

В. Б. Стешенко

ПЛИС фирмы ALTERA:
элементная база,
система проектирования
и языки описания аппаратуры

*3-е издание,
стереотипное*

УДК 621.3.049.77
ББК 32.844-02
С79

С79 Стешенко В.Б.

ПЛИС фирмы Altera: элементная база, система проектирования и языки описания аппаратуры. — М.: Издательский дом «Додэка-XXI», 2007. — 576 с.

ISBN 978-594120-112-9

В книге рассмотрены вопросы практического применения ПЛИС фирмы «Altera» при разработке цифровых устройств. Приведены краткие сведения об особенностях архитектуры и временных параметрах устройств. Рассмотрены САПР MAX+PLUS II и Quartus, языки описания аппаратуры AHDL, VHDL, VERILOG HDL. Приводятся примеры описания цифровых устройств на языках высокого уровня, а также примеры реализации некоторых алгоритмов. Приведены сведения о современных интерфейсах передачи данных, даны рекомендации по разработке печатных плат.

Цель книги — помочь начинающему разработчику в выборе элементной базы и дать представление о технологии проектирования устройств на ПЛИС.

УДК 621.3.049.77
ББК 32.844-02

ISBN 978-594120-112-9

© Стешенко В.Б., текст
© Издательский дом «Додэка-XXI», 2002
® Серия «Мировая электроника»

Все права защищены. Никакая часть этого издания не может быть воспроизведена в любой форме или любыми средствами, электронными или механическими, включая фотографирование, ксерокопирование или иные средства копирования или сохранения информации, без письменного разрешения издательства.

ПРЕДИСЛОВИЕ

Программируемые логические интегральные схемы (ПЛИС) — удобная в освоении и применении элементная база, альтернативы которой зачастую не найти. Структурно книга разбита на семь глав и три приложения.

В главе 1 дается обзор перспективных семейств ПЛИС фирмы «Altera» и краткие сведения об особенностях их архитектуры и временных параметрах устройств. В главе 2 рассмотрена САПР MAX+PLUS II. Глава 3 посвящена языку описания аппаратуры AHDL. Язык описания аппаратуры VHDL рассмотрен в главе 4, а аппаратуры VERILOG HDL — в главе 5. В главе 6 приводятся примеры описания цифровых устройств на языках высокого уровня. В главе 7 приведены примеры реализации некоторых алгоритмов. В приложении 1 рассмотрены особенности САПР Quartus. В приложении 2 — современные интерфейсы передачи данных. В приложении 3 даны рекомендации по разработке печатных плат.

Следует заметить, что книга ни в коей мере не подменяет собой фирменную документацию, без которой проектирование устройств просто невозможно. Ее цель — помочь начинающему разработчику в выборе элементной базы и дать представление о технологии проектирования устройств на ПЛИС.

Автор выражает огромную благодарность фирме «Гамма» и лично ее директору М.А.Кузнеченкову за осуществление издания. Автор также благодарит сотрудников фирмы «Гамма» С.Н.Шипулина, И.Г.Алексеева, А.А.Кулакова и зам. директора центра «Логические системы» В.Ю.Храпова за предоставленное программное обеспечение и информацию.

Автор благодарит редакторов Издательского дома «Додэка-XXI» за чуткое и внимательное отношение к рукописи.

Огромный вклад в работу над книгой внесли к.т.н. доцент Д.А. Губанов, аспирант Ю.М. Седякин, студенты А.В. Самохин, Г.В. Шишкин, инженеры Н.Н. Анищенко, А.В. Евстифеев, Р.Б. Гаврилов и др., которым автор выражает искреннюю благодарность.

В журнале Chip News (Новости о микросхемах) был опубликован цикл статей «Школа разработки аппаратуры цифровой обработки сигналов на ПЛИС», ставших основой книги. Автор признателен главному редактору журнала А.Г. Биленко и научному редактору А.А. Осипову за полезные дискуссии, способствовавшие появлению книги.

Наконец, работа над книгой была бы в принципе невозможна без поддержки и терпения семьи.

Автор надеется, что книга найдет понимание у читателя, и будет признателен за все отклики, которые можно присылать по адресу:

107005, Москва, 2-я Бауманская улица, д. 5. Кафедра СМ-5 «Автономные информационные и управляющие системы» МГТУ им. Н.Э. Баумана.

E-mail: steshenk@sm.bmstu.ru

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	13
ГЛАВА 1. ЭЛЕМЕНТНАЯ БАЗА	19
1.1. Семейство MAX3000	19
1.2. Семейство FLEX6000	33
1.3. Семейство MAX7000	46
1.4. Семейство FLEX8000	53
1.5. Семейство MAX9000	58
1.6. Семейство FLEX10K	65
1.7. Семейство APEX20K	78
1.8. Семейство Mercury	85
1.9. Семейство ACEX	102
1.10. Конфигурационные ПЗУ	104
1.11. Программирование и реконфигурирование в системе	110
ГЛАВА 2. СИСТЕМА ПРОЕКТИРОВАНИЯ MAX+PLUS II	115
2.1. Общие сведения	115
2.2. Процедура разработки проекта	121
2.3. Редакторы MAX PLUS II	140
2.4. Процесс компиляции	154
2.5. Верификация проекта	162
ГЛАВА 3. ЯЗЫК ОПИСАНИЯ АППАРАТУРЫ AHDL	167
3.1. Общие сведения	167
3.2. Использование чисел и констант в языке AHDL	171
3.2.1. Использование чисел	171
3.2.2. Использование констант	171
3.3. Комбинационная логика	172

3.3.1.	Реализация булевых выражений и уравнений	172
3.3.2.	Объявление NODE (узел)	173
3.3.3.	Определение групп	173
3.3.4.	Реализация условной логики	174
3.3.5.	Описание дешифраторов	177
3.3.6.	Использование для переменных значений по умолчанию	180
3.3.7.	Реализация логики с активным низким уровнем	182
3.3.8.	Реализация двунаправленных выводов	183
3.4.	Последовательностная логика	184
3.4.1.	Объявление регистров	184
3.4.2.	Объявление регистровых выходов	186
3.4.3.	Создание счетчиков	186
3.5.	Цифровые автоматы с памятью (state mashine)	187
3.5.1.	Реализация цифровых автоматов (state machine)	188
3.5.2.	Установка сигналов Clock, Reset и Enable	189
3.5.3.	Задание выходных значений для состояний	189
3.5.4.	Задание переходов между состояниями	190
3.5.5.	Присвоение битов и значений в цифровом автомате	190
3.5.6.	Цифровые автоматы с синхронными выходами	191
3.5.7.	Цифровые автоматы с асинхронными выходами	193
3.5.8.	Восстановление после неправильных состояний	194
3.6.	Реализация иерархического проекта	196
3.6.1.	Использование макрофункций системы MAX+PLUS II фирмы «Altera»	196
3.6.2.	Создание и применение пользовательских макрофункций	199
3.6.3.	Определение пользовательской макрофункции	199
3.6.4.	Импорт и экспорт цифровых автоматов (state machine)	201
3.7.	Управление синтезом	203
3.7.1.	Реализация примитивов LCELL и SOFT	203
3.7.2.	Значения констант по умолчанию	205
3.7.3.	Присвоение битов и значений в цифровом формате	205
3.8.	Элементы языка AHDL	205
3.8.1.	Зарезервированные ключевые слова	205
3.8.2.	Символы	207
3.8.3.	Имена в кавычках и без кавычек	208
3.8.4.	Группы	209
3.8.5.	Числа в языке AHDL	211
3.8.6.	Булевы выражения	211
3.8.7.	Логические операторы	212
3.8.8.	Выражения с оператором NOT	213

3.8.9. Выражения с операторами AND, NAND, OR, XOR, & XNOR	213
3.8.10. Арифметические операторы	214
3.8.11. Компараторы (операторы сравнения)	215
3.8.12. Приоритеты в булевых уравнениях	216
3.8.13. Примитивы	216
3.8.14. Порты	226
3.9. Синтаксис языка AHDL	229
3.9.1. Лексические элементы	229
3.9.2. Основные конструкции языка AHDL	230
3.9.3. Синтаксис объявления названия	233
3.9.4. Синтаксис оператора включения	233
3.9.5. Синтаксис задания константы	233
3.9.6. Синтаксис прототипа функции	233
3.9.7. Синтаксис оператора вариантов	234
3.9.8. Синтаксис секции подпроекта Subdesign	236
3.9.9. Синтаксис секции переменных	236
3.9.10. Синтаксис объявления цифрового автомата	237
3.9.11. Синтаксис объявления псевдонима цифрового автомата	237
3.9.12. Синтаксис логической секции	238
3.9.13. Синтаксис булевых уравнений	238
3.9.14. Синтаксис булевых уравнений управления	238
3.9.15. Синтаксис оператора CASE	238
3.9.16. Объявление по умолчанию	238
3.9.17. Синтаксис условного оператора IF	239
3.9.18. Синтаксис встроенных (in-line) ссылок на макрофункцию или примитив	239
3.9.19. Синтаксис объявления таблицы истинности	239
3.9.20. Синтаксис порта	239
3.9.21. Синтаксис группы	240
3.9.22. Синтаксические группы и списки	241
ГЛАВА 4. ЯЗЫК ОПИСАНИЯ АППАРАТУРЫ VHDL	243
4.1. Общие сведения	243
4.2. Алфавит языка	246
4.2.1. Комментарии	247
4.2.2. Числа	247
4.2.3. Символы	248
4.2.4. Строки	248
4.3. Типы данных	248

4.3.1. Простые типы	248
4.3.2. Сложные типы	249
4.3.3. Описание простых типов	250
4.4. Операторы VHDL	261
4.4.1. Основы синтаксиса	261
4.4.2. Объекты	262
4.4.3. Атрибуты	263
4.4.4. Компоненты	263
4.4.5. Выражения	264
4.4.6. Операторы	265
4.5. Интерфейс и тело объекта	266
4.5.1. Описание простого объекта	270
4.5.2. Объявление объекта проекта F	270
4.5.3. Поведенческое описание архитектуры	270
4.5.4. Поточковая форма	271
4.5.5. Структурное описание архитектуры	273
4.6. Описание конфигурации	274
4.7. Векторные сигналы и регулярные структуры	275
4.8. Задержки сигналов и параметры настройки	277
4.9. Атрибуты сигналов и контроль запрещенных ситуаций	279
4.10. Алфавит моделирования и пакеты	280
4.11. Описание монтажного «ИЛИ» и общей шины	283
4.12. Синтезируемое подмножество VHDL	285
4.12.1. Общие сведения	285
4.12.2. Переопределенные типы (Redefined types)	286
4.12.3. Методика верификации синтезируемого описания (Verification methodology)	286
4.12.4. Моделирование элементов аппаратуры (Modeling hardware elements)	288
4.12.5. Директивы компилятора (псевдокомментарии, Pragmas)	295
4.12.6. Синтаксис синтезируемого подмножества VHDL	297
4.13. Краткое описание синтаксиса синтезируемого подмножества VHDL	343
ГЛАВА 5. ЯЗЫК ОПИСАНИЯ АППАРАТУРЫ VERILOG HDL	363
5.1. Общие сведения	363
5.2. Операторы	365
5.3. Числа в Verilog	365
5.3.1. Целые числа (Integers)	365
5.3.2. Неопределенное и высокоимпедансное состояния (x and z values)	366

5.3.3. Отрицательные числа (Negative numbers)	366
5.3.4. Подчеркивание (Underscore)	366
5.3.5. Действительные числа (Real)	366
5.3.6. Строки (Strings)	367
5.4. Цепи в Verilog (Nets)	367
5.5. Регистры (Registers)	367
5.6. Векторы (Vectors)	369
5.7. Массивы (Arrays)	370
5.8. Регистровые файлы (Memories)	370
5.9. Элементы с третьим состоянием (Tri-state)	370
5.10. Арифметические операторы (Arithmetic operators)	372
5.11. Логические операторы (Logical operators)	373
5.12. Операторы отношения (Relational operators)	374
5.13. Операторы эквивалентности (Equality)	374
5.14. Поразрядные операторы (Bitwise operators)	375
5.15. Операторы приведения (Reduction operator)	376
5.16. Операторы сдвига (Shift operator)	377
5.17. Конкатенация (объединение, Concatenation)	377
5.18. Повторение (Replication)	378
5.19. Системные директивы (System tasks)	378
5.19.1. Директивы вывода результатов моделирования (Writing to standard output)	379
5.19.2. Контроль процесса моделирования (Monitoring a simulation)	380
5.19.3. Окончание моделирования (Ending a simulation)	382
5.20. Проектирование комбинационных схем, пример проектирования мультиплексора 4 в 1	382
5.20.1. Реализация на уровне логических вентилях (Gate level implementation)	382
5.20.2. Реализация мультиплексора с помощью логических операторов (Logic statement Implementation)	384
5.20.3. Реализация с помощью оператора выбора (CASE statement implementation)	385
5.20.4. Реализация с использованием условного оператора (Conditional operator Implementation)	387
5.20.5. Тестовый модуль (The stimulus module)	387
5.21. Модули проекта (Design blocks modules)	390
5.21.1. Тестирование	392
5.22. Порты (Ports)	393
5.23. Правила соединения (Connection rules)	394

5.23.1. Входы (inputs)	394
5.23.2. Выходы (outputs)	394
5.23.3. Двухнаправленные выходы (inouts)	394
5.23.4. Соответствие портов (Port matching)	394
5.23.5. Присоединение портов (Connecting ports)	394
5.24. Базовые блоки (Basic blocks)	395
5.24.1. Инициализация (Initial block)	395
5.24.2. Конструкция Always (Always block)	395
5.25. Пример проектирования последовательностного устройства: двоичный счетчик	396
5.25.1. Поведенческая модель счетчика (Behavioural model)	400
5.26. Временной контроль (Timing Control)	402
5.26.1. Задержки (delay)	402
5.26.2. Событийный контроль (event-based control)	403
5.27. Защелкивание (triggers)	403
5.28. Список сигналов возбуждения (sensitivity list)	404
5.29. Задержка распространения в вентиле (Gate delays)	404
5.30. Операторы ветвления (Branch statements)	404
5.30.1. Оператор IF (IF statement)	404
5.30.2. Оператор выбора (CASE statement)	406
5.30.3. Оператор ветвления (Conditional operator)	406
5.31. Циклы (Looping constructs)	407
5.31.1. Цикл WHILE (WHILE LOOP)	407
5.31.2. Цикл FOR (FOR LOOP)	408
5.31.3. Цикл REPEAT (REPEAT LOOP)	408
5.31.4. Вечный цикл (FOREVER LOOP)	409
5.32. Файлы в Verilog	409
5.32.1. Открытие файла (Opening a file)	409
5.32.2. Запись в файл (Writing to a file)	410
5.32.3. Закрытие файла (Closing a file)	410
5.32.4. Инициализация регистровых файлов (памяти) (Initialising memories)	410
5.33. Задание векторов входных сигналов для моделирования (Verilog input vectors)	412
5.34. Список операторов Verilog	414
5.35. Приоритет операторов	415
5.36. Ключевые слова (keywords)	415
5.37. Директивы компилятора	416
5.38. Типы цепей (Net types)	416

ГЛАВА 6. ПРИМЕРЫ ПРОЕКТИРОВАНИЯ ЦИФРОВЫХ УСТРОЙСТВ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКОВ ОПИСАНИЯ АППАРАТУРЫ VHDL И VERILOG	417
6.1. Общие сведения	417
6.2. Триггеры и регистры	418
6.2.1. Триггеры, тактируемые передним фронтом (Rising Edge Flipflop)	419
6.2.2. Триггеры, тактируемые передним фронтом, с асинхронным сбросом (Rising Edge Flipflop with Asynchronous Reset) . . .	420
6.2.3. Триггеры, тактируемые передним фронтом, с асинхронной предустановкой (Rising Edge Flipflop with Asynchronous Preset)	421
6.2.4. Триггеры, тактируемые передним фронтом, с асинхронным сбросом и предустановкой (Rising Edge Flipflop with Asynchronous Reset and Preset)	422
6.2.5. Триггеры, тактируемые передним фронтом, с синхронным сбросом (Rising Edge Flipflop with Synchronous Reset)	424
6.2.6. Триггеры, тактируемые передним фронтом, с синхронной предустановкой (Rising Edge Flipflop with Synchronous Preset)	425
6.2.7. Триггеры, тактируемые передним фронтом, с асинхронным сбросом и разрешением тактового сигнала (Rising Edge Flipflop with Asynchronous Reset and Clock Enable)	426
6.2.8. Защелка с разрешением выхода (D-Latch with Data and Enable)	427
6.2.9. Защелка с входом данных с разрешением (D-Latch with Gated Asynchronous Data)	428
6.2.10. Защелка с входом разрешения (D-Latch with Gated enable) . .	429
6.2.11. Защелка с асинхронным сбросом (D-Latch with Asynchronous Reset)	430
6.3. Построение устройств потоковой обработки данных (Datapath logic)	431
6.4. Счетчики	439
6.5. Арифметические устройства	443
6.6. Конечные автоматы (Finite state machine)	449
6.7. Элементы ввода-вывода	459
6.8. Параметризация	464
6.9. Специфика проектирования устройств с учетом архитектурных особенностей ПЛИС	466
6.10. Совместное использование ресурсов	468

6.11. Дублирование регистра	473
6.12. Создание описаний с учетом особенностей архитектуры ПЛИС (Technology Specific Coding Techniques)	476

ГЛАВА 7. ПРИМЕРЫ РЕАЛИЗАЦИИ АЛГОРИТМОВ ЦОС НА ПЛИС

485

7.1. Реализация цифровых фильтров на ПЛИС семейства FLEX фирмы «Altera»	485
7.2. Реализация цифровых полиномиальных фильтров	491
7.3. Алгоритмы функционирования и структурные схемы демодуляторов	495
7.4. Реализация генератора ПСП на ПЛИС	500
7.5. Примеры описания цифровых схем на VHDL	506
7.6. Реализация нейрона на AHDL	516
7.7. Построение быстродействующих перемножителей	529

Приложение 1. Система проектирования Quartus	535
---	-----

Приложение 2. Интерфейсы передачи данных и сопряжение устройств	537
---	-----

Приложение 3. Практические рекомендации по разработке печатных плат	568
---	-----

Литература	572
-------------------------	-----

ВВЕДЕНИЕ

Идея написания этой книги назревала в течение последних двух-трех лет, когда для многих разработчиков аппаратуры центральной оперативной системы (ЦОС) стало ясно, что программируемые логические интегральные схемы (ПЛИС) — удобная в освоении и применении элементная база, альтернативы которой зачастую не найти. Последние годы характеризуются резким ростом плотности упаковки элементов на кристалле, многие ведущие производители либо начали серийное производство, либо анонсировали ПЛИС с эквивалентной емкостью более 1 миллиона логических вентилей. Цены на ПЛИС (к сожалению, только лишь в долларовом эквиваленте) неуклонно падают. Так, еще год-полтора назад ПЛИС логической емкостью 100000 вентилей стоила в Москве в зависимости от производителя, приемки, быстродействия от 1500 до 3000 у.е., сейчас такая микросхема стоит от 100 до 350 у.е., т.е. цены упали практически на порядок, и эта тенденция устойчива. Что касается ПЛИС емкостью 10000...30000 логических вентилей, то появились микросхемы стоимостью менее 10 у.е.

Такая ситуация на рынке вызвала волну вопросов, связанных с подготовкой специалистов, способных проводить разработку аппаратуры цифровой обработки сигналов на ПЛИС, владеющих основными методами проектирования, ориентирующихся в современной элементной базе и программном обеспечении. Идя навстречу многочисленным пожеланиям предприятий, заинтересованных в подготовке молодых специалистов, владеющих современными технологиями, на кафедре СМ-5 «Автономные информационные и управляющие системы» МГТУ им. Н.Э. Баумана в программу четырехсеместрового курса «Схемотехническое проектирование микроэлектронных устройств» включен семестровый раздел «Проектирование аппаратуры обработки сигналов на ПЛИС», на основе лекционных и семинарских материалов которого и выходит этот цикл статей.

Приведем известную классификацию ПЛИС [1, 2, 3] по структурному признаку, так как она дает наиболее полное представление о классе задач, пригодных для решения на той или иной ПЛИС. Следует заметить, что общепринятой оценкой логической емкости ПЛИС является число эквивалентных вентилях, определяемое как среднее число вентилях «2И-НЕ», необходимых для реализации эквивалентного проекта на ПЛИС и базовом матричном кристалле (БМК). Понятно, что эта оценка весьма условна, поскольку ПЛИС не содержат вентилях «2И-НЕ» в чистом виде, однако для проведения сравнительного анализа различных архитектур она вполне пригодна. Основными критериями такой классификации являются наличие, вид и способы коммутации элементов логических матриц. По этому признаку можно выделить следующие классы ПЛИС.

Программируемые логические матрицы — наиболее традиционный тип ПЛИС, имеющий программируемые матрицы «И» и «ИЛИ». В зарубежной литературе соответствующими этому классу аббревиатурами являются FPLA (Field Programmable Logic Array) и FPLS (Field Programmable Logic Sequencers). Примерами таких ПЛИС могут служить отечественные схемы К556РТ1, РТ2, РТ21. Недостаток такой архитектуры — слабое использование ресурсов программируемой матрицы «ИЛИ», поэтому дальнейшее развитие получили микросхемы, построенные по архитектуре программируемой матричной логики (ПМЛ — Programmable Array Logic, PAL) — это ПЛИС, имеющие программируемую матрицу «И» и фиксированную матрицу «ИЛИ». К этому классу относится большинство современных ПЛИС небольшой степени интеграции. В качестве примеров можно привести отечественные интегральные схемы (ИС) КМ1556ХП4, ХП6, ХП8, ХЛ8, ранние разработки (середина—конец 80-х годов) ПЛИС фирм «Intel», «Altera», «AMD», «Lattice» и др. Разновидностью класса ПМЛ являются ПЛИС, имеющие только одну (программируемую) матрицу «И», например схема 85С508 фирмы «Intel».

Следующий традиционный тип ПЛИС — программируемая макрологика. Она содержит единственную программируемую матрицу «И-НЕ» или «ИЛИ-НЕ», но за счет многочисленных инверсных обратных связей способна формировать сложные логические функции. К этому классу относятся, например, ПЛИС PLHS501 и PLHS502 фирмы «Signetics», имеющие матрицу «И-НЕ», а также схема XL78С800 фирмы «Exel», основанная на матрице «ИЛИ-НЕ».

Вышеперечисленные архитектуры ПЛИС содержат небольшое число ячеек, к настоящему времени морально устарели и применяются для реа-

лизации относительно простых устройств, для которых не существует готовых ИС средней степени интеграции. Естественно, для реализации алгоритмов ЦОС они не пригодны.

ИС ПМЛ (Programmable Logic Device, PLD) имеют архитектуру, весьма удобную для реализации цифровых автоматов. Развитие этой архитектуры — программируемые коммутируемые матричные блоки (ПКМБ) — это ПЛИС, содержащие несколько матричных логических блоков (МЛБ), объединенных коммутационной матрицей. Каждый МЛБ представляет собой структуру типа ПМЛ, т.е. программируемую матрицу «И», фиксированную матрицу «ИЛИ» и макроячейки. ПЛИС типа ПКМБ, как правило, имеют высокую степень интеграции (до 10000 эквивалентных вентилях, до 256 макроячеек). К этому классу относятся ПЛИС семейства MAX5000 и MAX7000 фирмы «Altera», схемы XC7000 и XC9500 фирмы «Xilinx», а также большое число микросхем других производителей (фирмы «Atmel», «Vantis», «Lucent» и др.). В зарубежной литературе они получили название Complex Programmable Logic Devices (CPLD).

Другой тип архитектуры ПЛИС — программируемые вентиляльные матрицы (ПВМ), состоящие из логических блоков (ЛБ) и коммутирующих связей — программируемых матриц соединений. Логические блоки таких ПЛИС состоят из одного или нескольких относительно простых логических элементов, в их основе лежит таблица перекодировки (ТП — Look-up table, LUT), программируемый мультиплексор, D-триггер, а также цепи управления. Таких простых элементов может быть достаточно большое количество, у современных ПЛИС емкостью до 1 миллиона вентилях число логических элементов достигает нескольких десятков тысяч. За счет такого большого числа логических элементов они содержат значительное число триггеров, также некоторые семейства ПЛИС имеют встроенные реконфигурируемые модули памяти (РМП — Embedded Array Block, EAB), что делает ПЛИС данной архитектуры весьма удобным средством реализации алгоритмов цифровой обработки сигналов, основными операциями в которых являются перемножение, умножение на константу, суммирование и задержка сигнала. Вместе с тем возможности комбинационной части таких ПЛИС ограничены, поэтому совместно с ПВМ применяют ПКМБ (CPLD) для реализации управляющих и интерфейсных схем. В зарубежной литературе такие ПЛИС получили название Field Programmable Gate Array (FPGA). К FPGA (ПВМ) классу относятся ПЛИС XC2000, XC3000, XC4000, Spartan, Virtex фирмы «Xilinx», ACT1, ACT2 фирмы

«Actel», а также семейства FLEX8000 фирмы «Altera», некоторые ПЛИС фирм «Atmel» и «Vantis».

Множество конфигурируемых логических блоков (Configurable Logic Blocks, CLB) объединяются с помощью матрицы соединений. Характерными для FPGA архитектур являются элементы ввода-вывода (Input/Output Blocks, IOBs), позволяющие реализовать двунаправленный ввод/вывод, третье состояние и т.п.

Особенностью современных ПЛИС является возможность тестирования узлов с помощью порта JTAG (B-scan), а также наличие внутреннего генератора (Osc) и схем управления последовательной конфигурацией.

Фирма «Altera» пошла по пути развития FPGA-архитектур и предложила в семействе FLEX10K так называемую двухуровневую архитектуру матрицы соединений.

Логические элементы (ЛЭ) объединяются в группы — логические блоки (ЛБ). Внутри логических блоков ЛЭ соединяются посредством локальной программируемой матрицы соединений, позволяющей соединять любой ЛЭ с любым другим. Логические блоки связаны между собой и с элементами ввода-вывода посредством глобальной программируемой матрицы соединений (ГПМС). Локальная и глобальная матрицы соединений имеют непрерывную структуру — для каждого соединения выделяется непрерывный канал.

Дальнейшее развитие архитектур идет по пути создания комбинированных архитектур, сочетающих удобство реализации алгоритмов ЦОС на базе таблиц перекодировок и реконфигурируемых модулей памяти, характерных для FPGA-структур и многоуровневых ПЛИС с удобством реализации цифровых автоматов на CPLD-архитектурах. Так, ПЛИС APEX20K фирмы «Altera» содержат в себе логические элементы всех перечисленных типов, что позволяет применять ПЛИС как основную элементную базу для «систем на кристалле» (System-On-Chip, SOC). В основе идеи SOC лежит интеграция всей электронной системы в одном кристалле (например, в случае персонального компьютера (ПК) такой чип объединяет процессор, память, и т.д.). Компоненты этих систем разрабатываются отдельно и хранятся в виде файлов параметризуемых модулей. Окончательная структура SOC-микросхемы выполняется на базе этих «виртуальных компонентов» с помощью программ систем автоматизации проектирования (САПР) электронных устройств — EDA (Electronic Design Automation). Благодаря стандартизации в одно целое можно объединять «виртуальные компоненты» от разных разработчиков.

Как известно, при выборе элементной базы систем обработки сигналов обычно руководствуются следующими критериями отбора:

- быстродействие;
- логическая емкость, достаточная для реализации алгоритма;
- схемотехнические и конструктивные параметры ПЛИС, надежность, рабочий диапазон температур, стойкость к ионизирующим излучениям и т.п.;
- стоимость владения средствами разработки, включающая как стоимость программного обеспечения, так наличие и стоимость аппаратных средств отладки;
- стоимость оборудования для программирования ПЛИС или конфигурационных ПЗУ (постоянное запоминающее устройство);
- наличие методической и технической поддержки;
- наличие и надежность российских поставщиков;
- стоимость микросхем.

В данной книге рассматриваются вопросы проектирования устройств обработки информации на базе ПЛИС фирмы «Altera».

Фирма «Altera Corporation» (101 Innovation Drive, San Jose, CA 95134, USA, www.altera.com) была основана в июне 1983 года. В настоящее время последним достижением этой фирмы является семейство APEX20K.

