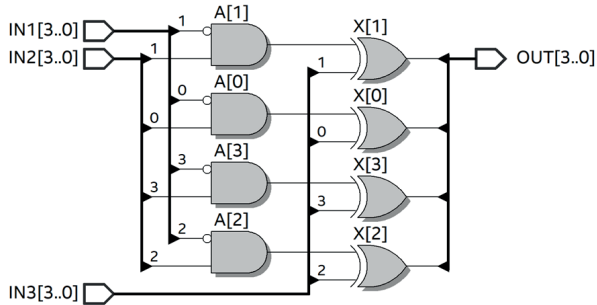


Рис. 30. Результат синтеза примера использования массива примитивов

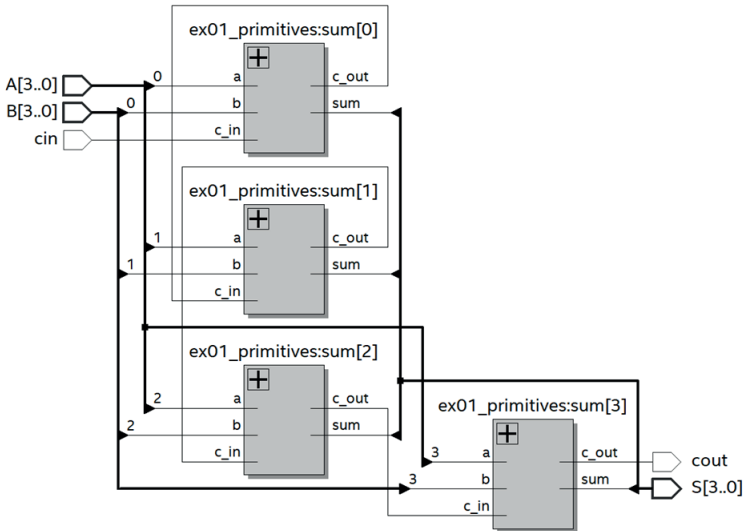


Рис. 31. Результат синтеза проекта 4-разрядного сумматора с использованием массива субмодулей

Если сравнить результаты синтеза на рис. 29 и 31, можно заметить, что они совпадают, за исключением именования модулей и цепей, при этом описание с использованием массива гораздо лаконичнее и уменьшает вероятность внесения ошибок при описании соединений модулей.

2.7. Функциональное описание модулей

Поведенческое описание (включая потоковое) представляет собой описание зависимости между входом и выходом модуля. Такой способ позволяет разработчику сосредоточиться на функциональности разрабатываемой системы.

Функциональное описание определяет функцию, реализуемую схемой, но не характеризует ее физическую реализацию или размещение на кристалле.

На основании описанной функции средство синтеза выполняет выбор вентиля и компонентов и реализует схему их соединений. При этом разработчик не может напрямую повлиять на результаты синтеза: они зависят как от разработанного описания, так и от инструмента синтеза. Для одного и того же функционального описания результаты синтеза в различных САПР могут быть различными.

Функциональное описание содержит функциональные конструкции Verilog. Эти конструкции могут появиться в пределах функций, задач или функциональных блоков `always`. Функции подразумевают только комбинационную логику, а блоки `always` могут подразумевать как комбинационную логику, так и триггеры, регистры или счетчики. Большинство модулей нижнего уровня разрабатываются в форме функциональных описаний.

2.7.1. Операции языка Verilog

Язык Verilog содержит набор операций, которые можно разбить на следующие группы [10, 21, 40, 41].

Побитные операции (bitwise) – логические операции над сигналами и шинами, которые выполняются над соответствующими парами разрядов (битов). Описание побитных операций приведено в табл. 2.

Таблица 2

Побитовые операции

Обозначение	Пример	Выполняемое действие
$\sim$	$\sim a$	Инверсия каждого разряда (бита) переменной a
$\&$	$a \& b$	Операция «И» над одноименными разрядами a и b
$ $	$a b$	Операция «ИЛИ» над одноименными разрядами a и b
$\wedge$	$a \wedge b$	Операция «исключающее ИЛИ» над одноименными разрядами a и b
$\sim \wedge$ или $\wedge \sim$	$a \sim \wedge b$	Операция «исключающее ИЛИ-НЕ» над одноименными разрядами a и b

Следующий пример иллюстрирует использование побитовых операций.

Пример 17. Использование побитовых операций

```
// объявление модуля
module ex17(

// описание входных и выходных портов
input wire [3:0] a, // 4-разрядная шина a
input wire [3:0] b, // 4-разрядная шина b
input wire c,d,
output wire not_bit, and_bit, xor_bit,
output wire [3:0] or_bit //4-разрядная выходная шина or
);

//побитовые операции
assign not_bit = ~c; //побитовая инверсия
assign and_bit = c & d; //побитовое логическое «и»
assign or_bit = a | b; //побитовое логическое «или»
assign xor_bit = c ^ d; // побитовое логическое
//исключающее или»
endmodule
```

Не все конструкции примера к настоящему моменту рассмотрены, но он в целом интуитивно понятен. На рис. 32 показаны результаты синтеза примера. Отметим, что внешний вид схемы, возможно, отличается от привычного, с которым студенты знакомятся при изучении таких дисциплин, как «Схемотехника» или «Теория цифровых автоматов». При-

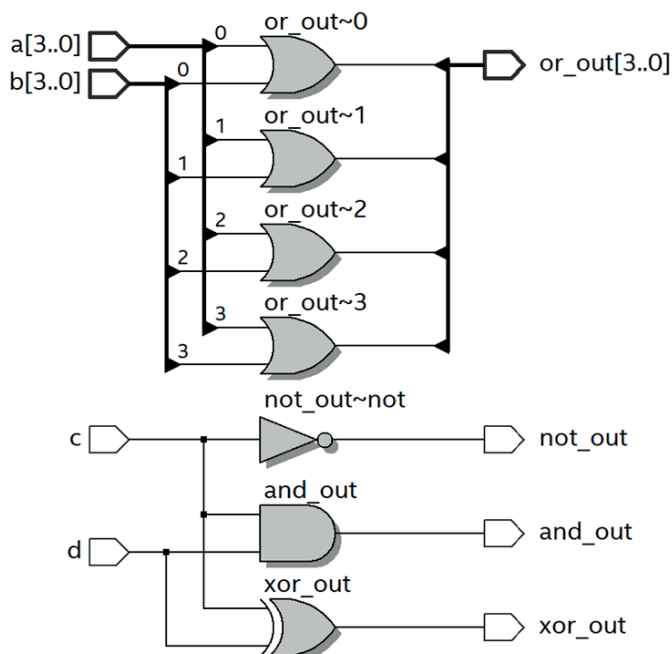


Рис. 32. Результат синтеза побитовых операций

чина в том, что российские (ранее – советские) стандарты для графического представления логических схем аналогичны европейским: в них логические примитивы отображаются в виде прямоугольных модулей с надписью, соответствующей реализуемой функции. Но большинство САПР интегральных схем и ПЛИС разработаны в Америке и поддерживают американские стандарты. В соответствии с ними и формируется изображение результатов синтеза.

Обратите внимание на формирование сигнала на шине `or_out`. Результаты синтеза наглядно иллюстрируют особенность выполнения побитовых операций: они выполняются над каждой парой соответствующих битов входных шин `a` и `b` и формируют соответствующие разряды выходной шины `or_out`.

Арифметические операции (arithmetic) реализуют основные арифметические действия над сигналами и шинами (табл. 3).

Таблица 3

Арифметические операции

Обозначение	Пример	Выполняемое действие
+	$a + b$	Арифметическое сложение a и b
–	$a - b$	Вычитание b из a
–	$-b$	Отрицательное b
*	$a * b$	Умножение a на b
/	a / b	Деление a на b
%	$a \% b$	Остаток от целочисленного деления a на b
**	$a ** b$	Возведение a в степень b

Логические операции (logical) служат для получения логических значений (ИСТИНА/ЛОЖЬ). Результат выполнения этих операций всегда является однобитным (табл. 4).

Таблица 4

Логические операции

Обозначение	Пример	Выполняемое действие
!	$!a$	Результат 1, если a равно 0, и 0 в противном случае
&&	$a \&\& b$	Результат 1, если и a , и b не равны 0, в противном случае результат 0
	$a b$	Результат 1, если или a , или b , или и a , и b не равны 0, в противном случае результат 0

Чтобы пояснить, чем отличаются побитовые и логические операции, сравним результаты выполнения следующих операций:

$1011 \& 1101 = 1001$ – результат содержит столько же разрядов, сколько операнды;

$1011 \&\& 1101 = 1$ – результат представляет собой однобитное логическое значение.

Необходимо обратить внимание, что операнд-приемник должен быть одноразрядным, чтобы избежать заполнения незадействованных разрядов результата нулями. Если эта рекомендация не учтена, может возникнуть трудно обнаруживаемая логическая ошибка. Разработчик мо-

жет ошибочно полагать, что все разряды операнда-приемника принимают одинаковое значение.

Представление логических значений многоразрядными числами и действия над ними требуют повышенного внимания, поскольку также могут служить источниками ошибок, не обнаруживаемых на этапе компиляции.

Следующий пример иллюстрирует использование логических операций.

Пример 18. Использование логических операций

```
// объявление модуля
module ex18(
input wire [3:0] b,
output not_log, and_log, or_log
);

//логические операции
assign not_log = ! a;      // операция «логическое не»
assign and_log = a && b;   // операция «логическое и»
assign or_log = a || b;   // операция «логическое или»
endmodule
```

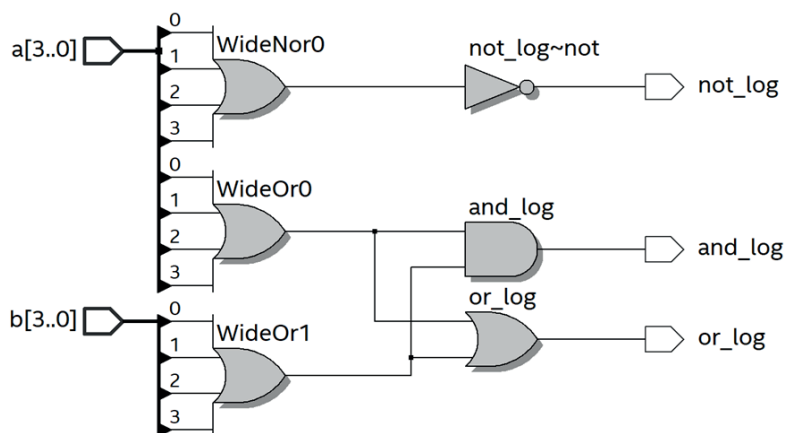


Рис. 33. Результат синтеза логических операций

На рис. 33 показаны результаты синтеза примера.

В языке Verilog накладываются ограничения для того, чтобы защитить файлы описания от ошибок, которые крайне тяжело обнаружить. В следующем примере первая строка не соответствует второй:

$$\begin{aligned} a \&\& b; a \mid \mid b \\ a \&\& b; a \mid \mid b \end{aligned}$$

В первой строке введен дополнительный символ пробела между символами $\&$ и $\mid$. Это может быть замысел разработчика, а может быть ошибочный ввод пробела. Для того, чтобы предотвратить подобную неоднозначность, Verilog требует разделить бинарную и унарную операции круглыми скобками. В силу этого требования первая строка примера является ошибочной. Поэтому правильно записывать так:

$$a \& (\& b); a \mid (\mid b)$$

Унарные побитовые операции – операции свертки (reduction) – выполняются попарно над разрядами одного операнда. Допустимые в Verilog операции свертки приведены в табл. 5.

Таблица 5

Унарные побитовые операции (операции свертки)

Обозначение	Пример	Выполняемое действие
$\&$	$\& b$	Операция «И» над всеми разрядами b
$\sim\&$	$\sim\& b$	Операция «И-НЕ» над всеми разрядами b
$\mid$	$\mid b$	Операция «ИЛИ» над всеми разрядами b
$\sim\mid$	$\sim\mid b$	Операция «ИЛИ-НЕ» над всеми разрядами b
$\wedge$	$\wedge b$	Операция «исключающее ИЛИ» над всеми разрядами b

Следующий пример иллюстрирует использование унарных побитовых операций.

Пример 19. Использование операций свертки

```
// объявление модуля
module ex8(
input wire [3:0] a,
```

```

output and_not_dig, and_dig, or_dig
);

//унарные побитовые операции
assign and_not_dig = ~& a;//операция «и-не»
assign and_dig = & a;      //операция «и»
assign or_dig = |a;        //операция «или»
assign par_dig = ^a;       //операция «исключающее или»
endmodule

```

На рис. 34 показаны результаты синтеза примера. В соответствии со смыслом операции «ИЛИ» наличие единицы в любом из разрядов a даст единицу в качестве результата свертки.

Аналогично, операция $\&a$ даст в результате «1» только в том случае, если все разряды переменной a равны «1».

Операция $\wedge a$ при этом вычислит четность переменной a , т. е. ответит на вопрос, является ли количество разрядов, установленных в единицу, четным.

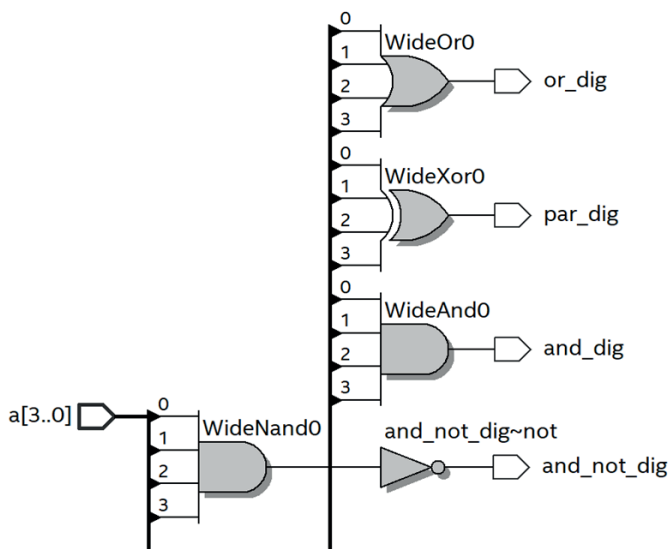


Рис. 34. Результат синтеза унарных побитовых операций

Операции отношения (relational). Кроме унарных побитовых операций в языке Verilog существуют еще две группы операций, дающих одноразрядный результат при многоразрядных операндах: это операции сравнения (табл. 6) и операции идентичности (табл. 7).

Операции отношения служат для выполнения операций сравнения между операндами. Результатом сравнения является логическое (однобитное) выражение «ИСТИНА» (1) или «ЛОЖЬ» (0) [10, 11].

Таблица 6

Операции сравнения

Обозначение	Пример	Выполняемое действие
==	a == b	Результат 1, если a равно b, и 0, если a не равно b
!=	a != b	Результат 1, если a не равно b, и 0, если a равно b
<	a < b	Результат 1, если a меньше b, и 0 в остальных случаях
>	a > b	Результат 1, если a больше b, и 0 в остальных случаях
<=	a <= b	Результат 1, если a меньше или равно b, и 0, если a больше b
>=	a >= b	Результат 1, если a больше или равно b, и 0, если a меньше b

Следующий пример иллюстрирует использование операций сравнения.

Пример 20. Использование операций сравнения

```
// объявление модуля
module ex9(
  input wire [3:0] a,
  input wire [3:0] b,
  output eq, not_eq, lt, gt, ge
);

//операции сравнения
assign eq = a==b;           //равно
assign not_eq = a!=b;       //не равно
assign lt = a < b;          //меньше
assign gt = a > b;          //больше
assign ge = a >= b;         //больше или равно
endmodule
```

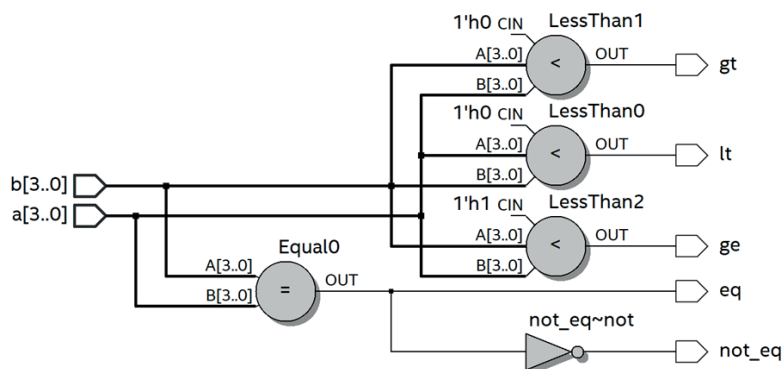


Рис. 35. Результат синтеза операций сравнения

На рис. 35 показаны результаты синтеза примера.

Операции тождества (equality) являются специфическими для языка описания аппаратуры. Все операции сравнения дают результат X (не определено), если хотя бы один разряд в операндах содержит логическое значение X или Z. В операциях идентичности явным образом сравниваются все возможные логические значения разрядов, включая 0, 1, X и Z [10, 11].

Таблица 7

Операции тождества (идентичности)

Обозначение	Пример	Выполняемое действие
===	a === b	Результат 1, если a по всем разрядам идентично b (с учетом значений X и Z), и 0 в противном случае
!==	a !== b	Результат 0, если a по всем разрядам идентично b (с учетом значений X и Z), и 1 в противном случае

Несколько необычные варианты операций «тождественно равно» и «тождественно не равно» проистекают из возможности появления значений z и x (unknown) при моделировании. Если хотя бы один из битов сравниваемых операндов равен этим значениям, обычное сравнение выполнить не удастся, а результатом операции === будет x (неизвестно). Однако можно произвести «буквальное сравнение» с помощью опера-

ции `===`, которая дает истинный результат, если в каждом бите обоих операндов находятся строго одинаковые значения, включая *Z* и *X*. Это можно проиллюстрировать таким примером:

```
if (a === 000x) ... // операции, показывающие,
                    // что младший разряд
                    // равен неизвестному значению
```

Можно добавить, что операции проверки тождественности имеет смысл использовать только при моделировании, где они полезны для подробной диагностики состояния моделируемой системы и появляющихся в ней ошибок. Приведенный пример позволяет представить возможности глубокой диагностики появляющихся логических ошибок, когда в модели будет проверяться не только появление значений «неизвестно», но еще и выявляться разряд, в котором они появляются.

Операции конкатенации/повторения (concatenation/replication) применяются для удобного манипулирования данными разной разрядности в одном выражении и формирования шин с не одноименными линиями (табл. 8) [10, 11, 41].

Таблица 8

Операции конкатенации и повторения

Обозначение	Пример	Выполняемое действие
{ }	{a,b}	Соединяет <i>a</i> и <i>b</i> , формируя шину (вектор) для использования в многоразрядном выражении (<i>a</i> – старший разряд, <i>b</i> – младший)
{ { }	{n{a}}	Повторяет <i>a</i> <i>n</i> раз

Операции конкатенации служат для группировки сигналов в шины. Операции повторения (replication) являются удобной формой объединения повторяющихся сигналов.

Например если *a*, *b*, *c*, *d* представляют собой одnorазрядные сигналы, то выражение {*a*, *b*, *c*, *d*} создаст четырехбитный сигнал, составленный из «сцепленных» разрядов *a*, *b*, *c*, *d* соответственно. Здесь символы { } служат для оформления операции конкатенации. Выполнив эту операцию, далее можно работать с результатом, как с единой шиной.

Операция повторения (репликации) является разновидностью операции конкатенации. Например, выражение $\{4\{a\}\}$ полностью аналогично записи $\{a, a, a, a\}$, т. е. число «4» обозначает количество повторений сигнала, указанного во внутренних фигурных скобках.

Пример 21. Использование операций конкатенации (склейки) и репликации

```
// объявление модуля
module ex10(
  input wire [3:0] a,
  input wire [3:0] b,
  input wire c,
  output [8:0] d
);

// операции конкатенации (склейки) и репликации
assign d = {a, c, {2{b[1:0]}}};
endmodule
```

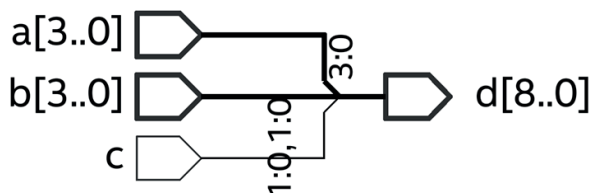


Рис. 36. Результат синтеза операций конкатенации (склейки) и репликации

На рис. 36 показаны результаты синтеза примера.

Результат синтеза выглядит не очень наглядно, и в целом проиллюстрировать эти очень полезные операции сложно. В данном случае в результате операции сформирована девятиразрядная шина вида

$$[a_3, a_2, a_1, a_0, c, b_1, b_0, b_1, b_0].$$

Операции сдвига (shift) являются аналогами подобных операций, используемых в языках программирования. Они позволяют записывать манипуляции с шинами, заключающиеся в сдвиге их значений на один или несколько разрядов (табл. 9) [10, 11, 41].

Таблица 9

Операции сдвига

Обозначение	Пример	Выполняемое действие
<<	$a \ll n$	Логический сдвиг a влево на n разрядов (справа вдвигаются нули)
>>	$a \gg n$	Логический сдвиг a вправо на n разрядов (слева вдвигаются нули)
>>>	$a \ggg n$	Арифметический сдвиг вправо на n разрядов. Старший (знаковый) разряд сохраняется, освободившиеся разряды заполняются значением знакового разряда
<<<	$a \lll n$	Арифметический сдвиг влево на n разрядов. Старший (знаковый) разряд сохраняется, освободившиеся разряды заполняются значением знакового разряда

Результаты синтеза этой операции не имеет смысла приводить, так как они представляют собой переставленные местами входные сигналы.

Отличием логического и арифметического сдвигов является поведение старшего разряда.

Если рассматривать многоразрядные шины как набор логических сигналов, то операции сдвига используются для анализа отдельных разрядов или групп разрядов. В этом случае заполнение «освободившихся» в результате сдвига разрядов нулями является естественным.

Если же шина рассматривается как двоичное число, то операции сдвига являются удобным способом умножения и деления двоичных чисел на целые степени двойки: 2, 4, 8, 16 и т. д. Однако в двоичном представлении чисел со знаком обычно старший (левый) разряд кодирует знак числа: значение «0» обычно соответствует положительному числу, а значение «1» – отрицательному. Соответственно при арифметическом сдвиге его значение не должно меняться. «Освободившиеся» в результате арифметического сдвига разряды числа должны заполняться арифметическими нулями. В случае отрицательного числа в обратном коде незначащий разряд соответствует единице.

Операция условного выбора (conditional) является тернарной, т. е. содержит 3 операнда: операнд-условие и два операнда выбора (табл. 10).

Таблица 10

Условный оператор

Обозначение	Пример	Выполняемое действие
?:	<sel>?<a>:	Если условие <sel> вычисляется как «ложное» (false), то в качестве результата будет использовано выражение . Если условие «истинно» (true), то будет использовано выражение <a>

Операция условного выбора используется при описании условной логики, например мультиплексоров.

2.7.2. Приоритет операций

При выполнении выражений с многоразрядными данными применяются следующие правила:

- все операнды расширяются до разрядности наибольшего операнда, включая операнд, стоящий слева от операции присваивания;
- разрядность операндов, образованных с помощью операций конкатенации и тиражирования, вычисляется перед выполнением выражения и рассматривается равноправно с разрядностью других операндов в выражении;
- операнды без знака расширяются слева нулями;
- операнды со знаком (signed) расширяются слева значением старшего значащего бита (бита знака).

При выполнении большинства арифметических операций все операнды выражения используются для установления того, как должна выполняться операция. При этом:

- если хотя бы один операнд является действительным, то применяется арифметика с плавающей точкой;
- если хотя бы один операнд является данным без знака, то применяется арифметика без знака;
- если все операнды со знаком (signed), то применяется арифметика со знаком;
- любой операнд может быть приведен к виду со знаком или без знака путем использования системных функций \$signed и \$unsigned.

При наличии нескольких операций в выражении они выполняются с учетом приоритета. Наивысшим приоритетом обладают унарные операции, низшим – операции конкатенации сигналов. Список операторов Verilog в порядке убывания приоритета приведен ниже [21, 22, 40]:

- унарные и операции редукции;
- возведение в степень;
- умножение, деление, остаток;
- сложение, вычитание;
- сдвиг;
- операции отношения;
- сравнение;
- логическое «И» &;
- побитное «исключающее ИЛИ», «исключающее ИЛИ-НЕ»;
- побитное «ИЛИ»;
- логическое «И»;
- логическое «ИЛИ»;
- условный оператор;
- операции конкатенации и репликации.

При необходимости изменения порядка вычисления следует использовать скобки.

Кроме того, расстановка скобок в ряде случаев помогает получить более эффективную схемотехническую реализацию при синтезе.

Вопросы и задания

1. Охарактеризуйте современные языки описания аппаратуры, их особенности.
2. Каковы современные тенденции развития языков описания аппаратуры?
3. В чем отличие между синтезируемым и несинтезируемым подмножеством Verilog?
4. Перечислите общие и специальные типы данных языка Verilog.
5. Перечислите уровни сигналов в Verilog и их назначение.
6. Арифметические и логические операторы Verilog. Унарные, бинарные, тернарные операторы.

7. Опишите структуру модуля на языке Verilog.
8. В чем отличие между модулем и экземпляром модуля (инстансом)?
9. В чем различие между головным модулем, submodule и базовым модулем?
10. Перечислите правила назначения входных и выходных портов модуля.
11. Каковы особенности и применение структурного способа описания модуля?
12. Каковы особенности и применение поведенческого способа описания модуля?
13. Для чего служат параметры модуля?
14. Для чего используются массивы элементов Verilog?
15. Что такое шина в Verilog? Какой порядок нумерации сигналов используется в шине и почему?

3. СИНТЕЗИРУЕМОЕ ОПИСАНИЕ АСИНХРОННЫХ (КОМБИНАЦИОННЫХ) СХЕМ

Асинхронная логика – разновидность взаимодействия логических элементов цифровых устройств, когда элементы действуют асинхронно, не подчиняясь глобальному генератору тактовых импульсов.

Асинхронные (комбинационные) схемы (АС) предназначены для выполнения определенных логических операций (функций алгебры логики) в устройствах вычислительной техники и дискретной автоматики. В АС выходная логическая функция определяется лишь комбинацией входных переменных, поступающих на входы схемы в текущий момент времени, и не зависит от их комбинации в предшествующее время.

То есть АС выполняют функцию обработки сигналов, но не выполняют функцию хранения. В идеальных моделях АС изменение состояния на выходе схемы в ответ на изменения входных сигналов происходит мгновенно. В реальных цифровых логических схемах между этими двумя событиями существует промежуток времени, связанный с прохождением переходных процессов в электронных компонентах, реализующих вентили.

Фактически все ранее приведенные примеры представляют собой описание асинхронных схем. Следовательно, они могут быть описаны структурным способом. Однако структурное описание больших проектов требует слишком больших затрат времени и труда разработчиков, а также значительных усилий для отладки проектов, поскольку «ручная» работа является источником большого числа ошибок. Проводя аналогии с программированием, можно сказать, что структурное описание подобно программированию на языке низкого уровня – ассемблере: хорошо написанная на таком языке программа эффективна и экономична, но процесс ее написания и отладки крайне трудоемок, а в случае больших проектов недопустимо продолжителен. Поэтому в большинстве случаев при разработке ПЛИС-проектов используется функциональный стиль описания в сочетании с элементами структурного стиля.

К синтезируемым конструкциям, описывающим функциональность модулей, относятся:

- операция непрерывного присваивания `assign`;
- процедурные блоки `always`;
- условные конструкции `case` и `if`;
- операторы языка Verilog;
- процедуры и функции.

3.1. Оператор непрерывного присваивания (назначения) `assign`

Источник сигнала (драйвер) цепи назначается либо при объявлении переменной, либо с использованием оператора непрерывного присваивания `assign`, который дает дополнительные возможности.

Общий формат оператора непрерывного назначения

```
assign (сила сигнала) # (задержка) <оператор присваивания>
```

Необязательные параметры «сила сигнала» и «задержка» (`delay`) игнорируются средствами синтеза. При написании синтезируемого кода задержки, как правило, не используют. Если в операторе непрерывного присваивания указана задержка, то она повлияет на результаты моделирования процессов в цепи, но при синтезе будет проигнорирована. В результате поведение сети при моделировании и реальной аппаратуре может отличаться по временным характеристикам – и проект может оказаться неработоспособным. Если функционирование разрабатываемого модуля требует соблюдения определенных временных интервалов, то введение данной конструкции не приведет к желаемому результату. Временные соотношения обеспечиваются введением синхронной логики и реализуются, как правило, с использованием таймеров-счетчиков.

Собственно оператор присваивания состоит из двух частей, которые отделены символом равенства (`=`). Этот оператор уже встречался ранее, в примерах использования операций языка Verilog.