Кроме того, фирма «Altera» выпускает CPLD семейств MAX3000, MAX7000, MAX9000 (устаревшие серии специально не упоминаются), FPGA семейств FLEX10K, FLEX8000, FLEX6000.

Дополнительным фактором при выборе ПЛИС фирмы «Altera» является наличие достаточно развитых бесплатных версий САПР. В Табл. В.1 приведены основные характеристики пакета MAX+PLUS II BASELINE версия 9.3 фирмы «Altera», который можно бесплатно «скачать» с сайта www.altera.com или получить на CD «Altera Digital Library», на котором содержится также и полный набор документации по архитектуре и применению ПЛИС.

Кроме того, ПЛИС фирмы «Altera» выпускаются с возможностью программирования в системе непосредственно на плате. Для программирования и загрузки конфигурации устройств опубликована схема загрузочного кабеля ByteBlaster и ByteBlasterMV. Следует отметить, что новые конфигурационные ПЗУ EPC2 и EPC16 позволяют программирование с помощью этого устройства, тем самым отпадает нужда в программаторе, что, естественно, снижает стоимость владения технологией.

ПЛИС фирмы «Altera» выпускаются в коммерческом и промышленном диапазоне температур.

Таблица В.1. Основные характеристики пакета MAX+PLUS II BASELINE
версии 9.3

Поддерживаемые устройства	ACEX, EPF10K10, EPF10K10A, EPF10K20, EPF10K30, EPF10K30A, EPF10K30E (до 30000 эквивалентных вентилях), EPM9320, EPM9320A, EPF8452A, EPF8282A, MAX7000, FLEX6000, MAX5000, MAX3000A, Classic
Средства описания проекта	Схемный ввод, поддержка AHDL, средства интерфейса с САПР третьих фирм, топологический редактор, иерархическая структура проекта, наличие библиотеки параметризуемых модулей
Средства компиляции проекта	Логический синтез и трассировка, автоматическое обнаружение ошибок, поддержка мегафункций по программам MegaCore и AMPP
Средства верификации проекта	Временной анализ, функциональное и временное моделирование, анализ сигналов, возможность использования программ моделирования (симуляторов) третьих фирм

Глава 1. Элементная база

1.1. Семейство MAX3000

Летом 1999 года на рынке стали доступны ПЛИС семейства MAX3000. Их архитектура близка к архитектуре семейства MAX7000, однако имеется ряд небольших отличий. В **Табл. 1.1** приведены основные параметры ПЛИС.

Таблица 1.1. Основные параметры ПЛИС семейства MAX3000

Параметр	ЕРМ3032А	ЕРМ3064А	ЕРМ3128А	ЕРМ3256А
Логическая емкость, количество эквивалентных вентилях	600	1 250	2 500	5 000
Число макроячеек	32	64	128	256
Число логических блоков	2	4	8	16
Число программируемых пользователем выводов	34	66	96	158
Задержка распространения сигнала вход/выход, t_{PD} [нс]	4.5	4.5	5	6
Время установки глобального тактового сигнала, t_{SU} [нс]	3.0	3.0	3.2	3.7
Задержка глобального тактового сигнала до выхода, t_{CO1} [нс]	2.8	2.8	3.0	3.3
Максимальная глобальная тактовая частота, f_{CNT} [МГц]	192.3	192.3	181.8	156.3

Микросхемы семейства MAX3000 выполнены по КМОП EPROM 0.35-микронной технологии, что позволило существенно удешевить их по сравнению с семейством MAX7000S. Все ПЛИС семейства

MAX3000 поддерживают технологию программирования в системе ISP (In-System Programmability) и периферийного сканирования (boundary scan) в соответствии со стандартом IEEE Std. 1149.1 — 1990 (JTAG). Элементы ввода/вывода (ЭВВ) позволяют работать в системах с уровнями сигналов 5, 3.3 и 2.5 В. Матрица соединений имеет непрерывную структуру, что позволяет реализовать время задержки распространения сигнала до 4.5 нс. ПЛИС семейства MAX3000 имеют возможность аппаратной эмуляции выходов с открытым коллектором (open drains pin) и удовлетворяют требованиям стандарта PCI. Имеется возможность индивидуального программирования цепей сброса, установки и тактирования триггеров, входящих в макроячейку. Предусмотрен режим пониженного энергопотребления. Программируемый логический расширитель позволяет реализовать на одной макроячейке функции до 32 переменных. Имеется возможность задания бита секретности (security bit) для защиты от несанкционированного тиражирования разработки.

Реализация функции программирования в системе поддерживается с использованием стандартных средств загрузки, таких, как ByteBlasterMV, BitBlaster, MasterBlaster, а также поддерживается формат JAM.

ПЛИС семейства MAX3000 выпускаются в корпусах от 44 до 208 выводов.

На **Рис. 1.1** представлена функциональная схема ПЛИС семейства MAX3000. Основными элементами структуры ПЛИС семейства MAX3000 являются:

- логические блоки (ЛБ, Logic array blocks, LAB);
- макроячейки (МЯ, macrocells);
- логические расширители (expanders) (параллельный (parallel) и разделяемый (shareble));
- программируемая матрица соединений (ПМС, Programmable Interconnect Array, PIA);
- элементы ввода/вывода (ЭВВ, I/O control block).

ПЛИС семейства MAX3000 имеют четыре вывода, закрепленных за глобальными цепями (dedicated inputs). Это глобальные цепи синхронизации сброса и установки в третье состояние каждой макроячейки. Кроме того, эти выводы можно использовать как входы или выходы пользователя для «быстрых» сигналов, обрабатываемых в ПЛИС.

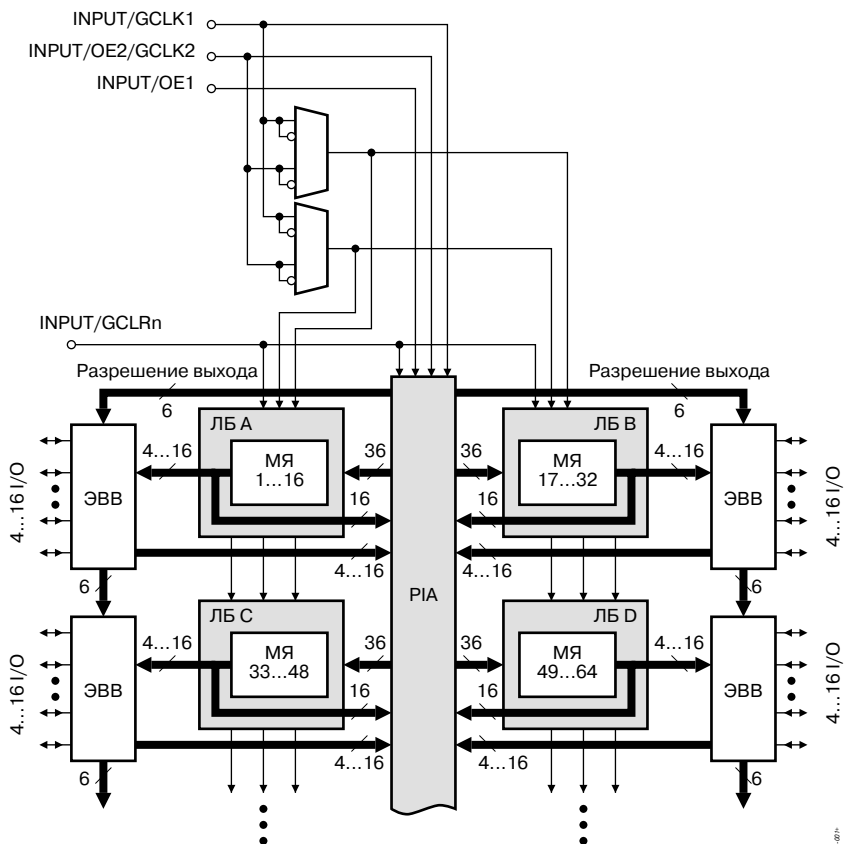


Рис.1.1. Функциональная схема ПЛИС семейства MAX3000

Как видно из **Рис. 1.1**, в основе архитектуры ПЛИС семейства MAX3000 лежат логические блоки, состоящие из 16 макроячеек каждый. Логические блоки соединяются с помощью программируемой матрицы соединений. Каждый логический блок имеет 36 входов с ПМС.

На **Рис. 1.2** приведена структурная схема макроячейки ПЛИС семейства MAX3000.

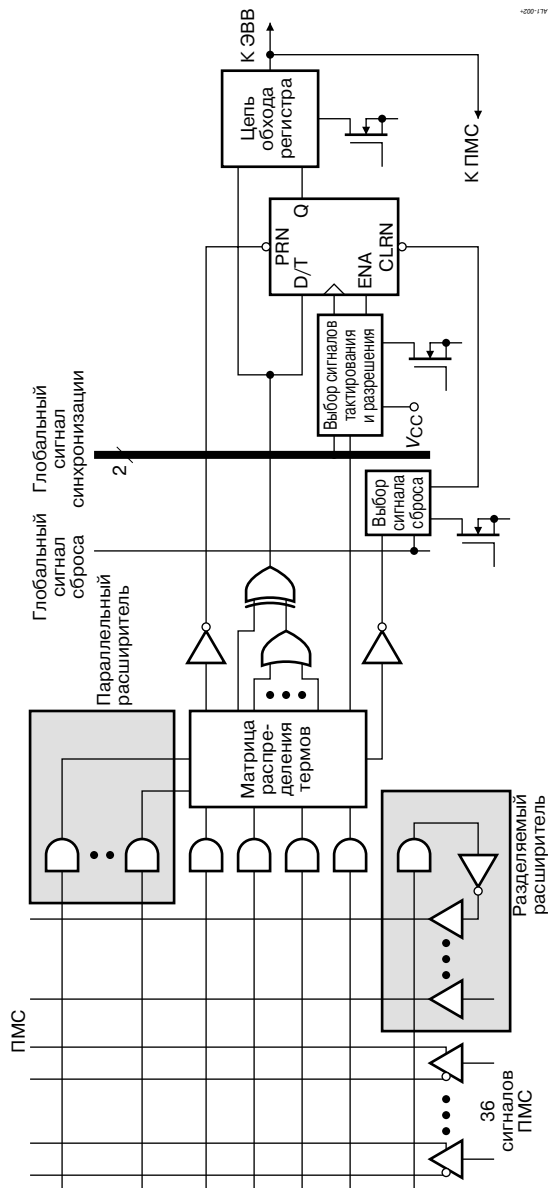


Рис. 1.2. Структурная схема макроячейки ПЛИС семейства MAX3000

МЯ ПЛИС семейства MAX3000 состоит из трех основных узлов:

- локальной программируемой матрицы (LAB local array);
- матрицы распределения термов (product-term select matrix);
- программируемого регистра (programmable register).

Комбинационные функции реализуются на локальной программируемой матрице и матрице распределения термов, позволяющей объединять логические произведения либо по «ИЛИ» (OR), либо по исключающему «ИЛИ» (XOR). Кроме того, матрица распределения термов позволяет скомутировать цепи управления триггером МЯ.

Режим тактирования и конфигурация триггера выбираются автоматически во время синтеза проекта в САПР MAX+PLUS II в зависимости от выбранного разработчиком типа триггера при описании проекта.

В ПЛИС семейства MAX3000 доступно 2 глобальных тактовых сигнала, что позволяет проектировать схемы с двухфазной синхронизацией.

Для реализации логических функций большого числа переменных используются логические расширители.

Разделяемый логический расширитель (**Рис. 1.3**) позволяет реализовать логическую функцию с большим числом входов, позволяя объединить МЯ, входящие в состав одного ЛБ. Таким образом, разделяемый расширитель формирует терм, инверсное значение которого передается матрицей распределения термов в локальную программируемую матрицу и может быть использовано в любой МЯ данного ЛБ. Как видно из **Рис. 1.3** имеются 36 сигналов локальной ПМС, а также 16 инверсных сигналов с разделяемых логических расширителей, что позволяет в пределах одного ЛБ реализовать функцию до 52 термов ранга 1.

Параллельный логический расширитель (**Рис. 1.4**) позволяет использовать локальные матрицы смежных МЯ для реализации функций, в которые входят более 5 термов. Одна цепочка параллельных расширителей может включать до 4 МЯ, реализуя функцию 20 термов. Компилятор системы MAX+PLUS II поддерживает размещение до 3 наборов не более чем по 5 параллельных расширителей.

На **Рис. 1.5** приведена структура программируемой матрицы соединений. На ПМС выводятся сигналы от всех возможных источников: ЭВВ, сигналов обратной связи ЛБ, специализированных выделенных выводов. В процессе программирования только необходимые сигналы «заводятся» на каждый ЛБ. На **Рис. 1.5** приведена структурная схема формирования сигналов ЛБ.

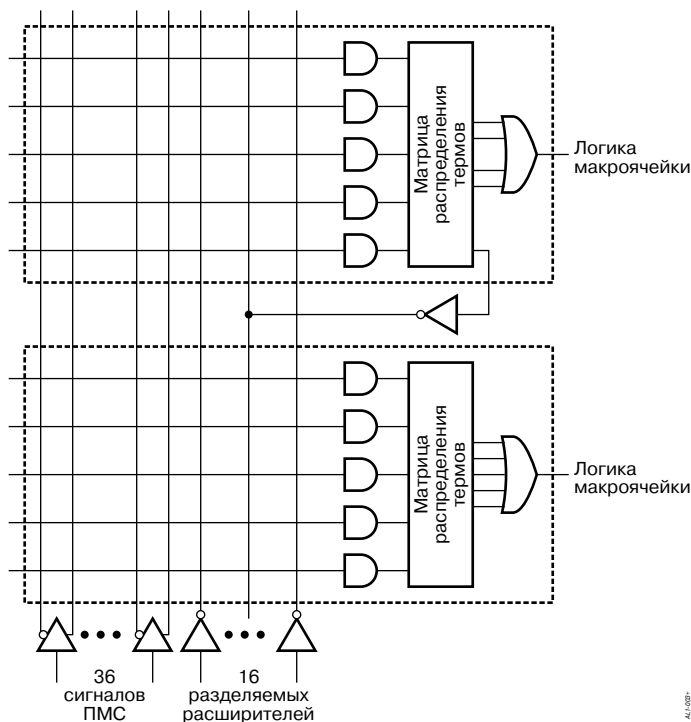


Рис. 1.3. Разделяемый логический расширитель

На **Рис. 1.6** приведена схема элемента ввода/вывода ПЛИС семейства MAX3000. ЭВВ позволяет организовать режимы работы с открытым коллектором и третьим состоянием.

ПЛИС семейства MAX3000 полностью поддерживают возможность программирования в системе в соответствии со стандартом IEEE Std. 1149.1 — 1990 (JTAG) с использованием соответствующих инструментальных средств. Повышенное напряжение программирования формируется специализированными схемами, входящими в состав ПЛИС семей-

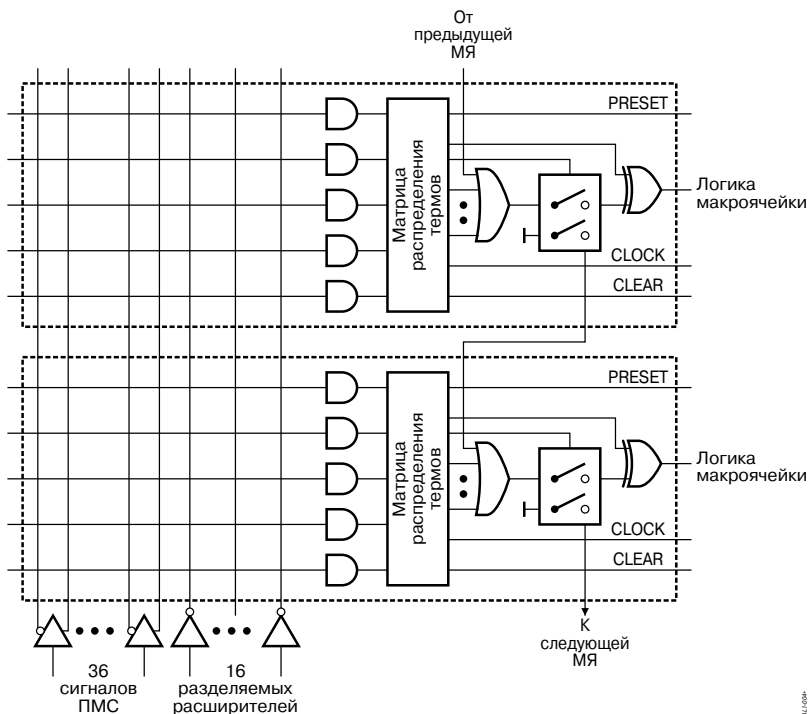


Рис. 1.4. Параллельный логический расширитель

ства MAX3000, из напряжения питания 3.3 В. Во время программирования в системе входы и выходы ПЛИС находятся в третьем состоянии и «слегка подтянуты» к напряжению питания. Сопротивления внутренних подстраивающих резисторов порядка 50 кОм. На **Рис. 1.7** приведены временные диаграммы программирования ПЛИС семейства MAX3000 через порт JTAG.

Значения временных параметров приведены в **Табл. 1.2**.

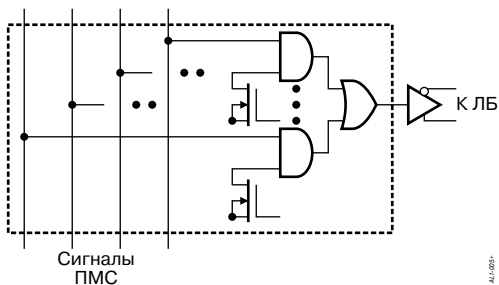


Рис. 1.5. Структура ПМС ПЛИС семейства MAX3000

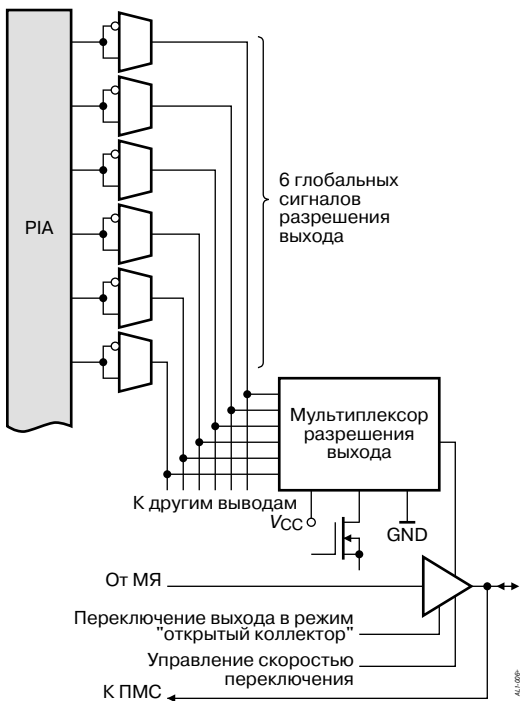


Рис. 1.6. Элемент ввода/вывода

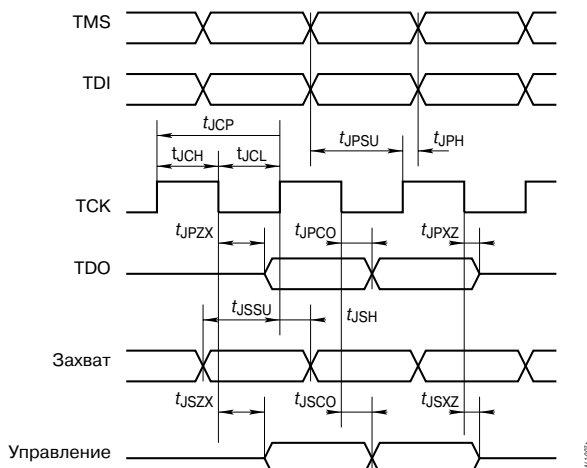


Рис. 1.7. Временные диаграммы программирования ПЛИС семейства MAX3000 через порт JTAG

Таблица 1.2. Временные параметры ПЛИС семейства MAX3000

Обозначение	Параметр	Значение	
		min	max
t_{JCP}	Период сигнала TCK [нс]	100	—
t_{JCH}	Длительность единичного уровня сигнала TCK [нс]	50	—
t_{JCL}	Длительность нулевого уровня сигнала TCK [нс]	50	—
t_{JPSU}	Время установления порта JTAG [нс]	20	—
t_{JPH}	Длительность сигнала JTAG	45	—
t_{JPCO}	Задержка распространения сигнала относительно такта JTAG [нс]	—	25
t_{JPZX}	Задержка перехода сигнала JTAG из третьего состояния [нс]	—	25
t_{JPXZ}	Задержка перехода сигнала JTAG в третье состояние [нс]	—	25
t_{JSSU}	Время установки регистра захвата [нс]	20	—
t_{JSH}	Длительность сигнала на входе регистра захвата [нс]	45	—
t_{JSCO}	Задержка обновления сигнала в регистре захвата относительно такта [нс]	—	25
t_{JSZX}	Задержка перехода сигнала регистра захвата из третьего состояния [нс]	—	25
t_{JSXZ}	Задержка перехода сигнала регистра захвата в третье состояние [нс]	—	25

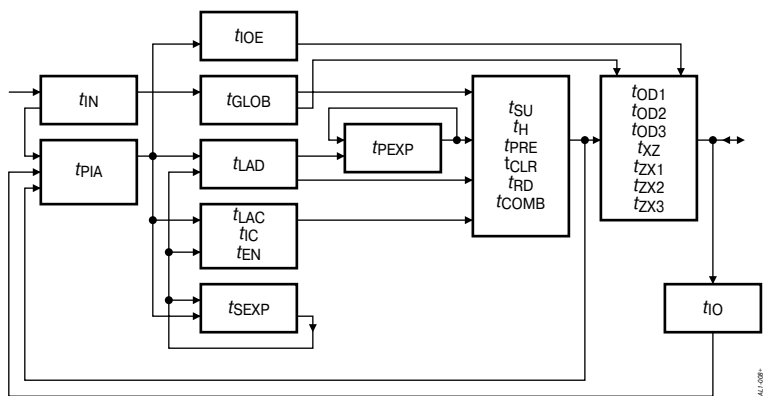


Рис. 1.8. Временная модель ПЛИС семейства MAX3000A

Временная модель ПЛИС семейства MAX3000A приведена на **Рис. 1.8**. ПЛИС семейства MAX3000A имеют предсказуемую задержку распространения сигнала, поэтому результаты временного моделирования в САПР MAX+PLUS II полностью адекватны поведению реальной схемы, в отличие от ПЛИС, выполненных по SRAM технологии. В **Табл. 1.3** приведено описание параметров временной модели для ПЛИС семейства MAX3000A с быстродействием –4 и –10.

Таблица 1.3. Параметры временной модели для ПЛИС семейства MAX3000A (все значения задержки в нс)

Обозначение	Параметр	Значение			
		–4		–10	
		min	max	min	max
t_{IN}	Задержка на входе и входном буфере	—	0.3	—	0.6
t_{IO}	Задержка на двунаправленном выводе и входном буфере	—	0.3	—	0.6
t_{SEXP}	Задержка разделяемого расширителя	—	1.9	—	4.9
t_{PEXP}	Задержка параллельного расширителя	—	0.5	—	1.1
t_{LAD}	Задержка в локальной программируемой матрице «И»	—	1.9	—	5.0
t_{LAC}	Задержка управляющего сигнала триггера в локальной программируемой матрице «И»	—	1.8	—	4.6
t_{IOE}	Внутренняя задержка сигнала разрешения	—	0.0	—	0.0

Таблица 1.3 (окончание)

Обозначение	Параметр	Значение			
		–4		–10	
		min	max	min	max
t_{OD1}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	0.3	—	0.7
t_{OD2}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	0.8	—	1.2
t_{OD3}	Задержка сигнала от выходного буфера до вывода, $\text{slew rate} = \text{on}$	—	5.3	—	5.7
t_{ZX1}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	4.0	—	5.0
t_{ZX2}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	4.5	—	5.5
t_{ZX3}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $\text{slew rate} = \text{on}$	—	9.0	—	10.0
t_{XZ}	Задержка сигнала в выходном буфере после сигнала запрещения выхода	—	4.0	—	5.0
t_{SU}	Время установки регистра	1.4	—	1.7	—
t_H	Время удержания сигнала на регистре	0.8	—	3.8	—
t_{RD}	Регистровая задержка	—	1.2	—	2.8
t_{COMB}	Комбинационная задержка	—	1.3	—	2.0
t_{IC}	Задержка изменения сигнала относительно тактового импульса	—	1.9	—	4.6
t_{EN}	Задержка разрешения регистра	—	1.8	—	4.6
t_{GLOB}	Задержка глобальных управляющих сигналов	—	1.0	—	1.8
t_{PRE}	Время предустановки регистра МЯ	—	2.3	—	5.2
t_{CLR}	Время сброса регистра МЯ	—	2.3	—	5.2
t_{PIA}	Задержка ПМС	—	0.7	—	1.7
t_{LPA}	Задержка за счет режима пониженного потребления	—	12	—	10.0

Таблица 1.4. Динамические параметры ПЛИС семейства MAX3000А
(временные параметры в нс, частоты в МГц)

Обозначение	Параметр	Значение			
		–4		–10	
		min	max	min	max
t_{PD1}	Задержка вход — комбинаторный выход	—	4.5	—	10.0
t_{PD2}	Задержка вход — регистровый выход	—	4.5	—	10.0
t_{SU}	Время установки глобального синхросигнала	3.0	—	6.6	—
t_H	Время удержания глобального синхросигнала	0.0	—	0.0	—
t_{CO1}	Задержка глобального синхросигнала до выхода	1.0	2.8	1.0	5.9
t_{CH}	Длительность высокого уровня глобального синхросигнала	2.0	—	4.0	—
t_{CL}	Длительность низкого уровня глобального синхросигнала	2.0	—	4.0	—
t_{ASU}	Время установки синхросигнала триггера МЯ	1.4	—	2.1	—
t_{AH}	Время удержания синхросигнала триггера МЯ	0.8	—	3.4	—
t_{ACO1}	Задержка синхросигнала триггера МЯ до выхода	—	4.4	—	10.4
t_{ACH}	Длительность высокого уровня синхросигнала триггера МЯ	2.0	—	4.0	—
t_{ACL}	Длительность низкого уровня синхросигнала триггера МЯ	2.0	—	4.0	—
t_{CPW}	Минимальная длительность сигналов сброса и установки триггера МЯ	2.0	—	2.0	—
t_{CNT}	Минимальный период глобального синхросигнала	—	5.2	—	11.2
f_{CNT}	Максимальная глобальная внутренняя тактовая частота	192.3	—	89.3	—
t_{ACNT}	Минимальный период синхросигнала триггера МЯ	—	5.2	—	11.2
f_{ACNT}	Максимальная внутренняя тактовая частота триггера МЯ	192.3	—	89.3	—
f_{MAX}	Максимальная тактовая частота	250	—	125.0	—

В **Табл. 1.4** приведены динамические параметры ПЛИС семейства MAX3000A. **Рис. 1.9** и **1.10** иллюстрируют задержки сигналов в ПЛИС семейства MAX3000A в зависимости от режима работы ПЛИС. На **Рис. 1.10** и **1.11** длительности переднего и заднего фронтов t_R и t_F соответственно равны 2 нс.

Таким образом, мы рассмотрели основные архитектурные особенности и принципы построения ПЛИС семейства MAX3000A. Следует еще раз заметить, что в книге намеренно не приводится информация о назначении контактов для различных корпусов, потребляемой мощности и т.д. Это связано с тем, что данная информация легкодоступна как на CD «Altera Digital Library», так и в Internet.

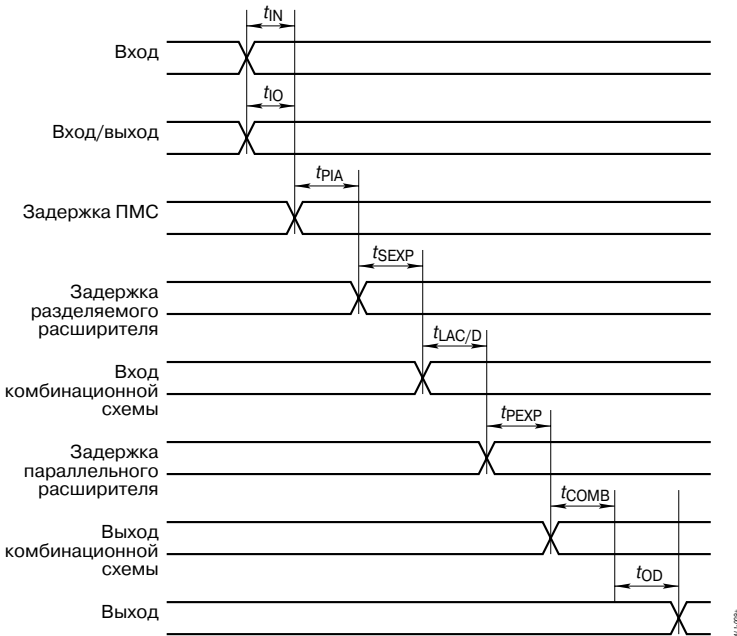


Рис. 1.9. Задержки в ПЛИС семейства MAX3000A

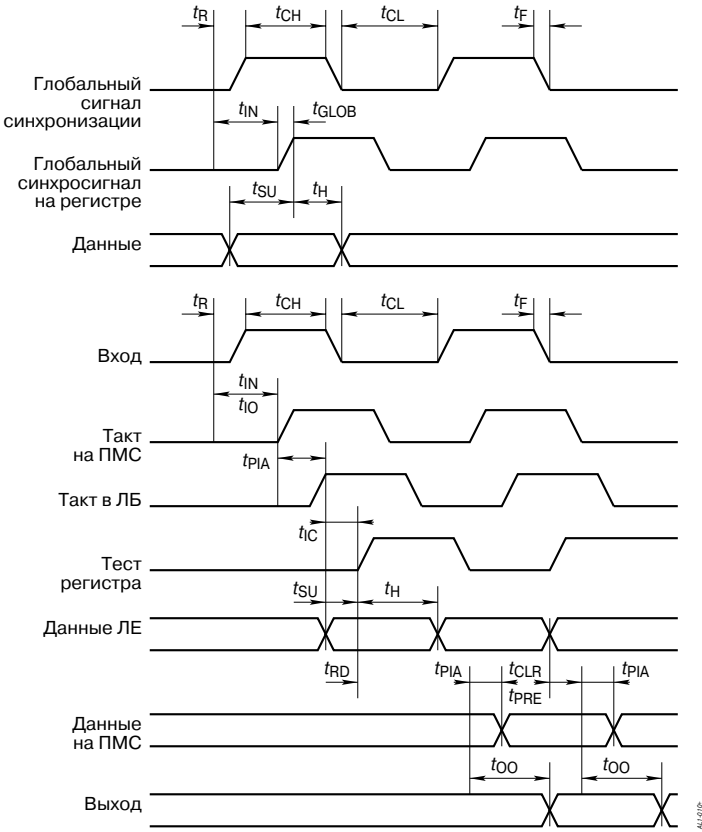


Рис. 1.10. Задержки в ПЛИС семейства MAX3000A

1.2. Семейство FLEX6000

Относительно новое семейство ПЛИС FLEX6000 появилось на рынке в конце 1997 года. По своим характеристикам оно является промежуточным между семействами FLEX8000 и FLEX10K. ПЛИС семейства FLEX6000 выпускаются по технологии 0.5 мкм SRAM (FLEX6000A по 0.35 мкм) с тремя слоями металлизации и обладают удачными характеристиками цена—производительность для реализации не очень сложных алгоритмов ЦОС. В **Табл. 1.5** приведены основные характеристики ПЛИС семейства FLEX6000.

Таблица 1.5. Основные характеристики ПЛИС семейства FLEX6000

Параметр	EPF6010	EPF6016	EPF6016A	EPF6024A
Логическая емкость, количество эквивалентных вентилей	10 000	16 000	16 000	24 000
Число логических элементов	800	1 320	1 320	1 960
Число логических блоков	80	132	132	196
Число программируемых пользователем выводов	160	204	204	215

Отличительной особенностью архитектуры ПЛИС семейства FLEX6000 является технология OptiFLEX, представленная на **Рис. 1.11**. В основе архитектуры OptiFLEX лежат логические блоки, каждый из которых объединяет по 10 логических элементов (Logic elements, LEs) с помощью локальной матрицы соединений. Особенностью архитектуры OptiFLEX является то, что каждый логический элемент может коммутироваться как на локальную матрицу соединений собственного логического блока, так и смежных (**Рис. 1.11**), тем самым расширяются возможности для трассировки.

На **Рис. 1.12** приведена структура ЛБ ПЛИС семейства FLEX6000. Как видно из **Рис. 1.12**, ЛБ имеет чередующуюся структуру (interleaved structure), объединяя на локальной матрице соединений (ЛМС, local interconnect) сигналы с двух смежных ЛБ. Кроме того, сигналы с ЛЭ и ЛМС

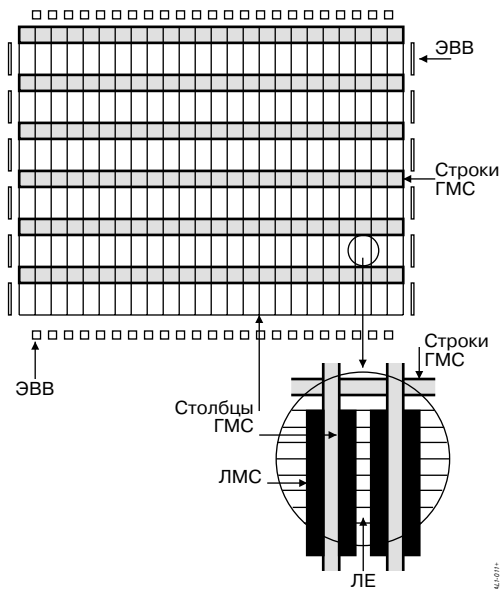


Рис. 1.11. Технология OptiFLEX

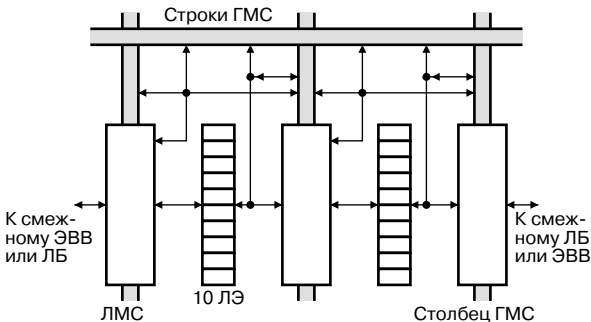


Рис. 1.12. Структура ЛБ FLEX6000

могут коммутироваться на строки и столбцы глобальной матрицы соединений (row and column interconnect), которые имеют непрерывную структуру, обеспечивающую минимальные задержки.

Каждый ЛБ и ЛЭ управляется выделенными глобальными сигналами (dedicated inputs), являющимися сигналами сброса, установки и синхронизации триггеров ЛЭ (**Рис. 1.13**).

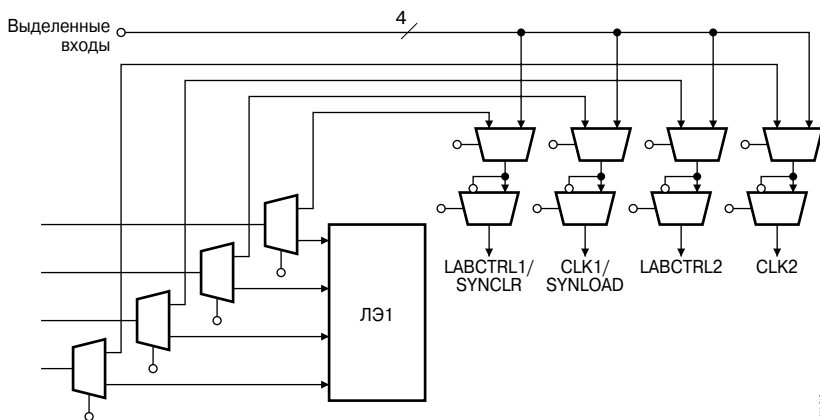


Рис. 1.13. Специализированные сигналы

На **Рис. 1.14** приведена структура ЛЭ ПЛИС семейства FLEX6000. В основе ЛЭ лежит четырехвходовая таблица перекодировок (ТП). Кроме того, в состав ЛЭ входят цепи ускоренного цепочечного переноса (carry-in, carry-out) и каскадирования (cascade-in, cascade-out). Триггер ЛЭ может быть сконфигурирован с помощью логики сброса-установки (clear/preset logic), тактируется одним из сигналов, выбираемых логикой тактирования (clock select). При необходимости сигнал с выхода ТП может быть подан на выход ЛЭ в обход триггера (register bypass).

Для обеспечения минимальной задержки при реализации сложных арифметических функций в таких устройствах, как счетчики, сумматоры, вычитатели и т.п., используется организация ускоренных цепочечных переносов (carry chain) между ЛЭ. Логика ускоренных переносов автоматически формируется компилятором САПР MAX+PLUS II или вручную при описании проекта.

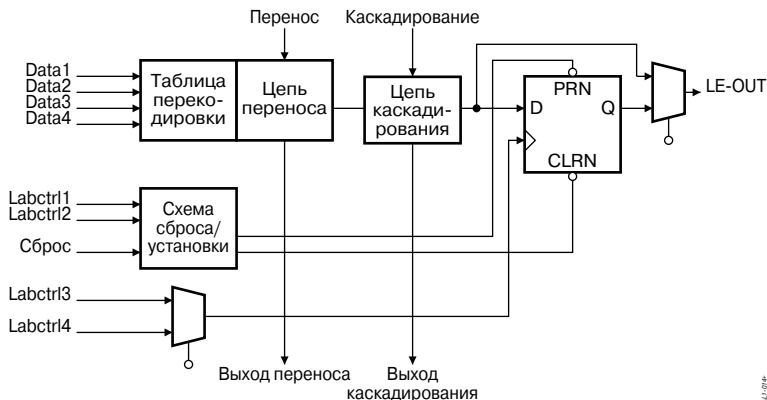


Рис. 1.14. Структура ЛЭ ПЛИС семейства FLEX6000

При организации цепочечных переносов первый ЛЭ каждого ЛБ не включается в цепочку цепочечных переносов, поскольку он формирует управляющие сигналы ЛБ. Вход первого ЛЭ в каждом ЛБ может быть использован для формирования сигналов синхронной загрузки или сброса счетчиков, использующих цепочечный перенос.

Цепочка переносов длиннее чем 9 ЛЭ автоматически формируется путем объединения нескольких ЛБ вместе, причем перенос формируется не в соседний ЛБ, а через один, т.е. из четного в четный, из нечетного ЛБ в нечетный. Например, последний ЛЭ в первом ЛБ в ряду формирует перенос во второй ЛЭ в третьем ЛБ в том же ряду. Отсюда ясно, что длина цепочки переносов не может быть больше, чем половина ряда.

На **Рис. 1.15** приведен пример реализации полного сумматора с использованием логики ускоренного переноса. В этом случае ТП сконфигурирован таким образом, что два ее входа формируют сигнал суммы, а два других входа — перенос.

При реализации многовходовых функций используется режим каскадирования ЛЭ (**Рис. 1.16**). ТП смежных ЛЭ реализуют частичные функции, а затем с помощью цепей каскадирования формируется выход функции многих переменных. Логика каскадирования строится либо по «И» (AND), либо по «ИЛИ» (OR).

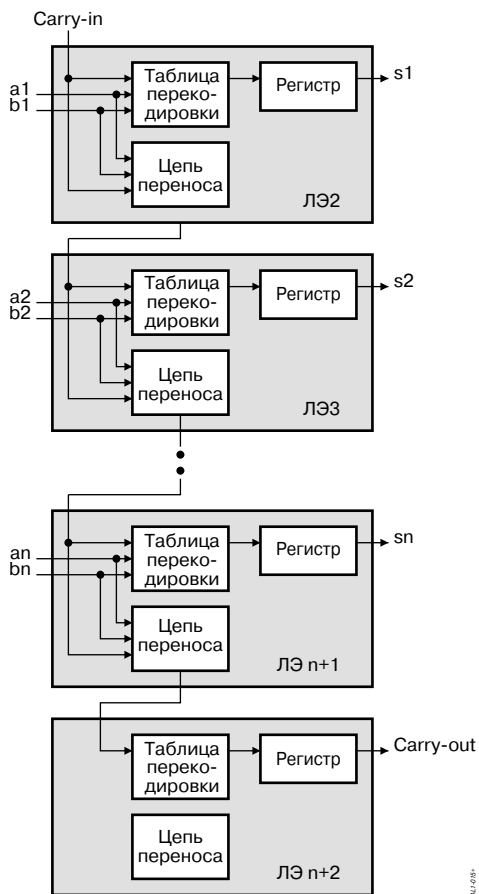


Рис. 1.15. Цепочечный перенос в сумматоре

При каскадировании по «И» возможно использование регистра последнего ЛЭ, при каскадировании по «ИЛИ» использование регистра невозможно, поскольку инвертор используется для реализации элемента «ИЛИ». Аналогично цепочечным переносам при каскадировании объединяются либо только четные, либо нечетные ЛЭ.

Рис. 1.16 иллюстрирует реализацию каскадирования для функции большого числа переменных. Так, при реализации 16-разрядного дешифратора адреса задержка составляет порядка 3.5 нс.

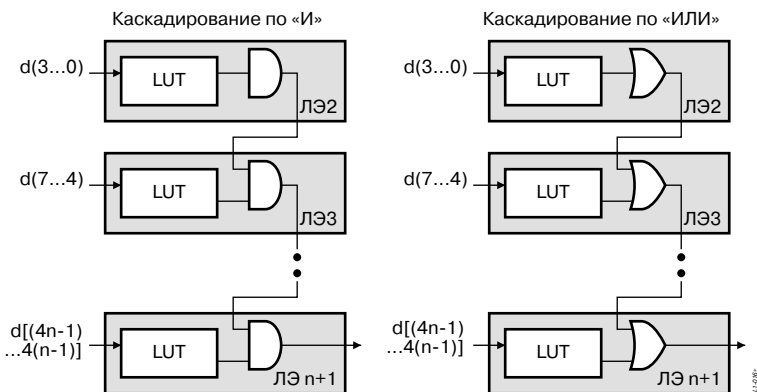


Рис. 1.16. Каскадирование ЛЭ

Каждый ЛЭ ПЛИС может быть сконфигурирован тремя способами (Рис. 1.17):

- нормальный режим (normal mode);
- арифметический режим (arithmetic mode);
- режим счетчика (counter mode).

Нормальный режим используется для реализации основных логических функций, комбинационных схем, дешифраторов с большим числом входов, когда возможность каскадного наращивания позволяет получить выигрыш во времени. В нормальном режиме ТП имеет четыре входа, источниками которых являются сигналы с ЛМС и цепочечные переносы.

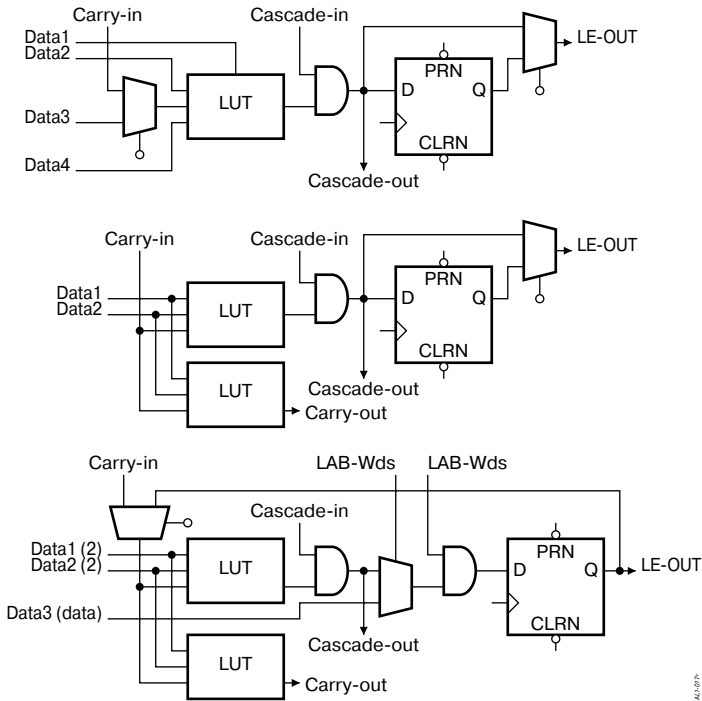


Рис. 1.17. Режимы конфигурации ЛЭ

Арифметический режим используется для реализации сумматоров, вычитателей, накопителей и компараторов. В арифметическом режиме ТП конфигурируется как две трехходовые ТП: одна для функции трех переменных, другая для сигнала ускоренного переноса.

В режиме счетчика возможна реализация с помощью ТП функций разрешения счета, реверса, синхронных сброса и загрузки данных в счетчик. Кроме того, формируется ускоренный перенос для реализации синхронных счетчиков с помощью двух трехходовых ТП, подобно арифметическому режиму.

Каждый ЛЭ имеет возможность глобальной асинхронной установки и сброса триггера, а также эмуляции внутренней шины с тремя состояниями.



Рис. 1.18. Коммутация ЛЭ на матрицы соединений

На **Рис. 1.18** приведена схема коммутации ЛБ и ЛЭ на локальную и глобальную матрицу соединений (ГМС). Следует отметить, что ГМС имеет непрерывную структуру как по строкам, так и по столбцам (т.н. FastTrack Interconnect). Как можно видеть из **Рис. 1.18**, ЛЭ имеют возможность коммутации входов и выходов как на ЛМС, так и на ГМС. Кроме того, на ЛЭ могут быть сформированы глобальные управляющие сигналы, такие, как внутренняя тактовая частота, сигналы асинхронного сброса и установки. Каждый ЛБ коммутируется на две ЛМС, тем самым улучшая возможности трассировки ПЛИС.

На **Рис. 1.19** приведена структурная схема элемента ввода/вывода. Как можно заметить, ЭВВ позволяет скомутировать данные как на глобальные цепи, так и на локальную матрицу соединений. Управление ЭВВ осуществляется с помощью глобального управляющего сигнала разрешения выхода (chip-wide output enable). Кроме того, можно задать режим пониженной скорости переключения ЭВВ, что позволяет снизить «звон», возникающий при высокой скорости переключения, правда, ценой 5 нс задержки.

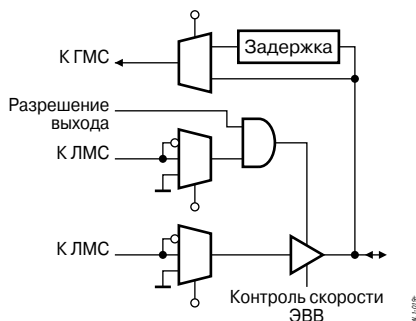


Рис. 1.19. Элемент ввода/вывода

ПЛИС семейства FLEX6000 поддерживают возможность конфигурации через порт JTAG, временные диаграммы приведены на **Рис. 1.20**. Временные параметры конфигурации по порту JTAG приведены в **Табл. 1.6**.

На **Рис. 1.21** приведена временная модель семейства FLEX6000, а в **Табл. 1.7** значения ее параметров.

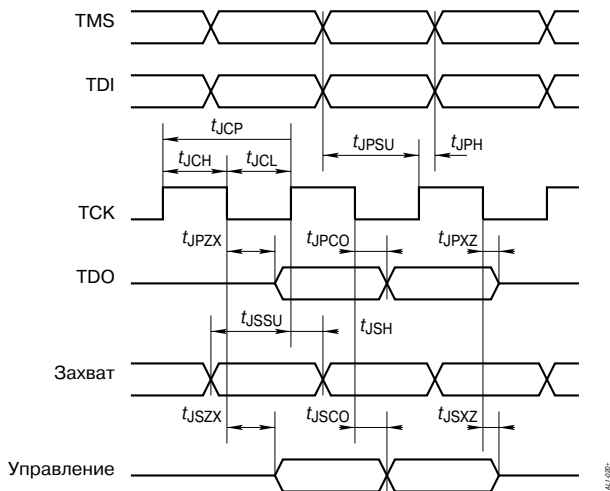


Рис. 1.20. Временные диаграммы конфигурации через порт JTAG

Таблица 1.6. Временные параметры конфигурации по порту JTAG

Обозначение	Параметр	Значение	
		min	max
t_{JCP}	Период сигнала TCK [нс]	100	—
t_{JCH}	Длительность единичного уровня сигнала TCK [нс]	50	—
t_{JCL}	Длительность нулевого уровня сигнала TCK [нс]	50	—
t_{JPSU}	Время установления порта JTAG [нс]	20	—
t_{JPH}	Длительность сигнала JTAG	45	—
t_{JPCO}	Задержка распространения сигнала относительно такта JTAG [нс]	—	25
t_{JPZX}	Задержка перехода сигнала JTAG из третьего состояния [нс]	—	25
t_{JPXZ}	Задержка перехода сигнала JTAG в третье состояние [нс]	—	25
t_{JSSU}	Время установки регистра захвата [нс]	20	—
t_{JSH}	Длительность сигнала на входе регистра захвата [нс]	45	—
t_{JSCO}	Задержка обновления сигнала в регистре захвата относительно такта [нс]	—	35
t_{JSZX}	Задержка перехода сигнала регистра захвата из третьего состояния [нс]	—	35

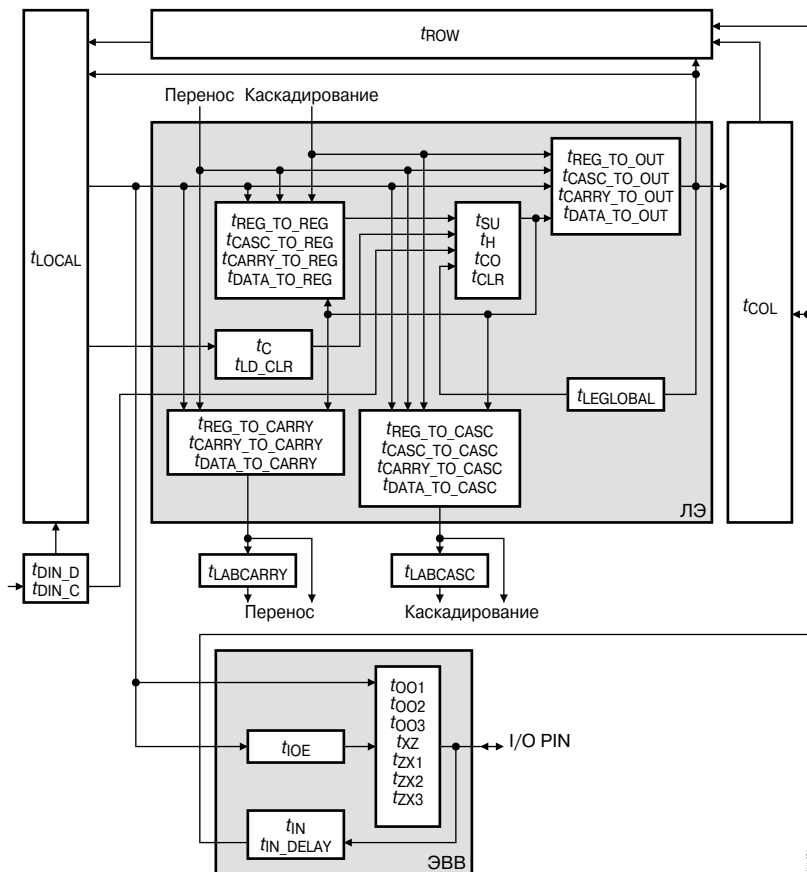


Рис. 1.21. Временная модель ПЛИС семейства FLEX6000

Таблица 1.7. Параметры временной модели

Обозначение	Параметр	Значение для ERF6010A-1 [нс]	
		min	max
$t_{\text{REG_TO_REG}}$	Задержка на ТП для обратной связи триггера ЛЭ в цепочке переноса	—	1.2
$t_{\text{CASC_TO_REG}}$	Задержка в цепи каскадирования до входа регистра	—	0.9
$t_{\text{CARRY_TO_REG}}$	Задержка в цепи переноса до входа регистра	—	0.9
$t_{\text{DATA_TO_REG}}$	Задержка входных данных ЛЭ до регистра	—	1.1
$t_{\text{CASC_TO_OUT}}$	Задержка от входа каскадирования до выхода ЛЭ	—	1.3
$t_{\text{CARRY_TO_OUT}}$	Задержка от входа переноса до выхода ЛЭ	—	1.6
$t_{\text{DATA_TO_OUT}}$	Задержка входных данных ЛЭ до выхода	—	1.7
$t_{\text{REG_TO_OUT}}$	Задержка данных с выхода регистра до выхода ЛЭ	—	0.4
t_{SU}	Время установки регистра	0.9	—
t_{H}	Время удержания сигнала на входе регистра после подачи синхроимпульса	1.4	—
t_{CO}	Задержка выходного сигнала регистра относительно такта	—	0.3
t_{CLR}	Задержка сброса регистра	—	0.4
t_{C}	Задержка управляющего сигнала на регистре	—	1.8
$t_{\text{LD_CLR}}$	Задержка сигнала синхронного сброса или загрузки регистра в режиме счетчика	—	1.8
$t_{\text{CARRY_TO_CARRY}}$	Задержка сигнала переноса от входа переноса до выхода переноса	—	0.1
$t_{\text{REG_TO_CARRY}}$	Задержка выходного сигнала регистра до выхода переноса	—	1.6
$t_{\text{DATA_TO_CARRY}}$	Задержка входных данных ЛЭ до выхода переноса	—	2.1
$t_{\text{CARRY_TO_CASC}}$	Задержка сигнала переноса от входа до выхода каскадирования	—	1.0
$t_{\text{REG_TO_CASC}}$	Задержка выходного сигнала регистра до выхода каскадирования	—	1.4
$t_{\text{CASC_TO_CASC}}$	Задержка сигнала каскадирования от входа каскадирования до выхода каскадирования	—	0.5
$t_{\text{DATA_TO_CASC}}$	Задержка входных данных ЛЭ до выхода каскадирования	—	1.1
t_{CH}	Длительность высокого уровня тактового сигнала регистра	2.5	—
t_{CL}	Длительность низкого уровня тактового сигнала регистра	2.5	—
t_{OD1}	Задержка сигнала от выходного буфера до вывода, $V_{\text{CCIO}} = 3.3 \text{ В}$, slew rate = off	—	1.9
t_{OD2}	Задержка сигнала от выходного буфера до вывода, $V_{\text{CCIO}} = 2.5 \text{ В}$, slew rate = off	—	4.1
t_{OD3}	Задержка сигнала от выходного буфера до вывода, slew rate = on	—	5.8

Таблица 1.7 (окончание)

Обозначение	Параметр	Значение для EPF6010A-1 [нс]	
		min	max
t_{XZ}	Задержка сигнала в выходном буфере после сигнала запрещения выхода	—	1.4
t_{ZX1}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 3.3$ В, $\text{slew rate} = \text{off}$	—	1.4
t_{ZX2}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 2.5$ В, $\text{slew rate} = \text{off}$	—	3.6
t_{ZX3}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $\text{slew rate} = \text{on}$	—	5.3
t_{IOE}	Задержка сигнала разрешения выхода	—	0.5
t_{IN}	Задержка сигнала во входном буфере	—	3.6
t_{IN_DELAY}	Задержка сигнала во входном буфере при введении дополнительной задержки	—	4.8
t_{LOCAL}	Задержка ЛМС	—	0.7
t_{ROW}	Задержка в строке ГМС	—	2.9
t_{COL}	Задержка в столбце ГМС	—	1.2
t_{DIN_D}	Задержка данных с выделенного вывода до входа ЛЭ	—	5.4
t_{DIN_C}	Задержка управляющих сигналов с выделенного вывода до входа ЛЭ	—	4.3
$t_{LEGLOBAL}$	Задержка сформированного в ПЛИС управляющего сигнала с выхода ЛЭ до входов других ЛЭ	—	2.6
$t_{LABCARRY}$	Задержка сигнала переноса в другой ЛБ	—	0.7
$t_{LABCASC}$	Задержка сигнала каскадирования в другой ЛБ	—	1.3
t_1	Тестовая задержка регистр-регистр	—	37.6
t_{DPR}	Тестовая задержка регистр-регистр через 4 ЛЭ, 3 ряда и 4 ЛМС	—	—
t_{INSU}	Время установки глобального синхросигнала на регистре ЛЭ	—	2.1
t_{INH}	Время удержания данных для глобального синхросигнала на регистре ЛЭ	—	0.2
t_{OUTCO}	Задержка появления данных на выходе для глобального	—	2.0

Величины времен задержек распространения сигнала по ГМС приводятся из расчета «худшего случая».