Правила описания оператора непрерывного присваивания `assign`:

- результатом присваивания (левая часть выражения) должны быть скалярные и векторные сигналы типа «цепь» или конкатенация скалярных и векторных цепей;

- непрерывные присваивания всегда отслеживают любые изменения операндов, и результат после указанной задержки принимает соответствующее им значение;
- операнды в непрерывных присваиваниях (т. е. справа в записи) могут быть векторами или скалярами, причем как цепями, так и переменными;
- выражение на правой стороне непрерывного назначения ничем не ограничено, кроме условия вычислимости; оно может содержать ссылку на функции, оператор выбора case, условный оператор if или другую процедурную конструкцию.

Непрерывные назначения позволяют моделировать комбинационную логику, как определяя взаимосвязь вентилей логики, так и задавая логическое выражение, которое управляет цепью. При моделировании значение сигнала на переменной типа wire пересчитывается постоянно, изменение сигнала в цепи происходит всякий раз, когда изменяется значение в правой части оператора присваивания.

Пример 22. Использование операции непрерывного присваивания

```
module ex04_assign(IN1, IN2, A, B, C, D, E, F, G, H);
    input IN1, IN2;
    output A, B, C, D;

    assign A=IN1;
    not (B, IN1);
    assign C = ~ IN1;

    and(D, IN1, IN2);
    assign E = IN1 & IN2;

    or(F, IN1, IN2);
    assign G = IN1 | IN2;

endmodule
```

Результат синтеза примера приведен на рис. 37. На рис. 37 видно, что описания логических функций с использованием операции непрерывного присваивания assign, содержащей логические функции, синтезированы в те же конструкции, что и описания с использованием примитивов языка Verilog.

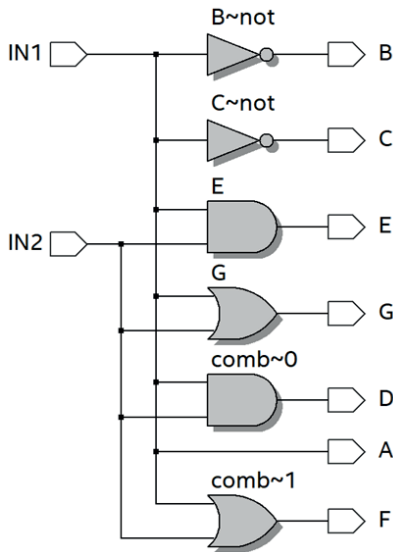


Рис. 37. Результат синтеза примера использования операции непрерывного присваивания

Также интуитивно понятно, что источник сигнала (драйвер) у цепи в общем случае должен быть один, т. е. для каждой цепи назначение источника выполняется однократно. В большинстве случаев при попытке назначить более одного драйвера сети возникнет ошибка, в частности, средство синтеза Quartus сообщит: “Error (№ ошибки): Net (имя цепи) cannot be assigned more than one value” (перевод: «Ошибка: цепи не может быть назначено более одного значения»). В ряде случаев множественное назначение драйверов может быть успешно обработано средствами синтеза, но использовать эту

возможность не рекомендуется, поскольку результат может быть некорректным либо не соответствовать ожиданиям разработчика. Средство синтеза не в состоянии определить, является ли такая ситуация осознанным решением или это результат ошибки; поэтому при синтезе в случае успешного назначения нескольких драйверов одной цепи выдается предупреждение.

В то же время одна и та же цепь может служить источником сигнала для нескольких модулей и участвовать в формировании драйверов нескольких цепей (шин). Если переменная типа *wire* объявлена, но не имеет подключения к драйверу, это не вызовет ошибки. При синтезе эта переменная будет проигнорирована, но средство синтеза выдаст предупреждение. Сообщение Quartus имеет вид: “Warning (№ предупреждения): Wire “имя цепи” has no driver” (перевод: «Предупреждение: цепь не имеет драйвера»).

Словесно описать работу модуля можно следующим образом: если на вход модуля поступает управляющий сигнал `enable = 1`, то со-

гласно сигналам на входах выбора s производится выбор между одной из трех входных шин и шиной Z (шина нулевых значений), а выбранная шина подключается к выходной шине. Если сигнал $enable = 0$, выходные шины также коммутируются с нулевой шиной Z .

Функциональные описания модуля с использованием оператора непрерывного присваивания assign можно выполнить несколькими способами.

Пример 23. Функциональное логическое описание модуля одно-разрядного сумматора

```
module add_a (a, b, p, s, p1);
input a, b, p;
output s, p1;
    assign s = (a ^ b) ^ p;
    assign p1 = (a & b) | ((a ^ b) & p);
endmodule
```

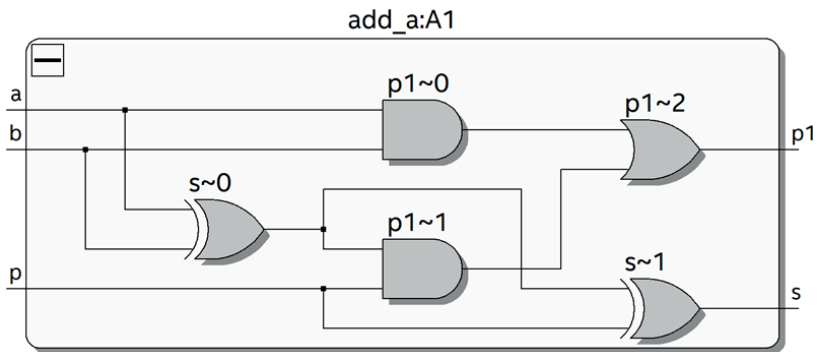


Рис. 38. Результат синтеза примера функционального логического описания модуля одноразрядного сумматора

В логических описаниях используются логические функции; в результате получается описание, близкое по форме и по особенностям применения к структурному описанию с использованием логических примитивов. В процессе синтеза может быть выполнена оптимизация полученной схемы. Результат синтеза примера приведен на рис. 38.

Пример 24. Функциональное арифметическое описание модуля одноразрядного сумматора

```
module add_b (a, b, p, s, p1);
input a, b, p;
output s, p1;
    assign {p1,s}=a+b+p;
endmodule
```

В арифметических описаниях используются арифметические операции. В процессе синтеза в зависимости от ресурсов выбранной ПЛИС арифметические операции могут быть реализованы в виде комбинационных схем либо с использованием встроенных аппаратных арифметических блоков. Результат синтеза примера приведен на рис. 39.

В арифметических описаниях используются арифметические операции. В процессе синтеза в зависимости от ресурсов выбранной ПЛИС арифметические операции могут быть реализованы в виде комбинационных схем либо с использованием встроенных аппаратных арифметических блоков.

Пример 25. Функциональное поведенческое описание модуля одноразрядного сумматора

```
module add_c (a, b, p, s, p1);
input a, b, p;
output s, p1;
    assign s = (a ? (b ? (p ? 1'b1 : 1'b0) : (p ? 1'b0 : 1'b1)) : (b ? (p ? 1'b0 : 1'b1) : (p ? 1'b1 : 1'b0)));
    assign p1 = (a ? (b ? 1'b1 : (p ? 1'b1 : 1'b0)) : (b ? (p ? 1'b1 : 1'b0) : 1'b0));
endmodule
```

Результат синтеза примера приведен на рис. 40.

В поведенческом описании используются условные операции или конструкции `if` и `case`, которые будут рассмотрены позже. Подобные конструкции обычно реализуются с использованием мультиплексоров, входящих в состав логических ячеек ПЛИС.

Деление функциональных описаний на способы условно; в реальных проектах все они могут использоваться и используются совместно.

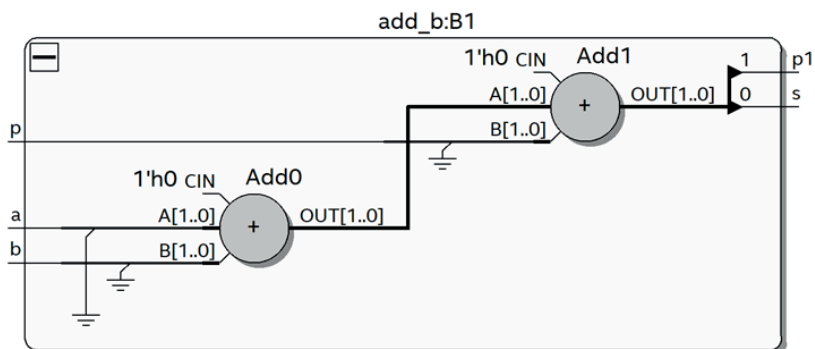


Рис. 39. Результат синтеза примера функционального арифметического описания модуля одноразрядного сумматора

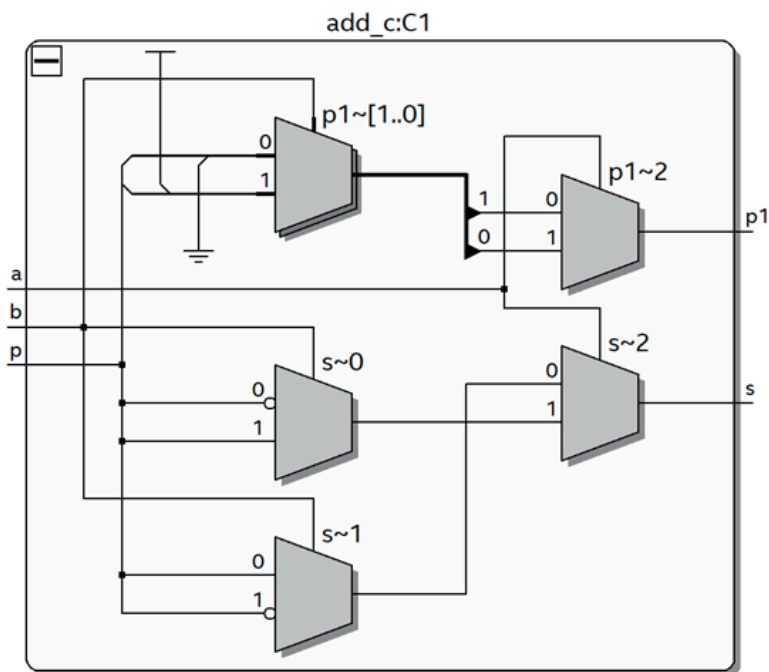


Рис. 40. Результат синтеза примера функционального поведенческого описания модуля одноразрядного сумматора

3.2. Поведенческий блок `always` в описаниях комбинационных схем

Операция непрерывного присваивания проста в использовании и интуитивно понятна, но построение описания только с использованием этих операций трудно читается. Кроме того, моделирование этих операций является трудоемким (времяемким) процессом. Это становится особенно критично в больших проектах. В том числе для решения этих проблем в язык Verilog введена конструкция процедурного блока `always`.

Поведенческий блок (behavioral block) (процедурный блок) `always` непрерывно выполняет последовательность определенных внутри него действий. Действия повторяются циклически, начиная с нулевого момента времени. Если в модуле есть несколько конструкций `always`, то каждая из них выполняется параллельно с другими и независимо от других [10, 14, 22, 29, 40].

Блоки `always` могут быть синтезируемыми. Они предназначены для функционального описания в некоторых случаях асинхронных, но в основном синхронных схем. В данном разделе мы опишем общий синтаксис данной конструкции и особенности ее применения именно для описания асинхронной логики.

Описание блока `always` имеет следующий синтаксис:

```
always @(<список чувствительности>) <тело блока>
```

Тело блока может содержать один или несколько операторов. Если требуемое действие не может быть выражено одним оператором, используются «операторные скобки» `begin/end`.

Список чувствительности (sensitivity list) представляет собой список сигналов, изменение которых должно приводить к выполнению действий и назначений, перечисленных внутри «операторных скобок» `begin/end`.

Для облегчения восприятия конструкции ее можно словесно описать так: «всегда (`always`) при условии изменения (`@`) значений из списка чувствительности (sensitivity list) выполнять операции, составляющие тело блока».

В списке чувствительности имена входных сигналов разделяются ключевым словом “or” или запятой:

```
always @(a or b or d) <statements>
always @(a, b, d) <statements>
```

Если вместо списка чувствительности указать символ «*», это будет означать, что в него будут включены все сигналы:

```
always @* <statements>
```

При описании выражений внутри поведенческих блоков с правой стороны от знака равенства можно использовать целочисленные переменные и типы wire или reg, а вот с левой стороны теперь используется только тип reg:

```
reg [3:0] c;
always @(a or b or d)
begin
  c = <выражение использующее входные сигналы a,b,d>;
end
```

Обратите внимание, что регистры, которым идет присвоение в таких поведенческих блоках, не обязательно будут выполнены в виде триггеров после синтеза. Процедурный блок может быть синтезирован как комбинационная логика в том случае, если в списке чувствительности перечислены все сигналы, встречающиеся в теле блока [21].

Пример 26. Описание комбинационной логики в блоке always

```
module fig1 (q, a, b);
output reg q;
input a,b;
//assign q = a & b;
always @(a, b)
begin
  q = a & b;
end
endmodule
```

Такая запись при синтезе эквивалентна выражению непрерывного присваивания, приведенного в виде комментария в коде данного примера. Результат синтеза обоих примеров будет одинаков.

Обратите внимание, что выходной порт q в первом примере должен быть описан как reg, поскольку в блоке always назначение получают

именно переменные этого типа. Но, несмотря на это, синтезатор, проанализировав код на языке Verilog, пришел к заключению, что фактически в этом блоке функция хранения отсутствует, поэтому задействовать регистр нет необходимости.

С точки зрения моделирования различие состоит в том, что для оператора `assign` повторный расчет выражения производится при изменении любого из сигналов, участвующих в выражении, а для блока `always` – только для сигналов, перечисленных в списке чувствительности. Для приведенных выше примеров результат будет одинаковым; однако возможны ситуации, когда сокращение списка чувствительности до необходимого минимума способно ускорить процесс моделирования за счет отказа от повторного перерасчета выражений, которые в конечном итоге не влияют на работу схемы [10, 11, 41].

Необходимо также иметь в виду, что в ряде случаев при синтезе блока `always` в схеме появляется триггер с управлением уровнем сигнала (защелка, или `latch`). Таким образом, в схему вводится элемент, который не предусматривается разработчиком, вносит дополнительную задержку и требует аппаратные ресурсы для своей реализации. Для ПЛИС с архитектурой FPGA использование защелок не рекомендуется производителями, поскольку они строятся на базе триггера и мультиплексора. При выборе между триггером и защелкой следует выбирать триггер, а появление защелки в комбинаторной схеме, вызванное неполным описанием условий, является стилистической ошибкой. Средство синтеза сообщает о появлении этого триггера, поскольку это может свидетельствовать об ошибке в коде или, как минимум, о его неточности.

Структура тела поведенческого блока имеет ряд особенностей. Поскольку он, как правило, используется для описания синхронных схем, детали будут изложены в соответствующем разделе. Рекомендуем ознакомиться с ним прежде, чем использовать поведенческий блок для описания как асинхронных, так синхронных схем.

3.3. Синтезируемая конструкция цикла `for`

Конструкция цикла может использоваться только внутри поведенческих блоков `always`. Интерпретация конструкции `for` в синтезируемых

и несинтезируемых описаниях существенно отличается. В синтезируемых описаниях конструкция `for` используется в основном для создания множественных экземпляров схем [21, 29, 40].

Синтаксис конструкции цикла `for`:

```
for (начальное присваивание; условие; шаг присваивания)
<процедурная операция>;
```

Конструкция `for` повторяет выполнение процедурных присваиваний или операторов такое число раз, которое задается при помощи трех характеристик:

- начальное присваивание – определяет начальное значение повторения;
- условие – определяет состояние, когда повторения должны быть завершены;
- шаг присваивания – определяет величину изменения индекса повторения обычно при помощи декремента или инкремента.

Группировка нескольких присваиваний осуществляется при помощи блока `begin-end`.

Пример 27. Использование конструкции `for` для описания асинхронной логики

```
module for_ex(
    input [3:0] a,
    input b,
    output reg [3:0] c
);

    integer i;
    always @ (a, b)
    begin
        for (i = 0; i < 4; i = i + 1)
            c[i] = b & a[i];
    end
endmodule
```

Результат синтеза такой схемы показан на рис. 41.

На рис. 42 видно, что оператор был интерпретирован САПР как четыре одинаковых фрагмента цифровой схемы, различающиеся индек-

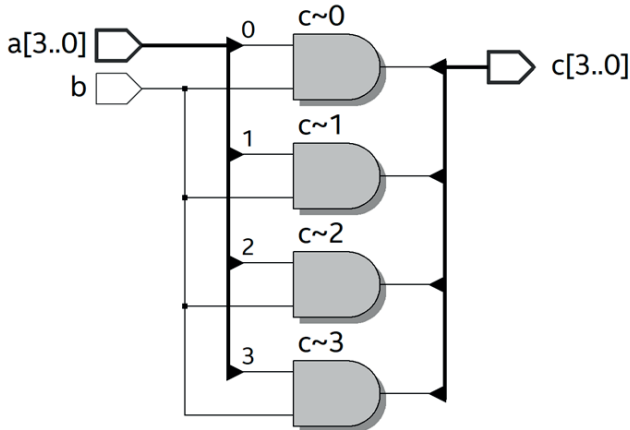


Рис. 41. Пример синтеза конструкции for

сами обрабатываемых сигналов. Такой вид использования – генерация множества экземпляров компонента, различающихся индексами сигналов, – является наиболее естественной формой применения оператора `for` в синтезируемых описаниях.

Необходимо сразу же подчеркнуть, что данный оператор не предназначен для создания счетчиков или других схем, которые после их синтеза работали бы заданное количество тактов. Подобные задачи решаются в языках описания аппаратуры в совершенно ином стиле, и здесь не следует искать прямых соответствий с языками программирования.

Аналогичного результата синтеза можно добиться с использованием массивов модулей.

3.4. Описание условной логики в комбинационных схемах

Для проверки условий (в том числе и для описания условной логики) Verilog предоставляет три типа конструкций:

- тернарная операция `?:` ;
- условный оператор `if`;
- условный оператор `case`.

3.4.1. Тернарная операция ?:

Синтаксис и особенности выполнения тернарной операции описаны в разд. «Операции языка Verilog». Рассмотрим ее использование на примере.

Пример 28. Применение тернарной операции ?:

```
module condit_stmts
#(parameter N = 4)
(
    input [N-1:0] a, b,
    input sel,
    output [N-1:0] y
);
    assign y = sel? b:a;
endmodule
```

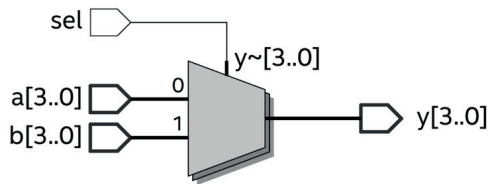


Рис. 42. Пример синтеза тернарного оператора

В результате синтеза получен множественный мультиплексор с одним управляющим входом, сигнал на котором совпадает с условием, проверяемым тернарной операцией (рис. 42).

3.4.2. Условная конструкция if

Синтаксис условного оператора `if`:

```
if (<условие>) <оператор>;
[else <оператор-иначе>;]
```

Если требуемое действие не может быть выражено одним оператором, используются «операторные скобки» `begin/end`. При необходи-

мости проверки множественных условий могут использоваться вложенные операторы if/else.

Пример 29. Применение конструкции if

```
module condit_stmts_1
#(parameter N = 4)
(
input [N-1:0] a, b,
input sel,
output reg [N-1:0] y
);
always @(a or b or sel)
begin
if (sel==1)
    y = b;
else y = a;
end
endmodule
```

Результат синтеза полной конструкции if совпадает с результатом синтеза тернарного оператора, приведенным на рис. 42, и представляет собой множественный мультиплексор с одним управляющим входом, сигнал на котором совпадает с условием оператора if [41, 42]. Поскольку в полном операторе if выбирается один из двух возможных вариантов, мультиплексор имеет две входные шины данных. Фактически мы получили четыре копии одноразрядного мультиплексора (по числу разрядов шин данных). Это стандартный ход средств синтеза HDL (исключение составляют буферные элементы, имеющие высокоимпедансное Z-состояние). Особенности работы средства синтеза требуют от HDL-программиста повышенного внимания к работе оператора if: если else-ветвь оператора не описана, возникает неопределенность в вопросе о том, какова должна быть реакция проектируемого устройства в этом случае. Как правило, синтезатор учитывает, что в этом случае состояние выхода сохраняет прежнее значение; значит, в схеме должен появиться элемент хранения. Поскольку в описании отсутствует привязка к фронту синхросигнала, синтезируется триггер, управляемый уровнем сигнала, – защелка (latch), упоминавшаяся ранее (рис. 43).

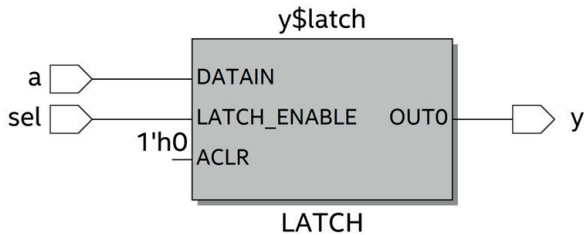


Рис. 43. Результат синтеза конструкции if с опущенной частью else

Пример 30. Пропущенная часть else в конструкции if

```

module if_without_else (input a, input sel, output reg y);
always @(a or sel)
begin
if (sel==1) y = a;
end
endmodule

```

Иными словами, средство синтеза не игнорирует пропущенную в коде ветку оператора if, а вместо мультиплексора устанавливает триггер-защелку, управляемый уровнем. При таком подходе для отсутствующей ветви оператора if на выходе устройства (или той переменной, что указана в существующей ветви оператора if) сохраняется последнее значение, установившееся при последнем активном состоянии селектора. По итогу синтеза САПР выдает предупреждение об установке триггера-защелки. Например, Quartus в данном случае сообщает: “Info (№ ошибки): Inferred latch for "(имя переменной)" at (имя модуля)” (перевод: «Информация: добавлен триггер-защелка для переменной (имя переменной) в модуле (имя модуля)»).

Рассмотрим пример использования вложенных конструкций if.

Пример 31. Описание мультиплексора 4-к-1 с использованием полных вложенных конструкций if/else

```

module fig (
input d, c, b, a,
input [1:0] sel,
output reg q);

```

```

always @ *
    if ( sel == 2'b00 ) q = a;
      else if ( sel == 2'b01 ) q = b;
        else if ( sel == 2'b10 ) q = c;
          else if ( sel == 2'b11 ) q = d;
            else q = 1'bx;

endmodule

```

В этом примере описан мультиплексор 4-к-1. Такой мультиплексор имеет мультиплексируемые входы данных a, b, c, d, двухразрядный вход селектора sel и выход данных q. Результат синтеза примера приведен на рис. 44.

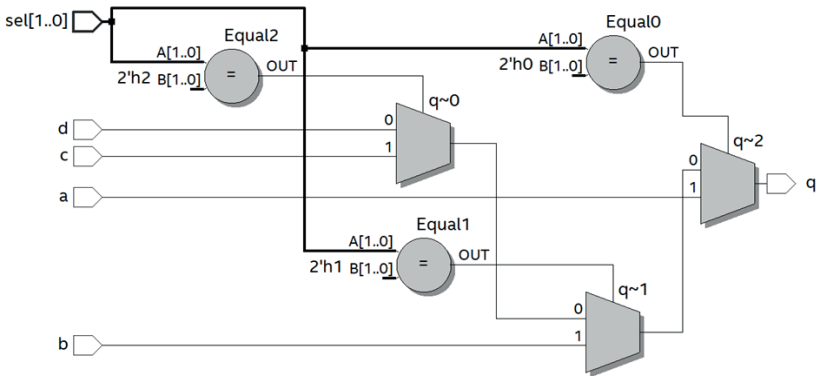


Рис. 44. Результат синтеза описания мультиплексора 4-к-1, описанного с использованием конструкции if

В приведенном листинге интерес представляет оператор по умолчанию, который присваивает выходу значение X («неизвестно»). С точки зрения цифровой схемотехники использование этой строки излишне, поскольку на двухразрядном входе sel всегда будет присутствовать одна из комбинаций уровней, перечисленных в четырех вариантах условий. Однако с точки зрения САПР возможны также комбинации вида 0Z, 1Z, ZZ и т. д., несмотря на то, что в реальной схеме они появляться не будут. Тем не менее перечисленные варианты не будут считаться исчерпывающим перечислением состояний такой схемы – и она не сможет быть реализована с помощью только комбинаторной логики. Для чисто комбинаторной схе-

мы требуется, чтобы ее выходное состояние было однозначно определено в каждый момент времени и для любой комбинации входных сигналов.

Таким образом, лишняя на первый взгляд строка, явно задающая действия по умолчанию, позволяет избежать появления защелок, которые не требуются по смыслу работы схемы. Кроме того, при моделировании появление значения «неизвестно» позволяет быстро диагностировать проблемы при формировании тестовых воздействий, поскольку выполнение ветки `else` возможно только в том случае, если в модели не удалось обеспечить правильные логические уровни для сигнала `sel`. Для сравнения можно привести следующий пример.

Пример 32. Описание мультиплексора 4-к-1 с использованием вложенных конструкций `if/else` и опущенным условием неопределенности входов

```
module fig (
    input d, c, b, a,
    input [1:0] sel,
    output reg q);

always @ *
    if      ( sel == 2'b00 ) q = a;
    else if ( sel == 2'b01 ) q = b;
    else if ( sel == 2'b10 ) q = c;
    else if ( sel == 2'b11 ) q = d;

endmodule
```

В данном примере в последней строке выходу присваивается значение входа `d`, поскольку предполагается, что если ни одно из предыдущих условий не выполняется, то значение селектора равно 11. С точки зрения синтеза это решение корректно, однако при моделировании оператор `q = d` будет выполняться и в случае, когда значение селектора не определено (т. е. равно XX). Таким образом, проблема в модели, заключающаяся в отсутствии корректного значения для сигнала-селектора, оказывается замаскированной, поскольку на выходе мультиплексора при моделировании все равно будет присутствовать какой-то сигнал, равный `d`. При этом будет неочевидно, появился этот уровень из-за действительного выбора входа `d` или это явилось результатом неверно описанной модели [41–43]. В то же время значение «неизвестно» ока-

жет соответствующее влияние на входы, к которым подключен выход мультиплексора, сделав «неизвестным» результат работы соответствующих блоков, что даст возможность идентифицировать проблему и проследить ее до того блока, в котором и произошло некорректное задание тестовых воздействий (рис. 45).

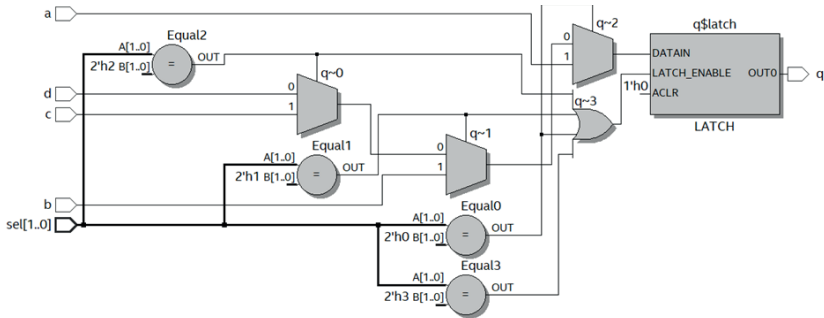


Рис. 45. Результат синтеза мультиплексора 4-к-1 с неполным описанием if

Можно также добавить, что состояния «неизвестно», «не инициализировано» обычно выделяются цветом при графическом построении временных диаграмм сигналов средствами моделирования, что позволяет быстро идентифицировать ошибки.

3.4.3. Конструкция множественного выбора case

Условная конструкция case соответствует оператору «выбор по переменной-селектору» в языках программирования и в Verilog имеет вид:

```
case (selector)
option1: <statement1>;
option2: <statement2>;
default: <if nothing else statement>;
endcase
```

Здесь selector – это выражение или переменная-селектор, на основании значения которого выполняется один из вариантов действий, перечисленных ниже в строках вида

<option n> : <statement n> ;

Допустимы множественные варианты выбора в одной строке, например:

```
2'b00, 2'b11 :
```

Аналогично конструкции if, если необходимо выполнить более одного оператора, они должны быть указаны в операторных скобках begin/end.

Пример 33. Описание мультиплексора 4-к-1 с использованием конструкции case

```
module fig (
    input d, c, b, a,
    input [1:0] sel,
    output reg q);

always @ *
case (sel)
    2'b00 : q = a;
    2'b01 : q = b;
    2'b10 : q = c;
    2'b11 : q = d;
    default : q = 1'bx;
endcase
endmodule
```

Если разработчик хочет получить в результате синтеза конструкции case мультиплексор, то для выбранных назначений сигнала должны использоваться все возможные комбинации (рис. 46). Если часть из них

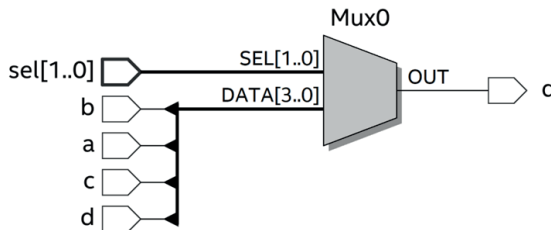


Рис. 46. Результат синтеза мультиплексора 4-к-1, описанного с использованием конструкции case

не используется, в описание необходимо включить ветвь `default` у оператора `case` (ветка оператора `case` по умолчанию), иначе результат синтеза будет содержать защелки (по аналогии с синтезом конструкции `for` без `else`) [44].