1.3. Семейство MAX7000

ПЛИС семейства MAX7000 являются первыми CPLD фирмы «Altera», выполненными по технологии ПЗУ с электрическим стиранием (EEPROM). В настоящее время выпускаются ПЛИС семейств MAX7000, MAX7000A, MAX7000B, MAX7000E, MAX7000S. Семейства MAX7000A и MAX7000B рассчитаны на работу в системах с напряжением питания 3.3 и 2.5 В соответственно, ПЛИС семейства MAX7000S является дальнейшим развитием 5-вольтового семейства MAX7000, допуская возможность программирования в системе. В настоящее время это семейство, пожалуй, является самым популярным CPLD у российских разработчиков. Это связано с тем, что для работы с ПЛИС этого семейства не требуется никаких серьезных затрат, поскольку пакет MAX+PLUS II BASELINE полностью поддерживает всех представителей этого семейства ПЛИС, а для программирования и загрузки конфигурации устройств опубликована схема загрузочного кабеля ByteBlaster и ByteBlasterMV. В **Табл. 1.8** приведены основные характеристики ПЛИС семейства MAX7000S.

Таблица 1.8. Основные характеристики ПЛИС семейства MAX7000S

Параметр	ЕРМ7032S	ЕРМ7064S	ЕРМ7128S	ЕРМ7160S	ЕРМ7192S	ЕРМ7256S
Логическая емкость, количество эквивалентных вентилях	600	1 250	2 500	3 200	3 750	5 000
Число макроячеек	32	64	128	160	192	256
Число логических блоков	2	4	8	10	12	16
Число программируемых пользователем выводов	36	68	100	104	124	164
Задержка распространения сигнала вход/выход, t_{PD} [нс]	5	5	6	6	7.5	7.5
Время установки глобального тактового сигнала, t_{SU} [нс]	2.9	2.9	3.4	3.4	4.1	3.9
Задержка глобального тактового сигнала до выхода, t_{CO1} [нс]	2.5	2.5	2.5	2.5	3.0	3.0
Максимальная глобальная тактовая частота, f_{CNT} [МГц]	175.4	175.4	147.1	149.3	125.0	128.2

Все ПЛИС семейства MAX7000S поддерживают технологию программирования в системе ISP и периферийного сканирования в соответствии со стандартом IEEE Std. 1149.1 — 1990 (JTAG). Элементы ввода/вывода позволяют работать в системах с уровнями сигналов 5 или 3.3 В. Матрица соединений имеет непрерывную структуру, что позволяет реализовать время задержки распространения сигнала до 5 нс. ПЛИС семейства MAX7000S имеют возможность аппаратной эмуляции выходов с открытым коллектором и удовлетворяют требованиям стандарта PCI. Имеется возможность индивидуального программирования цепей сброса, установки и тактирования триггеров, входящих в макроячейку. Предусмотрен режим пониженного энергопотребления. Программируемый логический расширитель позволяет реализовать на одной макроячейке функции до 32 переменных. Имеется возможность задания бита секретности для защиты от несанкционированного тиражирования разработки.

В отличие от архитектуры ПЛИС семейства MAX7000 (**Рис. 1.22**), ПЛИС семейства MAX7000S (**Рис. 1.23**) имеют возможность использования двух глобальных тактовых сигналов. На **Рис. 1.24** приведена структура макроячейки логического элемента ПЛИС MAX7000S. Как можно заметить, МЯ ПЛИС семейства MAX7000 не отличается от МЯ семейства MAX3000.

Аналогично ПЛИС семейства MAX3000, ПЛИС семейства MAX7000 имеют возможность использования параллельного и разделяемого расширителей, которые подробно описаны в параграфе 1.1. На **Рис. 1.25** приведена временная модель ПЛИС семейства MAX7000, а в **Табл. 1.9** и **1.10** ее параметры.

Таблица 1.9. Параметры временной модели ПЛИС семейства MAX7000
(все значения задержек и времен в нс)

Обозначение	Параметр	Значение для EPM70128S-10 [нс]	
		min	max
t_{IN}	Задержка на входе и входном буфере	—	2.0
t_{IO}	Задержка на двунаправленном выводе и входном буфере	—	2.0
t_{SEXP}	Задержка разделяемого расширителя	—	8.0
t_{PEXP}	Задержка параллельного расширителя	—	1.0
t_{LAD}	Задержка в локальной программируемой матрице «И»	—	6.0

Таблица 1.9 (окончание)

Обозначение	Параметр	Значение для EPM70128S -10	
		min	max
t_{LAC}	Задержка управляющего сигнала триггера в локальной программируемой матрице «И»	—	6.0
t_{IOE}	Внутренняя задержка сигнала разрешения	—	3.0
t_{OD1}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	4.0
t_{OD2}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	5.0
t_{OD3}	Задержка сигнала от выходного буфера до вывода, $\text{slew rate} = \text{on}$	—	8.0
t_{ZX1}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	6.0
t_{ZX2}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	7.0
t_{ZX3}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $\text{slew rate} = \text{on}$	—	8.0
t_{XZ}	Задержка сигнала в выходном буфере после сигнала запрещения выхода	—	6.0
t_{SU}	Время установки регистра	4.0	—
t_H	Время удержания сигнала на регистре	4.0	—
t_{RD}	Регистровая задержка	—	1.0
t_{COMB}	Комбинационная задержка	—	1.0
t_{IC}	Задержка изменения сигнала относительно тактового импульса	—	6.0
t_{GLOB}	Задержка разрешения регистра	—	6.0
t_{PRE}	Задержка глобальных управляющих сигналов	—	1.0
t_{PRE}	Время предустановки регистра МЯ	—	4.0
t_{CLR}	Время сброса регистра МЯ	—	4.0
t_{PIA}	Задержка ПМС	—	2.0
t_{LPA}	Задержка за счет режима пониженного потребления	—	13.0

Таблица 1.10. Параметры временной модели ПЛИС семейства MAX7000
(временные параметры в нс, частоты в МГц)

Обозначение	Параметр	Значение для EPM70128S-10	
		min	max
t_{PD1}	Задержка вход — комбинаторный выход	—	10.0
t_{PD2}	Задержка вход — регистровый выход	—	10.0
t_{SU}	Время установки глобального синхросигнала	7.0	—
t_H	Время удержания глобального синхросигнала	0.0	—
t_{CO1}	Задержка глобального синхросигнала до выхода	—	4.5
t_{CH}	Длительность высокого уровня глобального синхросигнала	4.0	—
t_{CL}	Длительность низкого уровня глобального синхросигнала	4.0	—
t_{ASU}	Время установки синхросигнала триггера МЯ	2.0	—
t_{AH}	Время удержания синхросигнала триггера МЯ	5.0	—
t_{ACO1}	Задержка синхросигнала триггера МЯ до выхода	—	10.0
t_{ACH}	Длительность высокого уровня синхросигнала триггера МЯ	4.0	—
t_{ACL}	Длительность низкого уровня синхросигнала триггера МЯ	4.0	—
t_{CPW}	Минимальная длительность сигналов сброса и установки триггера МЯ	4.0	—
t_{CNT}	Минимальный период глобального синхросигнала	—	10.0
f_{CNT}	Максимальная глобальная внутренняя тактовая частота	100	—
t_{ACNT}	Минимальный период синхросигнала триггера МЯ	—	10.0
f_{ACNT}	Максимальная внутренняя тактовая частота триггера МЯ	100	—
f_{MAX}	Максимальная тактовая частота	125.0	—

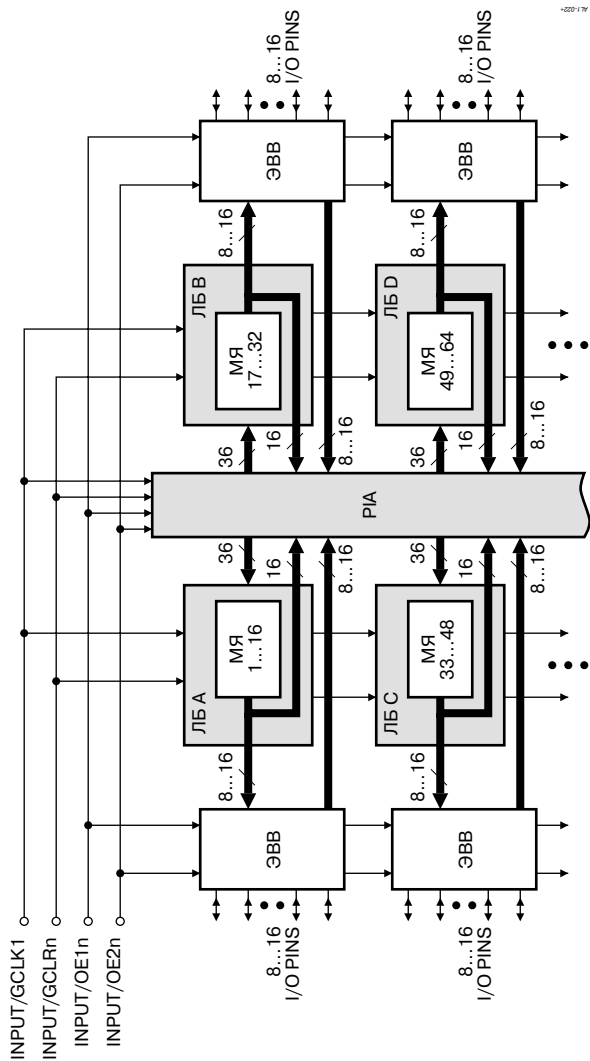


Рис. 1.22. Архитектура семейства MAX7000

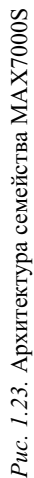




Рис. 1.24. Структура макроячейки семейства MAX7000

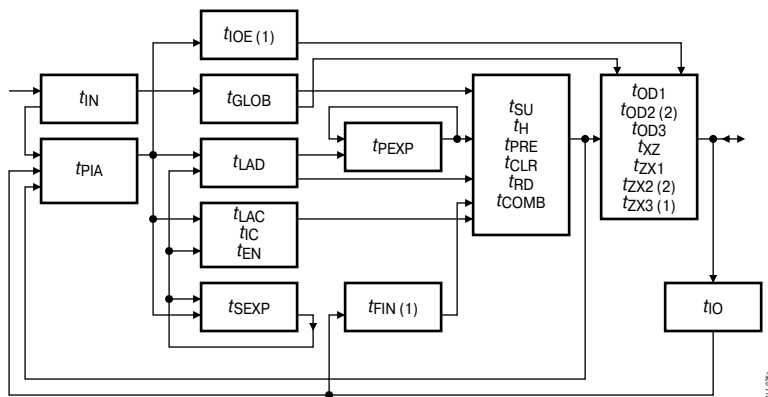


Рис. 1.25. Временная модель ПЛИС семейства MAX7000

1.4. Семейство FLEX8000

ПЛИС семейства FLEX8000 выпускаются по технологии 0.5 мкм SRAM с тремя слоями металлизации и пригодны для реализации не очень сложных алгоритмов ЦОС. В Табл. 1.11 приведены основные характеристики ПЛИС семейства FLEX8000. Данные ПЛИС обладают высокими характеристиками надежности, поэтому они достаточно часто выпускаются в индустриальном применении.

Таблица 1.11. Основные характеристики ПЛИС семейства FLEX8000

Параметр	EPF8282	EPF8452	EPF8636	EPF8820	EPF81188	EPF81500
Логическая емкость, количество эквивалентных вентилей	2 500	4 000	6 000	8 000	12 000	16 000
Число логических элементов	208	336	504	672	1 008	1 296
Число логических блоков	282	452	636	820	1 188	1 500
Число программируемых пользователем выводов	78	120	136	152	184	208

На **Рис. 1.26** приведена обобщенная функциональная схема ПЛИС семейства FLEX8000. Как можно заметить, в архитектуре ПЛИС семейства FLEX8000 много общего с рассмотренным в параграфе 1.2 семейством FLEX6000, однако поскольку семейство FLEX8000 было разработано значительно раньше, то имеется ряд отличий.

Логические блоки ПЛИС семейства FLEX8000 объединяют по 8 логических элементов на ЛМС, при этом в отличие от семейства FLEX6000 каждый ЛБ имеет возможность коммутации только на одну строку и столбец ГМС. Структура ЛБ ПЛИС семейства FLEX8000 приведена на **Рис. 1.27**.

Каждый ЛЭ, входящий в ЛБ, имеет возможность коммутации как на ЛМС, так и на строки и столбцы ГМС. На ЛМС поступает 24 входных сигнала со строки ГМС, а также 8 сигналов обратной связи. Управляющие сигналы формируются либо из глобальных выделенных управляющих сигналов, либо из сигналов ЛМС. Структура ЛЭ в целом практически подобна ЛЭ ПЛИС семейства FLEX6000.

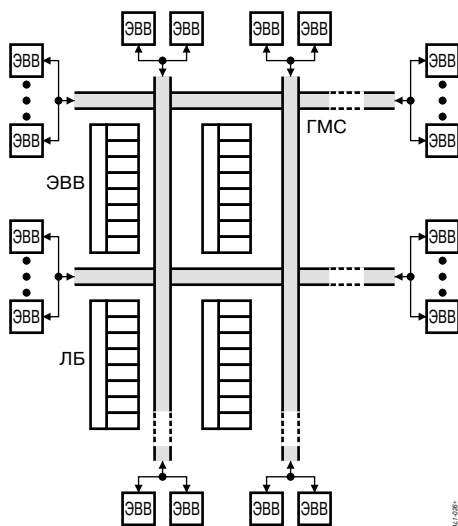


Рис. 1.26. Укрупненная структурная схема ПЛИС семейства FLEX8000

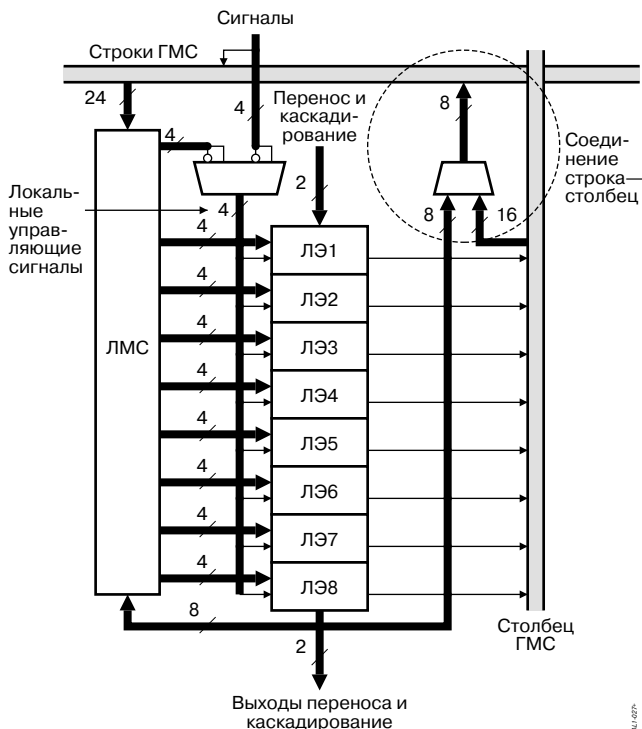


Рис. 1.27. ЛБ ПЛИС семейства FLEX8000

ЛЭ ПЛИС семейства FLEX8000 допускают каскадирование, а также цепочные переносы, имеют возможность конфигурации в нормальном, счетном и арифметическом режимах.

Временная модель ПЛИС FLEX8000 приведена на **Рис 1.28** и **1.29**. Основные параметры временной модели ПЛИС семейства FLEX8000 приведены в **Табл. 1.12**.

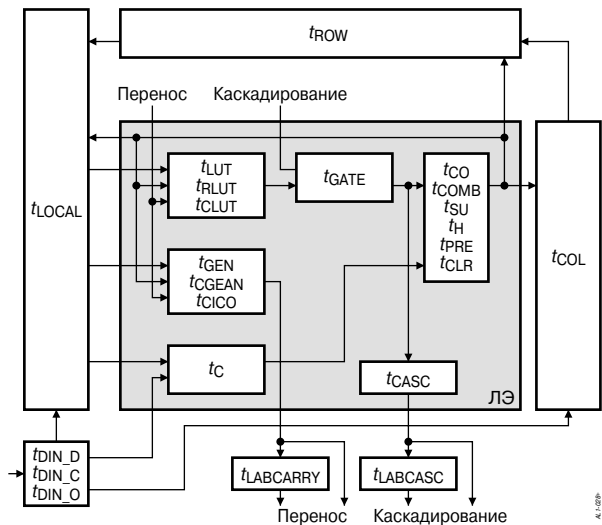


Рис. 1.28. Временная модель ПЛИС семейства FLEX8000

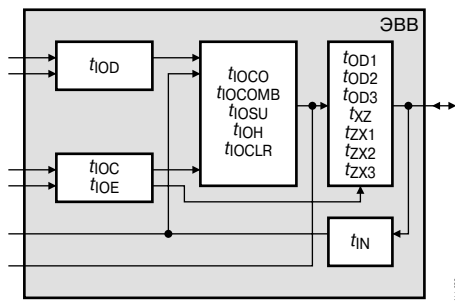


Рис. 1.29. Временная модель ПЛИС семейства FLEX8000

Таблица 1.12. Параметры временной модели ПЛИС семейства FLEX8000

Обозначение	Параметр	Значение для EPF8282A-2 [нс]	
		min	max
t_{IOD}	Задержка данных в регистре ЭВВ	—	0.7
t_{IOC}	Задержка сигнала управления регистра ЭВВ	—	1.7
t_{IOE}	Задержка сигнала разрешения выхода	—	1.7
t_{IOCO}	Задержка появления данных на выходе регистра ЭВВ после подачи синхроимпульса	—	1.0
t_{IOCOMB}	Задержка комбинационных схем ЭВВ	—	0.3
t_{IOSU}	Время установки регистра ЭВВ	1.4	—
t_{IOH}	Время удержания данных регистра ЭВВ	0.0	—
t_{IOCLR}	Задержка сброса регистра ЭВВ	—	1.2
t_{IN}	Задержка сигнала во входном буфере	—	1.5
t_{OD1}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	1.1
t_{OD3}	Задержка сигнала от выходного буфера до вывода, $\text{slew rate} = \text{on}$	—	4.6
t_{XZ}	Задержка сигнала в выходном буфере после сигнала запрещения выхода	—	1.4
t_{ZX1}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	1.4
t_{ZX3}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $\text{slew rate} = \text{on}$	—	4.9
t_{LUT}	Задержка данных в ТП	—	2.0
t_{CLUT}	Задержка формирования сигнала переноса в ТП	—	0.0
t_{RLUT}	Задержка сигнала обратной связи регистра в ТП	—	0.9
t_{GATE}	Задержка в логике каскадирования	—	0.0
t_{CASC}	Задержка в цепи каскадирования	—	0.6
t_{CICO}	Задержка данных со входа на выход ускоренного переноса	—	0.4
t_{CGEN}	Задержка распространения данных со входа на выход переноса	—	0.4
t_{CGENR}	Задержка сигнала обратной связи с регистра ЛЭ на цепь переноса	—	0.9
t_C	Задержка управляющего сигнала на регистре	—	1.6
t_{CH}	Длительность высокого уровня тактового сигнала регистра	4.0	—
t_{CL}	Длительность низкого уровня тактового сигнала регистра	4.0	—
t_{CO}	Задержка выходного сигнала регистра относительно такта	—	0.4
t_{COMB}	Задержка в комбинационной части ЛЭ	—	0.4
t_{SU}	Время установки регистра	0.8	—

Таблица 1.12 (окончание)

Обозначение	Параметр	Значение для EPF8282A-2 [нс]	
		min	max
t_H	Время удержания сигнала на входе регистра после подачи синхронимпульса	0.9	—
t_{PRE}	Задержка предустановки регистра	—	0.6
t_{CLR}	Задержка сброса регистра	—	0.6
$t_{LABCASC}$	Задержка сигнала каскадирования в другой ЛБ	—	0.3
$t_{LABCARRY}$	Задержка сигнала переноса в другой ЛБ	—	0.3
t_{LOCAL}	Задержка ЛМС	—	0.5
t_{ROW}	Задержка в строке ГМС	—	4.2
t_{COL}	Задержка в столбце ГМС	—	2.5
t_{DIN_D}	Задержка данных с выделенного вывода до входа ЛЭ	—	7.2
t_{DIN_C}	Задержка управляющих сигналов с выделенного вывода до входа ЛЭ	—	5.0
t_{DIN_IO}	Задержка управляющих сигналов с выделенного вывода до входов управления ЭВВ	—	5.0
t_{DPR}	Тестовая задержка регистр-регистр через 4 ЛЭ, 3 ряда и 4 ЛМС	—	15.8

1.5. Семейство MAX9000

Семейство ПЛИС MAX9000 имеет матричную структуру, подобную ПЛИС семейств FLEX6000 и FLEX8000, но выполнена по EPROM технологии, так же, как и ПЛИС семейств MAX3000 и MAX7000.

Микросхемы семейства MAX9000 имеют достаточно высокую логическую емкость и не требуют внешнего конфигурационного ПЗУ. Благодаря матричной структуре межсоединений они являются подходящей элементной базой для реализации алгоритмов ЦОС. Основные параметры ПЛИС семейства MAX9000 приведены в **Табл. 1.13**.

Таблица 1.13. Параметры ПЛИС семейства MAX9000

Параметр	ERM9320	ERM9400	ERM9480	ERM9560
Логическая емкость, количество эквивалентных вентиляей	600	1 250	2 500	5 000
Число макроячеек	320	400	480	560
Число логических блоков	484	580	676	772
Число программируемых пользователем выводов	168	159	175	216

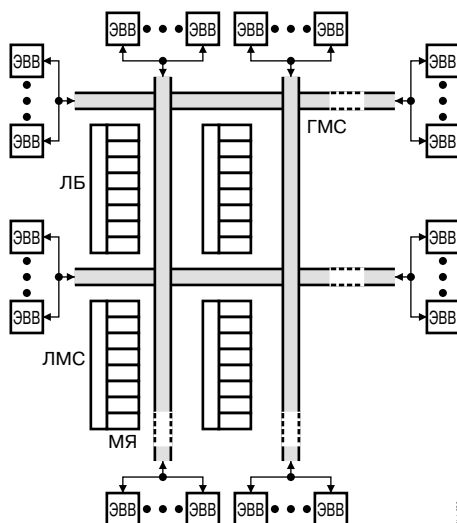


Рис. 1.30. Структурная схема ПЛИС семейства MAX9000

Структурная схема ПЛИС семейства MAX9000 приведена на **Рис. 1.30**. В основе архитектуры ПЛИС семейства MAX9000 лежит глобальная матрица соединений, разделенная на строки и столбцы. Макроячейки объединяются в логические блоки, содержащие по 16 МЯ каждый, а также локальную матрицу соединений. На **Рис. 1.31** приведена структурная схема логического блока ПЛИС семейства MAX9000.

Как можно видеть из **Рис. 1.31**, каждая МЯ имеет возможность коммутации как на локальную, так и на строки и столбцы глобальной матрицы соединений.

Собственно структура МЯ ПЛИС семейства MAX9000 показана на **Рис. 1.32**. Она практически не отличается от МЯ ПЛИС семейств MAX7000 или MAX3000. Так же, как и у ПЛИС этих семейств, имеется возможность использования параллельных и разделяемых расширителей.

Наличие элемента ввода/вывода ПЛИС семейства MAX9000 показано на **Рис. 1.33**. Наличие в ЭВВ тактируемого триггера позволяет, по сути, выполнять хранение входных или выходных данных в ЭВВ, не задействуя ресурсы МЯ.

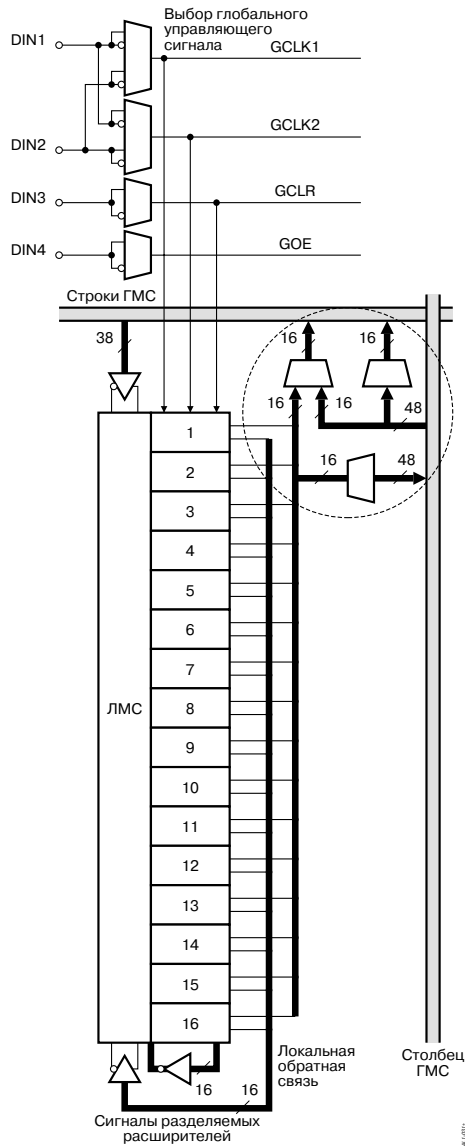


Рис. 1.31. Структурная схема логического блока ПЛИС семейства MAX9000

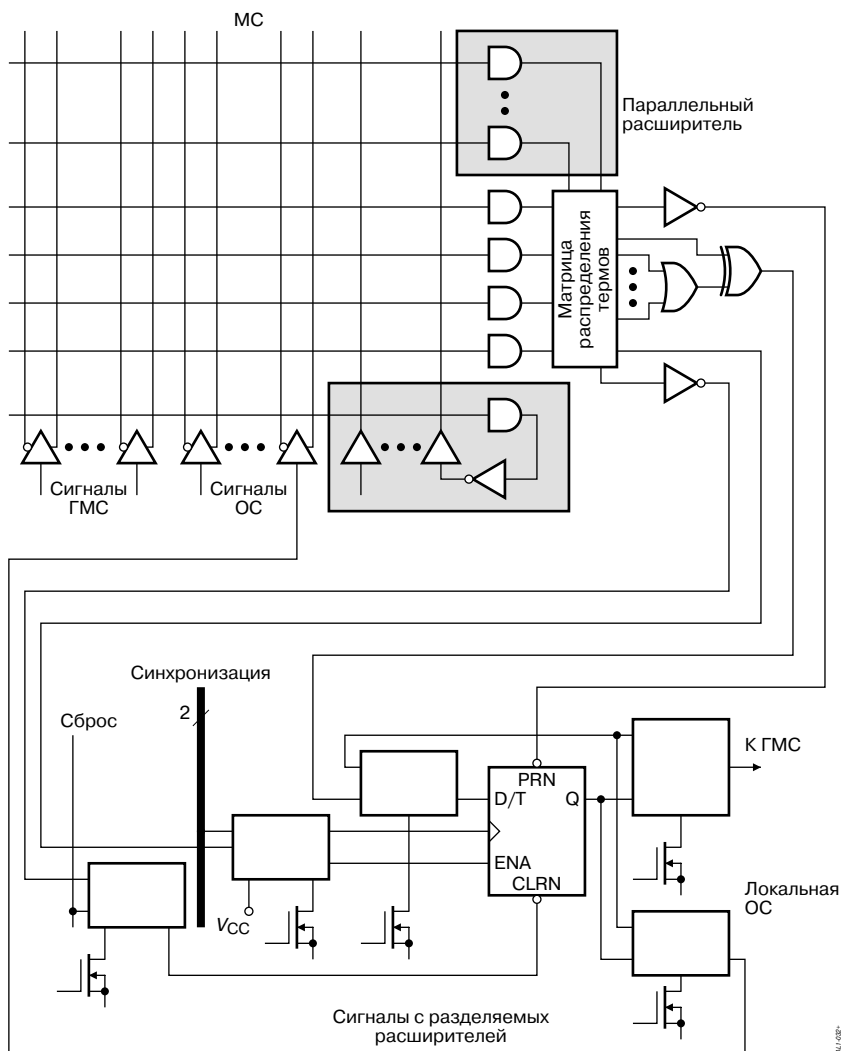


Рис. 1.32. МЯ ПЛИС семейства MAX9000

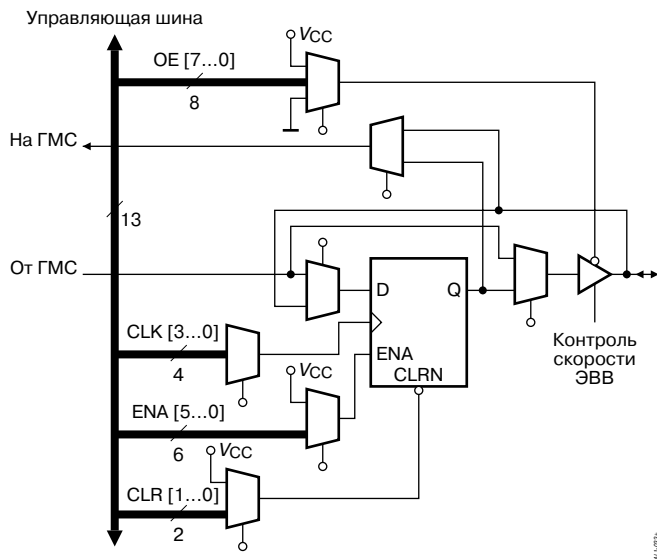


Рис. 1.33. ЭВБ ПЛИС семейства MAX9000

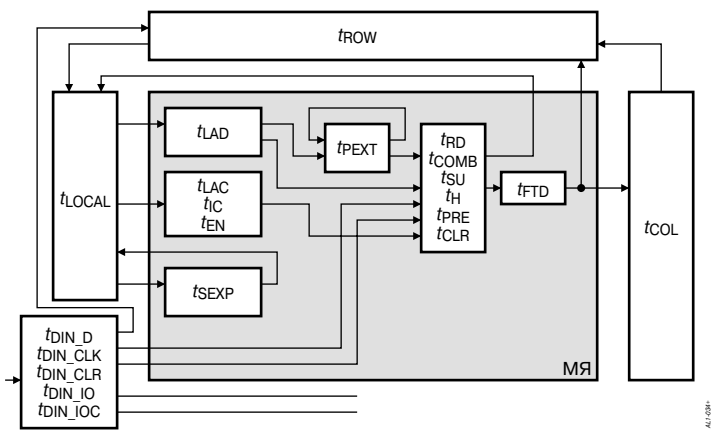


Рис. 1.34. Временная модель ПЛИС семейства MAX9000

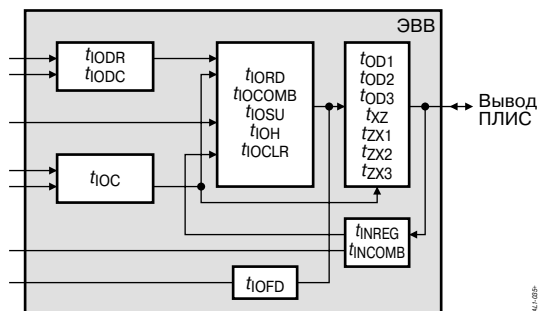


Рис. 1.35. Временная модель ЭВБ ПЛИС семейства MAX9000

Временная модель ПЛИС семейства MAX9000 приведена на Рис. 1.34. и 1.35, а ее параметры в Табл. 1.14.

Таблица 1.14. Параметры временной модели семейства MAX9000

Обозначение	Параметр	Значение для EPM9400-10 [нс]	
		min	max
t_{PD1}	Задержка вход—выход по строке	—	10.0
t_{PD2}	Задержка вход—выход по столбцу	—	10.8
t_{FSU}	Время установки глобального синхросигнала	3.0	—
t_{FH}	Время удержания глобального синхросигнала	0.0	—
t_{FCO}	Задержка глобального синхросигнала до выхода	1.0	4.8
t_{CNT}	Минимальный период глобального синхросигнала	—	6.9
f_{CNT}	Максимальная глобальная внутренняя тактовая частота	144.9	—
t_{LAD}	Задержка в локальной программируемой матрице ««И»»	—	3.5
t_{LAC}	Задержка управляющего сигнала триггера в локальной программируемой матрице «И»	—	3.5
t_{IC}	Задержка изменения сигнала относительно тактового импульса	—	3.5
t_{EN}	Задержка разрешения регистра	—	3.5
t_{SEXP}	Задержка разделяемого расширителя	—	3.5
t_{PEXP}	Задержка параллельного расширителя	—	0.5
t_{RD}	Регистровая задержка	—	0.5
t_{COMB}	Комбинационная задержка	—	0.4
t_{SU}	Время установки регистра	2.4	—
t_H	Время удержания сигнала в регистре	2.0	—

Таблица 1.14 (окончание)

Обозначение	Параметр	Значение для ЕРМ9400-10 [нс]	
		min	max
t_{PRE}	Время предустановки регистра МЯ	—	3.5
t_{CLR}	Время сброса регистра МЯ	—	3.7
t_{FTD}	Задержка глобальной ПМС	—	0.5
t_{LPA}	Задержка за счет режима пониженного потребления	—	10.0
t_{IODR}	Задержка ввода/вывода данных на строку ГМС	—	0.2
t_{IODC}	Задержка ввода/вывода данных на столбец ГМС	—	0.4
t_{IOC}	Задержка управляющего сигнала ЭВВ	—	0.5
t_{IORD}	Задержка данных на выходе ЭВВ относительно такта	—	0.6
t_{IOCOMB}	Задержка комбинационных схем ЭВВ	—	0.2
t_{IOSU}	Время установки регистра ЭВВ	2.0	—
t_{IOH}	Время удержания данных регистра ЭВВ	1.0	—
t_{IOCLR}	Задержка сброса регистра ЭВВ	—	1.5
t_{IOFD}	Задержка обратной связи регистра ЭВВ	—	0.0
t_{INREG}	Задержка во входном буфере ЭВВ	—	3.5
t_{INCOMB}	Задержка во входном буфере ЭВВ	—	1.5
t_{OD1}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	1.8
t_{OD2}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	2.3
t_{OD3}	Задержка сигнала от выходного буфера до вывода, $\text{slew rate} = \text{on}$	—	8.3
t_{XZ}	Задержка сигнала в выходном буфере после сигнала запрещения выхода	—	2.5
t_{ZX1}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	2.5
t_{ZX2}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	3.0
t_{ZX3}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $\text{slew rate} = \text{on}$	—	9.0
t_{LOCAL}	Задержка ЛМС	—	0.5
t_{ROW}	Задержка в строке ГМС	—	0.9
t_{COL}	Задержка в столбце ГМС	—	0.9
t_{DIN_D}	Задержка данных с выделенного вывода до входа ЛЭ	—	4.0
t_{DIN_C}	Задержка управляющих сигналов с выделенного вывода до входа ЛЭ	—	2.7
t_{DIN_IO}	Задержка управляющих сигналов с выделенного вывода до входов управления ЭВВ	—	2.5

1.6. Семейство FLEX10K

ПЛИС семейств FLEX10K, FLEX10KA, FLEX10KE являются на данный момент, пожалуй, самой популярной элементной базой для реализации алгоритмов ЦОС, построения сложных устройств обработки данных и интерфейсов. Это объясняется тем, что благодаря большой логической емкости, удобной архитектуре, включающей встроенные блоки памяти (ЕАВ — Embedded Array Block), достаточно высокой надежности и удачному соотношению цена—логическая емкость данные ПЛИС удовлетворяют разнообразным требованиям, возникающим у разработчика как систем ЦОС, так и устройств управления, обработки данных и т.п. В **Табл. 1.15** приведены основные сведения о ПЛИС семейства FLEX10K.

Таблица 1.15. ПЛИС семейства FLEX10K

Параметр	EPF10K10	EPF10K20	EPF10K30	EPF10K40	EPF10K50	EPF10K70	EPF10K100	EPF10K130	EPF10K250
Логическая емкость, количество эквивалентных вентилей	10 000	20 000	30 000	40 000	50 000	70 000	100 000	130 000	250 000
Число логических элементов	576	1 152	1 728	2 304	2 880	3 744	4 992	6 656	12 160
Встроенные блоки памяти	3	6	6	8	10	9	12	16	20
Объем памяти [бит]	6 144	12 288	12 288	16 384	20 480	18 432	24 576	32 768	40 960
Максимальное число выводов пользователя	150	189	246	189	310	358	406	470	470

В настоящее время выпускаются ПЛИС семейств FLEX10K с напряжением питания 5 В, FLEX10KA (V) с напряжением питания 3.3 В и FLEX10KE с напряжением питания 2.5 В. Кроме того, ПЛИС семейства FLEX10KE имеют емкость встроенного блока памяти 4096 бит, в отличие от ПЛИС остальных семейств, имеющих емкость ЕАВ 2048 бит.

Обобщенная функциональная схема ПЛИС семейства FLEX10K приведена на **Рис. 1.36**. В основе архитектуры лежат логические блоки, содержащие 8 ЛЭ и локальную матрицу соединений. Глобальная матрица соединений разделена на строки и столбцы, имеет непрерывную структуру (Fast Track Interconnect). Посередине строки располагаются встроенные блоки памяти. Кроме того, имеются глобальные цепи управления, синхронизации и управления вводом/выводом.

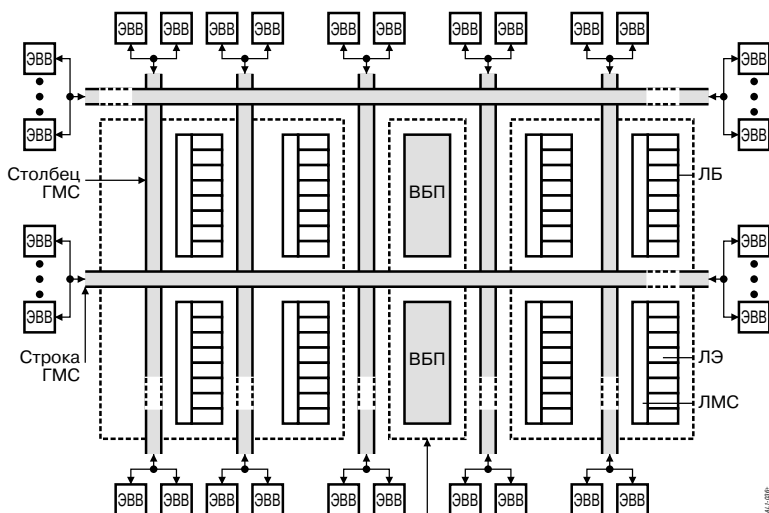


Рис. 1.36. Архитектура ПЛИС семейства FLEX10K

Встроенный блок памяти (ВБП) (**Рис. 1.37**) представляет собой оперативное запоминающее устройство (ОЗУ) емкостью 2048 (4096) бит и состоит из локальной матрицы соединений, собственно модуля памяти, синхронных буферных регистров, а также программируемых мультиплексоров.

Сигналы на вход ЛМС ВБП поступают со строки ГМС. Тактовые и управляющие сигналы поступают с глобальной шины управляющих сигналов. Выход ВБП может быть скоммутирован как на строку, так и на столбец ГМС.

Наличие синхронных буферных регистров и программируемых мультиплексоров позволяет конфигурировать ВБП как запоминающее устройство (ЗУ) с организацией 256×8 , 512×4 , 1024×2 , 2048×1 .

Наличие ВБП дает возможность табличной реализации таких элементов устройств ЦОС, как перемножители, арифметическое логическое устройство (АЛУ), сумматоры и т.п., имеющих быстродействие до 100 МГц (конечно, при самых благоприятных условиях реально быстродействие арифметических устройств, реализованных на базе ВБП, составляет 10...50 МГц).

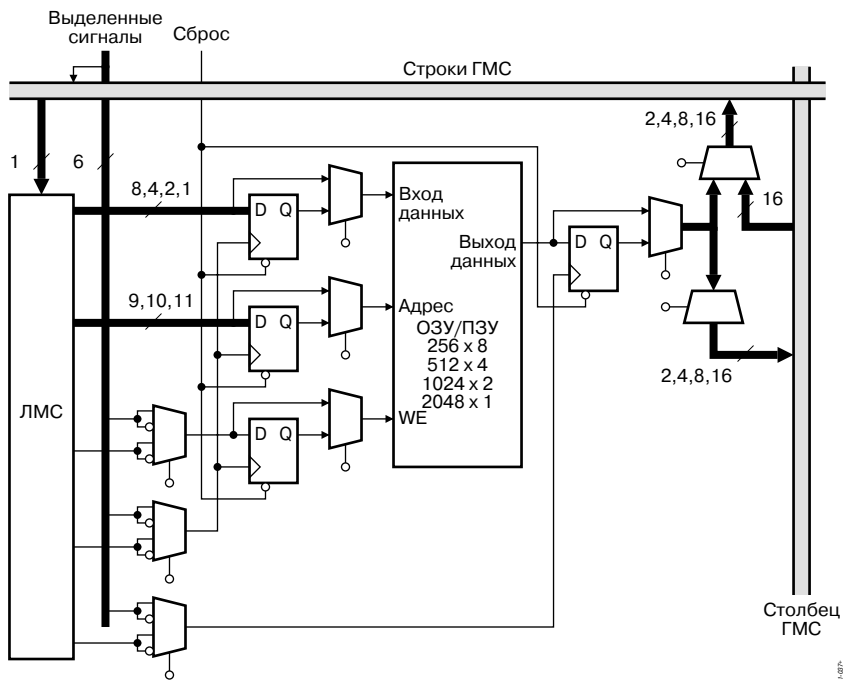


Рис. 1.37. Встроенный блок памяти

Все ПЛИС семейства FLEX10K совместимы по уровням с шиной PCI, имеют возможность как последовательной, так и параллельной загрузки, полностью поддерживают стандарт JTAG.

Структура логического блока ПЛИС семейства FLEX10K приведена на **Рис. 1.38**. Сигналы на вход ЛМС поступают как со строки ГМС, так и с выходов каждого из 8 ЛЭ, входящих в состав ЛБ. Сигналы с выхода ЛБ поступают как на строку, так и на столбец ГМС. Как можно заметить, архитектура ЛБ семейства FLEX10K напоминает архитектуру ЛБ ПЛИС семейства FLEX8000. Структура ЛЭ ПЛИС семейства FLEX10K приведена на **Рис. 1.39**. Как правило, архитектура ЛЭ всех семейств FLEX практически одинакова. С помощью схем организации

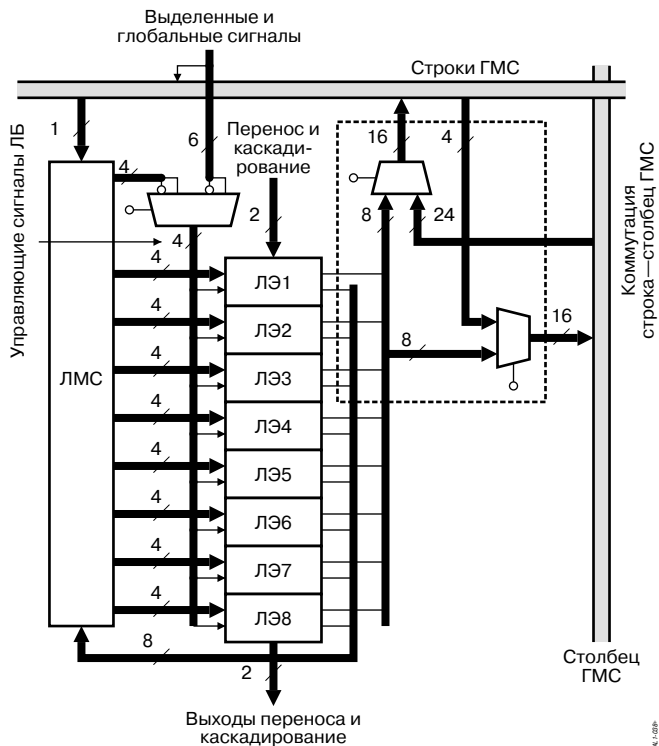


Рис. 1.38. Логический блок ПЛИС семейства FLEX10K

переносов (carry chain) и каскадирования (cascade chain) возможно расширение возможностей ЛЭ (подробнее о режимах конфигурации ЛЭ см. параграф 1.2).

Элемент ввода/вывода ПЛИС семейства FLEX10K соединяет канал строки или столбца ГМС с выводом микросхемы. Структурная схема ЭВВ приведена на **Рис. 1.40**. ЭВВ позволяет осуществить ввод/вывод бита данных с различными скоростями, временное хранение данных, эмуляцию открытого коллектора.



Рис. 1.39. Структура ЛЭ ПЛИС семейства FLEX10K

Наличие входного (input register) и выходного (output register) регистров позволяет хранить данные, что снижает логическую нагрузку на ЛЭ и высвобождает ресурсы ПЛИС для реализации других функций. Скорость переключения буфера ЭВВ может быть задана при конфигурации ПЛИС. Пониженная скорость переключения позволяет снизить уровень импульсных помех и «звона» в системе.

Следует помнить, что режим эмуляции открытого коллектора обеспечивает не слишком мощный выходной сигнал, поэтому при необходимости сопряжения с внешними схемами лучше использовать специализированные буферы (например, 74НС04, 1533ЛА8, 1533ЛН2 и т.п.). По крайней мере, при воздействии высокого напряжения проще (и дешевле) поменять буфер, а не всю ПЛИС (особенно в BGA корпусе). Временная модель ПЛИС представлена на **Рис. 1.41—1.44**, а ее основные параметры — в **Табл. 1.16**.

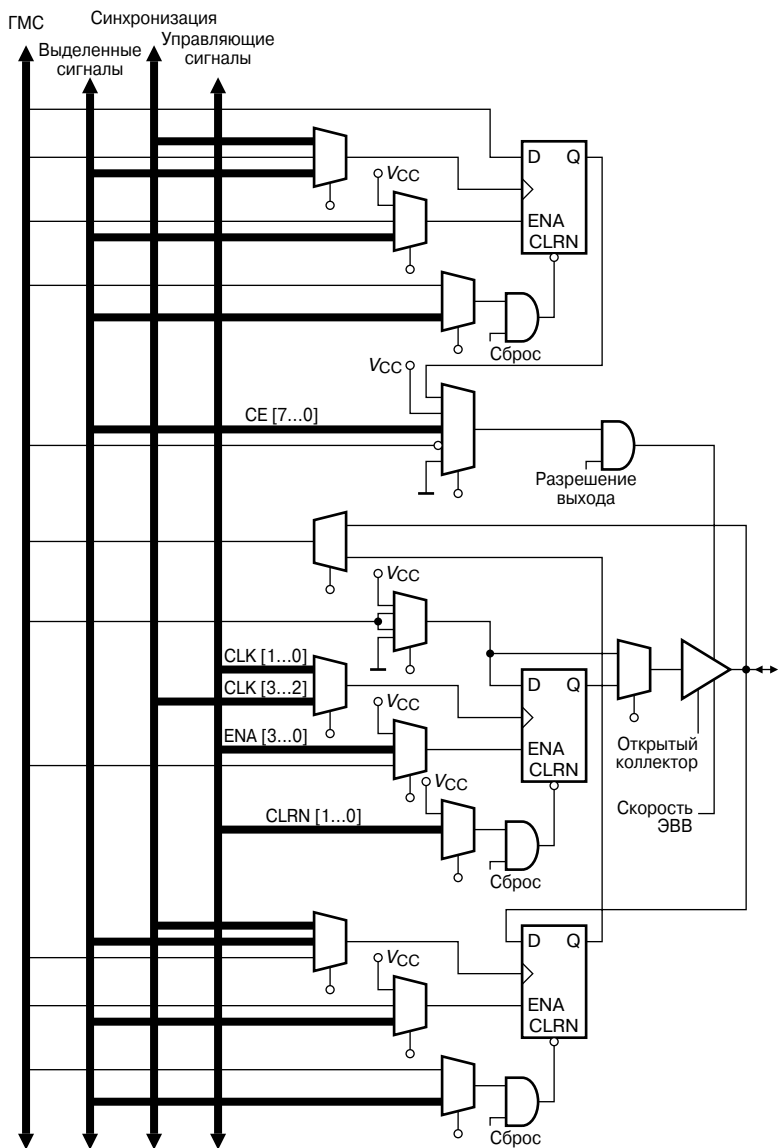


Рис. 1.40. Элемент ввода/вывода

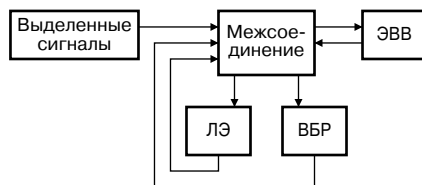


Рис. 1.41. Временная модель ПЛИС семейства FLEX10K

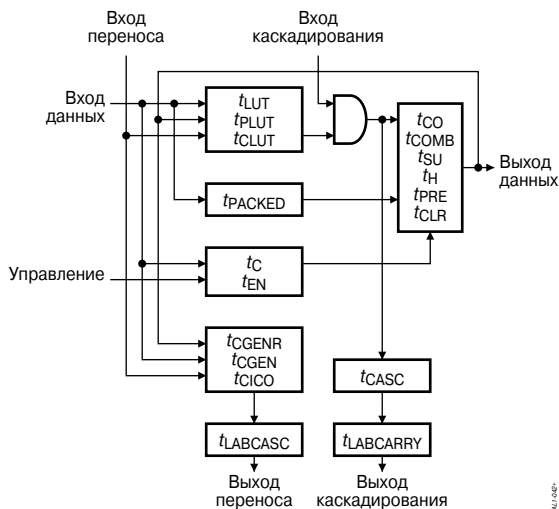


Рис. 1.42. Временная модель ЛЭ ПЛИС семейства FLEX10K

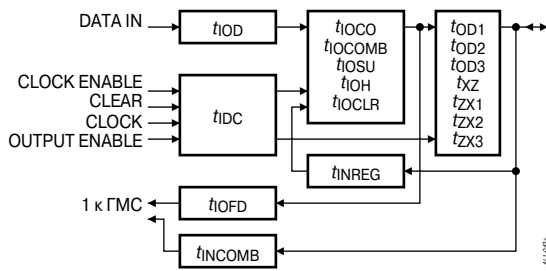


Рис. 1.43. Временная модель ЭВВ ПЛИС семейства FLEX10K

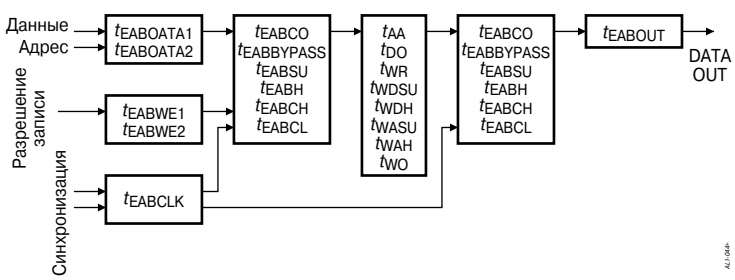


Рис. 1.44. Временная модель ВВП ПЛИС семейства FLEX10K

Таблица 1.16. Параметры временной модели ПЛИС семейства FLEX10K

Обозначение	Параметр	Значение для EPF10K10-3 [нс]	
		min	max
t_{LUT}	Задержка данных в ТП	—	1.4
t_{CLUT}	Задержка формирования сигнала переноса в ТП	—	0.6
t_{RLUT}	Задержка сигнала обратной связи регистра в ТП	—	1.5
t_{PACKED}	Задержка входных данных до отдельно сконфигурированного регистра (packed register)	—	0.6
t_{EN}	Задержка сигнала разрешения регистра ЛЭ	—	1.0
t_{CICO}	Задержка сигнала переноса от входа до выхода переноса	—	0.2

Таблица 1.16 (продолжение)

Обозначение	Параметр	Значение для EPF10K10-3 [нс]	
		min	max
t_{CGEN}	Задержка сигнала переноса от входа ЛЭ до выхода переноса	—	0.9
t_{CGENR}	Задержка сигнала от выхода регистра ЛЭ до выхода переноса	—	0.9
t_{CASC}	Задержка сигнала каскадирования от входа до выхода каскадирования	—	0.8
t_C	Задержка управляющего сигнала на регистре ЛЭ	—	1.3
t_{CO}	Задержка выходного сигнала регистра относительно такта	—	0.9
t_{COMB}	Задержка в комбинационной части ЛЭ	—	0.5
t_{SU}	Время установки регистра ЛЭ	1.3	—
t_H	Время удержания сигнала на входе регистра после подачи синхросигнала	1.4	—
t_{PRE}	Задержка предустановки регистра ЛЭ	—	1.0
t_{CLR}	Задержка сброса регистра ЛЭ	—	1.0
t_{CH}	Длительность высокого уровня тактового сигнала регистра	4.0	—
t_{CL}	Длительность низкого уровня тактового сигнала регистра	4.0	—
t_{IOD}	Задержка выходного сигнала ЭВВ	—	1.3
t_{IOC}	Задержка выходного сигнала регистра ЭВВ относительно сигналов управления	—	0.5
t_{IOCO}	Задержка выходного сигнала регистра ЭВВ относительно такта	—	0.2
t_{IOCOMB}	Задержка комбинационных схем ЭВВ	—	0.0
t_{IOSU}	Время установки регистра ЭВВ	2.8	—
t_{IOH}	Время удержания данных регистра ЭВВ	1.0	—
t_{IOCLR}	Задержка сброса регистра ЭВВ	—	1.0
t_{OD1}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	2.6
t_{OD2}	Задержка сигнала от выходного буфера до вывода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	4.9
t_{OD3}	Задержка сигнала от выходного буфера до вывода, $\text{slew rate} = \text{on}$	—	6.3
t_{XZ}	Задержка сигнала в выходном буфере после сигнала запрещения выхода	—	4.5
t_{ZX1}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 3.3 \text{ В}$, $\text{slew rate} = \text{off}$	—	4.5
t_{ZX2}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $V_{CCIO} = 2.5 \text{ В}$, $\text{slew rate} = \text{off}$	—	6.8

Таблица 1.16 (продолжение)

Обозначение	Параметр	Значение для ERF10K10-3 [нс]	
		min	max
t_{ZX3}	Задержка сигнала в выходном буфере после сигнала разрешения выхода, $slew\ rate = on$	—	8.2
t_{INREG}	Задержка в буфере ЭВВ	—	6.0
t_{IOFD}	Задержка в цепи обратной связи регистра ЭВВ	—	3.1
t_{INCOMB}	Задержка сигнала от входного буфера ЭВВ до ГМС	—	3.1
$t_{EABDATA1}$	Задержка данных или адреса ВБП до комбинационного выхода ВБП	—	1.5
$t_{EABDATA2}$	Задержка данных или адреса ВБП до регистрового выхода ВБП	—	4.8
t_{EABWE1}	Задержка данных ВБП относительно сигнала разрешения записи до комбинационного выхода ВБП	—	1.0
t_{EABWE2}	Задержка данных ВБП относительно сигнала разрешения записи до регистрового выхода ВБП	—	5.0
t_{EABCLK}	Задержка тактового импульса на регистре ВБП	—	1.0
t_{EABCO}	Задержка выхода ВБП относительно тактового импульса	—	0.5
$t_{EABYPASS}$	Задержка в цепи обхода регистра ВБП	—	1.5
t_{EABSU}	Время установки регистра ВБП	1.5	—
t_{EABH}	Время удержания регистра ВБП	2.0	—
t_{EABCH}	Длительность высокого уровня тактового сигнала регистра ВБП	4.0	—
t_{EABCL}	Длительность низкого уровня тактового сигнала регистра ВБП	5.8	—
t_{AA}	Время удержания адреса	—	8.7
t_{WP}	Длительность импульса записи ВБП	5.8	—
t_{WDSU}	Время установки данных до записи	1.6	—
t_{WDH}	Время удержания данных при сигнале записи в ВБП	0.3	—
t_{WASU}	Время установки адреса	0.5	—
t_{WAH}	Время удержания адреса	1.0	—
t_{WO}	Задержка данных на выходе ВБП относительно сигнала разрешения записи	—	5.0
t_{DD}	Задержка данных от входа до выхода ВБП	—	5.0
t_{EABOUT}	Задержка данных на выходе ВБП	—	0.5
t_{EABAA}	Время доступа адреса ВБП	—	13.7
$t_{EABRCCOMB}$	Длительность цикла асинхронного чтения из ВБП	13.7	—
$t_{EABRCREG}$	Длительность цикла синхронного чтения из ВБП	9.7	—
t_{EABWP}	Длительность импульса записи ВБП	5.8	—