«Идеальная» конструкция `case` должна содержать полный список возможных значений переменной-селектора, причем каждое значение должно упоминаться однажды. Если эта рекомендация не соблюдена, то возможными негативными последствиями могут быть добавление защелки и реализация приоритетного дешифратора (если одно и то же условие задано в разных вариантах выбора, требуется проверка приоритета).

Сравнив реализации мультиплексора с помощью оператора `if/else` и `case`, можно сделать вывод об их логической эквивалентности. Действительно, описанные аналогичным образом такие схемы будут давать одинаковые результаты при работе. Более того, для простых примеров схемная реализация, скорее всего, окажется идентичной (это зависит от особенностей конкретных САПР) [15, 21, 29, 40]. Однако для микросхем программируемой логики рекомендуется при наличии возможности использовать именно оператор `case`, поскольку вложенные операторы `if/else` формируют «каскадную» структуру, при которой следующее условие может быть проверено только после того, как подтверждено невыполнение предыдущего. Такая структура обуславливает дополнительную задержку при распространении сигналов, проверяемых в самом первом условном выражении. В то же время для оператора `case` существуют шаблоны проектирования, используемые САПР ПЛИС, которые позволяют, в частности, реализовать мультиплексоры с применением дополнительных ресурсов программируемых ячеек. Такая схема может оказаться быстрее и компактнее, чем схема, синтезированная из формально аналогичного оператора `if/else`. Разумеется, оператор `if/else` не может быть заменен при проверке сложных условий, которые не могут быть сведены к проверке единственной переменной-селектора.

Основные приемы описания комбинационных схем на языке Verilog рассмотрены. Большинство рассмотренных операторов и конструкций могут использоваться и в описании синхронных схем. В реальных ПЛИС-проектах строго разделить эти два вида описаний не представляется возможным и не требуется, так как они используются совместно.

3.5. Пример разработки асинхронной схемы: арифметико-логическое устройство

Арифметико-логическое устройство (АЛУ) – структурный блок процессора. Его задачей является выполнение арифметических и логических операций над данными, поступающими на вход. Обычно АЛУ бывает двухпортовым, т. е. принимает на вход два операнда. Если выполняемая операция является унарной, т. е. требует только одного операнда, второй вход может быть проигнорирован. Выполняемая операция задается кодом операции, который содержится в соответствующем поле машинной команды. Модуль АЛУ может быть выполнен как асинхронная схема.

Выполним разработку модуля двухпортового АЛУ.

Задание на разработку

Разрядность слова данных – 4 бит.

Разрядность кода операции – 4 бит, старший (левый) разряд резервный, всего система команд содержит 8 операций.

Перечень кодов и соответствующих им операций:

0000 – NOP, пустая операция, на выходе 0;

0001 – NOT, поразрядное «НЕ», унарная операция;

0010 – LSHIFT, сдвиг влево, унарная операция;

0011 – RSHIFT, сдвиг вправо, унарная операция;

0100 – MOV, пересылка (один из входов передается на выход);

0101 – AND, поразрядное логическое «И», бинарная операция (участвуют два операнда);

0110 – OR, поразрядное логическое «ИЛИ», бинарная операция;

0111 – SUM, арифметическое сложение, бинарная операция.

На первом этапе необходимо определить интерфейс разрабатываемого модуля, т. е. перечень входных и выходных портов.

Как можно заключить из описания работы АЛУ, у него есть информационные и управляющие входы.

На управляющий вход поступает код операции.

На информационные входы поступают обрабатываемые данные.

Разрядность обрабатываемых данных может быть произвольной, поэтому эту характеристику АЛУ можно описать в виде параметра. Тогда, изменяя его значение, мы сможем синтезировать АЛУ требуемой разрядности без редактирования описания модуля.

Разрядность кода операции зависит от реализуемой системы команд. Ее тоже можно вынести в качестве параметра, но изменение системы команд в любом случае потребует серьезной переработки всего описания модуля.

Выходной порт у АЛУ один. Поскольку в системе команд имеются операции сдвига и сложения, при выполнении которых может возникнуть перенос, разрядность выходного порта модуля АЛУ должна быть на 1 больше, чем разрядность информации, поступающей на вход.

Данной информации достаточно, чтобы описать модуль АЛУ.

Пример 34. Модуль АЛУ

```
module ALU
//параметр - разрядность
#(parameter N = 4)
(
//разрядность выходного порта на 1 больше
//выходной порт - результат
    output reg [N:0] c,

//управляющая шина кода операции
    input [3:0] instr,

// входные информационные порты - операнды
    input [N-1:0] a,
    input [N-1:0] b
);

//описание логики работы АЛУ
always @ (*)
    case (instr)
        4'b0000: c = z; // NOP
        4'b0001: c = ~a; // NOT
        4'b0010: c = a << 1; // LSHIFT
        4'b0011: c = a >> 1; // RSHIFT
        4'b0100: c = b; // MOV
        4'b0101: c = a & b; // AND
        4'b0110: c = a | b; // OR
        4'b0111: c = a + b; // SUM
        default: c = z;
    endcase
endmodule
```

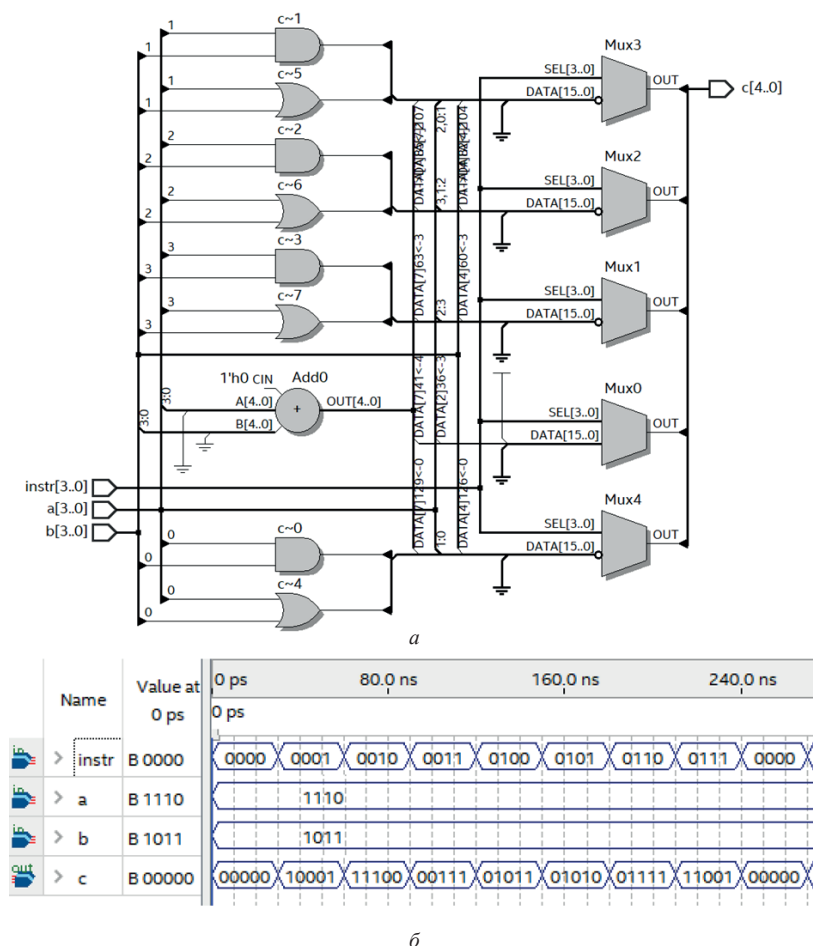


Рис. 47. Результат синтеза (а) и временные диаграммы работы 4-разрядного АЛУ (б)

Результат синтеза схемы и временные диаграммы ее работы приведены на рис. 47.

Для тестирования схемы АЛУ на входы a и b поданы два операнда, и над ними последовательно выполнены все команды, входящие в систему команд. Результаты выполнения операций отображаются на выходной шине c .

Резервный разряд кода операции позволяет нам наращивать систему команд: если понадобится добавить еще какие-либо операции, то в конструкцию множественного выбора case просто добавляется еще один вариант выбора с неиспользованным значением селектора.

Вопросы и задания

1. Дайте понятие асинхронной комбинационной схемы.
2. Каковы области применения асинхронных комбинационных схем?
3. В чем назначение операции непрерывного присваивания assign и как она используется в Verilog?
4. Перечислите способы функционального описания асинхронных комбинационных схем.
5. Приведите пример описания модуля с одной и той же функциональностью разными способами, охарактеризуйте преимущества и недостатки каждого.
6. Приведите пример использования поведенческого блока always в описаниях комбинационных схем.
7. Опишите конструкции цикла Verilog, их роль в синтезируемом подмножестве. Приведите пример.
8. Приведите пример использования тернарной операции «? :>» для описания условной логики.
9. Приведите пример использования условной конструкции if для описания условной логики.
10. Приведите пример использования конструкции множественного выбора case для описания условной логики.
11. Для чего служит опция default конструкции множественного выбора case?
12. В каких случаях при синтезе условной логики программа синтеза добавляет триггер-защелку (latch)?
13. Почему триггер-защелка (latch) является нежелательным компонентом синтезированной схемы?
14. Каким образом можно избежать появления триггера-защелки при синтезе условной логики?

4. СИНТЕЗИРУЕМОЕ ОПИСАНИЕ СИНХРОННЫХ СХЕМ

В рассмотренной ранее асинхронной (комбинационной) логике сигналы на выходе схемы зависят исключительно от уровня сигналов на их входе в текущий момент. Это позволяет реализовать любую логическую функцию, но не предоставляет возможности для хранения информации. Помимо этого использование сложных асинхронных схем в реальных устройствах порождает ряд проблем, связанных с временными задержками распространения сигналов. Поэтому в цифровых схемах часто используется синхронная (последовательностная, тактируемая) логика.

Синхронная (тактируемая) логика – разновидность взаимодействия логических элементов цифровых устройств, при которой изменение состояния устройства происходит после определенного события. Обычно таким событием является появление фронта (нарастающего или падающего) определенного сигнала – синхросигнала (тактового сигнала). В промежутках между синхронизирующими событиями система сохраняет свое состояние, т. е. реализует функцию хранения [10, 21, 39–41].

Структурно синхронная схема представляет собой цифровую схему, состоящую из вентилях, цепей распространения сигналов и собственно устройств хранения – триггеров. Изменение состояния триггера происходит по фронту сигнала синхронизации (синхросигнала). Одним из применений синхронной логики является построение цифровых автоматов.

Описание синхронной логики в языке Verilog выполняется с использованием поведенческих блоков `always`.

4.1. Поведенческий блок `always` при описании синхронных схем

При описании особенностей применения поведенческого блока `always` в асинхронных схемах было указано, что все изменения в них выполняются только при изменении сигналов, описанных в списке чув-

ствительности. Однако если в списке чувствительности просто указать цепь, несущую синхросигнал, то изменения будут происходить при смене уровня сигнала, а синхронная логика должна реагировать не на уровень, а на фронт (нарастающий или падающий).

Для описания именно фронта сигнала в языке Verilog используются специальные ключевые слова:

- переход из состояния 0 в состояние 1 – нарастающий фронт. Описывается в Verilog ключевым словом `posedge` (от `positive edge` – положительный фронт);
- переход из состояния 1 в состояние 0 – спадающий фронт, `negedge` (соответственно от `negative edge` – отрицательный фронт).

При моделировании условие `posedge clk` выполняется также и при переходах от X к 1 или от Z к 1, соответственно `negedge` – при переходе от X или Z к уровню 0.

Основное использование блока `always` в синтезируемых схемах – реализация синхронных схем, простейшим примером которых является D-триггер (D-flip-flop). Это устройство, имеющее вход данных `d` и выход данных `q`, а также тактовый вход `clk`. По фронту тактового сигнала выход принимает значение, равное действующему в этот момент на входе. D-триггер, таким образом, является простейшим элементом памяти. Описание D-триггера на Verilog с помощью процедурного блока `always` приведено ниже.

Пример 35. Описание D-триггера

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

Результат синтеза примера приведен на рис. 48.

В списке чувствительности данного примера появилось ключевое слово `posedge`. Этот момент времени соответствует, таким образом, нарастающему фронту сигнала `clk`. Внутри процедурного блока находится единственный оператор `q <= d`; При его выполнении выход `q` будет принимать значение, равное `d`. Поскольку выполнение этого оператора

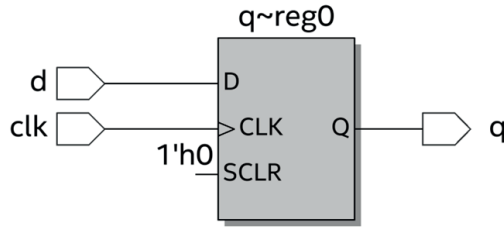


Рис. 48. Результат синтеза D-триггера

будет привязано к моментам положительных перепадов тактового сигнала, итогом синтеза для такого выражения будет D-триггер. Обратите внимание, что для сигнала *q* использован тип *reg*.

Комбинируя условия внутри процедурного блока, можно получить варианты триггера с различными особенностями поведения.

К управляющим сигналам триггера обычно относятся:

- разрешение работы (Clock Enable, Enable, CE) – сигнал, разрешающий изменение состояния триггера. Высокий уровень на этом входе разрешает выполнение оператора $q \leq d$;
- сброс или очистка – сигнал, устанавливающий триггер в состояние нуля. Различают синхронный сброс, выполняющийся по фронту тактового сигнала (обычно его и обозначают как *reset*), и асинхронный, выполняющийся немедленно при появлении на управляющем входе активного уровня, который обычно обозначают как *clr* (*clear*, очистка);
- *set* (установка) – сигнал, устанавливающий триггер в состояние единицы. Как и для сброса, различают синхронную установку (*set*), выполняющуюся по фронту тактового сигнала, и асинхронную (*preset*), выполняющуюся немедленно.

Кроме того, все управляющие сигналы могут иметь как высокий, так и низкий логический активный уровень. Например, если при появлении единицы на входе *reset* происходит сброс триггера, то говорят, что активным уровнем для входа *reset* является высокий, т. е. сигнал логической единицы [40, 41].

Поведение триггера при одновременном действии нескольких сигналов с противоположным смыслом определяется описанием на

Verilog. Например если сначала проверить сброс, а потом установку, то сброс будет обладать приоритетом, поскольку при успешной проверке условия его наступления будет выполнено присвоение выходу нуля, а дальнейшие проверки производиться не будут.

4.2. Синхронные и асинхронные управляющие сигналы

Рассмотрим способы реализации синхронных и асинхронных управляющих сигналов на примере сигнала «сброс». Сброс – сигнал, устанавливающий компоненты устройства в заранее определенное (исходное) состояние [44].

Различают синхронный сброс, выполняющийся по фронту тактового сигнала (обычно его и обозначают как *reset*), и асинхронный, выполняющийся немедленно при появлении высокого уровня, который обычно обозначают как *clr* (*clear*, очистка).

Сброс может быть добавлен к триггеру как часть комбинационной логики, генерирующей входные данные на входе *d* триггера (рис. 48). При этом соответственно увеличится время прохождения данных через всю цепь. Если же говорить о *FPGA*, то такое выполнение сброса приведет к дополнительному увеличению числа интерконнектов. В некоторых библиотеках, которые поставляют изготовители микросхем, могут присутствовать компоненты с отдельно выполненным входом синхронного сброса. Если таковые компоненты имеются, то использование их синхронных входов для сброса предпочтительнее, чем объединение сброса с комбинационной логикой на входах загрузки данных [44].

4.2.1. Синхронный сброс

Для того чтобы моделировать синхронный сброс, необходимо добавить в описание схемы на языке описания аппаратуры условие *if/else*. При этом рекомендуется в секции *if* описать условие для сброса, а в секции *else* – всю остальную комбинационную логику, необходимую для работы триггера.

Пример 36. Описание триггера с синхронным сбросом

```

module synchro
#(parameter N=3)
(
output reg [N-1:0] q,
input [N-1:0] d,
input clk, rst_n
);
always @(posedge clk)
    if (!rst_n) q <= 1'b0;
    else q <= d;
endmodule

```

Результат синтеза примера и временные диаграммы его работы приведены на рис. 49.

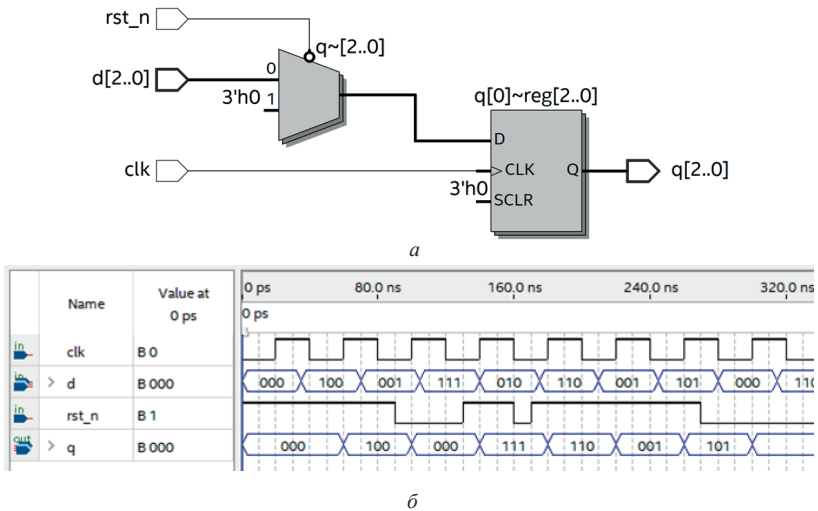


Рис. 49. Результат синтеза (а) и временные диаграммы работы (б) регистра с синхронным сбросом

Отметим, что сброс не входит в список чувствительности. Как мы знаем, именно по изменению тех сигналов, которые находятся в списке чувствительности, симулятор и определяет новое значение переменных. Поэтому для языка Verilog исключение сброса из списка чувстви-

ности показывает то, что этот сброс является синхронным, так как теперь симулятор производит новые назначения переменным только при прохождении активного фронта синхросигнала, а в момент изменения сигнала сброса (см. временные диаграммы на рис. 49).

Преимущества синхронного сброса: он позволяет получить полностью синхронную схему. Как и во всех других синхронных схемах, срабатывание только по фронту синхросигнала работает как фильтр для подавления очень коротких случайных импульсов на шине сброса, как это показано на временной диаграмме: на 160 нс можно видеть кратковременное падение сигнала сброса, которое могло вызвать срабатывание, но этого не произошло, так как ложный сигнал был снят раньше, чем пришел активный фронт синхросигнала. Но если ложные импульсы происходят около активного фронта синхросигнала, то сбой оборудования все равно может произойти.

Недостатки синхронных сбросов: разработчик аппаратуры должен быть уверен, что исходный сигнал, приходящий на вход «сброс», имеет достаточную продолжительность, чтобы за это время пришел хотя бы один синхроимпульс, иначе он будет «отфильтрован», так же как это происходит с ложными импульсами. На самом деле, если импульс на 160 нс не был ложным, а просто был снят раньше прихода фронта синхросигнала, он будет проигнорирован, т. е. снова возникнет ошибка. Поэтому длительность приходящего импульса должна быть не меньше, чем период синхрочастоты, и импульс сброса должен присутствовать в течение активного фронта синхрочастоты [44].

4.2.2. Асинхронный сброс

Асинхронный сброс – наиболее широко применяемый способ установки цифровых устройств в исходное состояние. Как правило, дополнительные входы асинхронного сброса и установки входят в состав триггеров. Вход сброса обычно имеет активный низкий уровень [40, 44].

Однако асинхронные сбросы могут быть очень опасными. Важно корректно определить способы подачи и снятия сигнала в схеме.

Пример 37. Описание триггера с асинхронным сбросом

```
module synchro
#(parameter N=3)
```

```
(  
output reg [N-1:0] q,  
input [N-1:0] d,  
input clk, rst_n  
);  
always @(posedge clk, negedge rst_n)  
    if (!rst_n) q <= 1'b0;  
    else q <= d;  
endmodule
```

В этом случае сигнал сброса входит в список чувствительности. Для языка Verilog добавление сигнала сброса к списку чувствительности заставляет симулятор сделать новое назначение переменным при изменении сигнала сброса, что и делает этот сигнал асинхронным. В примере блок процедуры `always` будет обновляться по переднему фронту сигнала сброса, а условный оператор `if` проверит правильность уровня этого сигнала. Анализируя временные диаграммы работы триггеров с синхронным и асинхронным сбросом, можно увидеть различие между ними: в случае асинхронного сброса переключение триггера в исходное состояние осуществляется немедленно после появления на входе сброса активного уровня сигнала. И даже кратковременная установка сигнала сброса в активный уровень на 160 нс вызвала сброс триггера.

Результат синтеза примера и временные диаграммы его работы приведены на рис. 50.

Недостатки асинхронных сбросов: самая большая проблема с асинхронными сбросами состоит именно в том, что они являются асинхронными и во время активного состояния сигнала, и в момент снятия активного состояния сигнала сброса. Причем если установка активного состояния сигнала не представляет трудностей, то момент снятия активного состояния сигнала сброса – это действительно проблема для разработчика.

Еще одна проблема асинхронного сброса состоит в том, что вход асинхронного сброса может быть чувствителен к ложным сигналам, которые могут возникнуть из-за шума или сбоев на плате, но проверка прохождения сигнала сброса при тестировании проекта в среде моделирования не выявляет никаких проблем. В реальной системе некорректно поданный асинхронный сигнал сброса может вызвать ложное срабатывание или перевести отдельные модули в неизвестное (метастабильное) состояние и, таким образом, не произвести правильный сброс. Однако

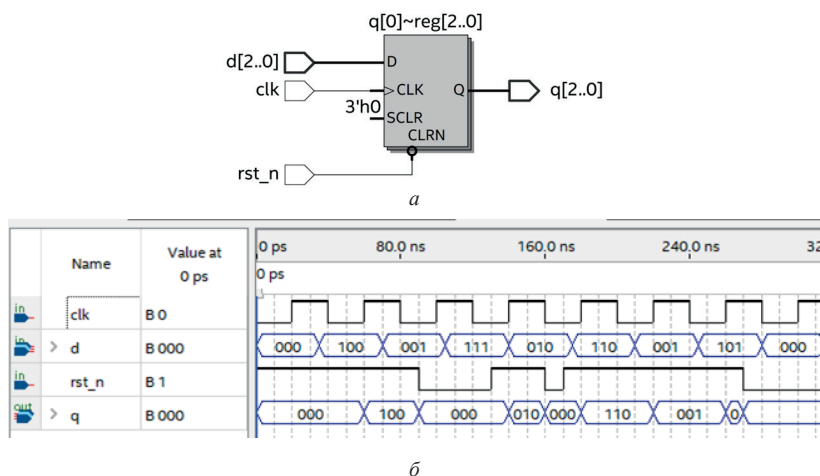


Рис. 50. Результат синтеза (а) и временные диаграммы работы (б) регистра с асинхронным сбросом

многие разработчики используют асинхронный сброс, не принимая никаких мер для его надежного срабатывания [44].

Поскольку сигнал сброса должен быть подан из одной точки ко многим триггерам, то он имеет большое разветвление (fanout) и под сигнал сброса обычно задействуется одна из глобальных цепей внутри кристалла. Временной анализ для такой цепи сброса должен быть выполнен после того, как произведено размещение проекта пользователя на кристалле. Это необходимо для того, чтобы гарантировать, что все требования по синхронизации соответствуют заданным [21, 44].

Вопросы корректного проектирования синхронных и асинхронных управляющих сигналов подробно рассматриваются при углубленном изучении проектирования на ПЛИС.

4.3. Блокирующее и неблокирующее присваивание

В приведенных выше примерах для описания присваивания были использованы две разные конструкции: оператор « $=$ » и оператор « $<=>$ ». Это не ошибка, а два различных оператора, соответствующих так назы-

ваемому блокирующему («=») и неблокирующему («<=») присваиваниям. Понимание особенности выполнения этих операторов крайне важно для разработки ПЛИС-проектов.

Блокирующее присваивание (оператор «=») называется так потому, что вычисления производятся строго в порядке, описанном в *always*-блоке. Второе выражение вычисляется только после первого, и результат второго выражения может зависеть от результата первого. Можно сказать, что этот вид присваивания максимально приближен к оператору присваивания в обычном программировании.

Неблокирующее присваивание в *always*-блоке (оператор "<=") обозначает факт одновременного запоминания вычисленных значений в соответствующих регистрах [21, 40, 42]. Выражения с неблокирующим присваиванием в *always*-блоке описывают процессы, происходящие в аппаратуре одновременно (параллельно), и вычисление изменений сигналов также будет выполнено параллельно, и само изменение произойдет во всех неблокирующих присваиваниях одновременно.

Наиболее наглядно принципиальную разницу этих двух операторов можно увидеть при анализе результатов их синтеза и на временных диаграммах.

Пример 38. Пример использования блокирующих операторов присваивания

```
module synchro(
    input wire clock,
    input wire [3:0]in,
    output reg [3:0]x,
    output reg [3:0]y,
    output reg [3:0]z );
    always @(posedge clock)
    begin
        x = in + 1;
        y = x + 1;

        z = y + 1;
    end
endmodule
```

Результат синтеза примера и временные диаграммы его работы приведены на рис. 51.

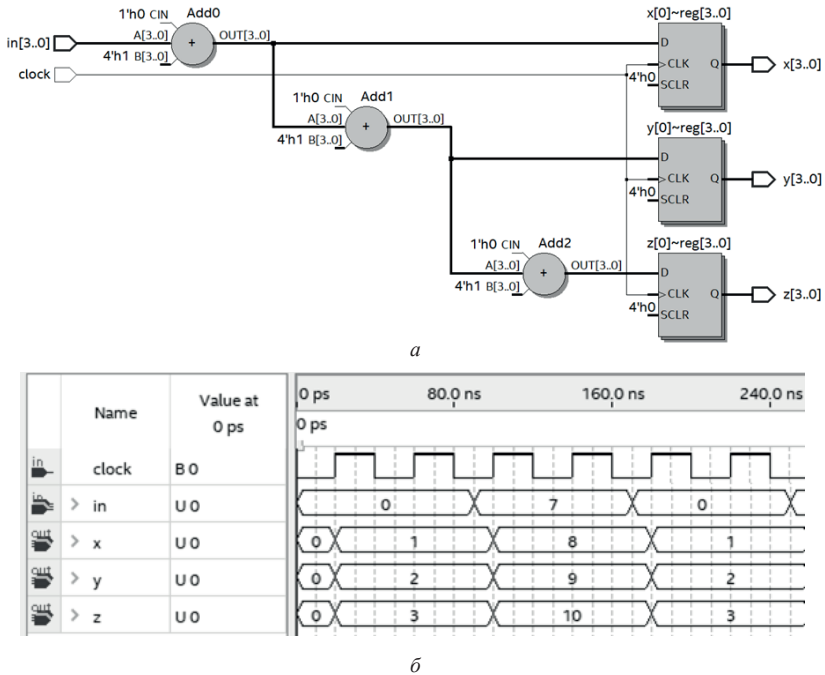


Рис. 51. Результат синтеза блокирующих операторов присваивания (а) и временные диаграммы работы (б) синтезированного устройства

На синтезированной схеме видно, что сигналы на информационных входах регистров x , y и z формируются только с участием асинхронной комбинаторной логики и одновременно заносятся в регистры по нарастающему фронту синхросигнала.

Пример 39. Пример использования неблокирующих операторов присваивания

```
module synchro(
    input wire clock,
    input wire [3:0] in,
    output reg [3:0] x,
    output reg [3:0] y,
    output reg [3:0] z );
    always @(posedge clock)

```

```

begin
    x <= in + 1;
    y <= x + 1;
    z <= y + 1;
end
endmodule

```

Как видно, различие в описаниях модулей из последних двух примеров состоит только в используемом виде оператора присваивания. Однако результаты их синтеза отличаются кардинально.

Результат синтеза примера и временные диаграммы его работы приведены на рис. 52.