Таблица 1.16 (продолжение)

Обозначение	Параметр	Значение для EPF10K10-3 [нс]	
		min	max
$t_{EABWCCOMB}$	Длительность цикла асинхронной записи в ВБП	7.3	—
$t_{EABWCREG}$	Длительность цикла синхронной записи в ВБП	13.0	—
t_{EABDD}	Задержка данных от входа до выхода ВБП	—	10.0
$t_{EABDATAO}$	Задержка данных на выходе ВБП относительно такта	—	2.0
$t_{EABDATASU}$	Время установки адреса или данных во входном регистре ВБП	5.3	—
$t_{EABDATAH}$	Время удержания адреса или данных на входе ВБП	0.0	—
$t_{EABWESU}$	Время установки сигнала WE	5.5	—
$t_{EABWESH}$	Время удержания сигнала WE	0.0	—
$t_{EABWDSU}$	Время установки входных данных ВБП без использования входного регистра	5.5	—
t_{EABWDH}	Время удержания входных данных ВБП без использования входного регистра	0.0	—
$t_{EABWASU}$	Время установки адреса ВБП без использования входного регистра	2.1	—
t_{EABWAH}	Время удержания адреса ВБП без использования входного регистра	0.0	—
t_{EABWO}	Задержка данных на выходе ВБП относительно сигнала разрешения записи	—	9.5
$t_{SAMELAB}$	Задержка данных в ЛМС	—	0.6
$t_{SAMEROW}$	Задержка передачи данных внутри одной и той же строки ГМС	—	3.6
$t_{SAMECOLUMN}$	Задержка передачи данных внутри одного и того же ГМС	—	0.9
$t_{DIFFROW}$	Задержка передачи данных по столбцу с одной строки ГМС на другую	—	4.5
$t_{TROWROWS}$	Задержка передачи данных с одной строки ГМС на другую	—	8.1
$t_{LEPERIPH}$	Задержка управляющего сигнала	—	3.3
$t_{LABCARRY}$	Задержка переноса в следующий ЛБ	—	0.5
$t_{LABCASC}$	Задержка каскадирования в следующий ЛБ	—	2.7
$t_{DIN2IOE}$	Задержка распространения с выделенного вывода до входа управления ЭВВ	—	4.8
t_{DIN2LE}	Задержка распространения с выделенного вывода до входа управления ЛБ или ВБП	—	2.6
$t_{DCLK2IOE}$	Задержка распространения тактового сигнала с выделенного вывода до входа тактирования ЭВВ	—	3.4
$t_{DCLK2LE}$	Задержка распространения тактового сигнала с выделенного вывода до входа тактирования ЛБ или ВБП	—	2.6

Таблица 1.16 (окончание)

Обозначение	Параметр	Значение для ERF10K10-3 [нс]	
		min	max
$t_{DIN2DATA}$	Задержка распространения данных с выделенного вывода до входа ЛБ или ВБП	—	4.3
t_{DPR}	Тестовая задержка регистр-регистр через 4 ЛЭ, 3 ряда и 4 ЛМС	—	16.1
t_{INSU}	Время установки глобального тактового импульса	5.5	—
t_{INH}	Время удержания данных относительно глобального тактового импульса	0.0	—
t_{OUTCO}	Время задержки выходных данных относительно глобального тактового импульса	2.0	6.7
$t_{ISUBIDIR}$	Время установки двунаправленного вывода относительно глобального тактового импульса	4.5	—
$t_{INHIBIDIR}$	Время удержания двунаправленного вывода относительно глобального тактового импульса	0.0	—
$t_{OUTCOBIDIR}$	Время задержки выходных данных на двунаправленном выводе относительно глобального тактового импульса	2.0	6.7
$t_{XZBIDIR}$	Задержка перехода выходного буфера в третье состояние	—	10.0
$t_{ZXBIDIR}$	Задержка перехода выходного буфера из третьего состояния	—	10.0

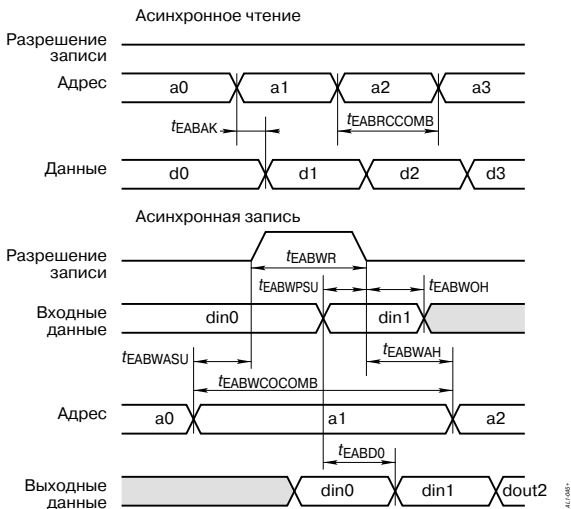


Рис. 1.45. Асинхронные режимы чтения и записи ВБП ПЛИС семейства FLEX10K

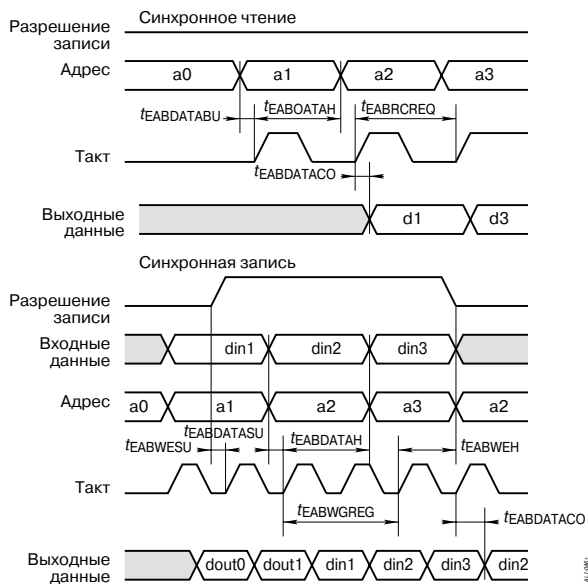


Рис.1.46. Синхронные режимы чтения и записи ВБП ПЛИС семейства FLEX10К

1.7. Семейство APEx20K

Развитие и разнообразие архитектур функциональных преобразователей, лежащих в основе базовых узлов ПЛИС, привели к тому, что в последние годы ПЛИС становятся основой для «систем на кристалле». В основе идеи SOC лежит интеграция всей электронной системы в одном кристалле (например, в случае ПК такой чип объединяет процессор, память и т.д.). Компоненты этих систем разрабатываются отдельно и хранятся в виде файлов параметризуемых модулей. Окончательная структура SOC-микросхемы выполняется на базе этих «виртуальных компонентов», называемых также «блоками интеллектуальной собственности» с помощью программ автоматизации проектирования электронных устройств. Благодаря стандартизации в одно целое можно объединять «виртуальные компоненты» от разных разработчиков.

Идея построения «систем на кристалле» подстегнула ведущих производителей ПЛИС к выпуску в конце 1998 — начале 1999 года изделий с эквивалентной емкостью 1000000 эквивалентных вентилях и более.

Примером новых семейств ПЛИС, пригодных для реализации «систем на кристалле», является семейство APEx20K фирмы «Altera», основные характеристики которого приведены в Табл. 1.17.

Таблица 1.17. Основные характеристики ПЛИС семейства APEx20K фирмы «Altera»

Параметр	EP20K 100	EP20K 160	EP20K 200	EP20K 300	EP20K 400	EP20K 600	EP20K 1000
Логическая емкость, количество эквивалентных вентилях	263 000	404 000	526 000	728 000	1 052 000	1 537 000	2 670 000
Число логических элементов	4 160	6 400	8 320	11 520	16 640	24 320	42 240
Встроенные блоки памяти	26	40	52	72	104	152	264
Максимальный объем памяти [бит]	53 248	81 920	106 496	147 456	212 992	311 296	540 672
Число макроячеек	416	640	832	1 152	1 664	2 432	4 224
Число выводов пользователя	252	320	382	420	502	620	780

Архитектура ПЛИС семейства APEX20K (**Рис. 1.47**) сочетает в себе достоинства FPGA ПЛИС с их таблицами перекодировок, входящими в состав логического элемента, логику вычисления СДНФ (совершенная дизъюнктивная нормальная форма), характерную для ПЛИС CPLD, а также встроенные модули памяти.

Отличительной особенностью ПЛИС семейства APEX20K является объединение ЛБ в так называемый мегаблок (megaLAB) (**Рис. 1.48**), имеющий собственную непрерывную матрицу соединений (megaLAB interconnect).

Такая организация соединений позволяет выделить дополнительные ресурсы для трассировки, кроме того, в каждом мегаблоке может быть полностью разведена та или иная функционально законченная часть сис-

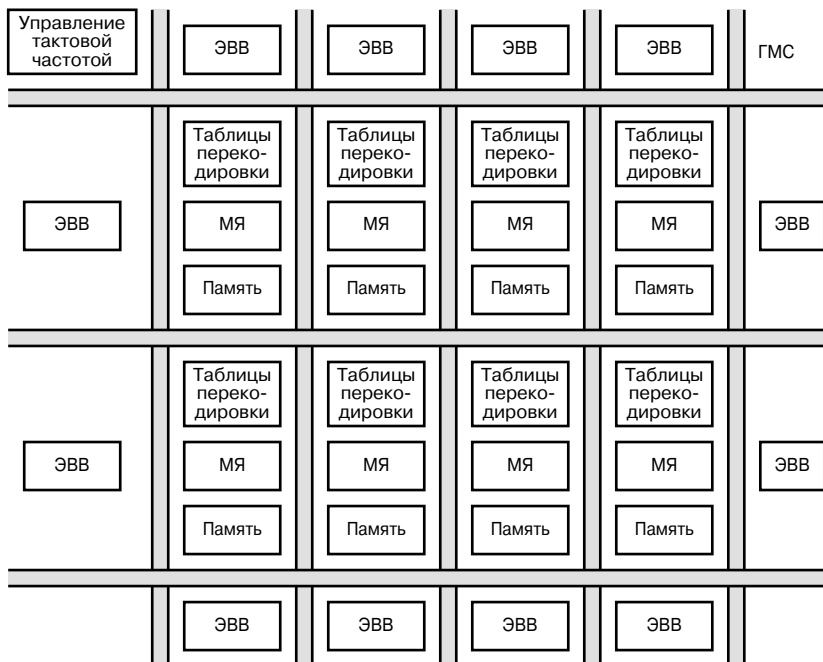


Рис. 1.47. Архитектура ПЛИС семейства APEX20K

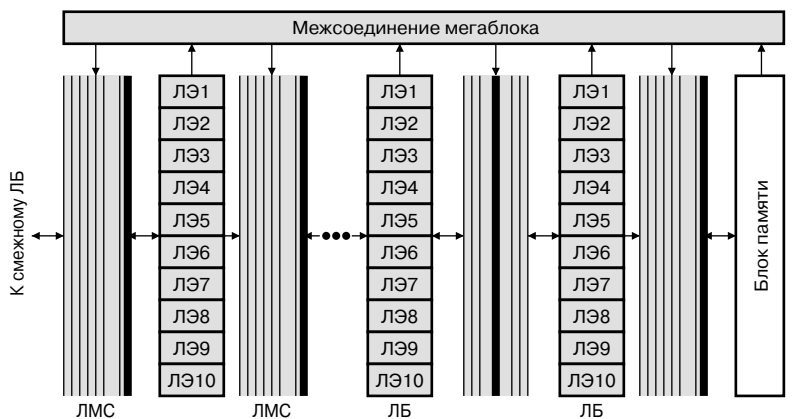


Рис. 1.48. Структура мегаблока ПЛИС семейства APEX20K

темы, что позволяет при ее модификации не перетрассировать этот участок и тем самым сохранить все заданные временные параметры. Также такая организация ПЛИС позволяет разумнее организовать соответствующее программное обеспечение, в том числе создать средства коллективной работы над проектом.

На **Рис. 1.49** представлена структура ЛБ ПЛИС семейства APEX20K. Каждый ЛБ состоит из 10 ЛЭ, имеющих структуру, показанную на **Рис. 1.50**. Как можно заметить, структура ЛБ объединяет все лучшее, что наработано в предшествующих семействах ПЛИС. Каждый ЛЭ имеет возможность коммутации на два столбца ГМС, подобно ПЛИС семейства FLEX6000. Матрица соединений мегаблока (МСМ) коммутируется на ЛМС ЛБ и на строки ГМС.

В отличие от семейств FLEX, ЛЭ ПЛИС семейства APEX20K имеет возможность формирования управляющих сигналов триггера как с помощью глобальных и локальных сигналов, так и используя сигналы мегаблока. Аналогично семействам FLEX ЛЭ может быть сконфигурирован в нормальном, арифметическом или счетном режиме, допускает каскадирование и цепочечный перенос.

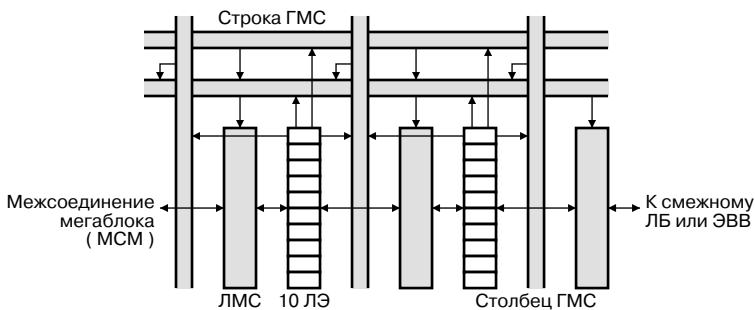


Рис. 1.49. Структура ЛБ

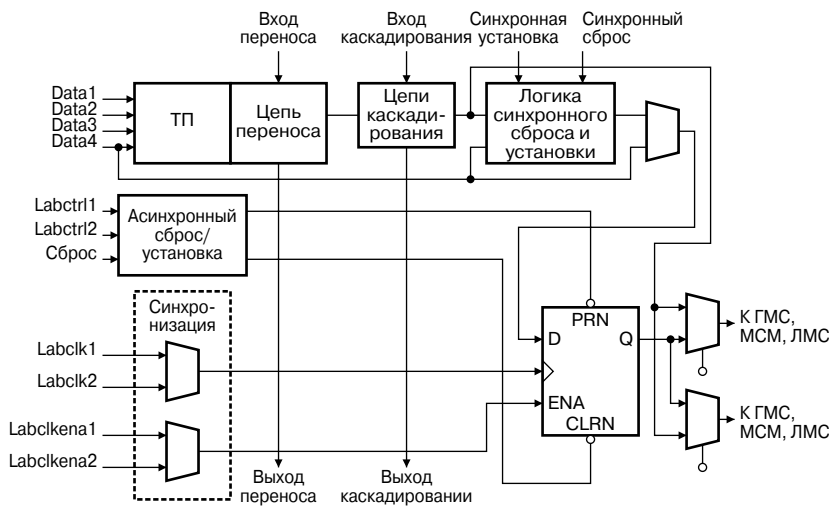


Рис. 1.50. Логический элемент ПЛИС семейства APEX20K

Структура соединений приведена на **Рис. 1.51**.

Подобно ПЛИС семейства MAX, в состав ПЛИС семейства APEX20K входят макроячейки, имеющие программируемую матрицу «И» и параллельный расширитель.

Отличительной особенностью ПЛИС семейства APEX20K являются системные блоки памяти (СБП, ESB — embedded system block), показанные на **Рис. 1.53**.

Отличительной особенностью СБП является то, что он может быть сконфигурирован как контекстно-адресуемая память (т.е. память, адресуемая по ее содержимому) или как двухпортовая память, что существенно расширяет возможности применения.

На **Рис. 1.54** приведена организация блока ввода/вывода (БВВ). Каждый БВВ может быть сконфигурирован в соответствии с различными уровнями логических сигналов, существует также два блока, поддерживающих скоростной интерфейс LVDS. Такая организация ввода/вывода позволяет использовать ПЛИС в системах с различными уровнями сигналов. Как и семейства FLEX, ПЛИС семейства APEX20K поддерживает спецификацию уровней PCI. В **Табл. 1.18** приведены временные параметры ПЛИС семейства APEX20K.

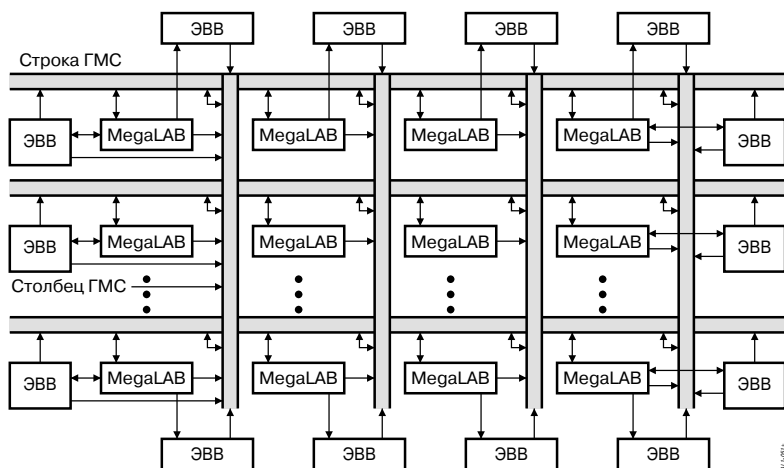


Рис. 1.51. Структура соединений ПЛИС семейства APEX20K

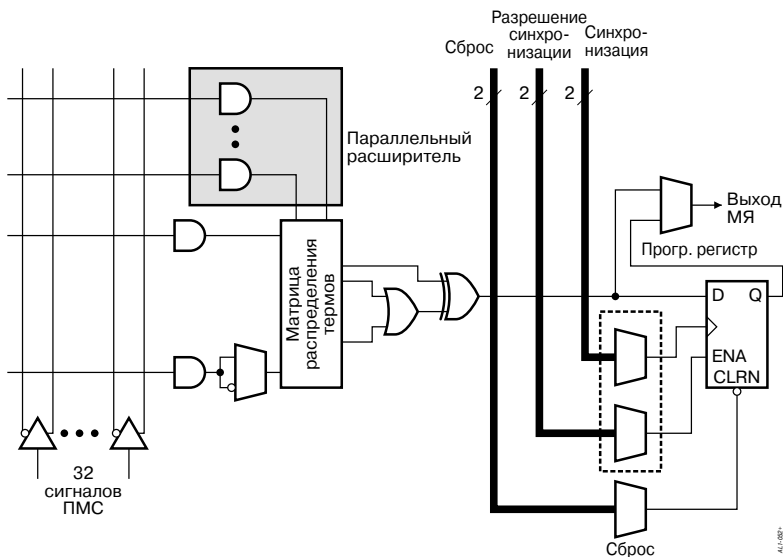


Рис. 1.52. Макроячейка ПЛИС семейства APEX20K

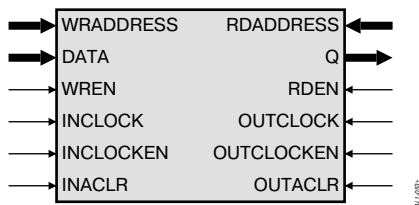


Рис. 1.53. Системный блок памяти

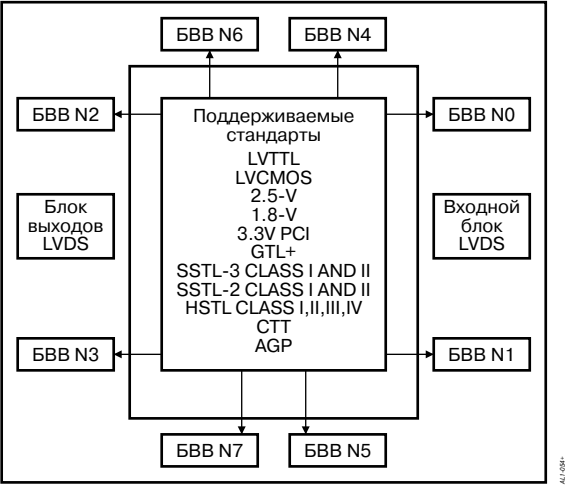


Рис. 1.54. Организация ввода/вывода

Таблица 1.18. Временные параметры ПЛИС семейства APX20K

Обозначение	Параметр	Значение для EP20K100–1 [нс]	
		min	max
t_{INSU}	Время установки глобального тактового импульса	2.1	—
t_{INH}	Время удержания данных относительно глобального тактового импульса	0.0	—
t_{OUTCO}	Время задержки выходных данных относительно глобального тактового импульса	2.0	4.0
t_{ISUBIDIR}	Время установки двунаправленного вывода относительно глобального тактового импульса	1.1	—
t_{INHBIDIR}	Время удержания двунаправленного вывода относительно глобального тактового импульса	0.0	—
$t_{\text{OUTCOBIDIR}}$	Время задержки выходных данных на двунаправленном выводе относительно глобального тактового импульса	2.0	4.0
t_{XZBIDIR}	Задержка перехода выходного буфера в третье состояние	—	4.8
t_{ZXBIDIR}	Задержка перехода выходного буфера из третьего состояния	—	5.9

1.8. Семейство Mercury

Новое семейство Mercury было выпущено на рынок фирмой «Altera» в начале 2001 года. Данные устройства представляют собой принципиально новый класс ПЛИС, ориентированных на приложения для коммуникаций и обработки сигналов.

Отличительными чертами семейства Mercury являются:

- интегрированные высокоскоростные приемопередатчики, поддерживающие синхронное восстановление данных (Clock Data Recovery, CDR), позволяющие организовать передачу и прием данных на скоростях до 1.25 Гбит/с (Gbps);

- архитектура логического элемента основана на традиционной таблице перекодировок, оптимизирована под высокие темпы обработки данных;

- новая архитектура быстрых межсоединений внутри кристалла для уменьшения задержек в критических путях;

- элементы ввода/вывода поддерживают множество стандартных интерфейсов обмена данными;

- ПЛИС семейства Mercury содержат до 14 400 логических элементов;

- тактирование с умножением частоты обеспечивается схемами фазовой автоподстройки частоты (Phase-Locked Loop, PLL) с программируемым коэффициентом умножения частоты и сдвигом фазы опорного сигнала;

- ПЛИС семейства Mercury имеют до 12 выходов ФАПЧ;

- специализированная схема для реализации аппаратных умножителей (как знаковых, так и беззнаковых), позволяющая реализовать умножители разрядностью до 16×16 ;

- ПЛИС семейства Mercury имеют встроенные системные блоки памяти (Embedded System Blocks, ESBs), на которых возможно реализовать разнообразные устройства памяти, такие, как четырехпортовое ОЗУ (quad-port RAM), двунаправленные двухпортовые ОЗУ (bidirectional dual-port RAM), буферы FIFO, память с адресацией по содержимому (Content-Addressable Memory, CAM);

- каждый системный блок памяти содержит 4096 бит и может быть сконфигурирован для использования как два однонаправленных двухпортовых ОЗУ по 2048 бит каждое.

В **Табл. 1.19** приведены основные характеристики ПЛИС семейства Mercury.

Таблица 1.19. Основные характеристики ПЛИС семейства Mercury

Параметр	EP1M120	EP1M350
Логическая емкость, количество эквивалентных вентиляей	120 000	350 000
Число высокоскоростных дифференциальных каналов ввода/вывода	8	18
Число логических элементов	4 800	14 400
Число системных блоков памяти	12	28
Объем встроенной памяти [бит]	49 152	114 688
Число пользовательских выводов	303	486

Особенности элементов ввода/вывода — поддержка огромного числа стандартных интерфейсов, таких как LVTTTL, PCI (до 66 МГц), PCI-X (до 133 МГц), 3.3-B AGP, 3.3-V SSTL, 3- и 2.5-B SSTL-2, GTL+, HSTL, CTT, LVDS, LVPECL и PCML.

Высокоскоростной дифференциальный интерфейс (High-Speed Differential Interface, HSDI) с встроенной синхронной схемой синхронного восстановления данных обеспечивает скорость передачи данных до 1.25 Гигабита в секунду для уровней LVDS, LVPECL и PCML. При использовании внешней синхронизации обеспечивается скорость до 840 Мбит/с для уровней LVDS, LVPECL и PCML.

Возможно использовать до 18 дифференциальных каналов на вход и до 18 на выход, поддерживая уровни LVDS, LVPECL или PCML. Гибкая встроенная схема LVDS TM обеспечивает производительность обмена до 332 Мбит/с по 100 каналам (для устройства EP1M350). Элементы ввода/вывода поддерживают удвоенную скорость обмена данными (Double Data Rate I/O, DDRIO), что позволяет работать с DDR SDRAM, памятью с нулевым возвращением шины (Zero Bus Turnaround, ZBT SRAM) и памятью с четырехкратным ускорением обмена (Quad Data Rate, QDR SRAM). Напряжение питания ПЛИС семейства Mercury составляет 1.8 В для внутренних ячеек (V_{CCINT}) и поддержку различных уровней для напряжения питания ЭВВ (V_{CCIO}) — 1.5, 1.8, 2.5, 3.3 В. Для работы с 5-вольтовыми схемами необходимы внешние подстраивающие резисторы. Структура межсоединений имеет многоуровневый характер, что обеспечивает хорошую трассируемость проекта.

В общем, можно сказать, что ПЛИС семейства Mercury интегрируют в себе встроенные дифференциальные ЭВВ, поддерживающие скоростной обмен данными и оптимизированную внутреннюю архитектуру. Архитектура ПЛИС семейства Mercury специально оптимизирована под использование мегафункций. При этом производительность этих устройств значительно возросла по сравнению с другими семействами (см. **Табл. 1.20**), что делает их очень привлекательными в сигнальных задачах.

Таблица 1.20. Производительность ПЛИС семейства Mercury

Приложение	Ресурсы ПЛИС		Производительность
	ЛЭ	СБП	
16-разрядный загружаемый счетчик	16	0	333 МГц
32-разрядный загружаемый счетчик	32	0	333 МГц
32-разрядный накапливающий сумматор	32	0	333 МГц
Мультиплексор 32 в 1	27	0	1.7 нс
32×64 FIFO	103	2	311 МГц

ПЛИС семейства Mercury построены по технологии КМОП SRAM и могут быть сконфигурированы либо с внешнего ПЗУ, либо от контроллера системы. Безусловно, поддерживаются все функции программирования в системе. В качестве средства разработки проектов на ПЛИС семейства Mercury используется пакет Quartus II.

Архитектура ПЛИС семейства Mercury состоит из рядов ЛЭ, выполняющих функции стандартной логики (row-based logic array), и рядов встроенных системных блоков памяти (row-based embedded system array), которые могут быть также сконфигурированы для реализации сложных функций.

Внутренние межсоединения в ПЛИС семейства Mercury представляют набор вертикальных и горизонтальных трасс различной длины и обеспечивающих различную скорость распространения сигнала. В отличие от ПЛИС других семейств, в ПЛИС семейства Mercury элементы ввода/вывода расположены по всей площади сигнала, что обеспечивает возможность реализации скоростного обмена. Каждый вывод управляется своим ЭВВ.

На **Рис. 1.55** приведена структура ПЛИС семейства Mercury. ПЛИС семейства Mercury имеют четыре выделенные линии тактовых сигналов (dedicated clock) и шесть выделенных глобальных линий сигналов управления. Следует заметить, что в ПЛИС семейства Mercury выделенные линии могут управляться внутренними сигналами, что значительно облегчает построение схем синхронизации (теперь не нужно «гонять» сигнал из ПЛИС в ПЛИС для реализации делителей частоты). Кроме того, цепи синхронизации может использовать ФАПЧ для умножения частоты.

В ПЛИС семейства Mercury реализован встроенный высокоскоростной дифференциальный интерфейс, позволяющий реализовать обмен со скоростью до 1.25 Гигабит в секунду. Структурная схема синхронного приема данных приведена на **Рис. 1.56**. Структурная схема синхронной передачи данных приведена на **Рис. 1.57**.

Схема высокоскоростного дифференциального интерфейса удобна для таких приложений, как Gigabit Ethernet, ATM, SONET, RapidIO, POS-PHY Level 4, Fibre Channel, IEEE Std. 1394, HDTV, SDTV. Схема поддерживает режимы с внешней синхронизацией (source-synchronous mode) и синхронным восстановлением данных (Clock Data Recovery (CDR) mode) (**Рис. 1.58**).

Каждый логический блок содержит 10 логических элементов, цепи переноса ЛЭ, схему аппаратного умножения (multiplier circuitry), сигналы управления и два типа межсоединений — локальную матрицу (local interconnect) и быстрые цепи для объединения таблиц перекодировки (Fast LUT). Структура ЛБ ПЛИС семейства Mercury приведена на **Рис. 1.59**.

Структурная схема управляющих сигналов ЛБ (LAB Control Signals) приведена на **Рис. 1.60**. Каждый ЛБ имеет выделенные тактовый сигнал (clock), сигнал разрешения тактового импульса (clock enable), асинхронный сброс (asynchronous clear), асинхронную предустановку (asynchronous preset), асинхронную загрузку (asynchronous load), синхронный сброс (synchronous clear) и сигнал синхронной загрузки (synchronous load). Одновременно может быть задействовано до 6 сигналов управления.

Структура логического элемента ПЛИС семейства Mercury напоминает структуру ЛЭ семейств FLEX и APEX. Каждый ЛЭ содержит четырехходовую ТП, цепи переноса и каскадирования, триггер. Единственное серьезное отличие — дополнительные цепи переноса, что позволяет строить высокопроизводительные арифметические устройства. Структура ЛЭ ПЛИС семейства Mercury приведена на **Рис. 1.61**.

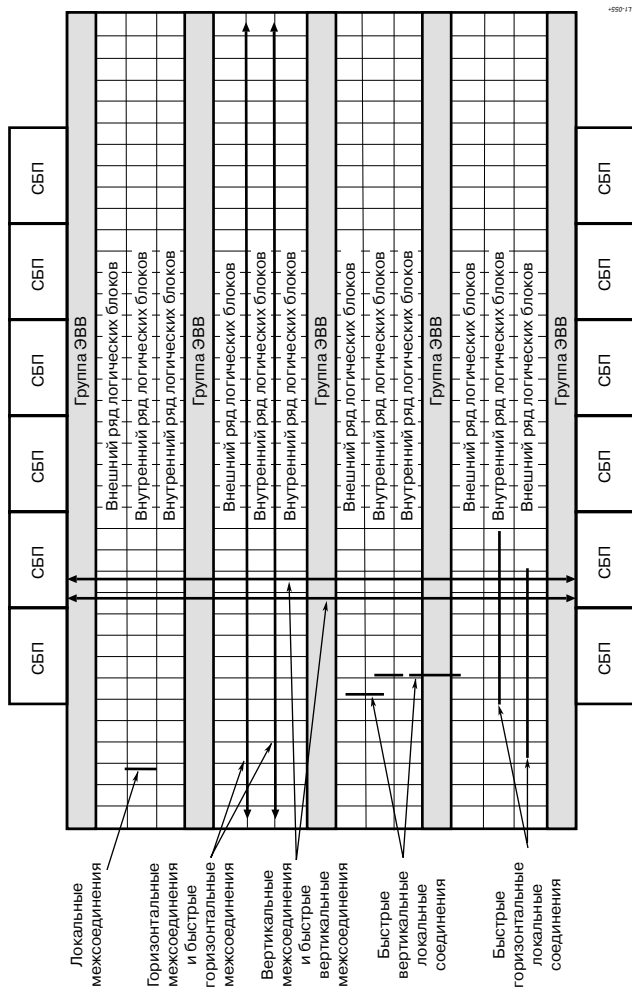


Рис. 1.55. Структура ПЛИС семейства Mercur

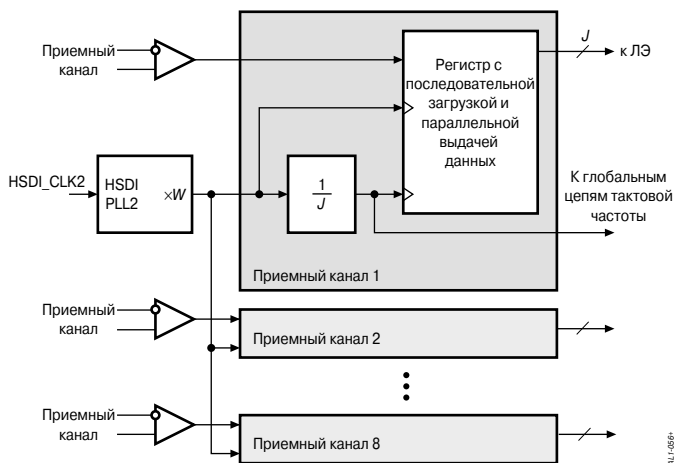


Рис. 1.56. Синхронный прием данных

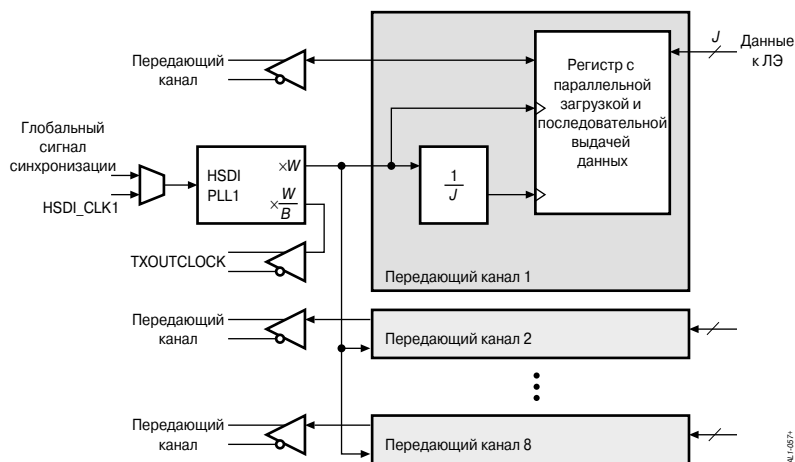


Рис. 1.57. Синхронная передача данных

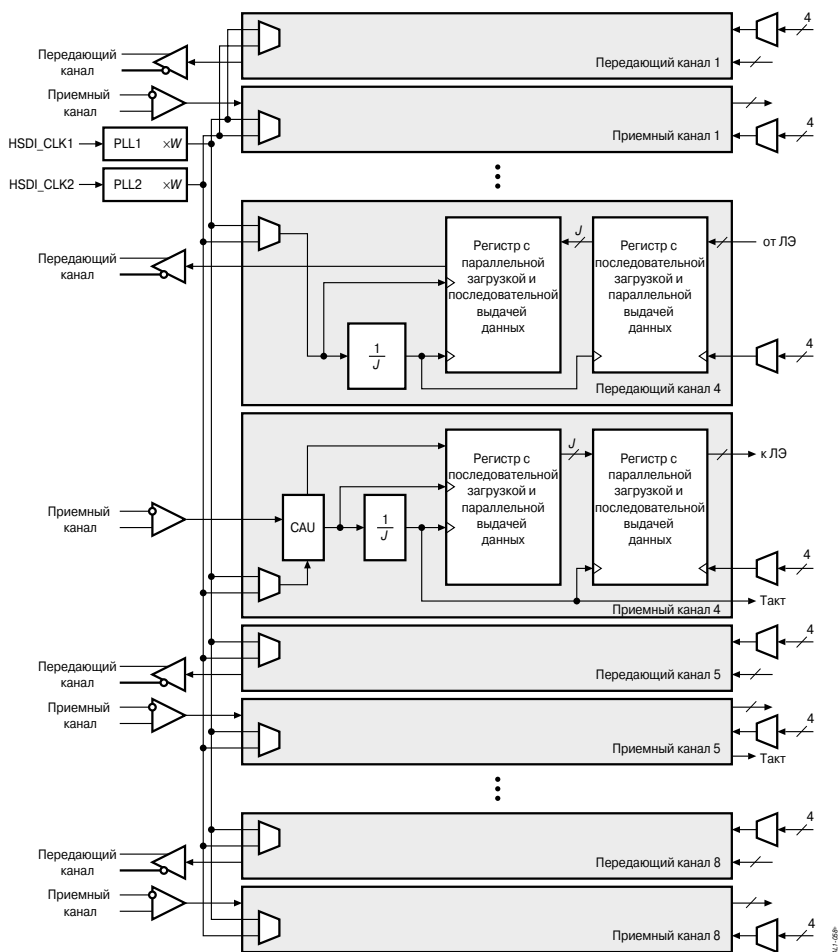
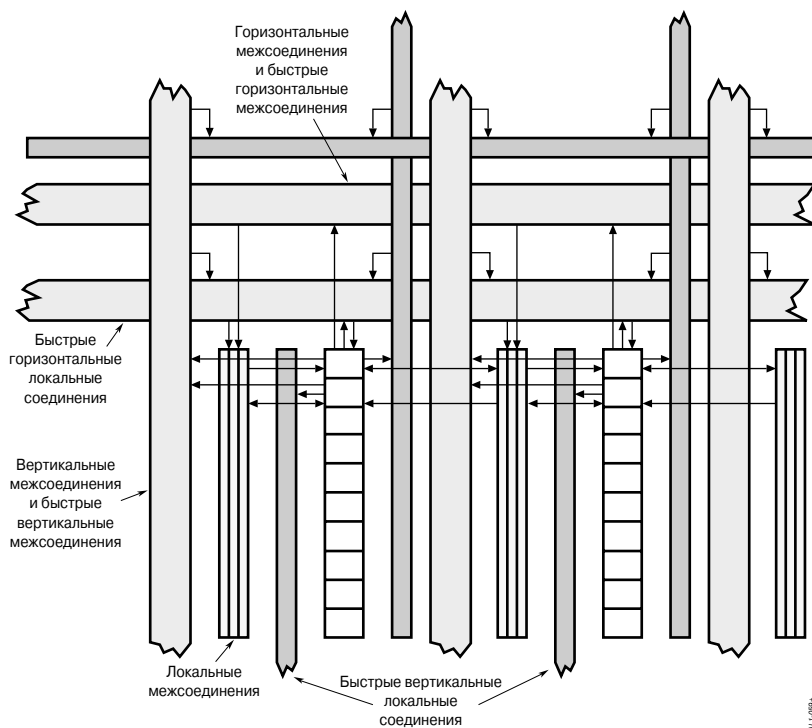


Рис. 1.58. Режим синхронного восстановления данных



AL1-089

Рис. 1.59. Логический блок ПЛИС семейства Mercury

Каждый ЛЭ может быть сконфигурирован в трех режимах: обычном (normal), арифметическом (arithmetic) и режиме аппаратного умножителя (multiplier). Нормальный и арифметический режимы во многом аналогичны соответствующим режимам работы ЛЭ ПЛИС семейств FLEX и APEx, отличие только в наличии дополнительных цепей переноса, что позволяет строить многоразрядные быстрые арифметические узлы. В режиме перемножителя (multiplier mode) можно реализовать скоростной перемножитель размерностью до 16×16 . Структурная схема этого режима приведена на **Рис. 1.62**.

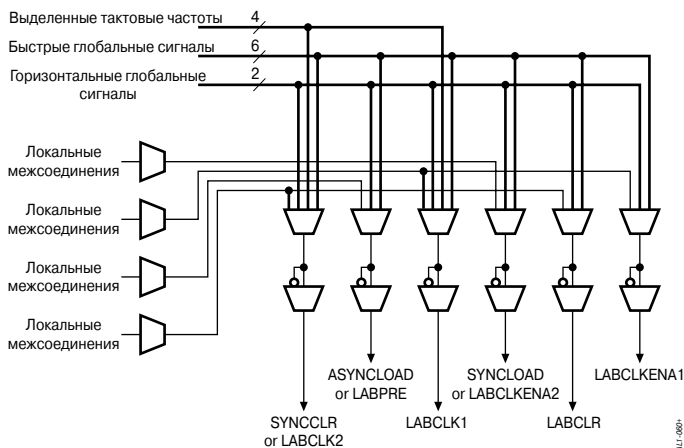


Рис. 1.60. Управляющие сигналы ЛБ ПЛИС семейства Меркурий

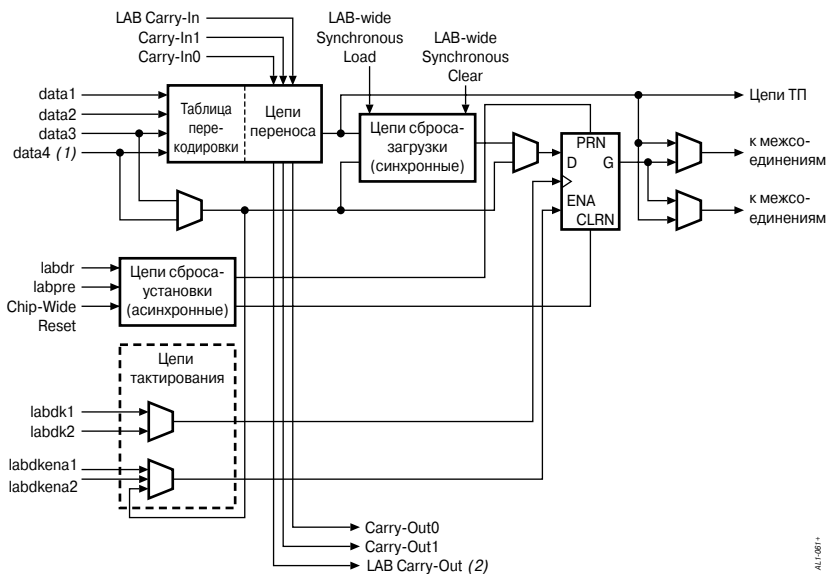


Рис. 1.61. Структура ЛЭ ПЛИС семейства Меркурий

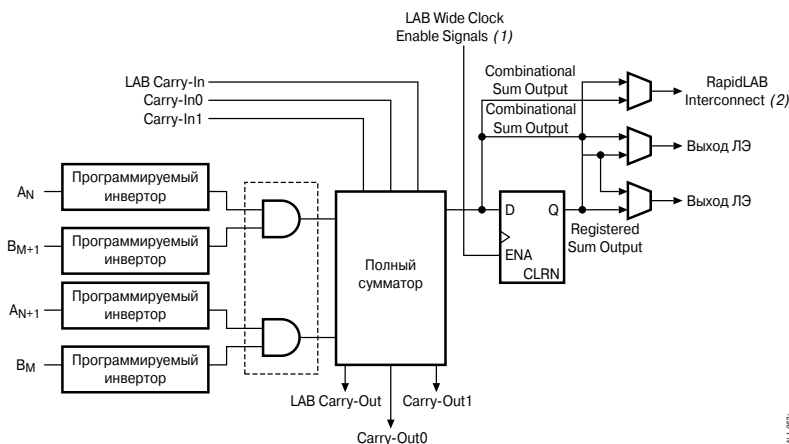


Рис. 1.62. Структура ЛЭ ПЛИС семейства Mercury в режиме перемножителя сигналов

Для построения перемножителей в ПЛИС семейства Mercury используется бинарное дерево. На **Рис. 1.63** приведена его структура. На двоичном дереве реализуется произведение 16-разрядных чисел $A[15:0]$ и $B[15:0]$, результат первой ступени — 16 шестнадцатиразрядных частичных произведений $A[15:0]B[15]$, $A[15:0]B[14]$, . . . $A[15:0]B[0]$. Частичные произведения группируются в пары и суммируются на второй ступени и т.д.

ПЛИС семейства Mercury имеют многоуровневую структуру межсоединений (Multi-Level FastTrack Interconnect), которая обеспечивает высокие скоростные характеристики ПЛИС.

Многоуровневая структура позволяет трассировать ответственные цепи непрерывно, что значительно повышает быстродействие. Ресурсы трассировки включают:

- горизонтальные линии соединений (row interconnect);
- приоритетные горизонтальные линии соединений (priority row interconnect) для быстрых сигналов;
- горизонтальные линии (RapidLAB), которые пересекают область в 10 ЛБ в центре кристалла (**Рис. 1.64**).

Вертикальные линии соединений представлены следующими видами:

- столбцы матрицы соединений (column interconnect);

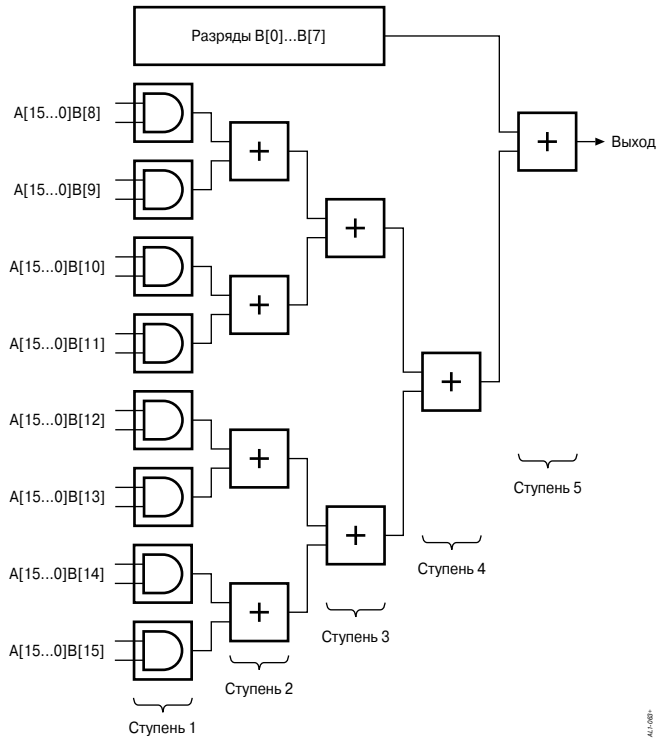


Рис. 1.63. Бинарное дерево умножения сигналов

— приоритетные столбцы (priority column interconnect) для быстрых сигналов;

— вертикальные межсоединения (Leap line) (**Рис. 1.65**).

Кроме того, внутри каждого ЛБ имеются специальные ресурсы трассировки, названные fastLUT, которые объединяют в единую цепь комбинационный выход таблицы перекодировки ЛЭ, не используя локальную матрицу соединений. Структура цепей fastLUT представлена на **Рис. 1.66**.

Таким образом, архитектура межсоединений ПЛИС семейства Mercury представляет собой сложную многоуровневую структуру, представление о которой дает **Табл. 1.21**.

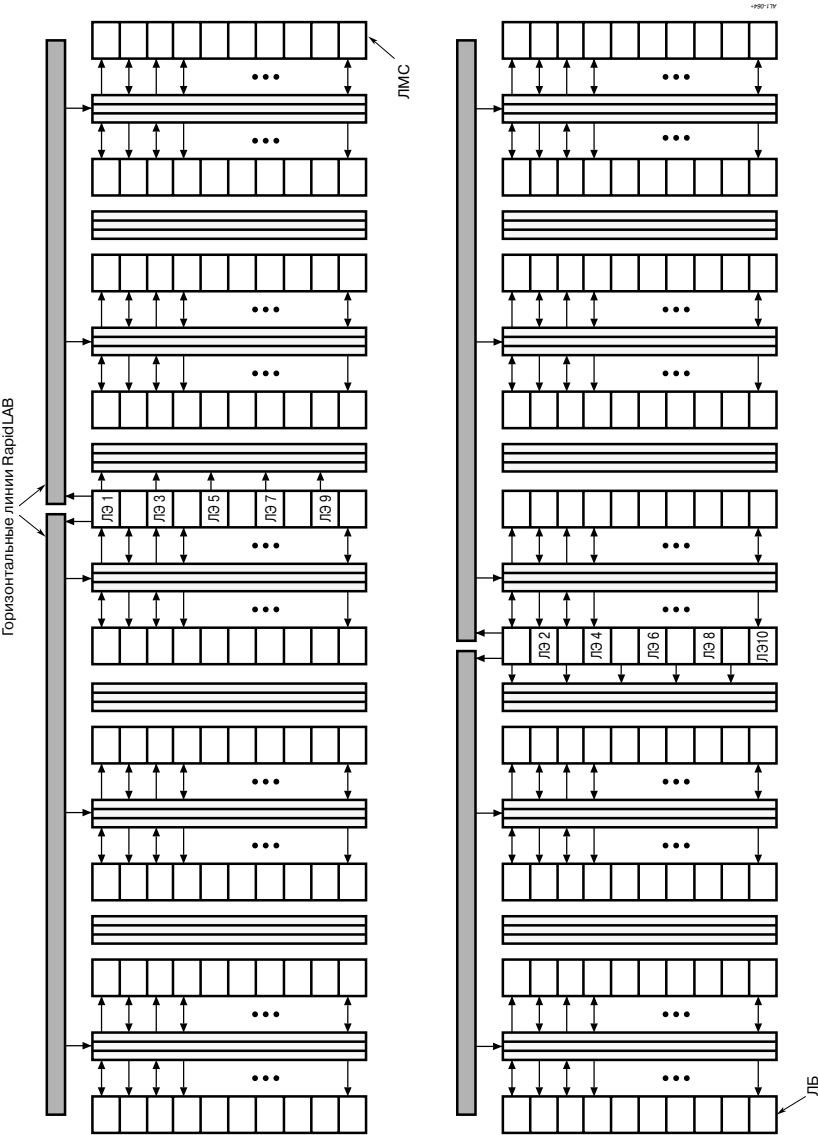


Рис. 1.64. Горизонтальные линии трассировки RapidLAB

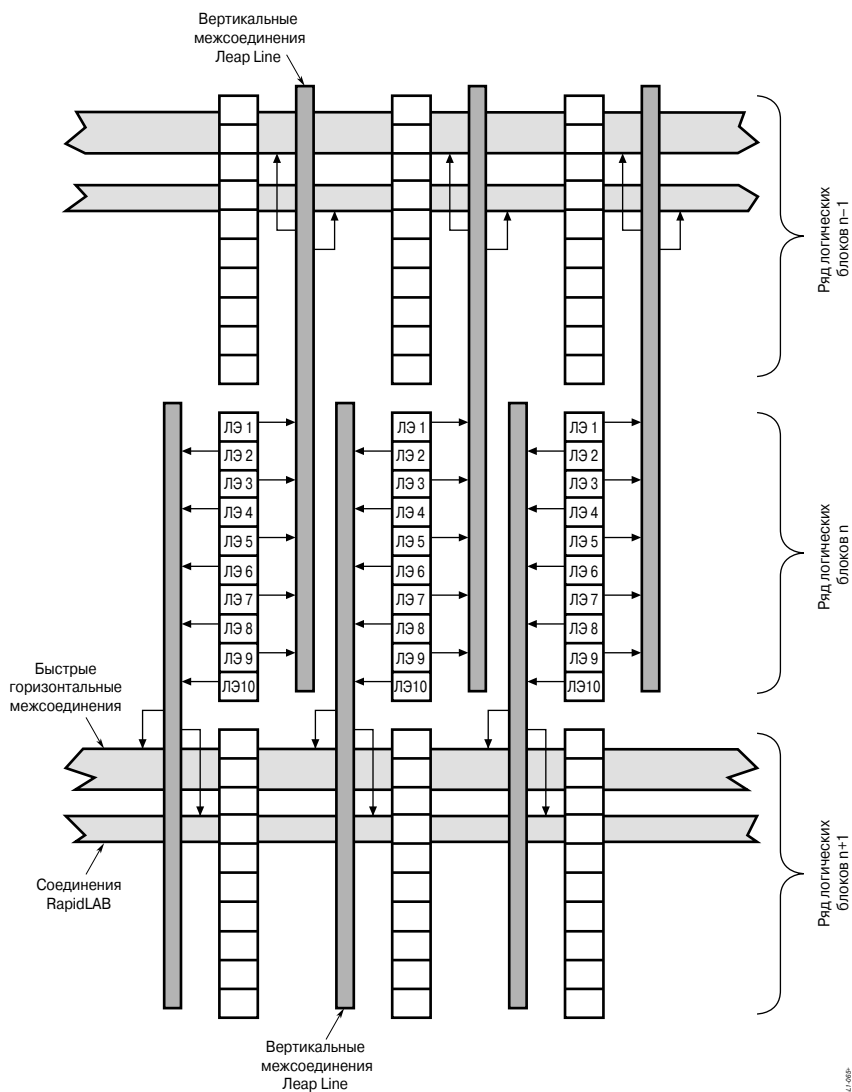


Рис. 1.65. Вертикальные межсоединения (Leap line)

Таблица 1.21. Архитектура межсоединений ПЛИС семейства Mercury

Источник	Путь										
	Логический элемент	Локальные линии соединений	Элемент ввода/вывода	Горизонтальные линии соединений СБП	Системный блок памяти	Горизонтальные межсоединения	Приоритетные горизонтальные межсоединения	Линии соединений быстродействующих ЛБ	Столбцы матрицы	Приоритетные столбцы матрицы	Вертикальные межсоединения
Логический элемент	+	+				+	+	+	+	+	+
Локальные линии соединений	+		+								
Элемент ввода/вывода		+				+	+		+	+	
Горизонтальные линии соединения СБП					+						
Системный блок памяти				+					+	+	+
Горизонтальные межсоединения		+									
Приоритетные горизонтальные межсоединения		+									
Линии соединений быстродействующих ЛБ	+	+									
Столбцы матрицы				+		+	+		+		
Приоритетные столбцы матрицы				+			+	+	+	+	
Вертикальные межсоединения				+		+	+	+	+		

Системный блок памяти позволяет реализовать различные типы блоков памяти — двух- и четырехпортовые ОЗУ, ПЗУ, буферы FIFO и контекстно адресуемую память. Структура системного блока памяти в режиме четырехпортового ОЗУ приведена на **Рис. 1.67**.

Различные варианты конфигурации системного блока памяти приведены на **Рис. 1.68**.

Структура элемента ввода/вывода ПЛИС семейства Mercury приведена на **Рис. 1.69**. ЭВВ ПЛИС семейства Mercury содержит двунаправленный буфер (bidirectional I/O buffer) и 3 регистра (registers) для обеспечения двунаправленного ввода/вывода данных. Элементы ввода/вывода в ПЛИС семейства Mercury объединяются в группы (I/O rowbands), что позволяет обеспечить кратчайший путь при трассировке кристалла (**Рис. 1.70**).