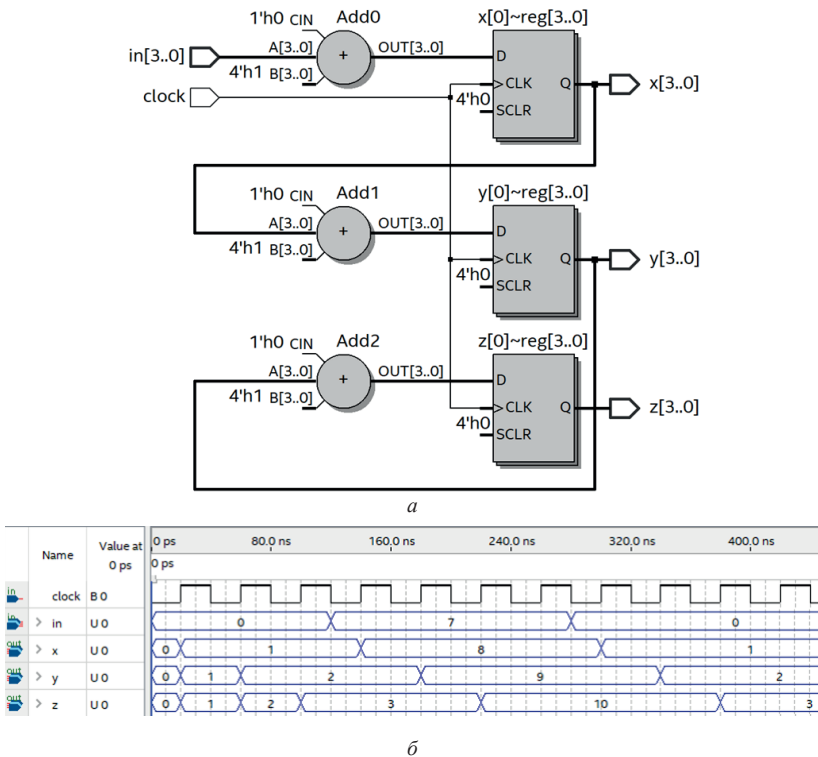


Рис. 52. Результат синтеза неблокирующих операторов присваивания (а) и временные диаграммы работы (б) синтезированного устройства

В этой схеме сигналы, участвующие в формировании информационных входов каждого следующего регистра, поступают с выхода предыдущего. Следовательно, изменение входного сигнала вызовет изменение в регистре x после прохождения первого синхроимпульса, регистр y изменится после второго, а z – после третьего.

Можно запомнить несложные практические правила:

- блокирующие присваивания используются для чисто комбинаторной логики;
- неблокирующие присваивания используются для синхронной (тактируемой) и смешанной логики.

Оба вида операторов присваивания могут использоваться в поведенческих блоках `always`.

4.4. Синтезируемое описание типовых синхронных схем, используемых в ПЛИС-проектах

Язык описания аппаратуры Verilog предоставляет разработчикам ИС и ПЛИС-проектов универсальные средства для описания цифровых устройств. Но в большинстве случаев даже самые сложные проекты содержат ряд модулей, обеспечивающих выполнение некоторого набора базовых функций.

В синхронной логике к этому перечню можно отнести:

- память различной организации (ROM, RAM) и с различной организацией шин адреса/данных;
- регистры с расширенной функциональностью, в первую очередь сдвиговые;
- счетчики с разной организацией счета: инкрементальные, декрементальные, реверсивные, с фиксированным или задаваемым коэффициентом пересчета, в коде Грея и др.

В большинстве средств САПР есть соответствующий набор шаблонов для этих конструкций, которые можно использовать в качестве отправной точки для проектирования.

Рассмотрим несколько примеров описаний перечисленных блоков.

4.4.1. Модули памяти

Ранее говорилось, что модули памяти могут быть описаны как массив регистров. При этом обращаться к отдельной ячейке памяти можно по номеру элемента в массиве. Однако обычно модули памяти имеют дополнительные управляющие сигналы и различные режимы работы, поэтому для описания памяти в проекте необходимо описать соответствующий модуль.

Пример 40. Описание модуля однопортового ПЗУ – ROM (Read Only Memory)

```
module memory
// задается разрядность данных и адреса
#(parameter D=8, parameter A=8)
(
    input [(A-1):0] addr, // входная шина адреса
    input clk,           // вход синхросигнала
    output reg [(D-1):0] q // выходная шина данных
);

// Объявление памяти как массива регистров
reg [D-1:0] rom[2**A-1:0];

//Следующий далее фрагмент используется для задания на-
чального значения ячейкам памяти
// Конструкция initial не является синтезируемой,
// она используется для проверки работы модуля
initial
    begin
        rom[1]<=1;
        rom[2]<=2;
        rom[3]<=3;
        rom[4]<=5;
    end

// Инициализация ПЗУ с помощью $ readmemb.
// Занесите содержимое памяти в файл single_port_rom_init.
txt.
// Раскомментируйте следующий ниже блок initial
// Внимание! Без файла содержимого памяти
// этот проект не скомпилируется.
```

```

// initial
// begin
//     $readmemb("single_port_rom_init.txt", rom);
// end
// описание работы памяти
always @ (posedge clk)
    begin
        q <= rom[addr];
    end
endmodule

```

Результат синтеза примера и временные диаграммы его работы приведены на рис. 53. В данном примере мы ограничились занесением данных и проверкой только первых четырех ячеек. Интерес представляет результат синтеза: как видно на рисунке, синтезатор не создал массив регистров, а задействовал встроенный модуль памяти, размещенный на целевой ПЛИС. Временная диаграмма показывает подаваемые на выходную шину модуля данные из ячеек памяти, адресуемых по входной шине адреса. Начальное

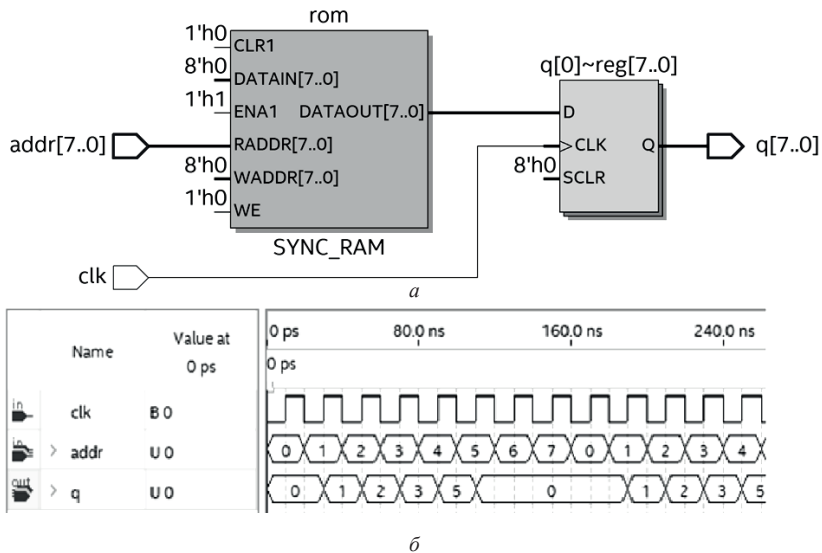


Рис. 53. Результат синтеза модуля ROM (а) и временные диаграммы работы (б) синтезированного устройства

содержимое памяти было задано непосредственно в описании модуля с целью отладки и в дальнейшем должно быть удалено. В примере в комментариях дано краткое указание, каким образом можно загрузить в ROM начальные значения из файла, пользуясь системными функциями Verilog.

Наиболее часто в ПЛИС-проектах возникает необходимость в модулях оперативной памяти ОЗУ (RAM – Random Access Memory), которые выполняют как чтение, так и запись. Пример описания модуля ОЗУ приведен ниже.

Пример 41. Описание модуля однопортового ОЗУ – RAM (Random Access Memory)

```
module memory
// задается разрядность данных и адреса
#(parameter D=4, parameter A=3)
(
    input [(D-1):0] data,
    input [(A-1):0] addr,
//we – сигнал разрешения записи
    input we, clk,
    output [(D-1):0] q
);

// Объявляется массив регистров – собственно модуль памяти
    reg [D-1:0] ram[2**A-1:0];
// Объявляется регистр адреса
    reg [A-1:0] addr_reg;

    always @ (posedge clk)
    begin
// Запись
        if (we)
            ram[addr] <= data;
            addr_reg <= addr;
    end

// Непрерывное присваивание пропускает на выход новые (за-
// носимые) значения данных
// Это типичное поведение однопортовой памяти
    assign q = ram[addr_reg];
endmodule
```


выполняется повторное занесение новых данных и их чтение. Временные диаграммы подтверждают правильное функционирование модуля.

Существуют и более сложные конструкции памяти. Описание самых распространенных из них обычно представлено в виде шаблонов и стандартных библиотечных модулей программных сред разработки ПЛИС-проектов.

4.4.2. Регистры с расширенной функциональностью

При выполнении большого числа операций требуется производить сдвиг обрабатываемого слова данных, поскольку это позволяет осуществить его поразрядную обработку; на базе сдвиговых регистров может быть реализован распределитель тактовых сигналов; сдвиг на N двоичных разрядов влево эквивалентен умножению числа на 2^N . Существуют и другие варианты использования этого действия. Сдвиг может выполняться различными способами: в разных направлениях (вправо или влево), циклически или с заполнением освободившихся разрядов определенным значением, с использованием разряда переноса и т. д.

Рассмотрим описание модуля регистра с расширенной функциональностью на примере реверсивного сдвигового регистра. Реверсивный сдвиговый регистр позволяет выполнить поразрядный сдвиг содержащейся в нем информации вправо или влево. Направление сдвига задается управляющим сигналом. Данные из разряда, который выталкивается из регистра, заносятся в выходной триггер. В освобождающийся в результате сдвига разряд заносится информация с информационного входа.

Пример 42. Описание модуля реверсивного сдвигового регистра

```
module ShiftReg
// Параметр N – разрядность регистра
#(parameter N=8)
(
    input clk, // вход синхросигнала
    enable,    // управляющий сигнал разрешения работы
    d,         // управляющий сигнал направления сдвига
               // высокий уровень – сдвиг влево,
               // низкий уровень – сдвиг вправо.
    input sr_in, //информационный вход
```

```

output reg sr_out, //выходной триггер
output reg [N-1:0] sr //регистр
);

always @ (posedge clk)
begin
    if (enable == 1'b1)
// если работа разрешена
    begin
        if (d == 1'b1)
//если сдвиг влево
        begin
            sr_out <= sr[N-1];
            sr[N-1:1] <= sr[N-2:0];
            sr[0] <= sr_in;
        end
    else
//если сдвиг вправо
    begin
        sr_out <= sr[0];
        sr[N-2:0] <= sr[N-1:1];
        sr[N-1] <= sr_in;
    end
end
end
endmodule

```

Результат синтеза примера и временные диаграммы его работы приведены на рис. 55. На временных диаграммах видно, как единичное значение заносится в регистр с информационного входа и перемещается в соответствии с направлением, заданным управляющим сигналом. Изменения состояния регистра происходят в момент прихода нарастающего фронта синхроимпульса.

4.4.3. Таймеры-счетчики

В работе реальных цифровых устройств большое значение имеет обеспечение определенных временных интервалов, например времени удержания сигнала, интервала между моментом поступления сигнала о начале передачи данных и началом их приема и т. д. При создании

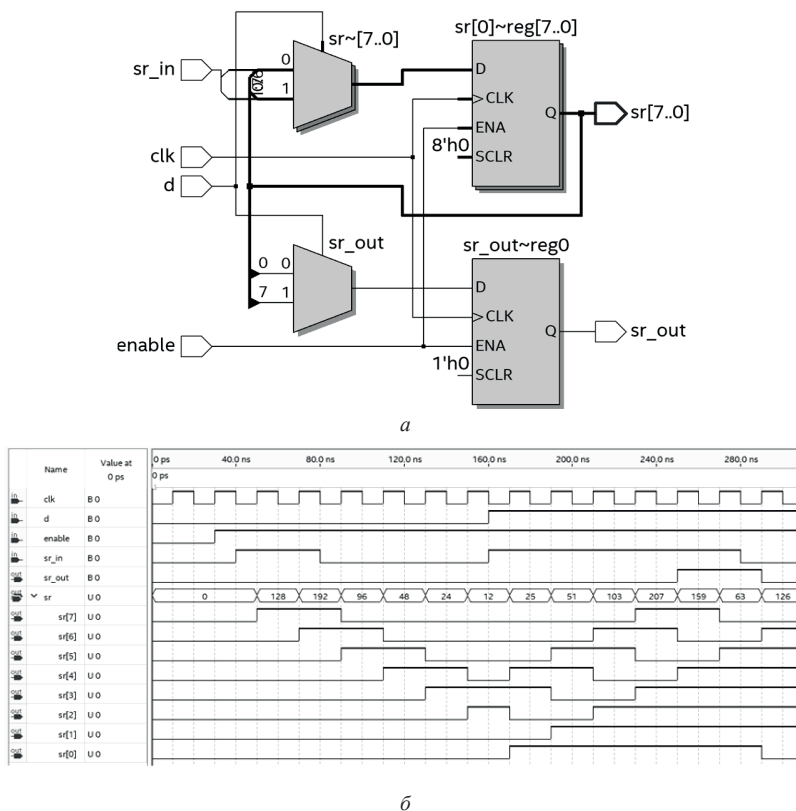


Рис. 55. Результат синтеза модуля реверсивного сдвигового регистра (а) и временные диаграммы работы (б) синтезированного устройства

ПЛИС-проекта на языке описания аппаратуры возникает вопрос: каким образом должны быть обеспечены требуемые временные соотношения? Языки описания аппаратуры, в частности Verilog, имеют такую конструкцию, как временные задержки, которая может быть неправильно интерпретирована. Начинающие разработчики могут воспринять задержки сигналов как инструмент для описания подобных временных зависимостей. Однако задержки в Verilog не относятся к синтезируемому подмножеству, а используются только при моделировании. Для обеспечения требуемых временных параметров в ПЛИС-проектах используются счетчики.

Чтобы корректно спроектировать счетчик временной задержки, необходимо знать требуемую задержку T и частоту синхросигнала ν . Зная эти величины, можно вычислить коэффициент пересчета счетчика задержки K , который определяет, через какое число тактовых сигналов счетчик должен выдавать выходной сигнал о достижении необходимого интервала времени: $K = T/\nu$. K должно быть целым числом, если результат дробный, округление выполняется в большую сторону, так как обычно незначительное превышение требуемого времени задержки не является критическим.

Предположим, необходимо обеспечить задержку в 100 нс при частоте синхросигнала 10 нс. В этом случае коэффициент пересчета $K = 10$.

Пример 43. Описание модуля счетчика временной задержки

```
module DelCount
// LIMIT - коэффициент пересчета
// N - разрядность счетчика
#(parameter LIMIT=10; N=4)
(
    input clk, enable, reset
    output reg P, //выходной сигнал, формируемый
                //при достижении коэффициента пересчета
    output reg [N-1:0] count
);

always @ (posedge clk or posedge reset)
begin
    if (reset)
        count <= 0;
    else if (enable == 1'b1)
        if (count < LIMIT-1)
            begin
                count <= count + 1;
                P<=1'b0;
            end
        else
            begin
                count <= 0;
                P <= 1'b1;
            end
    end
end
endmodule
```

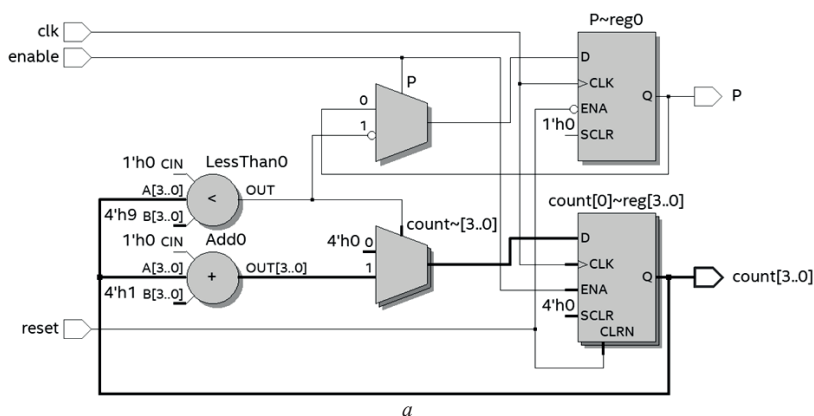
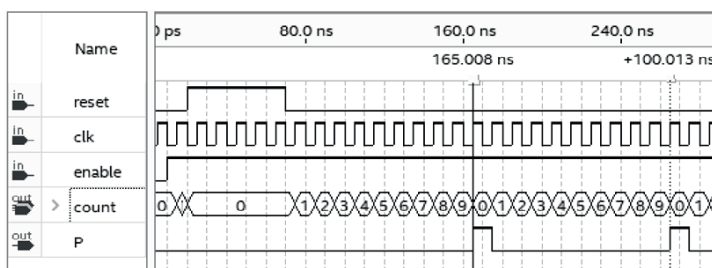


Рис. 56. Результат синтеза модуля таймера-счетчика временной задержки (а) и временные диаграммы работы (б) синтезированного устройства

Результат синтеза примера и временные диаграммы его работы приведены на рис. 56.

В данном случае текущие значения счетчика count выведены в качестве выходных сигналов для упрощения проверки работы модуля. В случае счетчика задержки полезным является только сигнал P, формируемый по истечению требуемого интервала. На шкале времени окна временных диаграмм выставлены отметки, подтверждающие, что интервал между нарастающими фронтами сигнала P соответствует заданию.

Счетчик задержки можно реализовать и на базе вычитающего счетчика; коэффициент пересчета может быть переменным и заноситься во внутренний регистр при начальной установке счетчика.

4.5. Пример разработки модуля памяти: регистровый файл для АЛУ

В п. 3.5 описан модуль АЛУ. Данные на этот модуль поступают через внешние порты, сам модуль является асинхронным, т. е. сигналы на его выходе меняются в зависимости от входных сигналов в произвольный момент времени. В процессорах АЛУ, как правило, подключается к регистровому файлу.

В большинстве случаев процессоры реализуют двухадресные операции, когда в операции участвуют два операнда, а результат ее выполнения записывается в один из них. Операнд, в который записывается результат, называется операндом-приемником, второй – операндом-источником. Если команда является одноадресной, то и источником, и приемником является один и тот же операнд.

Задание на разработку

Регистровый файл должен иметь две шины адреса, которые определяют, данные из каких регистров будут переданы на АЛУ.

Одна из этих шин (та, которая соответствует операнду-приемнику) определяет также, в какой регистр будет записан результат.

Регистровый файл должен иметь две выходные шины данных, по которым на АЛУ будут переданы операнды, и одну входную шину данных, подключенную к выходу АЛУ. Данные с этой шины записываются в операнд-приемник.

Разработка описания регистрового файла

Регистровый файл является синхронной схемой, следовательно, он должен иметь вход синхросигнала. Однако если мы попытаемся выполнить операции чтения (выдачи) операндов и записи результата по одному и тому же фронту синхросигнала, то обнаружим, что этого сделать невозможно.

По фронту синхросигнала сначала должен отработать регистровый файл и выдать операнды на АЛУ.

После этого АЛУ должно выполнить требуемое действие, которое является асинхронным, но требует времени на прохождение сигналов от входа к выходу.

И только после этого можно записать результат в регистр-приемник.

Следовательно, нам потребуются два синхросигнала: для чтения регистрового файла и для записи.

В наборе шаблонов проектирования, предоставляемых Quartus, есть модуль “True Dual Port RAM with dual clocks”, который реализует подобный функционал. После незначительной доработки шаблона в соответствии с заданием получаем описание модуля регистрового файла, приведенное ниже.

Пример 44. Модуль двухпортового регистрового файла с отдельными синхросигналами чтения и записи

```
module regfile_dual
// используем параметры, предлагаемые шаблоном:
// разрядность шин адреса и данных
#(parameter DATA_WIDTH=4, parameter ADDR_WIDTH=3)
(
    output reg [(DATA_WIDTH-1):0] a,
    output reg [(DATA_WIDTH-1):0] b,
// выходные регистры данных

    input [(DATA_WIDTH-1):0] data,
// входная шина данных

    input [(ADDR_WIDTH-1):0] read_addr, write_addr,
// две входные шины адреса

    input we, read_clock, write_clock
// два синхросигнала
);

// объявляется массив регистров с требуемыми параметрами
reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];

// Следующая ниже конструкция initial
// заносит начальные значения
// в указанные регистры регистрового файла
// этот код не синтезируется и используется при отладке
initial
begin
    ram[1] <= 4'b0001;
    ram[2] <= 4'b0010;
end

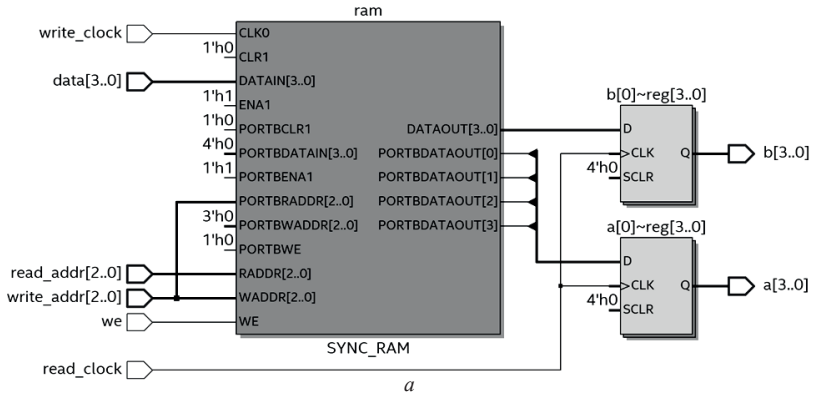
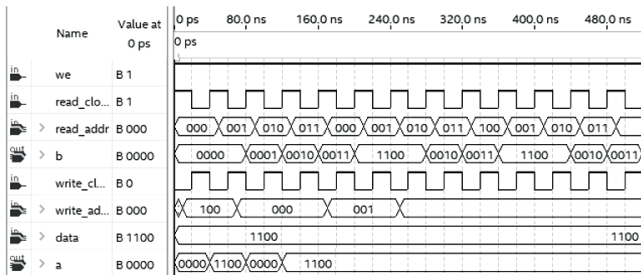
always @ (posedge write_clock)
```

```

//Запись
begin
    if (we)
        ram[write_addr] <= data;
    end

    always @ (posedge read_clock)
    begin
        // Чтение
        b <= ram[read_addr];
        a <= ram[write_addr];
    end
endmodule

```



б

Рис. 57. Модуль двухпортового регистрового файла с отдельными синхросигналами чтения и записи (а) и временные диаграммы его работы (б)

Результат синтеза этого модуля и диаграммы его работы приведены на рис. 57.

Как видно на рисунке, регистровый блок реализован на встроенном модуле памяти. Для проверки работы модуля в текст его описания введена конструкция *initial*, которая задает содержимое ячеек регистрового файла в начальный момент моделирования. Сигналы на временной диаграмме сгруппированы в соответствии с группами ввода/вывода.

4.6. Пример разработки иерархического проекта: процессор

Имея разработанные и отлаженные порознь модули АЛУ и регистрового файла, можно собрать из них элементарный микропроцессор.

Для этого необходимо объединить АЛУ и регистровый файл в общий проект. При этом будет создан новый модуль верхнего уровня.

Задание на разработку

В состав процессора входят АЛУ, регистровый файл и компоненты, обеспечивающие их работу.

На управляющий вход (шину команд) процессора поступает команда. Длина командного слова – 10 бит.

Команда содержит 4-разрядное поле кода операции и два 3-разрядных поля адресов операндов.

Первый адрес определяет регистр-источник, второй – регистр-приемник. При выполнении одноадресных команд сигналы на шине адреса регистра-источника игнорируются.

На вход также поступают сигналы синхронизации и сигнал сброса.

Сброс является асинхронным; по нарастающему фронту сигнала сброса обнуляются или устанавливаются в начальное состояние все регистры процессора. Регистровый файл сигналом сброса не управляется.

Результат работы АЛУ записывается в регистр операнда-приемника и в выходной регистр данных, с которого поступает на выходную шину данных.

Выходными портами процессора являются шина данных и одно-разрядный сигнал переноса, формируемый при выполнении операций.

Разработка описания процессора

Согласно заданию на вход процессора поступает команда, которую затем нужно разделить на поля и раздать на исполняющие модули. Одним из полей является код операции, поступающий непосредственно на АЛУ. Этот код определяет выполняемое АЛУ действие. АЛУ является асинхронной схемой, и если на его входе произойдет изменение любого сигнала, включая код операции, это может повлиять на результат его работы, а значит, и на результат работы всего процессора.

Для того чтобы этого избежать, данные с входной шины команд должны сохраняться в регистре команд.

Разработанный регистровый файл требует двух синхросигналов, по одному выполняется чтение операндов, по второму – запись результата.

Следовательно, логично организовать работу процессора в виде циклов выполнения команд. Каждый цикл выполняется за три такта:

- на первом такте с шины команд считывается команда и заносится в регистр команд, код операции и адреса операндов с регистра команд подаются на АЛУ и регистровый файл;
- на втором такте выполняется чтение операндов из ячеек регистрового файла, операнды подаются на АЛУ;
- на третьем такте результат работы АЛУ заносится в регистр-приемник и в выходной регистр процессора.

Для организации цикла исполнения команд понадобится распределитель тактовых сигналов. Модули АЛУ и регистрового файла описаны выше.

Пример 45. Процессор

```
module CLK_distributor
// распределитель синхроимпульсов
// на основе сдвигового регистра
(
output reg [2:0] order,
// выходной сигнал распределителя -
// 3 синхросигнала для 3 тактов выполнения команды

input wire reset, clk
);
always @ (posedge clk)
begin
    if (reset) order <= 3'b001;
```

```

        else
            order <= {order[1:0], order[2]};
        end
    endmodule

module Processor
# (parameter
// в виде параметров задаем
N = 4,           // разрядность обрабатываемых данных,
INSTR = 4,       // длину кода операции,
ADR = 3,         // разрядность шины адреса
L = INSTR+ADR*2) // вычисляем длину командного слова
(
input clk, reset,
input [(INSTR+ADR*2-1):0] instruction, // шина команд
output reg [N-1:0] result, // выходная шина данных
output reg carry // выход переноса
);

wire [N-1:0] w_a;
wire [N-1:0] w_b;
wire [N:0] w_c;
// шины, используемые для подключения АЛУ

wire w_we = 1;
reg [INSTR-1:0] r_cop;
reg [(ADR-1) : 0] r_adr_op_s;
reg [(ADR-1) : 0] r_adr_op_r;
// регистры полей команды – код операции и адреса операндов

wire [2:0] order;
// цепи распределения внутреннего синхроимпульсов

CLK_distributor clk1
// экземпляр (инстанс) распределителя синхроимпульса
(
.order (order),
.reset(reset),
.clk(clk)
);
ALU
//экземпляр (инстанс) АЛУ
#(.N(N))
alu_1

```

```

(
    .c (w_c),
    .instr (r_cop),
    .a (w_a),
    .b (w_b)
);

regfile_dual
// экземпляр (инстанс) регистрового файла
#(.DATA_WIDTH(N), .ADDR_WIDTH(ADR))
regfile_1
(
    .data(w_c),
    .read_addr(r_adr_op_s),
    .write_addr(r_adr_op_r),
    .we (w_we),
    .read_clock (order [1]),
    .write_clock (order [2]),
    .a(w_a),
    .b(w_b)
);

always @ (posedge order[0] or posedge reset)
// процедурный блок always, описывающий работу
// регистров хранения полей команды
begin
    if (reset)
        begin
            r_cop <= 0;
            r_adr_op_s <= 0;
            r_adr_op_r <= 0;
        end
    else
        begin
            r_cop <= instruction [(L-1):(L-INST)];
            r_adr_op_s <= instruction [(ADR*2-1) : ADR];
            r_adr_op_r <= instruction [(ADR-1) : 0];
        end
    end

always @ (posedge order [2] or posedge reset)
// процедурный блок always, описывающий работу
// выходного регистра данных и переноса
if (reset)

```

```

begin
    result <= 4'b0000;
    carry <= 1'b0;
end
else
begin
    {carry, result} <= w_c;
end
end

endmodule

```

В данном примере процессор описан в смешанном стиле: в нем как присутствуют структурные компоненты (модули нижнего уровня – АЛУ и регистровый файл), так и заданы их подключение и фрагменты функционального (поведенческого) описания (регистры хранения

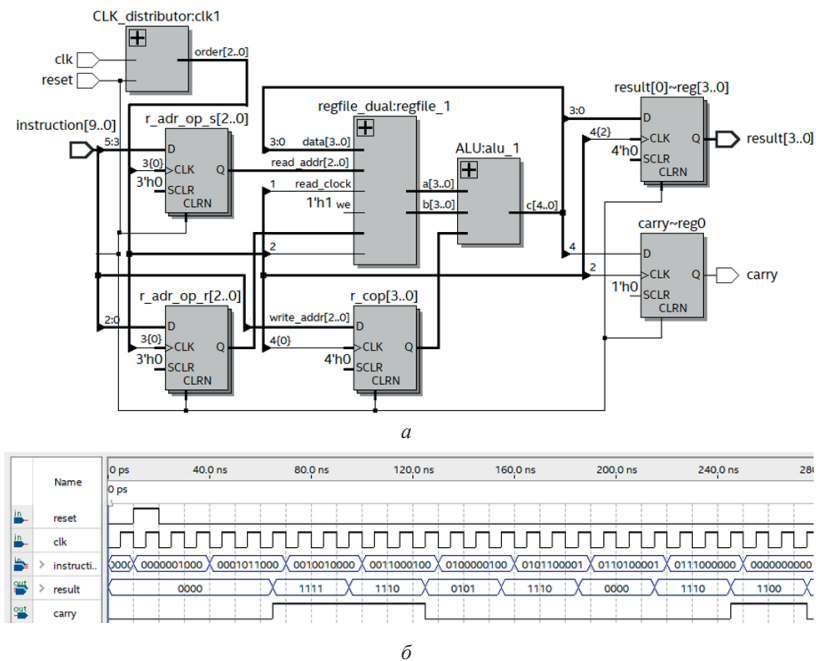


Рис. 58. Результат синтеза модуля АЛУ с регистровым файлом (а) и временные диаграммы его работы (б)

полей команды и выходные регистры). Результат синтеза и временные диаграммы приведены на рис. 58.

Временные диаграммы работы данного модуля слишком сложны для их анализа вручную, поэтому для тестирования сложных проектов используются специальные модули – тестбенчи. Способы и средства анализа временных диаграмм сложных проектов и порядок описания тестбенчей являются обширной темой, выходящей за рамки данного пособия.

4.7. Синтезируемое описание цифровых автоматов

В логических схемах значения выходных переменных определяются только комбинацией значений переменных на входах в данный момент времени. Поэтому их называют комбинационными схемами. В общем случае выходные переменные могут зависеть от значений входных переменных не только в данный момент, но и от их предыдущих значений. Иначе говоря, значения выходных переменных определяются последовательностью значений входных переменных, в связи с чем схемы с такими свойствами называют последовательностными схемами, или цифровыми автоматами. Если входные и выходные состояния автомата принимают конечный набор значений, которые можно рассматривать как символы некоторого алфавита, то они могут быть отнесены к конечным автоматам. Особого внимания заслуживают конечные автоматы с двузначным структурным алфавитом, зависимости между входными и выходными переменными которых выражаются булевыми функциями. При технической реализации таких автоматов используются преимущественно двоичные элементы и двузначная логика [45–47].

В реальных условиях сигналы представляются непрерывными функциями времени; поэтому для надежного различения сигналов требуется, чтобы новые значения на входах появлялись после окончания переходных процессов, связанных с предыдущими значениями. При рассмотрении логической структуры автоматов обычно отвлекаются от существования этих процессов и считают, что переменные изменяются не непрерывно, а мгновенно в некоторые моменты времени, называемые тактами. Предполагается, что тактовые моменты определяются синхро-

низирующими сигналами. Таким образом, вводится понятие дискретного автоматного времени, причем переменные зависят не от физического времени, а от номера такта, т. е. вместо непрерывных функций рассматриваются дискретные значения.

Кроме входных и выходных переменных можно выделить некоторую совокупность промежуточных переменных, которые связаны с внутренней структурой автомата. В комбинационных схемах промежуточные переменные непосредственно не участвуют в соотношениях «вход – выход». Напротив, выходные функции цифровых автоматов в качестве своих аргументов, кроме входных переменных, обязательно содержат некоторую совокупность промежуточных переменных, характеризующих состояние схемы. Набор всех возможных состояний, которые присущи данной схеме, называется множеством состояний.

Ясно, что цифровые автоматы схемы должны обладать способностью сохранять предыдущее состояние до следующего такта, поэтому их называют также автоматами с памятью. В качестве памяти могут использоваться элементы задержки, на выходах которых повторяются входные воздействия со сдвигом во времени на интервал между тактами. Широко применяются также различные запоминающие элементы, например триггеры, способные сохранять состояние на выходах до тех пор, пока оно не изменяется в результате воздействия на их входы.

Теория конечных и цифровых автоматов (FSM, Final State Machine) довольно сложна и разнообразна, поскольку они имеют очень широкое практическое применение в различных областях знаний. Поэтому остановимся только на основных моментах [45–47].

С теоретической точки зрения конечным автоматом K называется набор

$$K = (A, Z, W, \delta, \lambda, a1),$$

где $A = \{a1, \dots, a0\}$ – множество (алфавит) состояний (имеются в виду внутренние состояния автомата, states); $Z = \{z1, \dots, zN\}$ – множество входных сигналов (входной алфавит); $W = \{w1, \dots, wM\}$ – множество выходных сигналов (выходной алфавит); δ – функция переходов, определяющая состояние автомата в момент времени $(t + 1)$ в зависимости от состояния автомата и входного сигнала в момент времени t . Иначе говоря

$$as = \delta(am, z)$$

где am – состояния автомата в момент времени t ; z – входной сигнал в момент времени t ; as – состояния автомата в момент времени $(t + 1)$; λ – функция выходов.

В теории различают два типа цифровых автоматов: автоматы Мура и автоматы Мили.

Если выходные значения зависят как от внутреннего состояния, так и от входных сигналов, то автомат называется автоматом Мили.

Для автомата Мили

$$wm = \lambda(aq, zn),$$

где aq – текущее состояние автомата; z – текущий входной сигнал; wm – значение выходного сигнала.

Если же выходные значения зависят только от внутреннего состояния, то такой автомат называют автоматом Мура.

Для автомата Мура

$$w_m = \lambda(a_q),$$

$a0$ – начальное состояние автомата в момент времени $t = 0$.

Функционирование конечного автомата происходит следующим образом. В дискретные моменты времени $t = 0, 1, 2, 3, \dots$ на вход автомата поступает входной сигнал – один из элементов входного алфавита Z , а на выходе появляется выходной сигнал – один из элементов выходного алфавита W . В этот же момент времени автомат из состояния as переходит в состояние am , в котором он будет находиться в момент времени $(t + 1)$.

Таким образом, конечный автомат работает по следующей схеме: если текущее состояние am и на входе сигнал zn , то следующее состояние $as = \delta(am, z)$ и на выход следует выдать $wm = \lambda(aq, zn)$.

Входные и выходные сигналы автомата следует понимать обобщенно. То есть у конечного автомата может быть не один физический входной сигнал и не один физический выходной. Под входным сигналом следует понимать всю совокупность физических входных сигналов. Аналогично под выходным сигналом следует понимать всю совокупность физических выходных сигналов [45, 46].

Цифровые автоматы в цифровой технике используются в большинстве устройств, в которых выполняется некоторая последовательность действий. В качестве примеров можно привести реализации различных интерфейсов и каналов обмена, устройство управления процессором, устройства управления различными системами и механизмами (например исполнительными устройствами «умного дома»). Описанные выше синхронные схемы (сдвиговые регистры, счетчики и т. д.), строго говоря, являются цифровыми автоматами.

Все операции, которые реализуются цифровыми автоматами, могут быть реализованы программно; более того, любая компьютерная программа может быть интерпретирована как описание цифрового автомата. Однако программная реализация во многих случаях является избыточной и медленной. Современные ПЛИС имеют достаточное количество ресурсов, и использование автоматных моделей позволяет получить быстродействующую и в то же время наглядную реализацию при приемлемых затратах.

4.7.1. Формы задания конечных автоматов

При проектировании цифровых автоматов используются различные формы их задания [45, 46].

Формальный способ – это непосредственное задание набора $(A, Z, W, \delta, \lambda, a1)$. Используется в основном в теории конечных автоматов.

Таблица переходов – табличное представление функции δ . Обычно в такой таблице каждому столбцу соответствует одно состояние, а строке – один допустимый входной символ. В ячейке, на пересечении строки и столбца, записываются состояние, в которое должен перейти автомат, если в данном состоянии он считал данный входной символ, и соответствующий выходной сигнал.

Диаграмма состояний (или иногда граф переходов) – графическое представление множества состояний и функции переходов. Представляет собой размеченный ориентированный граф, вершины которого – состояния конечного автомата, дуги – переходы из одного состояния в другое, а метки дуг – символы, по которым осуществляется переход из одного состояния в другое. Метки дуг должны содержать информацию об условии перехода вдоль данной дуги.

Заметим, что если раньше мы говорили о переходах между состояниями в терминах функции δ , которая по текущему состоянию am и текущему входному сигналу zn выдавала следующее состояние $as = \delta(am, zn)$, то теперь мы говорим о переходах между состояниями в терминах выполнения условий перехода:

$$(\text{текущее_состояние} = am) \wedge (\text{текущий_входной_сигнал} = zn) \Rightarrow \text{следующее_состояние} = as$$

В этом случае функция δ задается в виде условий переходов, перечислением всех возможных текущих состояний и входных сигналов и сопоставлением им соответствующих следующих состояний. То же касается и выходных сигналов и функции λ .

Граф-схема алгоритма (ГСА) содержит столько же информации, сколько и диаграмма состояний, но ГСА более детальна. ГСА проще преобразовать в HDL-код. Также ГСА можно расширить до граф-схемы алгоритма с трактом данных (algorithmic state machine with data path).

4.7.2. Кодирование состояний

В коде на языке описания аппаратуры состояниям даются символичные имена, используя перечислимый тип данных. В железе этим символическим именам будут соответствовать некоторые последовательности битов. Существует несколько схем кодирования состояний. Пусть n – число состояний конечного автомата.

Бинарная (или последовательная) схема – простая нумерация состояний последовательными двоичными числами. Использует минимальное количество битов, равное $\log 2n$.

Код Грея (Gray code assignment). Нумерация состояний последовательными элементами последовательности Грея. Также использует минимальное количество битов, равное $\log 2n$.

One-hot назначение. Каждому состоянию соответствует один бит. Поэтому только один бит кода имеет значение «1» (так называемый hot) в каждый момент времени. Для кода требуется n бит.

One-hot с нулевым начальным состоянием. То же, что и one-hot, но использует представление, в котором все биты нулевые. Обычно «нулевое» состояние используется как начальное или основное, так

как оно легко достигается, если активировать вход асинхронного сброса D-триггеров. Требуется для кодирования $(n - 1)$ бит.

Существуют и другие способы кодирования. Схема кодирования выбирается из соображений скорости, количества используемой памяти и дополнительного функционала, который может быть привязан к разрядам кода состояния. В настройках синтеза можно выбрать схему кодирования, в том числе задать ее автоматический выбор.

Выходы автомата Мура зависят только от текущего состояния автомата, сохраненного в регистре состояния. Комбинационная схема, реализующая выходы Мили, напротив, зависит не только от текущего состояния, но и от входных сигналов. Обобщенный конечный автомат можно изобразить в виде следующей структурной схемы (рис. 59).

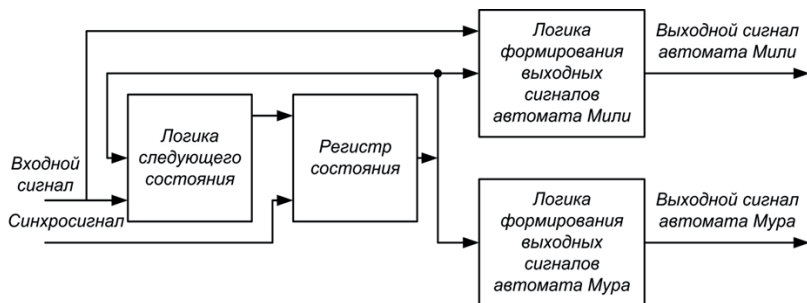


Рис. 59. Универсальная структурная схема автомата Мура/Мили

На каждый активный фронт сигнала синхронизации `clk` происходит запись следующего состояния в «Регистр состояния». Следующее состояние вычисляется комбинационной схемой «Логика следующего состояния» на основании текущего состояния и входных сигналов. Выходные сигналы вычисляются двумя комбинационными схемами: комбинационной схемой «Логика формирования выходных сигналов автомата Мура» на основании только текущего состояния `state_reg` и комбинационной схемой «Логика формирования выходных сигналов автомата Мили» на основании текущего состояния и входных сигналов [45–47].

Замечание: если в конечном автомате имеется комбинационная логика Мили, то такой конечный автомат имеет комбинацион-

ный путь (или комбинационный тракт, или комбинационный канал, combinational path).

Поскольку переход между состояниями осуществляется только по фронту сигнала синхронизации, то выходы автомата Мура являются в большей степени синхронными, чем выходы автомата Мили. Выходы автомата Мура изменяются практически сразу после переключения регистра состояния, а значит, практически синхронны с ним. Выходы автомата Мили же зависят и от входных сигналов, которые могут изменяться в произвольный момент времени. Следовательно, изменение выходов автомата Мили может происходить также в произвольный момент времени.

Комбинационная схема «Логика следующего состояния» реализует функцию δ , а комбинационная схема «Логика формирования выходных сигналов автомата Мура» вместе с комбинационной схемой «Логика формирования выходных сигналов автомата Мили» реализуют функцию λ .

Для обеспечения стабильной и безотказной работы используется сброс автомата в начальное состояние. Этим обеспечивается инициализация автомата в заранее predetermined состоянии при первом тактовом импульсе. В случае если сброс не предусмотрен, невозможно предсказать, с какого начального состояния автомат начнет функционирование, что может привести к сбоям в работе всей системы. Особенно эта ситуация актуальна при включении питания системы. Поэтому рекомендуется использовать схемы сброса и начальной установки при проектировании устройств на ПЛИС [44, 48].

Код, описывающий цифровой автомат, может быть структурирован в виде трех разделов, соответствующих логике следующего состояния, логике текущего состояния и выходной логике. В описании ПЛИС-проекта эти разделы могут быть охарактеризованы и структурированы разными способами.

Логика текущего состояния задается одним регистром с разрядностью достаточной, чтобы охватить все возможные состояния разрабатываемого цифрового автомата. Поскольку большинство цифровых автоматов требуют, чтобы синтезатор схем установил его в начале работы в известное начальное состояние, то для этой цели может быть использован или асинхронный, или синхронный сброс.

Логика следующего состояния лучше всего задается в Verilog с использованием конструкции множественного выбора case в блоке always.

Параметром-селектором конструкции case является регистр, хранящий код текущего состояния. Варианты выбора case должны описать все возможные переходы между состояниями. Ветка default конструкции является полезной, поскольку обычно число состояний автомата меньше, чем можно закодировать в слове состояния. Использование default значительно упрощает борьбу с «тупиковыми» состояниями, попав в которое однажды, выйти уже невозможно.

Способ генерации выходной логики зависит от того, какого типа автомат используется.

Для описания состояний и их кодирования рекомендуется использовать параметр, поскольку он позволяет вносить изменения в назначение состояния в единственном месте и дать состояниям автомата значимые имена.

Поскольку цифровые автоматы являются часто используемыми компонентами ПЛИС-проектов, для них разработаны соответствующие шаблоны и дополнительные средства анализа. Если автомат описан в соответствии с предлагаемым шаблоном, на основе его описания на Verilog средствами синтеза может быть синтезирована схема переходов.

Рассмотрим примеры синтеза описания цифрового автомата.

4.7.3. Описание конечного автомата Мура

Автомат Мура имеет выходы, которые являются функцией только текущего состояния. Далее приведен пример автомата Мура, созданный на основе шаблона, предлагаемого средой Quartus Prime.

Пример 46. Автомат Мура с четырьмя состояниями

```
// Quartus Prime Verilog Template
// 4-State Moore state machine
module FSM_Moore
(
    input clk, in, reset,
    output reg [1:0] out
);
// Объявление регистра состояния
reg [1:0] state;
// Описание кодирования состояний
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
```

```
// Описание сигнала на выходе, зависящее только от текущего
состояния
always @ (state) begin
    case (state)
        S0:
            out = 2'b01;
        S1:
            out = 2'b10;
        S2:
            out = 2'b11;
        S3:
            out = 2'b00;
        default:
            out = 2'b00;
    endcase
end

// Описание переходов в следующее состояние
always @ (posedge clk or posedge reset) begin
    if (reset)
        state <= S0;
    else
        case (state)
            S0:
                state <= S1;
            S1:
                if (in)
                    state <= S2;
                else
                    state <= S1;
            S2:
                if (in)
                    state <= S3;
                else
                    state <= S1;
            S3:
                if (in)
                    state <= S2;
                else
                    state <= S3;
        endcase
    end
endmodule
```

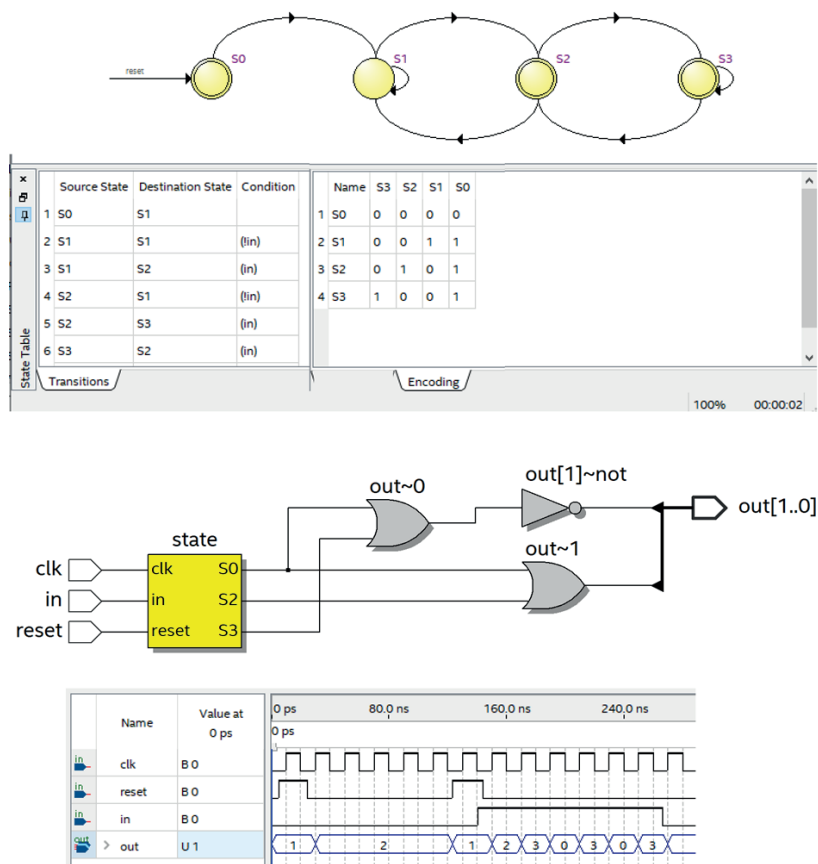


Рис. 60. Синтезированная диаграмма переходов, результат синтеза и временные диаграммы работы автомата Мура с четырьмя состояниями

Результат синтеза примера и временные диаграммы его работы приведены на рис. 60.

На временной диаграмме можно проследить изменения выходного сигнала автомата, но не его состояние. Видно, что изменения выходных сигналов происходят сразу после прихода нарастающего фронта сигнала асинхронного сброса или синхросигнала.

4.7.4. Описание конечного автомата Мили

Конечный автомат Мили имеет выходы, которые являются функцией и текущего состояния, и основных входов системы [46].

Пример 47. Автомат Мура с четырьмя состояниями (шаблон, предоставляемый Quartus Prime)

```
// Quartus Prime Verilog Template 4-State Mealy state
machine
module FSM_Mealy
(
    input clk, in, reset,
    output reg [1:0] out,
    output reg [1:0] state
);

// Описание кодирования состояний
parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;

// Описание переходов в следующее состояние
always @ (posedge clk or posedge reset) begin
    if (reset)
        state <= S0;
    else
        case (state)
            S0: if (in)
                state <= S1;
                else
                state <= S1;
            S1: if (in)
                state <= S2;
                else
                state <= S1;
            S2: if (in)
                state <= S3;
                else
                state <= S1;
            S3: if (in)
                state <= S2;
                else
                state <= S3;
```

```
        default:
            state <= S0;
        endcase
    end

// описание выходных сигналов в зависимости от
// текущего состояния и входных сигналов
    always @ (state or in)
    begin
        case (state)
            S0:    if (in)
                    out = 2'b00;
                else
                    out = 2'b10;
            S1:    if (in)
                    out = 2'b01;
                else
                    out = 2'b00;
            S2:    if (in)
                    out = 2'b10;
                else
                    out = 2'b01;
            S3:    if (in)
                    out = 2'b11;
                else
                    out = 2'b00;
            default:
                out = 2'b00;
        endcase
    end
endmodule
```

Результат синтеза примера и временные диаграммы его работы приведены на рис. 61.

На временных диаграммах видно, что выходные сигналы автомата Мили изменяются как при изменении состояний, так и при изменении входных сигналов. И если первое происходит по нарастающему фронту сигнала сброса или синхросигнала, то второе может произойти в произвольный момент времени. В результате, например на 150 нс модели-

рования, можно увидеть кратковременный скачок выходного сигнала, который может стать причиной сбоя в логике работы схем, зависящих от данного автомата.

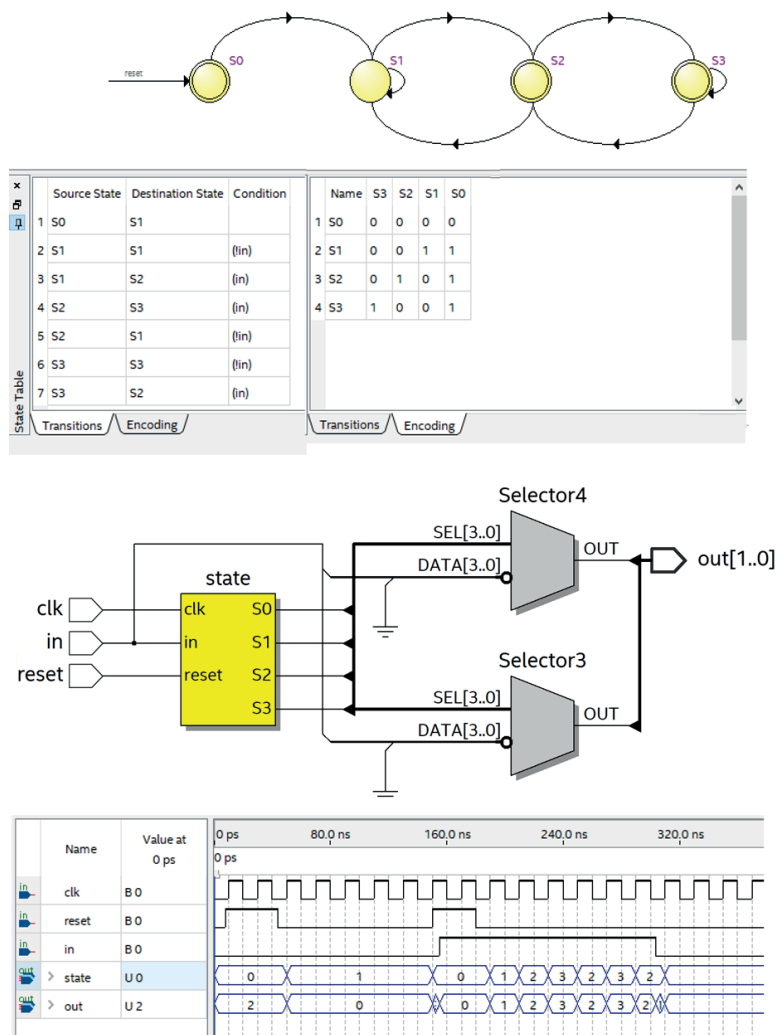


Рис. 61. Синтезированная диаграмма переходов, результат синтеза и временные диаграммы работы автомата Мили с четырьмя состояниями

4.7.5. Сравнительный анализ выходных сигналов автоматов Мура и Мили

Конечному автомату Мили обычно требуется меньше состояний для выполнения той же задачи, чем конечному автомату Мура. Это происходит из-за того, что выходные сигналы являются функциями не только текущего состояния, но и входных сигналов; поэтому можно задать несколько возможных значений выходных сигналов в одном состоянии.

Конечный автомат Мили имеет меньшее время отклика. Поскольку выходные сигналы Мили являются функциями входных сигналов конечного автомата, они могут изменить свои значения сразу же, как только определенное условие, содержащее входные сигналы, будет выполнено. В синхронной схеме различие между конечным автоматом Мура и конечным автоматом Мили обычно означает задержку первого относительно второго на один такт синхронизирующего сигнала.

В конечном автомате Мили длительность и временная зависимость выходного сигнала определяются входным сигналом, т. е. выходной сигнал Мили не имеет фиксированной длительности и может быть очень коротким. Если входной сигнал содержит «иголки» (кратковременные всплески длительностью меньше периода синхронизирующего сигнала), а такие сигналы часто встречаются на выходе комбинационных схем, то эти «иголки» могут быть выведены на выход конечного автомата. Именно такая ситуация и изображена на рис. 61. Выходной сигнал Мура, напротив, изменяет свое значение синхронно с синхронизирующим сигналом, и длительность импульсов в выходном сигнале Мура приблизительно совпадает с периодом синхронизирующего сигнала.

В зависимости от ситуации правильно выбирайте тип используемого конечного автомата.

4.8. Пример разработки цифрового автомата: устройство управления процессором

В п. 4.6 приведен пример разработки процессора на основе распределителя синхроимпульсов. Но, как правило, устройство управления, входящее в состав процессора, реализуется в виде цифрового авто-

мата. Это позволяет описать работу более логично и последовательно, реализовать более сложные функции и обеспечивает более простую модифицируемость кода на языке описания аппаратуры.

Задание на проектирование полностью совпадает с заданием из п. 4.6. В ходе разработки процессора был сделан вывод о том, что работа процессора реализуется в виде повторяющихся циклов выполнения команд. Каждый цикл состоит из трех тактов:

- на первом такте с шины команд команда считывается и заносится в регистр команд, код операции и адреса операндов с регистра команд подаются на АЛУ и регистровый файл;
- на втором такте выполняется чтение операндов из ячеек регистрового файла, операнды подаются на АЛУ;
- на третьем такте результат работы АЛУ заносится в регистр-приемник и в выходной регистр процессора.

Соответственно можно выделить три состояния, в которых процессор пребывает во время работы, и, согласно требованиям к цифровым автоматам, необходимо добавить исходное состояние, в котором автомат находится после включения или сброса.

Процессор на основе цифрового автомата использует модули АЛУ и регистрового файла, описанные выше. Поскольку процессор является синхронной схемой и его выходные сигналы должны изменяться в соответствии с приходом фронта синхросигнала, для его описания выбираем автомат Мура.

Пример 48. Процессор на основе автомата Мура

```
module Processor_FSM
# (parameter
// в виде параметров задаем
N = 4,           // разрядность обрабатываемых данных,
INSTR = 4,       // длину кода операции,
ADR = 3,         // разрядность шины адреса
L = INSTR+ADR*2) // вычисляем длину командного слова
(
input clk, reset,
input [(INSTR+ADR*2-1):0] instruction, // шина команд
output reg [N-1:0] result, // выходная шина данных
output reg carry // выход переноса
);
```

```
// Описание кодирования состояний
parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3=3'b11;

wire [N-1:0] w_a;
wire [N-1:0] w_b;
wire [N:0] w_c;
// шины, используемые для подключения АЛУ

reg [INSTR-1:0] r_cop;
reg [(ADR-1) : 0] r_adr_op_s;
reg [(ADR-1) : 0] r_adr_op_r;
// регистры полей команды

reg [1:0] state;
//регистр состояний
reg [1:0] order;
//выходной сигнал

ALU
//экземпляр (инстанс) АЛУ
#(.N(N))
alu_1
(
    .c (w_c),
    .instr (r_cop),
    .a (w_a),
    .b (w_b)
);

regfile_dual
// экземпляр (инстанс) регистрового файла
#(.DATA_WIDTH(N), .ADDR_WIDTH(ADR))
regfile_1
(
    .data(w_c),
    .read_addr(r_adr_op_s),
    .write_addr(r_adr_op_r),
    .we (w_we),
    .read_clock (order [0]),
    .write_clock (order [1]),
    .a(w_a),
    .b(w_b)
);
```

```

always @ (posedge clk or posedge reset) begin
// процедурный блок always
// описывающий работу устройства управления
    if (reset)

// обработка асинхронного сброса
        begin
            state <= S0;
            r_cop <= 0;
            r_adr_op_s <= 0;
            r_adr_op_r <= 0;
            result <= 4'b0000;
            carry <= 1'b0;
            order <= 2'b00;
        end
    else

// Описание переходов в следующее состояние
        case (state)
            S0:
                begin
                    r_cop <= instruction [(L-1):(L-INSTN)];
                    r_adr_op_s <= instruction [(ADR*2-1) : ADR];
                    r_adr_op_r <= instruction [(ADR-1) : 0];
                    order <= 2'b00;
                    state <= S1;
                end
            S1:
                begin
                    order <= 2'b01;
                    state <= S2;
                end
            S2:
                begin
                    order <= 2'b10;
                    {carry, result} <= w_c;
                    state <= S0;
                end
            default:
                state <= S0;
        endcase
    end
endmodule

```

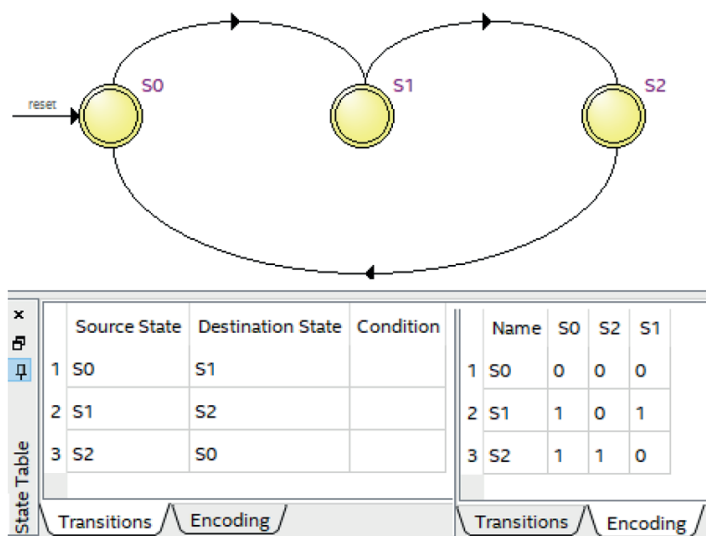


Рис. 62. Диаграмма переходов для устройства управления процессором

Временные диаграммы работы процессора совпадают с приведенными в п. 4.7. Поскольку описание автомата было сделано в соответствии с шаблоном, среда разработки Quartus сформировала для него диаграмму переходов (рис. 62).

Примеры реализации процессора, приведенные в этом разделе, являются учебными. Число состояний и диаграмма переходов реальных процессоров значительно сложнее, поскольку помимо работы с регистровым файлом процессор получает данные из ОЗУ (при этом используются разные способы адресации), взаимодействует с памятью команд, выполняет условные переходы и прерывания и т. д.

Вопросы и задания

1. Дайте определение понятию синхронной (тактируемой) схемы.
2. В чем отличие между синхронными и асинхронными цифровыми схемами?

3. В чем назначение поведенческого блока `always` при описании синхронных схем?
4. Для чего служит список чувствительности поведенческого блока `always`? Какие виды событий могут в него входить?
5. Дайте определение понятиям синхросигнала, уровня синхросигнала, фронта синхросигнала.
6. В чем разница между синхронными и асинхронными управляющими сигналами?
7. В чем преимущества, недостатки синхронных управляющих сигналов и какие проблемы они порождают?
8. В чем преимущества, недостатки асинхронных управляющих сигналов и какие проблемы они порождают?
9. Как описывается, функционирует, в какую схему синтезируется и как применяется блокирующее присваивание?
10. Как описывается, функционирует, в какую схему синтезируется и как применяется неблокирующее присваивание?
11. Какие варианты организации памяти могут использоваться для ПЛИС-проектов?
12. Какие входные, выходные сигналы и параметры используются при описании модулей памяти?
13. Приведите пример регистра с расширенной функциональностью. Разработайте описание модуля такого регистра на Verilog.
14. Каким образом обеспечиваются необходимые временные задержки в ПЛИС-проектах?
15. Каково назначение таймер-счетчиков в ПЛИС-проектах? Как определяется коэффициент пересчета?
16. Перечислите основные типы цифровых автоматов, охарактеризуйте особенности их функционирования.
17. Перечислите способы описания цифровых автоматов.
18. Перечислите способы кодирования состояний, используемые в цифровых автоматах, охарактеризуйте их преимущества и недостатки.
19. Приведите пример описания цифрового автомата Мили на Verilog.
20. Приведите пример описания цифрового автомата Мура на Verilog.

5. БИБЛИОТЕКИ МОДУЛЕЙ. IP-ЯДРА

Сегодня проектирование на ПЛИС основывается на многократном использовании ранее наработанных и верифицированных готовых модулей – блоков интеллектуальной собственности IP. С использованием IP как конструктивных элементов собирается проект на ПЛИС или СнК. Многократное использование готовых модулей не только способствует повышению эффективности работы отдельных проектировщиков, но и обеспечивает возможность распределенного проектирования систем. Сегодня проектировщики из разных областей знаний, территориально удаленные друг от друга, могут объединять усилия для создания конечного продукта.

5.1. Применение IP-ядер в ПЛИС-проектах

В настоящее время рынок ПЛИС – один из наиболее динамично развивающихся. ПЛИС применяются во многих отраслях техники. На данный момент нет однозначной, удовлетворяющей всех разработчиков аппаратуры методологии получения конфигурации ПЛИС по функциональной модели устройства на системном уровне. Наиболее популярный подход к решению этой проблемы – применение технологии IP-ядер (Intellectual Property Cores) [14, 22, 49, 50].

5.1.1. Виды IP-ядер

IP-ядро (Intellectual Property core) – логический блок, или блок данных, используемый в ПЛИС- и ASIC-микросхемах, который реализует некоторую законченную функциональность. IP-ядро представляет собой программную или аппаратную реализацию некоторого компонента системы, например центрального процессорного устройства, Ethernet-контроллера или PCI-интерфейса.

Наличие как аппаратных (hard), так и синтезируемых (soft) блоков от различных поставщиков позволяет разработчикам использовать

эти блоки в своих проектах, таким образом добавляя в проект функции без затрат на повторную разработку. Помимо снижения временных, интеллектуальных и материальных затрат многократное использование IP-ядер значительно уменьшает риск возникновения ошибок при проектировании. Сегодня проектирование на основе IP-ядер подразумевает сборку цифровой части из готовых блоков, разработку и доработку собственных IP-ядер и включение в проект отдельных аналоговых модулей в виде «черных ящиков».

IP-ядро (IP-блок) – это проектный блок, который, возможно, пригоден к повторному использованию.

Виртуальный компонент – это проектный блок, специально созданный с целью повторного использования, который был разработан и сертифицирован для максимального повторного использования.

Виртуальный сокет – описание всех проектных интерфейсов, которым должен соответствовать IP-блок, а также модели проектирования и информация по интеграции, которые должны поставляться вместе с IP-блоком, необходимые для облегчения интеграции блока в проект.

Ожидается, что в ближайшее время в состав IP-ядер, применяемых при проектировании, войдут радиочастотные компоненты, модули микроэлектромеханических систем (Micro-Electro Mechanical Systems, MEMS) и т. д. Разработка модулей каждого конкретного типа имеет свои особенности и требует высокой квалификации разработчиков в конкретной области проектирования. Объединение в одной команде разработчиков большого числа узких специалистов технически затруднительно и экономически нецелесообразно. С распространением технологии повторного использования IP-ядер процесс разработки в значительной степени будет заменен процессом интеграции, т. е. объединения в одном проекте IP-блоков как собственной, так и сторонней разработки. Создание библиотек многократно используемых IP-блоков в большой степени зависит от развития стандартизованных процессов их проектирования и интеграции. Библиотеки, соответствующие этим стандартам, предлагающие наборы разнообразных IP-блоков, собственно и предоставляют возможность многократного их использования [49, 51].

5.1.2. Требования к IP-ядрам

Основные требования к виртуальным компонентам:

- процессы разработки и верификации виртуальных компонентов должны соответствовать установившимся процессам проектирования и стандартам качества;
- вместе с виртуальными компонентами должны поставляться модели и данные проектирования, ассоциированные документы проектирования, сценарии, информация по параметрам и другие дополнительные данные, соответствующие одному из признанных стандартов повторного использования IP, например стандартам VSIA либо какому-то другому внутреннему или внешнему набору стандартов;
- виртуальные компоненты должны хотя бы один раз быть изготовлены с последующим определением характеристик;
- виртуальные компоненты должны хотя бы один раз быть использованы какой-либо внешней командой разработчиков с полученными результатами данного использования;
- виртуальные компоненты должны сопровождаться оценкой качества на основе промышленного стандарта по качеству, например такого как OpenMORE (разработанного компаниями Synopsys и Mentor Graphics) или стандарта качества VSI Quality (который воспринимает OpenMORE в качестве одного из входных параметров).

В течение последних десяти лет разработки в области повторного использования IP фокусировались на задании стандартов и процессов, позволяющих превратить хаотичное повторное использование IP-блоков в хорошо понятный и надежный процесс приобретения и повторного использования виртуальных компонентов.

Требования к IP-библиотекам к данному моменту в общем сформулированы. Важным требованием является обязательное сопровождение полным набором IP-моделей, документации и других вещей, требуемых соответствующими стандартами. Они должны быть иерархически организованы и классифицированы для поддержки быстрого поиска. Наконец, метаданные о IP-блоках («данные о данных» – документационная информация, поддерживающая быструю идентификацию, поиск и оценку) должны быть организованы в базы данных с возможностью быстрого поиска [49–51].

Однако при платформенном проектировании, в противоположность блочному подходу, также важно, чтобы IP-библиотеки поддерживали хранение, поиск и извлечение данных на базе всей платформы, а не только на базе отдельных блоков. При этом общие функциональные блоки, такие как процессоры и стандартные интерфейсы, могут входить в несколько платформ. Поэтому важно иметь базу данных, позволяющую сохранить информацию о блоке один раз, и иметь к ней доступ несколькими разными путями – как к отдельному блоку в блочном стиле проектирования СнК и как к части платформы.

5.2. Soft-процессоры

При разработке системы на кристалле, реализующей в одном корпусе ПЛИС функции процессора и периферийных устройств, целесообразно использовать готовые микропроцессорные ядра. Их применение позволяет ощутимо сократить длительность цикла проектирования разрабатываемой системы.

Программный процессор (soft-процессор) – это IP-ядро, представляющее собой синтезируемое описание модуля микропроцессора на языке описания аппаратуры, которое реализуется с использованием только логического синтеза.

Архитектура soft-процессора может быть как универсальной, так и оптимизированной для решения конкретных прикладных задач, например цифровой обработки сигналов или реализации алгоритмов искусственного интеллекта. При этом параллелизм soft-процессора, т. е. максимальное число его ядер, фактически ограничен лишь ресурсами конкретной ПЛИС. Возможность проектировать и быстро реализовывать специализированный процессор может быть полезной в специфических проектах.

Большинство фирм, разрабатывающих и производящих ПЛИС и системы автоматизированного проектирования для них, предлагают свои soft-процессоры. В табл. 11 приведены основные сведения о наиболее распространенных на сегодняшний день soft-процессорах.

Soft-процессор Nios

В рамках курса «ПЛИС» наибольший интерес представляет семейство программных процессоров Nios – встраиваемые процессоры

общего назначения с перестраиваемой конфигурацией. Эти процессоры используются в ПЛИС Intel-FPGA (Altera) семейств Cyclone, Arria, Stratix и HardCopy.

Таблица 11

Soft-процессоры ведущих мировых производителей

Название	Производитель	Открытый исходный код	Примечание
TSK3000A	Altium	Royalty Free	32-битный RISC-процессор модифицированной гарвардской архитектуры в стиле R3000
TSK51/52	Altium	Royalty Free	8-битный, совместимый с системой команд Intel 8051, lower clock cycle alternative
OpenSPARC T1	Sun	Да	64-битный
MicroBlaze	Xilinx	Нет	32-битный RISC-процессор гарвардской архитектуры
PicoBlaze	Xilinx	Да	8-битный
Nios, Nios II	Altera	Нет	–
Cortex-M1	ARM	Нет	–
Mico32	Lattice	Да	–
LEON 3/4	ESA	Да	Совместим со SPARC V8
OpenRISC	OpenCores	Да	32-битный; Done in ASIC, Altera, Xilinx
AEMB	Shawn Tan	Да	MicroBlaze EDK 3.2 compatible Verilog core
OpenFire	Virginia Tech CCM Lab	Да	Двоично совместимый с MicroBlaze
PacoBlaze	Pablo Bleyer	Да	Совместимый с PicoBlaze
xr16	Jan Gray	Нет	16-битный RISC-процессор + ШК описаны в журнале Circuit Cellar Magazine #116-118
Zet	Zeus Gómez Marmolejo	Да	Клон x86 ПК, совместимый с 8086

Процессор построен по архитектуре RISC (англ. Reduced Instruction Set Computer, набор коротких команд), в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения – коротким. Команды

выполняются за один цикл частоты синхронизации. Процессор имеет 32-разрядные шины данных и адреса, 32 регистра общего назначения и 32 источника внешних прерываний. Арифметические и логические операции выполняются над операндами, находящимися в регистрах общего назначения. Обмен информацией между регистрами и памятью осуществляется путем выполнения команд Load и Store. Машинное слово процессора Nios II имеет ширину 32 бита, такой же размер имеют его регистры. Для адресации байтов в слове используется little-endian – стиль, при котором менее значимые байты расположены по меньшим адресам в ОП. Процессор имеет гарвардскую архитектуру, т. е. использует отдельные шины для данных и команд [52, 53]. Кроме того, разработчику предоставляется возможность определить до 256 собственных инструкций процессора, что является принципиальным отличием soft-процессоров от обычных. В библиотеке заранее присутствует несколько инструкций (например, математика с плавающей точкой), которые можно подключить при настройке ядра. В дополнение к процессорному ядру разработан набор вспомогательных IP-ядер. В число таких блоков входят таймеры, которые можно настроить практически на любую частоту, контроллеры памяти, интерфейсы и т. д.

На рис. 63 приведен пример архитектурной организации кристалла с использованием Nios II.

Процессор Nios II использует для адресации 32-битный адрес; при этом память является адресуемой по байтам. С помощью команд можно записывать/считывать слова (32), полуслова (16) и байты данных (8). Чтение или запись данных по адресам, которые не связаны с памятью или периферийными устройствами, приводит к неопределенным результатам. В процессоре Nios II используются следующие способы адресации.

Непосредственная адресация: 16-битный операнд присутствует в команде. Он может быть дополнен знаковым разрядом до 32 разрядов при выполнении арифметической операции.

Регистровая адресация: операнды находятся в регистрах процессора.

Относительная регистровая адресация: адрес операнда получается путем суммирования содержимого регистра и знакового 16-разрядного смещения, находящегося в самой команде.

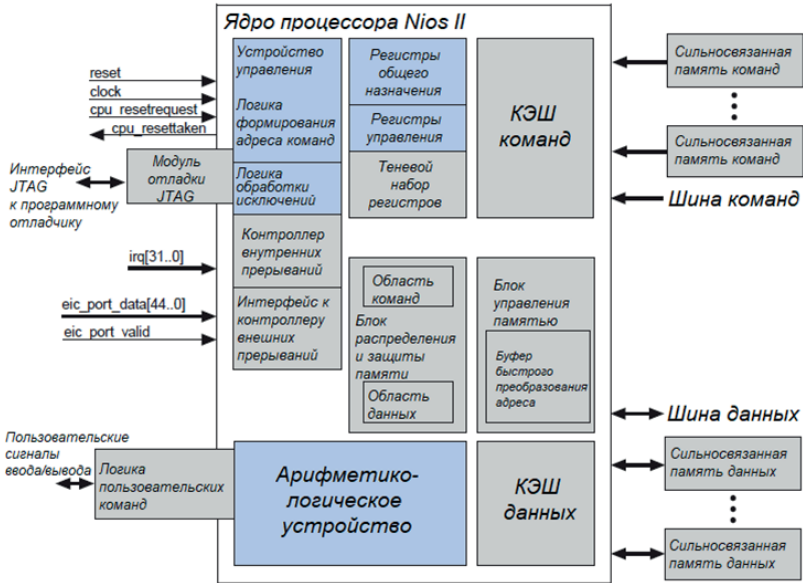


Рис. 63. Структурная схема процессора Nios II [53]

Косвенная регистровая адресация: содержимое регистра является адресом операнда. Этот способ эквивалентен предыдущему способу, когда смещение равно нулю.

Абсолютная адресация: 16-битный абсолютный адрес операнда может быть определен путем использования смещения относительно регистра r0, который всегда равен нулю [52, 53].

Процессорная система Nios II может содержать кэш-память команд и данных, которые реализуются с помощью блоков встроенной памяти в FPGA. Параметры кэш-памяти определяются во время создания системы. Добавление этой памяти значительно повышает производительность процессорной системы, особенно если в качестве основной памяти используется внешняя SDRAM-память.

Кэш-память команд и данных является памятью прямого отображения. Кэш-память команд может быть использована в стандартной и быстрой версиях процессора. При этом она организуется по 8 слов

в одной строке, а ее размер определяется пользователем во время создания системы.

Кэш-память данных может быть реализована только в быстрой версии процессора Nios II/f. Она может быть организована по 4, 16 или 32 байта в одной строке. Общий размер также определяется пользователем при генерации системы.

Сильносвязанная память (Tightly Coupled Memory) представляет собой память, реализованную внутри кристалла FPGA и подключенную к процессору напрямую без использования шины Avalon. В этой памяти могут храниться как инструкции, так и данные. Доступ к этой памяти осуществляется всегда в обход кэш-памяти. Настройка параметров этой памяти, также как и кэш-памяти, осуществляются во время проектирования процессорной системы [52].

Библиотечные элементы для периферии процессора: универсальный асинхронный приёмопередатчик (UART), таймер, параллельный ввод-вывод, интерфейс SRAM, SDRAM-контроллер, интерфейс FLASH-памяти, последовательный периферийный интерфейс (SPI), контроллер I2C, модулятор ширины импульса (PWM), IDE-контроллер диска, контроллер локальной сети 10/100 Ethernet (MAC), контроллер USB. Этот список не полный, но он даёт представление об уровне разработок библиотечных элементов. Подключив к процессору требуемые библиотечные элементы, можно программно сформировать необходимый для конкретного применения микроконтроллер [52, 53].

Вопросы и задания

1. Дайте определение понятия IP-ядро.
2. Для каких целей применяются IP-ядра при проектировании ИС и создании ПЛИС-проектов?
3. Каковы основные требования к IP-ядрам?
4. Дайте определение термина soft-процессор.
5. В каких случаях в ПЛИС-проектах используют soft-процессоры?
6. Опишите архитектуру и основные особенности организации soft-процессора Nios II.

6. ПРАКТИКУМ ПО КУРСУ «ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ»

В данном учебном пособии рассматривается раздел дисциплины «Программируемые логические интегральные схемы», посвященный разработке и синтезу ПЛИС-проектов и программированию ПЛИС [54].

6.1. Аппаратное обеспечение для выполнения работ

В ходе выполнения лабораторных работ курса ПЛИС студенты осваивают создание ПЛИС-проектов на базе ПЛИС производства Intel FPGA (Altera). Двойное название используемых микросхем объясняется тем, что компания Intel в декабре 2015 года сообщила о завершении процесса приобретения компании Altera, специализирующейся на выпуске программируемых вентильных матриц (FPGA). Сумма сделки составила 16,7 млрд долларов. Это самая крупная покупка Intel за всю историю существования компании. Сейчас под общим названием Intel FPGA корпорация Intel выпускает широкий спектр аппаратных решений.

Отладочные платы для СБИС Altera выпускаются как фирмой Intel, так и ее многочисленными партнерами. Особое место среди них занимает фирма Terasic (Тайвань) – платиновый DSN-партнер (Design Solutions Network) Intel FPGA. Она производит отладочные платы для всех современных семейств СБИС Intel FPGA. Все платы Terasic сопровождаются подробной документацией. С отладочными платами поставляются дистрибутивы, множество примеров проектирования, демонстрационных примеров и других материалов, упрощающих процесс освоения платы и проектирования на ней [55].

6.2. Программное обеспечение для выполнения работ

Интегрированное средство создания ПЛИС-проектов – пакет Quartus, ранее принадлежавшее компании Altera, также перешло в соб-

ственность Intel и теперь называется Intel Quartus. Информацию о изделиях линейки Intel FPGA и пакете проектирования можно найти на официальном сайте корпорации Intel по ссылке: <https://www.intel.com/content/www/us/en/products/programmable.html>.

САПР Quartus – среда для проектирования и отладки проектов на ПЛИС Altera. До версии 15.0 среда имела наименование Quatus II. Начиная с версии 15.1 она называется Quartus Prime. Последняя вышедшая на данный момент версия – Quartus Prime v18.1.

Среда Quartus относится к САПР ПЛИС (EDA Tool), т. е. поддерживает весь цикл проектирования цифровых устройств на основе программируемой и реконфигурируемой логики, например ПЛИС семейства MAX3000 и ППВМ семейств Flex10K, APEX20K, ACEX1K, Cyclon. Quartus содержит в своем составе средства для разработки устройств при помощи графического схемного редактора или на языках описания аппаратуры Verilog и VHDL, компилятор и редактор для размещения разработанной схемы на логику целевого устройства, средства анализа временных характеристик устройства, таких как время задержки сигнала между входом и выходом и максимальная тактовая частота работы устройства, редактор временных диаграмм для тестирования и отладки разрабатываемого устройства, программатор для переноса конфигурации устройства из проекта в интегральные схемы (ИС).

Аналогами и конкурентами Quartus II являются MAX+PLUS II той же фирмы Altera, предлагавшаяся ранее для создания подобных проектов, и Xilinx ISE фирмы Xilinx для поддержки ПЛИС и ППВМ собственного производства.

На сайте доступна для скачивания версия пакета Quartus Prime Light Edition со свободной лицензией, которую можно использовать в учебных целях, а также для написания небольших проектов. Quartus Prime Light Edition накладывает ограничения на поддерживаемые семейства устройств, некоторые функции и поддерживаемые операционные системы. Облегченная версия пакета имеет ограниченные возможности анализа и оптимизации проекта, но является вполне достаточной для освоения основных приемов разработки ПЛИС-проектов и может использоваться для знакомства с программой, а также в академических целях и для научных исследований.

6.3. Общие требования к выполнению практических работ

Курс практических работ состоит из двух видов заданий: тематических практических работ и заключительного проекта.

По окончании выполнения практической работы обучающиеся готовят отчет. Работу засчитывают по результатам защиты отчета.

Отчет по практической работе должен содержать:

- общее и индивидуальное задания;
- разработанную схему в формате схемного ввода или описания на языке Verilog с комментариями;
- результаты логического и (если это предусмотрено заданием) временного моделирования схемы в виде скриншотов временных диаграмм;

- вывод по результатам выполнения работы.

Защита практической работы состоит из следующих этапов:

- демонстрация работоспособности, корректности и эффективности разработанной схемы;
- ответы на вопросы по структуре схемы и ее описанию на языке описания аппаратуры;
- обсуждение результатов тестирования схемы и выводов по практической работе;
- ответы на теоретические вопросы по теме.

6.4. Тематические практические работы

Цель тематических работ: изучение особенности организации ПЛИС, средств и инструментов разработки ПЛИС-проектов, подходов к описанию модулей различных типов.

Практическая работа 1. Интегрированная среда разработки Quartus: основные возможности и инструменты

Порядок выполнения практической работы

1. Ознакомиться с описанием практической работы.
2. Получить вариант задания у преподавателя.

Часть 1 (на занятии):

3. Разработать схему проектируемого асинхронного устройства с использованием стандартных логических компонентов.
4. Создать новый проект в среде Quartus.
5. Выполнить ввод разработанной схемы, используя инструменты схемного ввода среды Quartus.
6. Выполнить синтез проекта, проанализировать сообщения различных этапов синтеза, исправить ошибки.
7. Создать символ для разработанного модуля.

Часть 2 (самостоятельная работа):

8. Создать новый проект в среде Quartus.
9. Выполнить установку нескольких модулей, разработанных в части 1, и их соединение согласно заданию, добавить элементы синхронной логики.
10. Выполнить синтез проекта, проанализировать сообщения различных этапов синтеза, исправить ошибки.

Описание работы

1. Создание нового проекта

Для создания нового проекта можно использовать мастер создания нового проекта New Project Wizard, который открывается при запуске Quartus Prime или в процессе работы. Для этого в среде САПР Intel Quartus Prime выбираем пункт меню:

File -> New Project Wizard.

Создание нового проекта с помощью мастера включает пять шагов описания свойств проекта, в том числе подключение имеющихся файлов проектов, IP-ядер или библиотеки, задание типа целевой ПЛИС и указание списка дополнительных инструментов проектирования (Electronic Design Automation, EDA), которые будут использоваться в разработке и отладке проекта.

1. Задание имени проекта, папки, куда он будет помещен, и имени модуля верхнего уровня проекта.

Не задавайте имена файлов, начинающихся с цифры, содержащих кириллицу, т. е. русскими буквами, не рекомендуется использовать в именах файлов или папок пробел.

Если папки с указанным именем не существует, то Quartus предлагает ее создать.

2. Задание типа проекта (Project type).

В практических работах обычно создается новый пустой проект – Empty project.

Кроме этой возможности среда предлагает разработчику несколько шаблонов проекта, с которыми вы можете ознакомиться самостоятельно.

3. Добавление существующих файлов в проект.

Если вы предполагаете использовать модули из предыдущих проектов, их можно добавить или перейти к следующему шагу.

4. Выбор типа целевой ПЛИС.

При выполнении полного цикла проектирования результатом будет файл программирования связей ПЛИС (файл прошивки), который формируется для конкретной ПЛИС.

Если разработчик не выбирает целевую ПЛИС, она выбирается по умолчанию.

Целевую ПЛИС можно задать позже, через пункт меню

Assignments -> Device

5. Выбор дополнительных инструментов для проекта.

Здесь предлагается добавить дополнительные инструменты проектирования (EDA tools), используемые совместно с Quartus Prime. Для выполнения практических работ данного раздела необходимости в подключении EDA нет.

6. Завершение.

Это итоговое диалоговое окно мастера нового проекта Quartus Prime.

В нем отображается описание свойств проекта, определенных при его создании.

Отсюда можно вернуться к предыдущим этапам описания свойств проекта для их коррекции либо нажать кнопку Finish, после чего Quartus Prime создает и открывает пустой проект.

2. Создание файла схемного ввода

Создание новых файлов выполняется из раздела File главного меню программы:

File -> New

В окне выбора отмечаем тип создаваемого файла. Поскольку первые практические работы предназначены для освоения инструментов схемного ввода, то выбираем файл схемного ввода:

File -> New -> Block Diagram / Schematic File

Обычно по умолчанию имя нового файла имеет стандартный вид, отличный от указанного нами при создании проекта. Его нужно сохранить под требуемым именем.

File -> Save As.

Если же задать другое имя файла, то лучше заново указать проекту, какой именно файл будет использоваться в проекте как содержащий модуль верхнего уровня. Это можно сделать через меню:

Project -> Set as Top-Level Entity.

3. Редактирование файла схемного ввода

Для редактирования файла схемного ввода необходимо открыть библиотеку элементов, выбрав инструмент, как показано на рис. 64.

Далее в списке элементов выбирается нужный и размещается на поле схемного ввода.

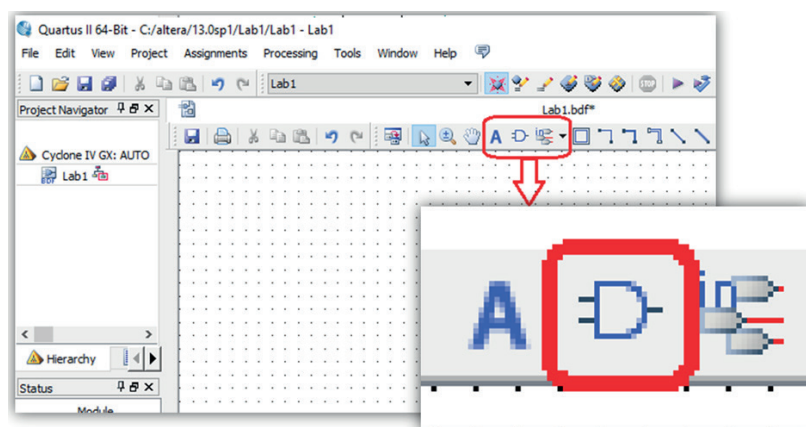


Рис. 64. Открытие библиотеки элементов

Соединения между элементами выполняются с помощью Orthogonal node tool или Diagonal node tool, которые находятся на панели инструментов (рис. 65).

Входные и выходные сигналы разрабатываемого модуля необходимо определить и обозначить направление передачи сигнала, выбрав соответствующий инструмент на панели (рис. 66).

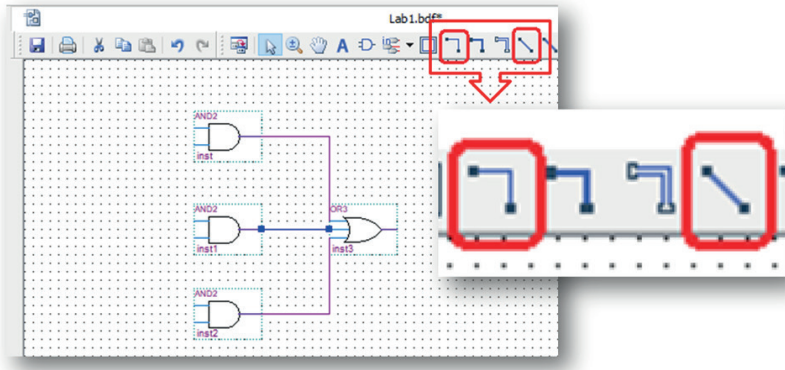


Рис. 65. Инструменты для выполнения межсоединений

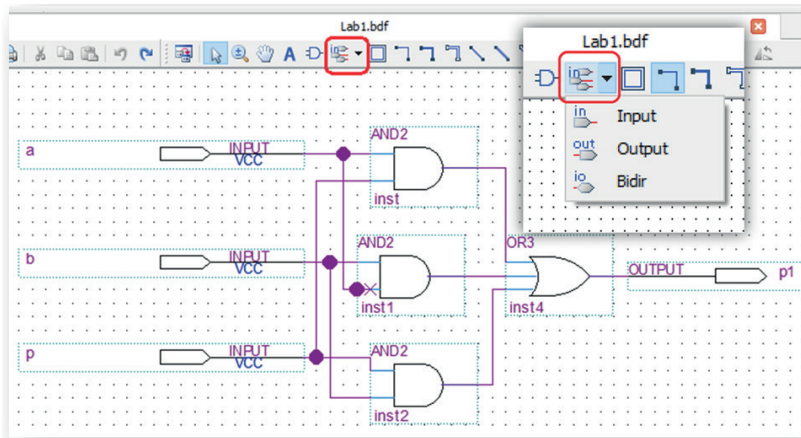


Рис. 66. Инструменты задания направления передачи сигнала

4. Компиляция проекта

Компиляция проекта – анализ, синтез, размещение и подготовка файла прошивки ПЛИС. Этот процесс требует много времени, поэтому его лучше запускать после того, как проект промоделирован и отлажен на логическом уровне.

Анализ и синтез – проверка синтаксиса и подготовка для логического моделирования. На данном этапе работы этого достаточно.

Выбор необходимых этапов компиляции проекта может быть выполнен на панели инструментов (рис. 67) или из главного меню программы:

Processing -> Start [выбрать действие из выпадающего меню]

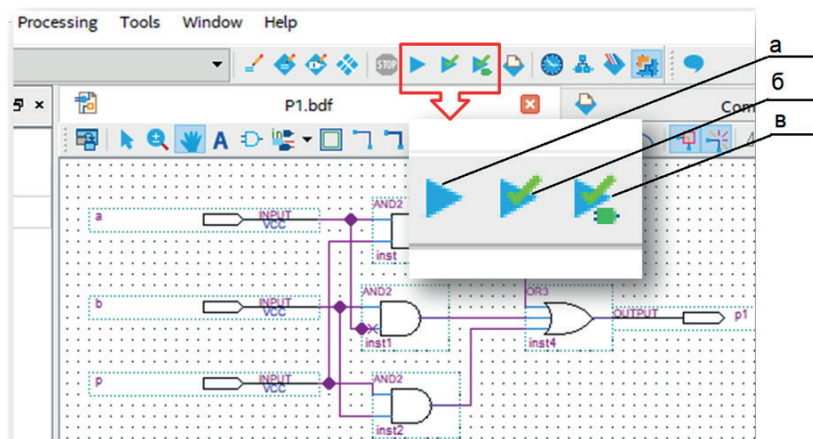


Рис. 67. Выбор необходимых этапов компиляции проекта может быть выполнен на панели инструментов: а – полная компиляция; б – анализ и предобработка; в – анализ и синтез

Если необходима полная компиляция проекта, то в меню выбирается следующее действие:

Processing -> Start Compilation

5. Создание графического символа для разработанного модуля

Если в дальнейшем планируется использовать разработанный модуль в составе других проектов, описываемых графически, для него

рекомендуется создать графический символ, посредством которого данный модуль будет отображаться на схемах.

Графический символ создается из меню:

File -> Create / Update -> Create Symbol Files for Current File

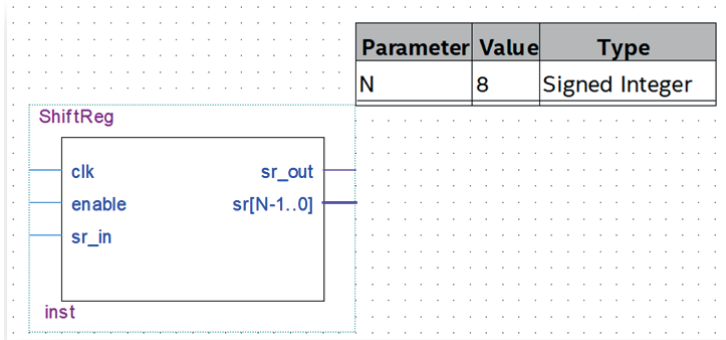


Рис. 68. Пример графического символа

Пример графического символа приведен на рис. 68.

Практическая работа 2. Цикл проектирования и программирования ПЛИС в среде Quartus

Порядок выполнения

Выполняется на занятии.

1. Ознакомиться с описанием практической работы. Вариант задания совпадает с вариантом практической работы 1.
2. Открыть проект модуля, созданного в ходе практической работы 1.
3. Выбрать целевую ПЛИС в соответствии с типом используемой отладочной платы.
4. Выполнить привязку модуля к входным и выходным контактам (пинам) ПЛИС.
5. Выполнить полный синтез проекта, проанализировать сообщения различных этапов синтеза, исправить ошибки.

6. Подключить к компьютеру отладочную плату и выполнить программирование (прошивку) ПЛИС.

7. Протестировать разработанный модуль средствами отладочной платы. При необходимости выполнить отладку проекта.

Описание работы

1. Открытие существующего проекта

Открыть существующий проект можно при запуске Quartus либо из главного меню:

File -> Open Project

2. Выбор типа целевой ПЛИС

Результатом практической работы будет файл программирования связей ПЛИС (файл прошивки), который формируется для определенного типа ПЛИС.

Подготовленный проект будет скомпилирован и загружен в ПЛИС на отладочной плате, которую вы будете использовать. Информацию о типе микросхемы можно найти в документации к плате. Выбор целевой ПЛИС выполняется на 4 шаге создания нового проекта в мастере (см. описание практической работы 1 и рис. 69). Если ПЛИС не выбрана разработчиком, она выбирается по умолчанию.

На отладочных платах установлены следующие ПЛИС:

- плата DE-1 – Cyclone II (EP2C20F484C7);
- плата MiniDiLab – Cyclone III (EP305E144C8N);
- плата DE2-115 – Cyclone IV (EP4CE115F29C7N);
- плата DE-1-SoC- Cyclone IV (5CSEMA5F31C6N).

Тип целевой ПЛИС можно задать или изменить позже, через пункт меню:

Assignments -> Device

При изменении целевой ПЛИС потребуются доработка и повторная компиляция проекта.

3. Привязка модуля к входным и выходным контактам (пинам) ПЛИС

Каждое устройство на отладочной плате получает управляющие и информационные сигналы от определенных выводов (пинов) ПЛИС,

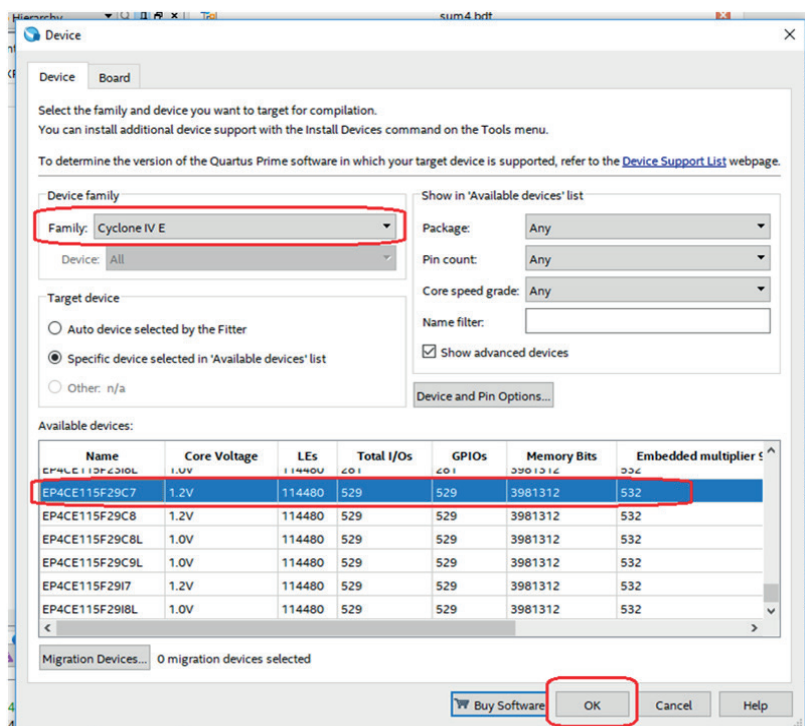


Рис. 69. Выбор целевой ПЛИС

установленной на плате. Если в процессе работы проектируемого модуля предполагаются получение или выдача сигналов устройствам платы, необходимо указать, каким именно выводам микросхемы ПЛИС соответствуют входы и выходы модуля.

Информацию о подключении устройств платы можно получить из документации. Обычно в проектах, создаваемых на практических занятиях, в качестве устройств ввода/вывода используются кнопки, сдвижные переключатели, световые индикаторы и семисегментные индикаторы.

Подключение кнопок

Схема подключения кнопок показана на рис. 70, назначение выводов ПЛИС для кнопок приведено в табл. 12.

Таблица 12

Назначение выводов (пинов) для кнопок

Название сигнала	Номер пина ПЛИС	Описание
KEY [0]	PIN_M23	Push-button [0]
KEY [1]	PIN_M21	Push-button [1]
KEY [2]	PIN_N21	Push-button [2]
KEY [3]	PIN_R24	Push-button [3]

Подключение сдвижных переключателей

Схема подключения сдвижных переключателей показана на рис. 71, назначение выводов ПЛИС для них приведено в табл. 13.

Таблица 13

Назначение выводов (пинов) для сдвижных переключателей

Название сигнала	Номер пина ПЛИС	Описание
SW[0]	PIN_AB28	Slide Switch[0]
SW[1]	PIN_AC28	Slide Switch[1]
SW[2]	PIN_AC27	Slide Switch[2]
SW[3]	PIN_AD27	Slide Switch[3]
SW[4]	PIN_AB27	Slide Switch[4]
SW[5]	PIN_AC26	Slide Switch[5]
SW[6]	PIN_AD26	Slide Switch[6]
SW[7]	PIN_AB26	Slide Switch[7]
SW[8]	PIN_AC25	Slide Switch[8]
SW[9]	PIN_AB25	Slide Switch[9]
SW[10]	PIN_AC24	Slide Switch[10]
SW[11]	PIN_AB24	Slide Switch[11]
SW[12]	PIN_AB23	Slide Switch[12]
SW[13]	PIN_AA24	Slide Switch[13]
SW[14]	PIN_AA23	Slide Switch[14]
SW[15]	PIN_AA22	Slide Switch[15]
SW[16]	PIN_Y24	Slide Switch[16]
SW[17]	PIN_Y23	Slide Switch[17]

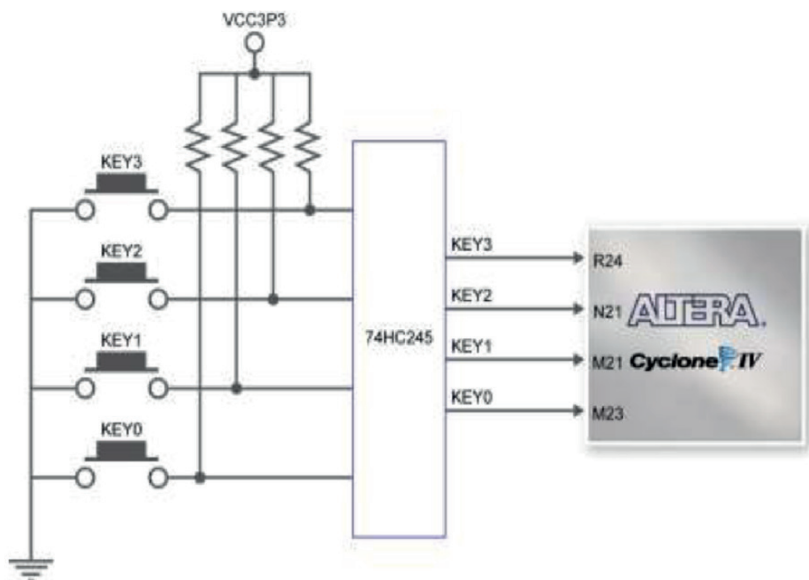


Рис. 70. Схема подключения кнопок к выводам ПЛИС

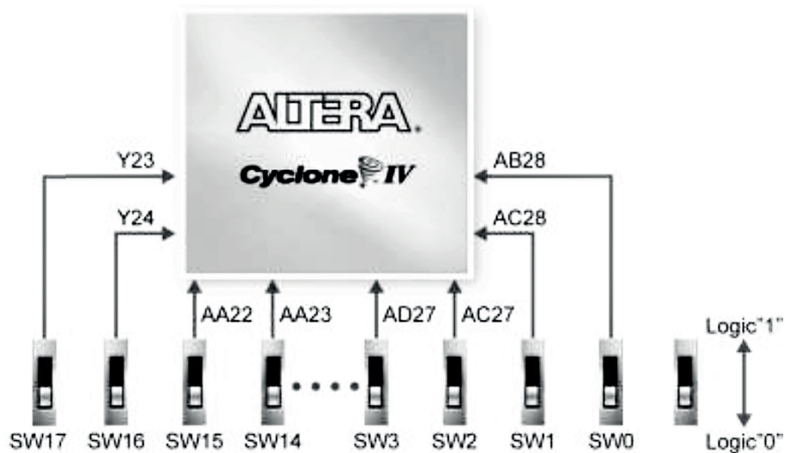


Рис. 71. Схема подключения сдвижных переключателей к выводам ПЛИС

Подключение световых индикаторов (светодиодов)

Схема подключения световых индикаторов показана на рис. 72, назначение выводов ПЛИС для них приведено в табл. 14.

Таблица 14

Назначение выводов (пинов) для световых индикаторов

Название сигнала	Номер пина ПЛИС	Описание
LEDR [0]	PIN_G19	LED Red [0]
LEDR [1]	PIN_F19	LED Red [1]
LEDR [2]	PIN_E19	LED Red [2]
LEDR [3]	PIN_F21	LED Red [3]
LEDR [4]	PIN_F18	LED Red [4]
LEDR [5]	PIN_E18	LED Red [5]
LEDR [6]	PIN_J19	LED Red [6]
LEDR [7]	PIN_H19	LED Red [7]
LEDR [8]	PIN_J17	LED Red [8]
LEDR [9]	PIN_G17	LED Red [9]
LEDR [10]	PIN_J15	LED Red [10]
LEDR [11]	PIN_H16	LED Red [11]
LEDR [12]	PIN_J16	LED Red [12]
LEDR [13]	PIN_H17	LED Red [13]
LEDR [14]	PIN_F15	LED Red [14]
LEDR [15]	PIN_G15	LED Red [15]
LEDR [16]	PIN_G16	LED Red [16]
LEDR [17]	PIN_H15	LED Red [17]
LEDG [0]	PIN_E21	LED Green [0]
LEDG [1]	PIN_E22	LED Green [1]
LEDG [2]	PIN_E25	LED Green [2]
LEDG [3]	PIN_E24	LED Green [3]
LEDG [4]	PIN_H21	LED Green [4]
LEDG [5]	PIN_G20	LED Green [5]
LEDG [6]	PIN_G22	LED Green [6]
LEDG [7]	PIN_G21	LED Green [7]
LEDG [8]	PIN_F17	LED Green [8]

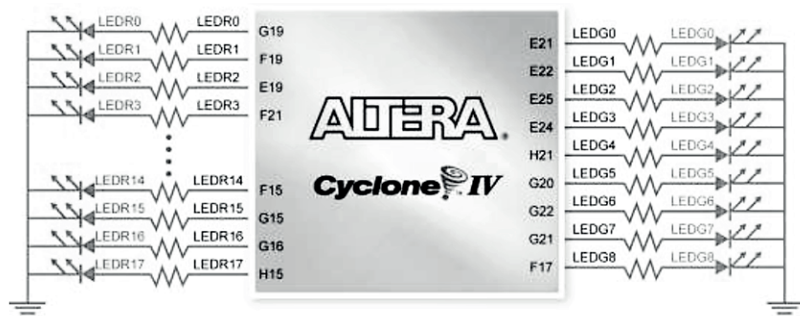


Рис. 72. Схема подключения сдвижных переключателей к выводам ПЛИС

Подключение семисегментного индикатора

Схема подключения семисегментного индикатора показана на рис. 73, назначение выводов ПЛИС для них приведено в табл. 15.

Таблица 15

Назначение выводов (пинов) для семисегментных индикаторов

Название сигнала	Номер пина ПЛИС	Описание
HEX0 [0]	PIN_G18	Seven Segment Digit 0 [0]
HEX0 [1]	PIN_F22	Seven Segment Digit 0 [1]
HEX0 [2]	PIN_E17	Seven Segment Digit 0 [2]
HEX0 [3]	PIN_L26	Seven Segment Digit 0 [3]
HEX0 [4]	PIN_L25	Seven Segment Digit 0 [4]
HEX0 [5]	PIN_J22	Seven Segment Digit 0 [5]
HEX0 [6]	PIN_H22	Seven Segment Digit 0 [6]
HEX1 [0]	PIN_M24	Seven Segment Digit 1 [0]
HEX1 [1]	PIN_Y22	Seven Segment Digit 1 [1]
HEX1 [2]	PIN_W21	Seven Segment Digit 1 [2]
HEX1 [3]	PIN_W22	Seven Segment Digit 1 [3]
HEX1 [4]	PIN_W25	Seven Segment Digit 1 [4]
HEX1 [5]	PIN_U23	Seven Segment Digit 1 [5]
HEX1 [6]	PIN_U24	Seven Segment Digit 1 [6]

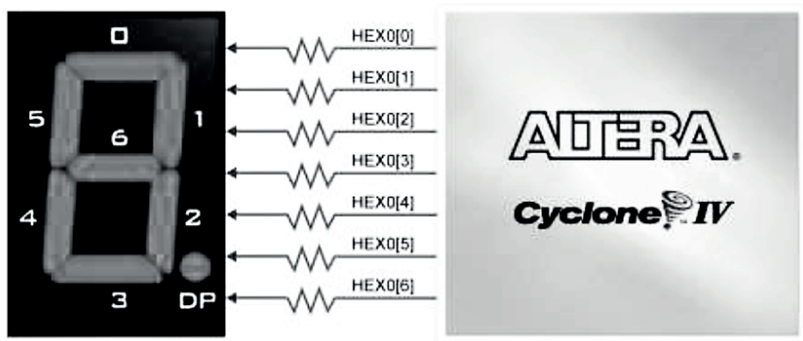


Рис. 73. Схема подключения семисегментного индикатора к выводам ПЛИС

После того как определены выходы ПЛИС, к которым будет подключен проект, их необходимо внести в конфигурацию проекта, используя планировщик выводов (Pin Planner). Для этого запускаем планировщик из главного меню:

Assignments -> Pin Planner

В нижней части рабочего окна планировщика находится таблица, в которой перечислены входы и выходы модуля, как показано на рис. 74.

Необходимо указать в стб. Location номер пина, соответствующего данному выходу. После назначения пинов необходимо выполнить полную компиляцию проекта:

Processing -> Start Compilation

4. Программирование (прошивка) ПЛИС

Для программирования ПЛИС отладочную плату необходимо подключить к компьютеру через порт USB Blaster. Затем нужно запустить программу прошивки ПЛИС из меню:

Tools -> Programmer

Рабочее окно программы-программатора приведено на рис. 75.

При необходимости добавьте в список оборудования USB-Blaster. Если возникли проблемы с установкой USB-Blaster, через диспетчер

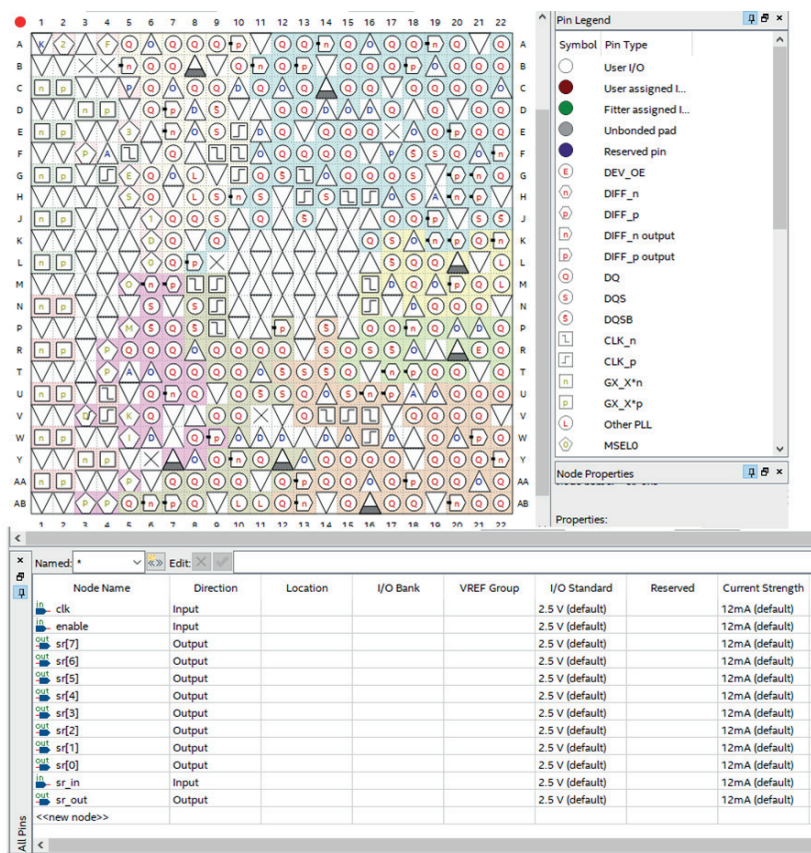


Рис. 74. Рабочее окно планировщика выводов PinPlanner

устройств установите драйверы из папки Quartus. Эти проблемы обычно возникают при первом запуске программатора, в дальнейшем настройки сохраняются и не требуют повторной установки.

В окне выбора файла прошивки программатора необходимо выбрать файл прошивки. Имя файла совпадает с именем модуля верхнего уровня проекта, расширение файла прошивки .sof:

Select Programming File <имя файла прошивки.sof>

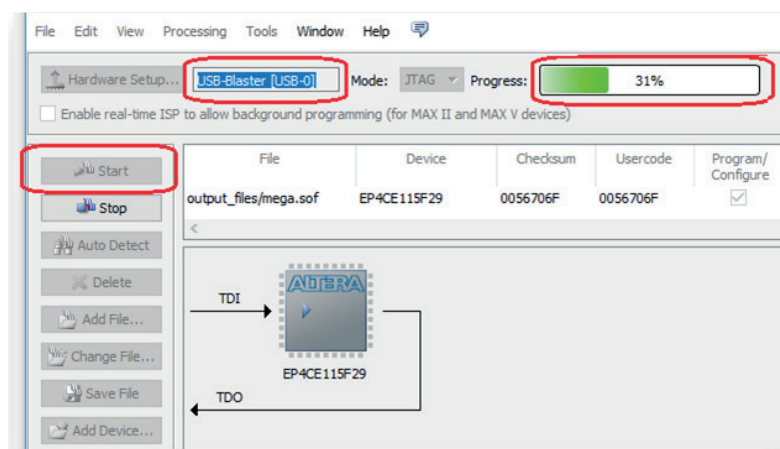


Рис. 75. Рабочее окно программатора

Затем нужно нажать кнопку Start. В окне Progress можно наблюдать за ходом процесса программирования ПЛИС. При успешном завершении процесса об этом появится сообщение.

5. Тестирование проекта

С помощью используемых устройств ввода/вывода отладочной платы проверить работоспособность проекта для различных комбинаций входных сигналов.

Практическая работа 3. Логическое и временное моделирование в среде Quartus

Порядок выполнения практической работы

Часть 1 (на занятии):

1. Ознакомиться с описанием практической работы.
2. Вариант задания совпадает с вариантом практической работы 1.
3. Открыть проект модуля, разработанного в ходе практической работы 1.
4. Выбрать целевую ПЛИС в соответствии с типом используемой отладочной платы.
5. Выполнить полный синтез проекта.

6. Вызвать средство моделирования University Program VWF.
7. Сформировать тестовые входные воздействия (wavefront).
8. Выполнить функциональное и временное моделирование проекта.
9. Проанализировать полученные временные диаграммы, выполнить отладку проекта, если это необходимо. Оценить временные задержки.

Часть 2 (самостоятельная работа):

10. В проект, выполненный в ходе практической работы 1, внести изменения, указанные преподавателем.
11. Выполнить полный синтез проекта.
12. Выполнить функциональное моделирование проекта с использованием University Program VWF и его отладку при необходимости.
13. Выполнить временное моделирование проекта с использованием University Program VWF и оценку временных характеристик полученной схемы (время задержки распространения сигнала).
14. Проанализировать полученные временные диаграммы, выполнить отладку проекта, если это необходимо. Оценить временные задержки.

Описание работы

Тестирование ПЛИС-проекта можно проводить непосредственно после прошивки реальной ПЛИС, как это было предложено сделать в практической работе 2. Но это решение требует значительных временных и экономических затрат: полный синтез и прошивка больших проектов требуют значительного времени, а ресурс ПЛИС по числу циклов перепрограммирования ограничен. Поэтому обычно тестирование проекта на реальном оборудовании выполняется на заключительной стадии работы, непосредственно перед сдачей проекта, а на этапе разработки тестирование и отладка проекта выполняются с использованием моделирования.

Вопросы автоматизации моделирования и тестирования ПЛИС-проекта с использованием языка описания аппаратуры и инструмента ModelSim будут рассмотрены в следующем разделе курса. На данном этапе для моделирования ПЛИС-проектов предлагается использовать средство генерации входных воздействий и моделирования, встроенное в среду Quartus, – University Program VWF.

University Program VWF позволяет выполнить функциональное моделирование, при котором полученные временные диаграммы будут отражать изменения сигналов без учета времени переходных процессов и передачи сигналов в реальных схемах, и временное моделирование, при котором эти задержки учитываются и соответствуют временным характеристикам целевой ПЛИС.

1. Открытие проекта

Перед началом моделирования открыть проект и выполнить анализ и синтез проекта:

Processing -> Start -> Start Analysis & Synthesis

Затем из главного меню вызвать University Program VWF:

File -> New -> University Program VWF

2. Выбор входных и выходных портов для моделирования

В окне редактора временных диаграмм выбрать порты, участвующие в моделировании. Для этого в главном меню окна вызвать функцию редактирования:

Edit -> Insert -> Insert Node or Bus -> Node Finder -> List

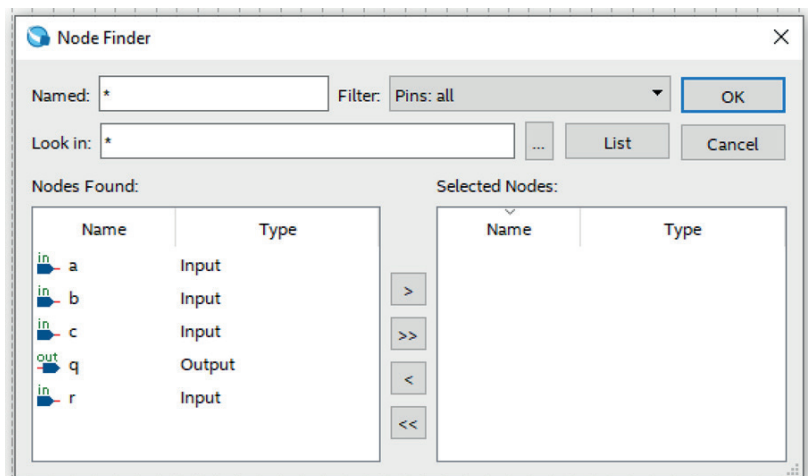


Рис. 76. Окно выбора портов для моделирования

После этого в левом поле окна появится список найденных портов модуля (рис. 76).

Те порты, которые вы бы хотели проконтролировать в процессе моделирования, необходимо перенести в правое поле окна с помощью стрелок. Выбор завершить нажатием кнопки “OK”.

3. Задание входных воздействий

Для задания сигналов на входах модуля можно воспользоваться инструментами задания сигналов, расположенными на панели инструментов, над окном ввода (рис. 77). Назначение инструментов является интуитивно понятным; кроме того, они снабжены выпадающими подсказками.

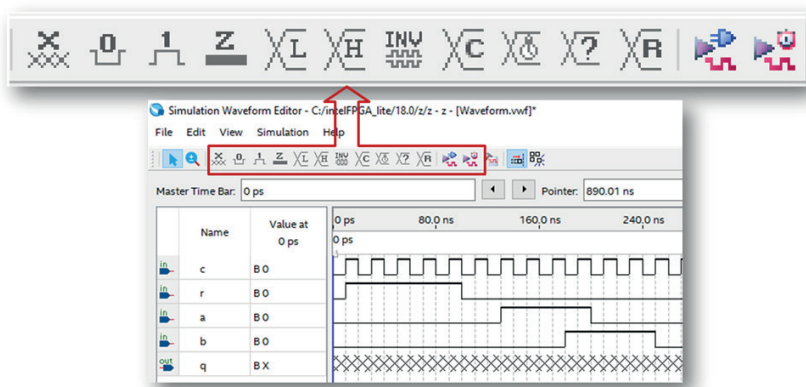


Рис. 77. Панель инструментов задания входных сигналов

4. Запуск процесса моделирования

Запуск моделирования выполняется с панели инструментов или меню Simulation (рис. 78).

Из меню выбором соответствующей строки можно запустить как функциональное, так и временное моделирование.

5. Анализ временных диаграмм

После запуска и выполнения моделирования открывается окно результатов моделирования. Они представлены также в форме временных диаграмм. Входные воздействия сохраняются в файле и могут быть использованы при последующей доработке проекта.

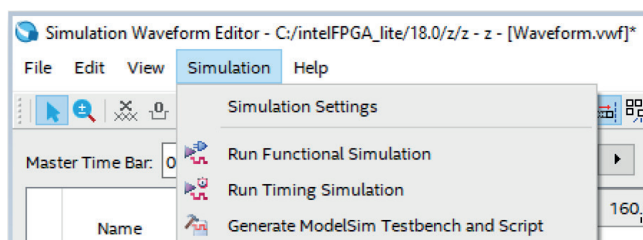
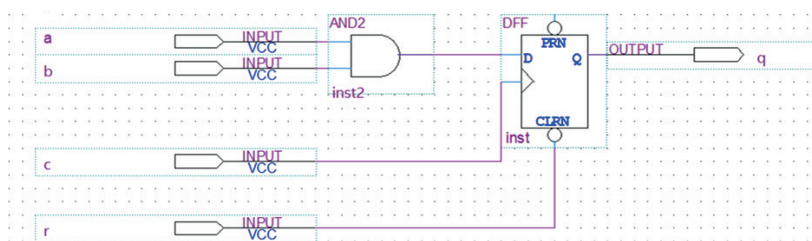
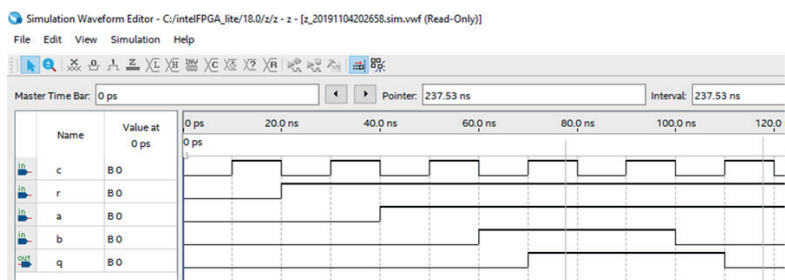


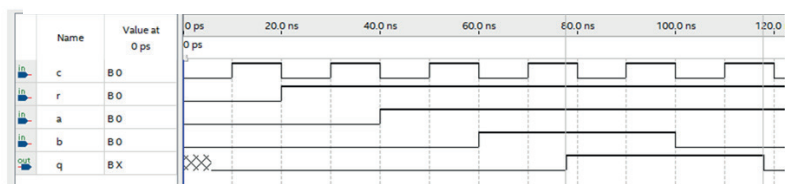
Рис. 78. Меню запуска моделирования



a



б



в

Рис. 79. Пример диаграмм функционального и временного моделирования: *a* – тестируемый модуль; *б* – результаты функционального моделирования; *в* – результаты временного моделирования

Примеры диаграмм функционального и временного моделирования приведены на рис. 79. На временных диаграммах можно увидеть временные задержки изменения выходного сигнала при временном моделировании относительно их изменений при функциональном моделировании. Для удобства просмотра можно менять временной масштаб диаграмм (Ctrl + колесо мышки) и представление данных на многорядных шинах (позиционировать указатель мыши на шину, щелкнуть правой кнопкой мыши, в выпадающем меню выбрать Radix и в выпавшем списке выбрать желаемое представление: двоичное, восьмеричное, шестнадцатеричное, десятичное со знаком или без знака и т. д.).

Практическая работа 4. Описание асинхронных схем на Verilog. Условная логика. Двоичная арифметика

Порядок выполнения

Часть 1 (на занятии):

1. Ознакомиться с описанием практической работы.
2. Получить вариант задания у преподавателя.
3. Разработать описание проектируемого асинхронного устройства на языке Verilog.
4. Создать новый проект, ввести описание, выполнить синтез проекта, проанализировать сообщения различных этапов синтеза, исправить ошибки синтаксиса.
5. Синтезировать RTL-представление для разработанного модуля, проанализировать результат синтеза. При появлении на схеме регистров-«защелок» (latch) исправить исходное описание проекта таким образом, чтобы они были убраны из результатов синтеза.
6. Провести функциональное моделирование проекта средствами University Program VWF, проанализировать полученные временные диаграммы, при необходимости выполнить отладку проекта.
7. Реализовать программирование (прошивку) ПЛИС и протестировать разработанный модуль средствами отладочной платы. При необходимости выполнить отладку проекта.

Часть 2 (самостоятельная работа):

8. В проект, созданный в ходе выполнения на занятии практической работы 4, внести изменения в соответствии с полученным вариантом.

9. Осуществить полный синтез проекта.
10. Провести функциональное моделирование проекта с использованием University Program VWF и его отладку при необходимости.

Описание работы

1. Создание нового проекта

Порядок создания нового проекта на языке описания аппаратуры не отличается от изложенного в практической работе 1.

Далее в проекте создается файл на языке Verilog:

```
File -> New -> Verilog HDL File
```

При необходимости можно сгенерировать описание на Verilog для модуля, созданного посредством схемного ввода:

```
File -> Create/Update -> Create HDL Design File From  
Current File -> Verilog HDL File
```

Например для модуля, рассмотренного в описании практической работы 3, был получен следующий файл на Verilog:

```
module z(  
    a,  
    b,  
    c,  
    r,  
    q  
);  
  
input wire    a;  
input wire    b;  
input wire    c;  
input wire    r;  
output reg    q;  
  
wire    SYNTHESIZED_WIRE_0;  
  
always@(posedge c or negedge r)  
begin  
    if (!r)  
        begin  
            q <= 0;  
        end  
end
```

```

else
  begin
    q <= SYNTHESIZED_WIRE_0;
  end
end
assign SYNTHESIZED_WIRE_0 = a & b;
endmodule

```

2. Ввод описания и редактирование файла

Эти действия выполняются средствами редактора среды Quartus, который имеет интуитивно понятный интерфейс и стандартный набор функций редактирования.

3. Анализ, синтез, компиляция и моделирование проекта

Выполняются теми же средствами, что и в случае проекта, созданного с использованием схемного ввода.

4. Синтез RTL-представления

Для просмотра результатов синтеза в форме логической схемы используется инструмент Netlist Viewers. Его вызов выполняется из меню:

Tools -> Netlist Viewers -> RTL Viewer

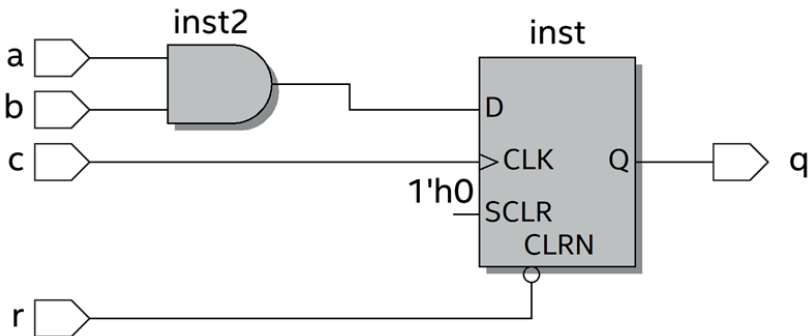


Рис. 80. Пример результата синтеза RTL-представления

Результат синтеза RTL-представления для приведенного выше описания представлен на рис. 80.

Практическая работа 5. Использование параметров при описании модуля на Verilog.

Использование библиотечных модулей и MegaWizzard

Порядок выполнения

Выполняется на занятии

1. Ознакомиться с описанием практической работы. Работа выполняется на базе проекта, созданного в результате практической работы 4.
2. Получить вариант задания у преподавателя.
3. Ввести в описание проекта параметр «Разрядность обрабатываемой информации».
4. Ввести в проект модуль, сгенерированный с использованием MegaWizzard в соответствии с полученным вариантом.
5. Выполнить синтез, функциональное моделирование с использованием University Program VWF и отладку проекта при необходимости.

Описание работы

1. Открытие каталога IP-ядер и библиотечных модулей

Выполняется из меню:

Tools -> IP Catalog

Окно выбора библиотечных IP-модулей обычно располагается справа от окна текстового редактора среды Quartus.

2. Выбор требуемого модуля и запуск MegaWizzard

Каталог IP-ядер представляет собой иерархическую структуру с наименованиями модулей на английском языке.

После выбора требуемого модуля MegaWizzard запускается автоматически.

3. Генерация описания IP-модуля с использованием MegaWizzard

Процесс генерации описания IP-модуля с использованием MegaWizzard выполняется автоматически на основе характеристик и параметров, задаваемых пользователем на этапе подготовки к генерации. Этот этап представляет собой последовательность сменяющихся окон ввода и зависит от выбранной функции. Запросы в окнах ввода сопрово-

ждаются комментариями и подсказками. При возникновении затруднений проконсультируйтесь у преподавателя.

На последнем шаге MegaWizard предлагает подключить файл со сгенерированным описанием модуля к текущему проекту.

4. Использование IP-модуля, сгенерированного MegaWizard

Для использования полученного модуля необходимо добавить его экземпляр (инстанс) в текущий проект.

Описание портов сгенерированного модуля находится в директории проекта в Verilog-файле с именем вида:

`<имя сгенерированного модуля>_inst.v`

5. Анализ, синтез, компиляция и моделирование проекта

Выполняются теми же средствами, которые использовались в ранее описанных практических работах.

Практическая работа 6. Описание модулей памяти на Verilog. Использование шаблонов проектирования

Часть 1 (на занятии):

1. Ознакомиться с описанием практической работы.
2. Получить вариант задания у преподавателя.
3. На основе шаблона проектирования разработать описание регистра с расширенной функциональностью в соответствии с вариантом задания.

4. Выполнить полный синтез проекта, функциональное моделирование с использованием University Program VWF и отладку проекта при необходимости.

5. Выполнить программирование (прошивку) ПЛИС и протестировать разработанный модуль средствами отладочной платы. При необходимости выполнить отладку проекта.

Часть 2 (самостоятельная работа):

6. В соответствии с индивидуальным заданием описать модуль памяти на основе шаблона проектирования.

7. Выполнить полный синтез проекта, функциональное моделирование с использованием University Program VWF и отладку проекта при необходимости.

Описание работы

1. Открытие библиотеки шаблонов

Открытие библиотеки шаблонов выполняется из меню:

Edit -> Insert Template

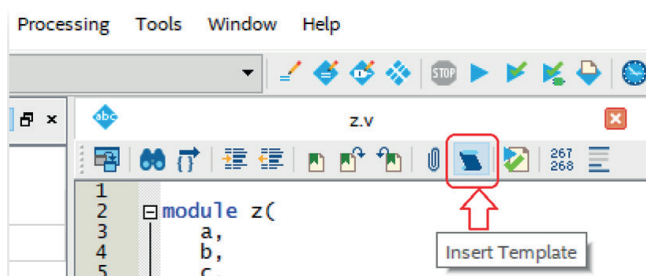


Рис. 81. Открытие библиотеки шаблонов

Или выбором инструмента на панели, как показано на рис. 81.

2. Поиск требуемого шаблона

Каталог шаблонов представляет собой иерархическую структуру с наименованиями модулей на английском языке и является удобным инструментом, позволяющим повысить производительность разработки. Он включает полные описания (Full Designs) модулей памяти, авто-

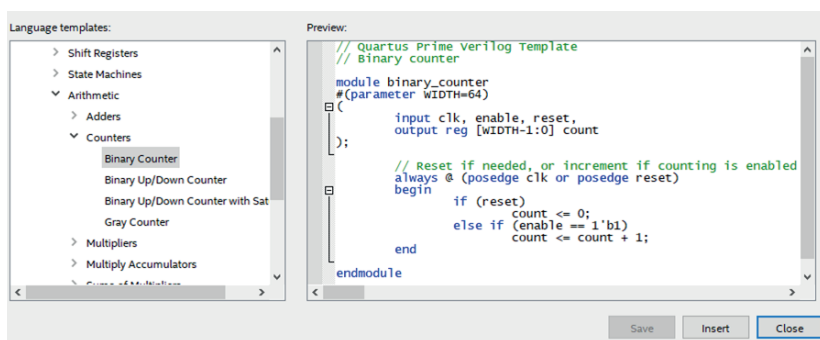


Рис. 82. Окно предварительного просмотра шаблона

матов Миля и Мура и т. д., типовые конструкции Verilog, арифметические и логические операции, счетчики и т. д.

Шаблоны модулей памяти находятся в разделе:

Verilog HDL Full Designs RAMs and ROMs

3. Вставка шаблона в проект

После выбора требуемого шаблона в окне предварительного просмотра предьявляется его текст (рис. 82). В случае подтверждения выбора нажатием кнопки Insert текст шаблона размещается в месте положения курсора в редактируемом файле и в дальнейшем может быть отредактирован в соответствии с заданием.

4. Анализ, синтез, компиляция и моделирование проекта

Выполняются теми же средствами, которые использовались в ранее описанных практических работах.

Практическая работа 7. Описание таймеров-счетчиков на Verilog

Выполняется на занятии

1. Ознакомиться с описанием практической работы.
2. Получить вариант задания у преподавателя.
3. На основе шаблона проектирования разработать описание модуля таймера-счетчика с требуемыми параметрами.
4. Осуществить полный синтез проекта, функциональное моделирование с использованием University Program VWF и отладку проекта при необходимости.
5. Выполнить программирование (прошивку) ПЛИС и протестировать разработанный модуль средствами отладочной платы. При необходимости выполнить отладку проекта.

Описание работы

Шаблоны счетчиков находятся в разделе библиотеки шаблонов:

Verilog -> HDL -> Full Designs -> Arithmetic -> Counters

Все приемы и действия, необходимые для выполнения работы, освоены в предыдущих практических работах.

Практическая работа 8. Описание цифрового автомата на Verilog

Выполняется на занятии

1. Ознакомиться с описанием практической работы.
2. Получить вариант задания у преподавателя.
3. На основе шаблона проектирования или самостоятельно разработать описание цифрового автомата выбранного типа в соответствии с заданием на лабораторную работу.
4. Осуществить полный синтез проекта, функциональное моделирование с использованием University Program VWF и отладку проекта при необходимости.
5. Выполнить программирование (прошивку) ПЛИС и протестировать разработанный модуль средствами отладочной платы. При необходимости выполнить отладку проекта.

Описание работы

Шаблоны описания цифровых автоматов находятся в разделе библиотеки шаблонов:

Verilog HDL -> Full Designs -> State Mashines

Синтез диаграммы переходов автомата возможен только при полном соблюдении формы шаблона. Вызов синтеза диаграммы переходов:

Tools -> Netlist Viewers -> State Mashine Viewer

Все прочие приемы и действия, необходимые для выполнения работы, освоены в предыдущих практических работах.

6.5. Итоговый ПЛИС-проект по разделу «Синтезируемые описания на языке Verilog»

Цель итогового проекта – получение навыков применения инструментов и приемов разработки ПЛИС-проекта в комплексе, для решения прикладной задачи.

Порядок выполнения итогового проекта:

1. Сформировать группу из двух-трех человек.
2. Выбрать вариант задания.
3. Обсудить вариант задания с преподавателем.
4. Разделить проект на модули, определить ответственность за разработку каждого модуля.
5. Осуществить разработку и тестирование отдельных модулей.
6. Провести сборку, отладку и тестирование проекта.
7. Выполнить программирование (прошивку) ПЛИС и протестировать разработанный модуль средствами отладочной платы.

Возможные варианты задания:

1. Собственный soft-процессор со специализированной системой команд на основе проектов АЛУ и регистрового файла.

2. «Игровой автомат». Пример игры:

На индикаторе загорается случайное десятичное (случайная комбинация может быть взята с таймера или может быть дан генератор псевдослучайных чисел). Задача игрока – перевести в двоичный код и набрать на свитчах. Оцениваются время и правильность перевода.

3. Модуль ассоциативной памяти.
4. Формальный нейрон для нейронной сети.
5. Модуль помехозащищенного кодирования и декодирования информации с восстановлением ошибок.

ЗАКЛЮЧЕНИЕ

ПЛИС – удобная в освоении и применении элементная база, альтернативы которой зачастую не найти.

Развитие ПЛИС существенно изменило задачи и возможности проектировщиков цифровой аппаратуры. Для большинства электронных изделий малой и средней серийности наиболее перспективной является их реализация на базе ПЛИС.

Прогресс микроэлектроники привёл к созданию ПЛИС с очень высокой степенью интеграции, применение которых, совместно с использованием современных микропроцессоров и микроконтроллеров, позволяет в короткие сроки выполнить разработку и начать производство электронной аппаратуры нового поколения. Последние годы характеризуются резким ростом плотности упаковки элементов на кристалле. Цены на ПЛИС в долларовом эквиваленте неуклонно падают.

Данное направление активно развивается. При этом развивается как сама аппаратная база ПЛИС, появляются новые изделия универсального применения или рассчитанные на определенную прикладную область (цифровая обработка сигналов, нейронные сети, Интернет вещей и др.), так и средства проектирования. Поэтому, несмотря на то, что в данном пособии приведены основные, базовые, понятия в области программируемых логических интегральных схем и приемы их проектирования, материал нуждается в уточнении и обновлении.

Можно с уверенностью сказать, что навыки эффективного использования ПЛИС при разработке широкой номенклатуры цифровых устройств являются востребованными на рынке труда и перспективными в дальнейшем.

СПИСОК ЛИТЕРАТУРЫ

1. Shannon C. E. A symbolic analysis of relay and switching circuits // *Electrical Engineering*. – 1938. – Т. 57. – № 12. – С. 713–723.
2. История вычислительной техники [Электронный ресурс] // Материал из Википедии – свободной энциклопедии. – URL: https://ru.wikipedia.org/wiki/История_вычислительной_техники (дата обращения: 12.09.2019).
3. Орлов С. А., Цилькер Б. Я. Организация ЭВМ и систем: учеб. для вузов. – 3-е изд. – Стандарт третьего поколения. – Издат. дом «Питер», 2014.
4. Беркс А., Голдстейн Г., Нейман Д. Предварительное рассмотрение логической конструкции электронного вычислительного устройства // *Кибернетический сборник*. – 1964. – № 9. – С. 7–67.
5. Гуров В. Чуканов В. Архитектура и организация ЭВМ [Электронный ресурс] // Национальный открытый университет «Интуит». – URL: <https://www.intuit.ru/studies/courses/60/60/info> (дата обращения: 09.09.2019).
6. Палташев Т. Т. Введение в проблему разработки и производства СБИС // *Gat china Nanoelectronics*, Phoenix, Arizona. – 2009.
7. Филатов А. Становление нового сегмента рынка FPGA. Экономические системные решения на FPGA // *Электроника: Наука, технология, бизнес*. – 2005. – № 3. – С. 20–23.
8. Волощенко А. Тенденция развития программируемых логических интегральных схем // *Надежность и качество: труды Международного симпозиума*. – 2012. – Т. 2.
9. Максфилд К. Проектирование на ПЛИС. Архитектура, средства и методы. – М.: Издат. дом «Додэка-XXI», 2007.
10. Максфилд К. Проектирование на ПЛИС. Курс молодого бойца. – М.: ДМК Пресс, 2015.
11. Стешенко В. Б. ПЛИС фирмы «ALTERA»: элементная база, система проектирования и языки описания аппаратуры. М.: Издат. дом «Додэка-XXI», 2002.
12. Угрюмов Е. П. Цифровая схемотехника. – 3-е изд. – СПб.: БХВ-Петербург, 2010.

13. Уилкинсон Б. Основы проектирования цифровых схем. М.: Издат. дом «Вильямс», 2004.
14. Грушвицкий Р. И., Мурсаев А. Х., Угрюмов Е. П. Проектирование систем на микросхемах с программируемой структурой. – СПб.: БХВ-Петербург, 2006.
15. Тарасов И. Е., Певцов Е. Ф. Программируемые логические схемы и их применение в схемотехнических решениях. – М.: МГТУ МИРЭА, 2012.
16. Киселев А. Lattice Semiconductor: шаг вперед в развитии программируемой логики ПЛИС // Компоненты и Технологии. – 2007. – № 75.
17. Микушин А. В. Цифровые устройства и микропроцессоры. – СПб.: БХВ-Петербург, 2010.
18. Ашихмин А. С. Цифровая схемотехника. Шаг за шагом. – М.: Диалог-МИФИ, 2008.
19. Палагин А. В., Яковлев Ю. С. Особенности проектирования компьютерных систем на кристалле ПЛИС // Математические машины и системы. – 2017. – № 2.
20. Шагурин И. И., Шалтырев В. А. Создание систем на кристалле на основе ПЛИС с использованием синтезируемых процессорных ядер // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). – М.: ИППМ РАН, 2006. – № 1. – С. 382–385.
21. Наваби З. Проектирование встраиваемых систем на ПЛИС. – М.: ДМК Пресс, 2016.
22. Певцов Е. Ф., Тарасов И. Е., Миннебаев В. М. Автоматизированное проектирование цифровых схем: учеб. пособие. – М.: МГТУ МИРЭА. – 2016.
23. Книшев Д. А. ПЛИС фирмы «Xilinx»: описание структуры основных семейств. – М.: ДМК Пресс, 2010.
24. Шауэрман А. Архитектура ПЛИС. Часть 1. Логический элемент [Электронный ресурс] // Лаборатория Электронных Средств Обучения (ЛЭСО) СибГУТИ. – URL: http://www.labfor.ru/articles/fpga_arch_le (дата обращения: 10.08.2019).
25. Справочник по электронным компонентам [Электронный ресурс] // ООО «Рынок Микроэлектроники»: офиц. сайт. – URL: <http://www.gaw.ru> http://www.labfor.ru/articles/fpga_arch_le (дата обращения: 17.08.2019).

26. Ковач Н. Архитектура ПЛИС (FPGA) [Электронный ресурс] // ООО «ИНПРО ПЛЮС»: офиц. сайт. – URL: <https://marsohod.org/index.php/ourblog/11-blog/265-fpga> (дата обращения: 17.09.2019).
27. Схемотехника ЭВМ. Лабораторный практикум: учеб. пособие / под ред. Б. Н. Ковригина. – Изд. 3-е, перераб. и доп. – М.: МИФИ, 2006.
28. Каршенбойм И. Краткий курс HDL. Часть 11. Асинхронные частоты, пересечение клоковых доменов и синхронизация // Компоненты и технологии. – 2009. – № 92.
29. Поляков А. К. Языки VHDL и VERILOG в проектировании цифровой аппаратуры. – М.: СОЛОН-Пресс, 2009.
30. Ефремов Н. В., Бородин А. А. Инструментальные средства проектирования и отладки систем на программируемых кристаллах компании Altera: учеб. пособие. – М.: МГТУ им. Н. Э. Баумана, 2012.
31. Xilinx [Электронный ресурс] // Материал из Википедии – свободной энциклопедии. – URL: <https://ru.wikipedia.org/wiki/Xilinx> (дата обращения: 7.09.2019).
32. Тарасов И. ПЛИС Xilinx и цифровая обработка сигналов. Особенности, преимущества, перспективы // Электроника: наука, технология, бизнес. – 2011. – № 3. – С. 70–74.
33. Lattice Semiconductor – шаг вперед в развитии программируемой логики ПЛИС [Электронный ресурс] // Все о радиоэлектротехнике. – URL: <http://elektors.ru/radioelektronika/mikroshemy/2855-lattice-semiconductor-shag-vpered-v-razvitii-programmiruemoy-logiki-plis.html> (дата обращения: 17.09.2019).
34. Норенков И. П. Основы автоматизированного проектирования: учеб. для вузов. – М.: Изд-во МГТУ им. Н. Э. Баумана, 2009.
35. Стешенко В. Проектирование СБИС. Стили и этапы проекта // Компоненты и технологии. – 2003. – № 30.
36. Мурсаев А. Х. Практикум по проектированию на языках VerilogHDL и SystemVerilog. – М.: Лань, 2017.
37. Бибило П. Н. Основы языка VHDL: учеб. пособие. – М.: Либроком, 2014.
38. Суворова Е. А., Шейнин Ю. Е. Проектирование цифровых систем на VHDL. – СПб.: БХВ-Петербург, 2003.

39. Потехин Д. С., Тарасов И. Е. Разработка систем цифровой обработки сигналов на базе ПЛИС. – М.: Горячая линия – Телеком, 2007.
40. Стемпковский А. Л. Основы логического синтеза средствами САПР Synopsys с использованием Verilog HDL: учеб. пособие. – М.: МИЭТ, 2005.
41. Стешенко В. Б., Попова Т. В., Малашевич Д. Б. Основы HDL Verilog как средства проектирования цифровых устройств. – М.: МИЭТ, 2006.
42. Емец С. Verilog – инструмент разработки цифровых электронных схем // Компоненты и технологии. – 2001. – № 13.
43. Каршенбойм И. Краткий курс HDL. Часть 2. Описание языка Verilog // Компоненты и технологии. – 2008. – № 81.
44. Каршенбойм И. Краткий курс HDL. Часть 10. Несколько слов о «сбросе» // Компоненты и Технологии. – 2009. – № 90.
45. Хопкрофт Д. Э. Введение в теорию автоматов, языков и вычислений. – М.: Изд-во «Вильямс», 2008.
46. Гуренко В. В. Введение в теорию автоматов: учеб. пособие – М.: МГТУ им. Н.Э. Баумана, 2013.
47. Постников А. И., Непомнящий О. В., Макуха Л. В. Прикладная теория цифровых автоматов: учеб. пособие. – Красноярск: СФУ, 2017.
48. Борисенко Н. Технические аспекты построения управляющих автоматов при проектировании цифровых устройств на основе современных ПЛИС // Компоненты и технологии. – 2011. – № 12.
49. Немудров В., Мартин Г. Системы-на-кристалле. Проектирование и развитие. – М.: Техносфера, 2004.
50. Денисов А. О документировании проектов для ПЛИС // Компоненты и Технологии. – 2010. – № 110.
51. Системы на кристалле (технологии высокоуровневого синтеза): учеб.-метод. пособие [Электронный ресурс] / О. В. Непомнящий и др. – Красноярск: СФУ, 2014.
52. Nios_II [Электронный ресурс] // Национальная библиотека им. Н. Э. Баумана. – URL: https://ru.bmstu.wiki/Nios_II (дата обращения: 7.09.2019).
53. Nios II Classic Processor Reference Guide [Электронный ресурс] // Altera: Innovation Drive. – URL: <https://www.intel.com/content/>

dam/ www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu_nii5v1.pdf
(дата обращения: 7.09.2019).

54. Сиротинина Н. Ю., Непомнящий О. В. Программируемые логические интегральные схемы: учеб-метод. материалы к изучению дисциплины [Электронный ресурс]. – Красноярск: СФУ, 2018.

55. Terasic DE2-115 User Manual [Электронный ресурс] // Intel FPGA. URL: <https://www.intel.com/content/dam/www/programmable/us/en/portal/dsn/42/doc-us-dsnbk-42-1404062209-de2-115-user-manual.pdf>
(дата обращения: 27.09.2019).

Учебное издание

Сиротинина Наталья Юрьевна
Непомнящий Олег Владимирович
Постников Александр Иванович
Недорезов Дмитрий Александрович

ПРОГРАММИРУЕМЫЕ ЛОГИЧЕСКИЕ ИНТЕГРАЛЬНЫЕ СХЕМЫ

Учебное пособие

Редактор *М. В. Саблина*
Корректор *В. Р. Наумова*
Компьютерная верстка *О. А. Кравченко*
Фото на обложке:
*[https://akspic.ru/image/14025-centralnyj_
processor-tehnologii-sinij-cpu-elektronnaya_tehnika](https://akspic.ru/image/14025-centralnyj_processor-tehnologii-sinij-cpu-elektronnaya_tehnika)*

Подписано в печать 24.04.2020. Печать плоская. Формат 60×84/16
Бумага офсетная. Усл. печ. л. 14,0. Тираж 100 экз. Заказ № 10229

Библиотечно-издательский комплекс
Сибирского федерального университета
660041, Красноярск, пр. Свободный, 82а
Тел. (391) 206-26-16; <http://bik.sfu-kras.ru>
E-mail: publishing_house@sfu-kras.ru