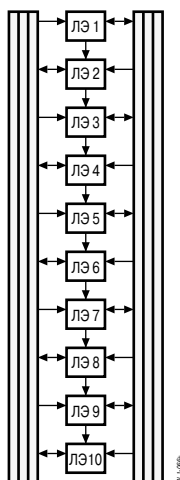


Рис. 1.66. Структура цепей fastLUT

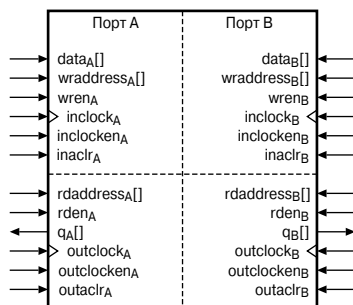
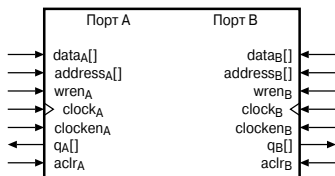
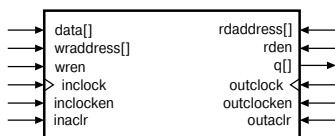


Рис. 1.67. Системный блок памяти

Двухнаправленная двухпортовая память



Двухпортовая память



Однопортовая память

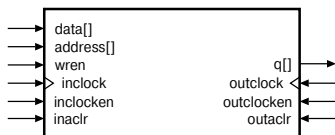


Рис. 1.68. Варианты конфигурации СБП

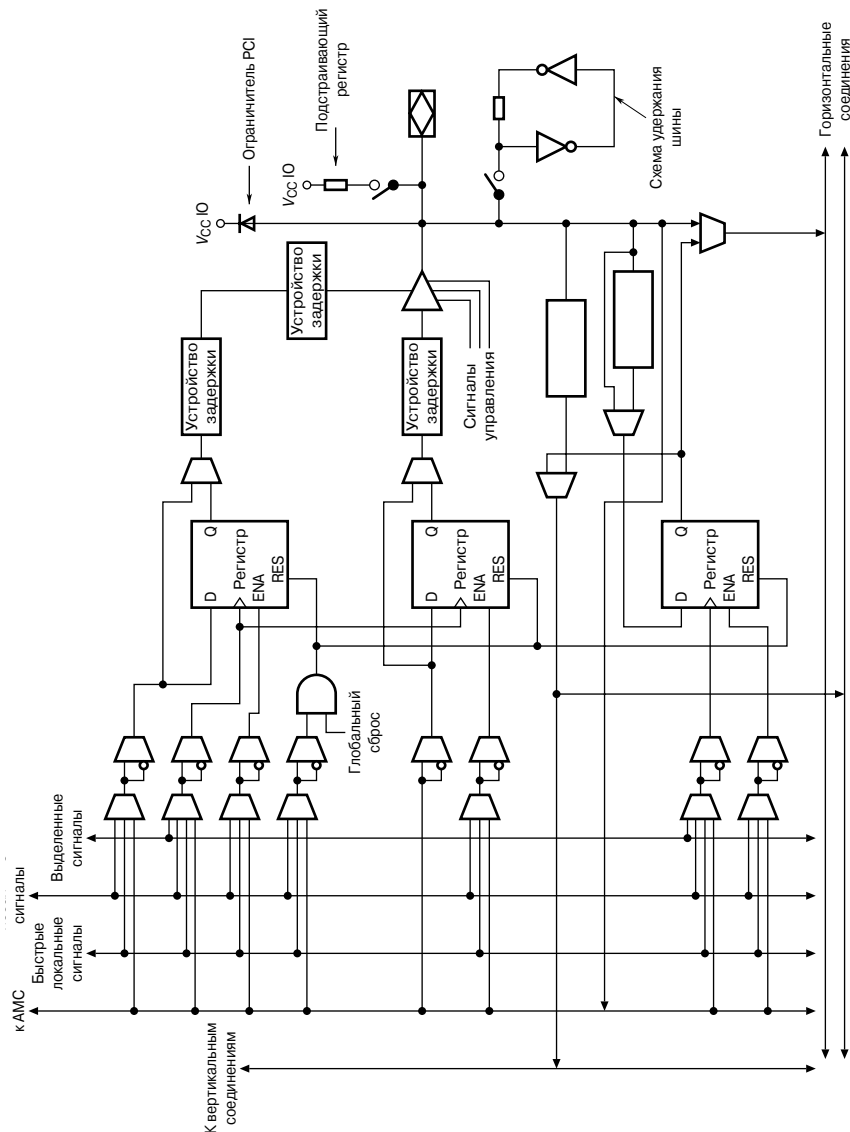


Рис. 1.69. Структура элемента ввода/вывода ПЛИС семейства Mercury

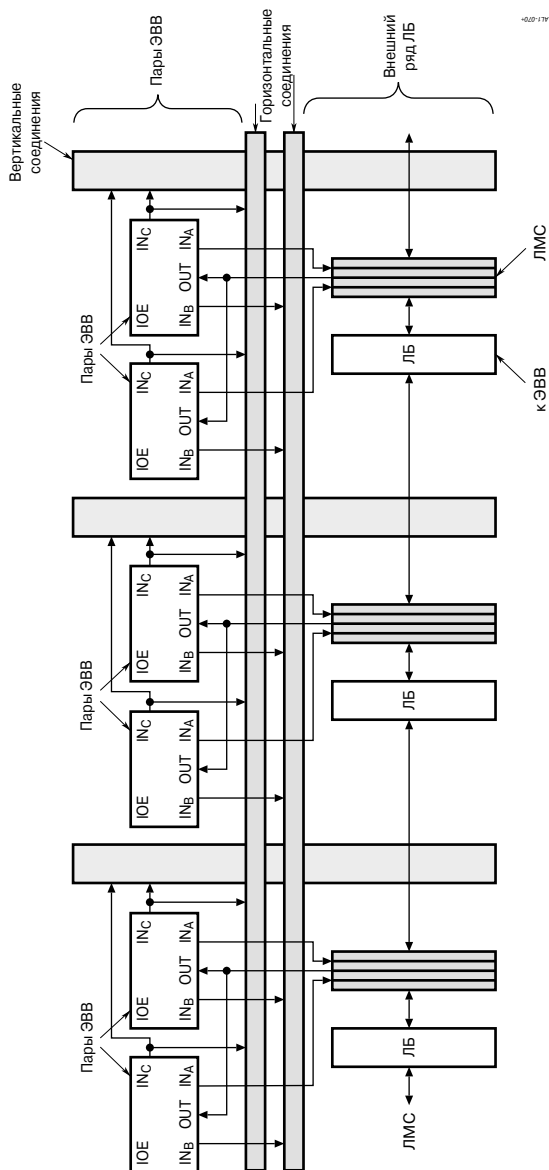


Рис. 1.70. Группы ЭВВ I/O rowbands

ПЛИС семейства Mercury имеют специализированные выделенные линии (FAST1, FAST2, FAST3, FAST4, FAST5 и FAST6), обеспечивающие функции глобального управления.

ЭВВ ПЛИС семейства Mercury поддерживают следующие интерфейсы:

- LVTTTL
- LVCMOS
- 1.8-V
- 2.5-V
- 3.3-V PCI
- 3.3-V PCI-X
- 3.3-V AGP
- LVDS
- LVPECL
- PCML
- GTL+
- HSTL class I and II
- SSTL-3 class I and II
- SSTL-2 class I and II
- CTT

Все ПЛИС семейства Mercury полностью поддерживают периферийное сканирование в соответствии со стандартом IEEE Std. 1149.1 — 1990 (JTAG).

1.9. Семейство ACEX

Выпущенные фирмой «Altera» устройства ACEX представляют собой дешевую и быстродействующую альтернативу ПЛИС семейства FLEX10K. Они настолько схожи в архитектуре, что здесь приведем только краткие сведения о новом семействе.

В ПЛИС семейства ACEX имеются встроенные блоки памяти, позволяющие реализовать двухпортовую память до 16 разрядов данных. ПЛИС семейства ACEX имеют эквивалентную емкость до 100 000 вентилей (Табл. 1.22).

Таблица 1.22. Основные сведения о ПЛИС семейства ACEX

Параметр	EP1K10	EP1K30	EP1K50	EP1K100
Логическая емкость эквивалентных вентилей	10 000	30 000	50 000	100 000
Максимальное число системных элементов	56 000	119 000	199 000	257 000
Число логических элементов	576	1 728	2 880	4 992
Число логических блоков	3	6	10	12
Максимальный объем ОЗУ [бит]	12 288	24 576	40 960	49 152
Максимальное число выводов пользователя	130	171	249	333

Таблица 1.23. Производительность ПЛИС семейства ACEX

Приложение	Ресурсы ПЛИС		Производительность		
	ЛЭ	СБП	Градации скорости		
			–1	–2	–3
16-разрядный загружаемый счетчик	16	0	200 МГц	188 МГц	128 МГц
16-разрядный накапливающий сумматор	16	0	200 МГц	188 МГц	128 МГц
Мультиплексор 32 в 1	10	0	3.2 нс	4.3 нс	5.5 нс
Мультиплексор с 3-каскадной обработкой данных	544	0	93 МГц	86 МГц	64 МГц
256×16 ОЗУ (скорость считывания)	0	1	212 МГц	181 МГц	131 МГц
256×16 ОЗУ (скорость записи)	0	1	142 МГц	128 МГц	94 МГц

Приложение	Ресурсы ПЛИС		Производительность		
	ЛЭ		Градации скорости		
			–1	–2	–3
16-разрядный 8-выводной параллельный фильтр с ограничительной импульсной характеристикой	420		185	175	122
8-разрядный, 512-точечный счетверенный быстродействующий преобразователь функции	1.854		47.4 мкс 100 МГц	57.8 мкс 82 МГц	76.5 мкс 62 МГц
a16450 универсальный асинхронный приемопередатчик	342		66 МГц	57 МГц	44 МГц

Каждый ВБП содержит 4 096 бит памяти. Поддерживается интерфейс с уровнями 2.5, 3.3 и 5.0 В. Элементы ввода/вывода обеспечивают темп обмена данными до 250 МГц. В **Табл. 1.23** приведены примеры производительности ПЛИС семейства ACEX на различных проектах.

На **Рис. 1.71** приведена структурная схема ПЛИС семейства ACEX. Как видим, фактически повторена архитектура ПЛИС семейства FLEX10K, но с пониженным напряжением питания и потреблением за счет применения новой технологии изготовления микросхем.

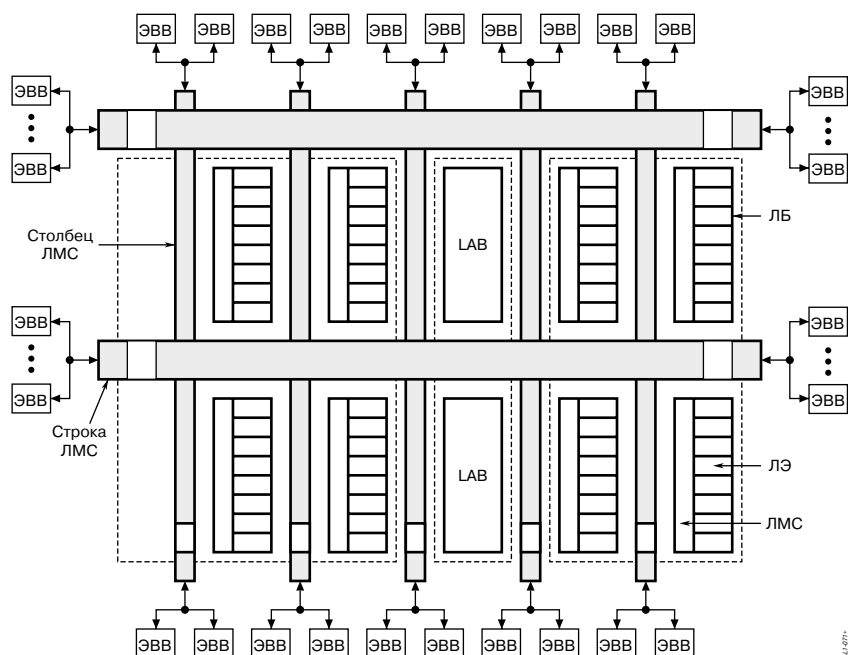


Рис. 1.71. Структурная схема ПЛИС семейства ACEX

1.10. Конфигурационные ПЗУ

Для хранения конфигурационной информации ПЛИС используются последовательные ПЗУ. На Рис. 1.72 и 1.73 показаны конфигурационные ПЗУ фирмы «Altera». В Табл. 1.24 приведены основные данные по конфигурации различных ПЛИС с помощью конфигурационных ПЗУ. При необходимости используется каскадное включение нескольких ПЗУ. На Рис. 1.74—1.78 приведены схемы включения конфигурационных ПЗУ и ПЛИС различных семейств. На Рис. 1.79 приведены временные диаграммы конфигурации ПЛИС.

Таблица 1.24. Типы ПЛИС и соответствующие им конфигурационные ПЗУ

ПЛИС	ПЗУ
EP20K100	EPC2
EP20K200	2×EPC2
EP20K400	3×EPC2
EPF10K10, EPF10K10A	EPC2, EPC1, EPC1441
EPF10K20	EPC2, EPC1, EPC1441
EPF10K30E	EPC2, EPC1
EPF10K30, EPF10K30A	EPC2, EPC1, EPC1441
EPF10K40	EPC2, EPC1
EPF10K50, EPF10K50V, EPF10K50E	EPC2, EPC1
EPF10K70	EPC2, EPC1
EPF10K100, EPF10K100A, EPF10K100B, EPF10K100E	EPC2, EPC1
EPF10K130V	EPC2, 2×EPC1
EPF10K130E	2×EPC2, 2×EPC1
EPF10K200E	2×EPC2, 3×EPC1
EPF10K250E	2×EPC2, 4×EPC1
EPF8282A	EPC1, EPC1441, EPC1064
EPF8282AV	EPC1, EPC1441, EPC1064V
EPF8452A	EPC1, EPC1441, EPC1213
EPF8636A	EPC1, EPC1441, EPC1213
EPF8820A	EPC1, EPC1441, EPC1213
EPF81188A	EPC1, EPC1441, EPC1213
EPF81500A	EPC1, EPC1441
EPF6010A	EPC2, EPC1, EPC1441
EPF6016, EPF6016A	EPC2, EPC1, EPC1441
EPF6024A	EPC2, EPC1, EPC1441

Кроме использования ПЗУ ПЛИС можно конфигурировать с использованием контроллера системы, в которой применена ПЛИС. В Табл. 1.25 приведены возможные режимы конфигурации ПЛИС.

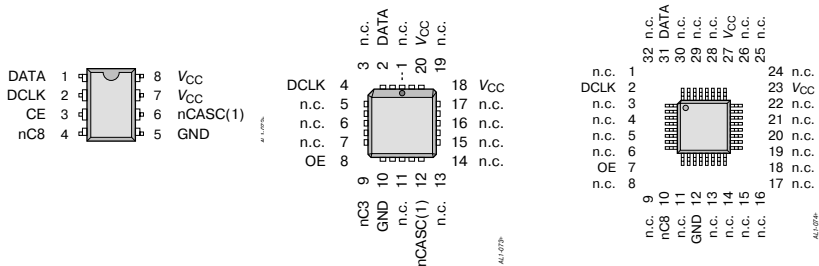


Рис. 1.72. Конфигурационные ПЗУ, программируемые с помощью программатора

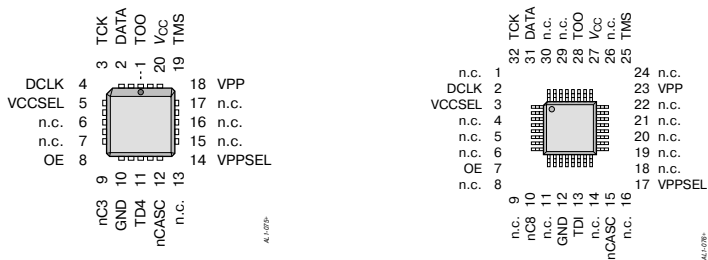


Рис. 1.73. Конфигурационные ПЗУ EPC2, программируемые по JTAG

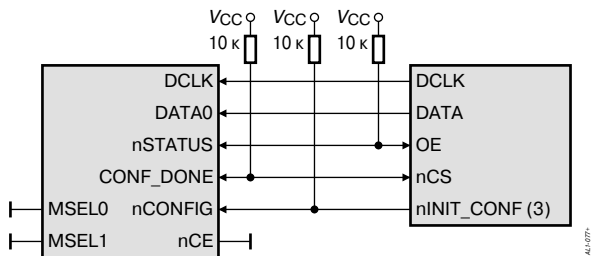


Рис. 1.74. Конфигурация ПЛИС семейств FLEX6000, FLEX10K, APEX20K при помощи ПЗУ EPC2, EPC1, EPC1441

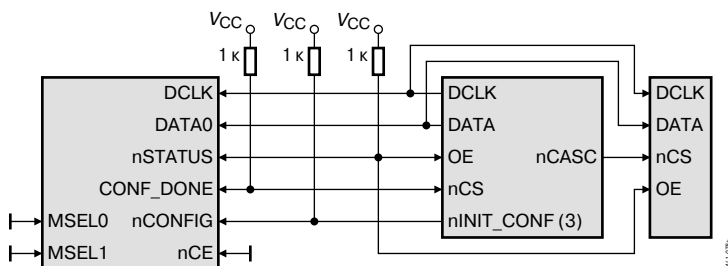


Рис. 1.75. Конфигурация ПЛИС семейств FLEX6000, FLEX10K, APEX20K при помощи двух ПЗУ EPC2, EPC1, EPC1441

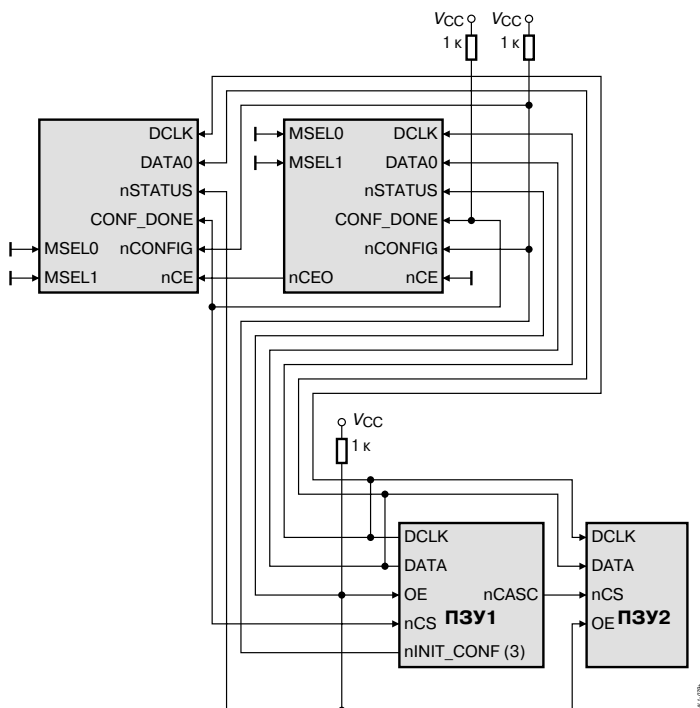


Рис. 1.76. Конфигурация нескольких ПЛИС семейств FLEX6000, FLEX10K, APEX20K при помощи ПЗУ EPC2, EPC1, EPC1441

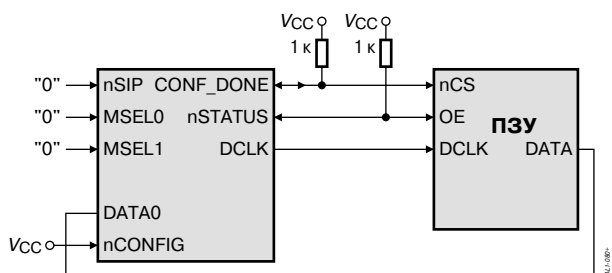


Рис. 1.77. Конфигурация ПЛИС семейства FLEX8000 при помощи ПЗУ EPC1, EPC1441, EPC1213, EPC1064

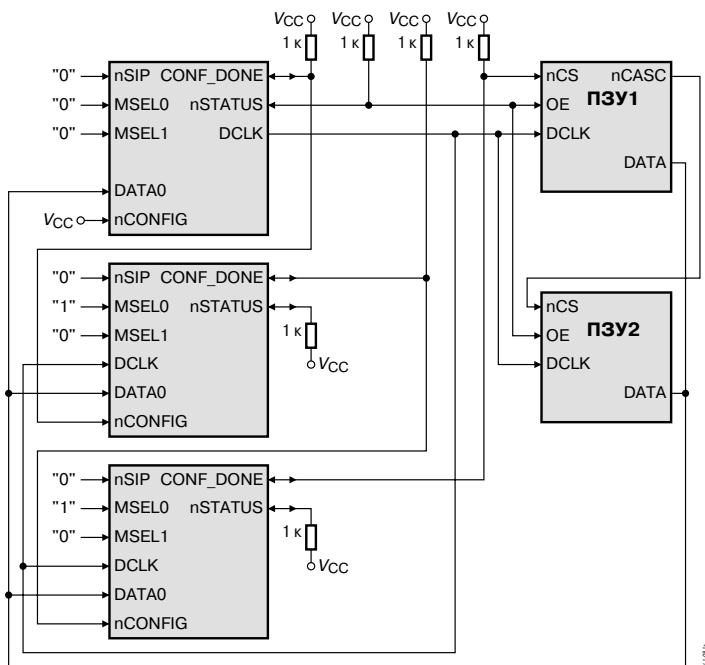


Рис. 1.78. Конфигурация нескольких ПЛИС семейства FLEX8000 при помощи ПЗУ EPC1, EPC1213

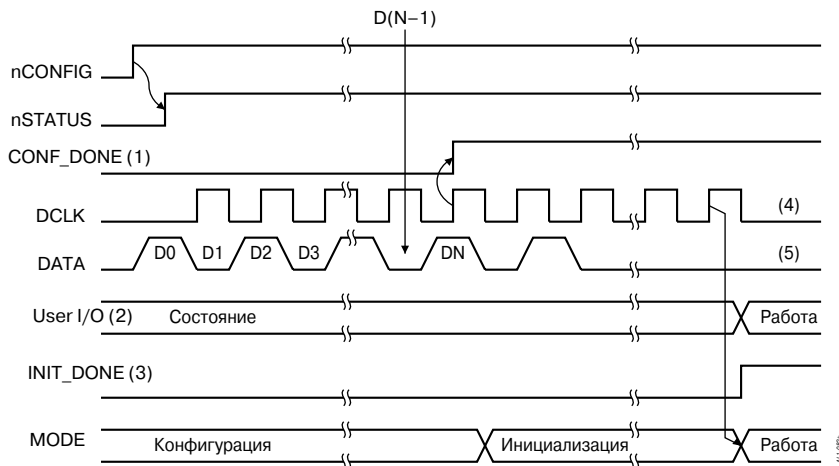


Рис. 1.79. Временные диаграммы конфигурации ПЛИС

Таблица 1.25. Режимы конфигурации ПЛИС

Режим конфигурации	Применение
Passive Serial (PS) Пассивный последовательный	Конфигурация по последовательному синхронному порту микропроцессора (МП) или устройству ByteBlaster, BitBlaster, MasterBlaster
Passive Parallel Synchronous (PPS) Пассивный параллельный синхронный	Конфигурация по параллельному синхронному порту МП
Passive Parallel Asynchronous (PPA) Пассивный параллельный асинхронный	Конфигурация по параллельному асинхронному порту МП. МП адресует ПЛИС как память
Passive Serial Asynchronous (PSA) Пассивный последовательный асинхронный	Конфигурация по последовательному асинхронному порту микропроцессора
JTAG	Используется стандарт IEEE Std. 1149.1—1990

1.11. Программирование и реконфигурирование в системе

Понятие «программирование в системе» относится к тем ПЛИС, которые позволяют произвести программирование непосредственно в составе системы без использования программатора, на смонтированной плате, причем программирование ПЛИС или конфигурационного ПЗУ может производиться многократно. Реконфигурирование в схеме (In-Circuit Reconfigurability, ISR) позволяет произвести перезагрузку данных в ПЛИС, построенной по SRAM технологии «на лету», то есть без выключения питания системы и последующей загрузки новой конфигурации. Свойства ISP и ISR характерны практически для всех современных ПЛИС, выпускаемых ведущими фирмами-производителями. Рассмотрим некоторые особенности архитектуры ПЛИС, позволяющие реализовать механизм ISP.

Как правило, микросхемы CPLD семейств MAX7000S, A, B, E, MAX 3000A, MAX9000 фирмы «Altera» программируются в системе через стандартный четырехконтактный JTAG-интерфейс. Программное обеспечение создает конфигурационную последовательность, которая загружается в ПЛИС с помощью специализированного загрузочного кабеля (ByteBlaster, BitBlaster или MasterBlaster для устройств фирмы «Altera»). Кроме того, для программирования таких ПЛИС можно использовать стандартный JTAG-тестер или простой интерфейс, эмулирующий последовательность команд JTAG.

В качестве примера рассмотрим схему конфигурационного кабеля ByteBlaster MV, опубликованную фирмой «Altera» [3] и предназначенную для программирования ПЛИС семейств конфигурации MAX7000S, A, B, E, MAX3000A, MAX9000. На **Рис. 1.80** приведена принципиальная электрическая схема устройства.

На схеме резисторы, помеченные (1), имеют номинал 100 Ом (реально можно использовать резисторы от 50 до 150 Ом) и предназначены для защиты линий ввода/вывода. Конечно, в экстренных случаях можно обойтись без них, но следует помнить, что «скупой платит четырежды», и не экономить на спичках (менять поврежденную ПЛИС обойдется дороже). Подтягивающие резисторы (pull-up resistors), обозначенные (2), имеют номинал 2.2 кОм (ясно, что на деле от 1 до 3.3 кОм). В качестве шинного формирователя желательно иметь микросхему 74HC244 (рос-

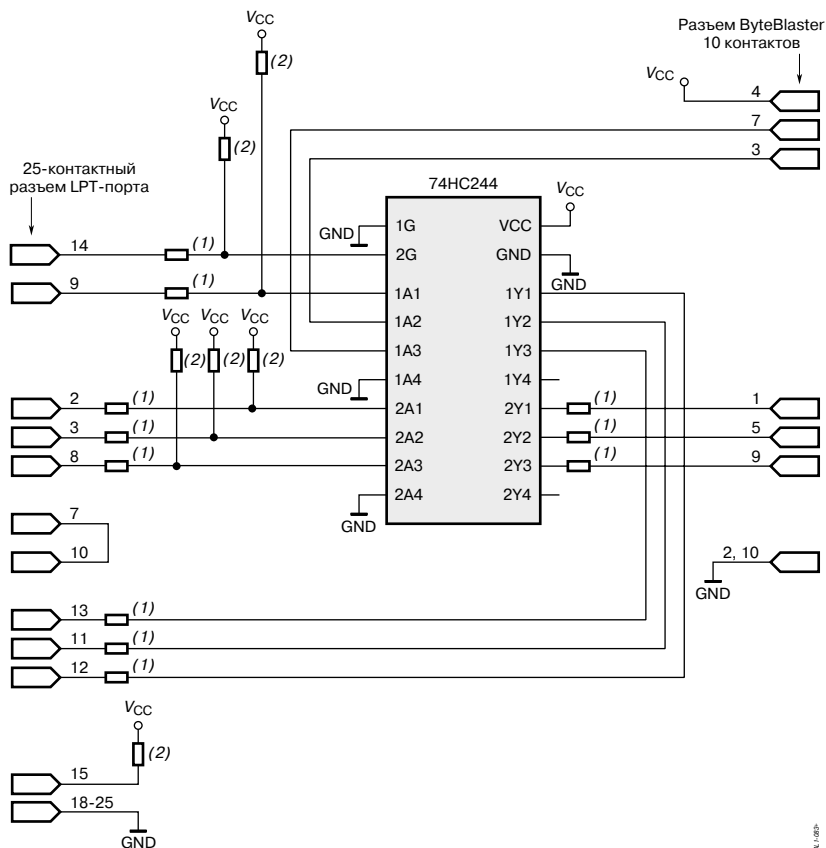


Рис. 1.80. Принципиальная схема загрузочного кабеля ByteBlaster MV

сийский или белорусский аналог — 1564АП5), хотя у автора один из ByteBlasterMV был собран на микросхеме серии 74AC (просто под рукой другого не оказалось) и успешно работает. Если вы работаете только с 5-вольтовыми ПЛИС, то пойдут и серии 1533, и 555 (74ALS и 74LS), но тогда гарантий успешного программирования никто не даст. Вообще говоря, наверное, не следует заниматься изобретением велосипеда, благо, штатная схема не содержит дефицитных компонентов, ее себестоимость

порядка 1.5...2 долларов. Следует помнить, что длина кабеля от параллельного порта до ByteBlasterMV не должна превышать 100...120 см (хотя в стандартном устройстве, поставляемом фирмой «Altera», схема смонтирована непосредственно в корпусе разъема, но это неудобно в работе), длину кабеля от ByteBlasterMV до платы с установленной ПЛИС не стоит делать больше 25 см.

На **Рис. 1.81** приведена распайка разъема устройства ByteBlasterMV. На **Рис. 1.81** все размеры приведены в дюймах. Обычно используют стандартный разъем на 10-жильный ленточный кабель под обжим. Назначение контактов разъема кабеля ByteBlasterMV в различных режимах приведено в **Табл. 1.26**.

Таблица 1. 26. Назначение контактов разъема кабеля ByteBlasterMV

Контакт разъема ByteBlaster	Режим последовательной пассивной конфигурации (PS Mode)		Режим программирования по порту JTAG (JTAG mode)	
	Сигнал	Назначение	Сигнал	Назначение
1	DCLK	Тактовый сигнал	TCK	Тактовый сигнал
2	GND	Сигнальная земля	GND	Сигнальная земля
3	CONF_DONE	Контроль завершения конфигурации	TDO	Данные с ПЛИС
4	V_{CC}	Напряжение питания	V_{CC}	Напряжение питания
5	nCONFIG	Контроль кофигурации	TMS	Контроль автомата JTAG
6	—	Не подключен	—	Не подключен
7	nSTATUS	Состояние конфигурации	—	Не подключен
8	—	Не подключен	—	Не подключен
9	DATA0	Данные в ПЛИС	TDI	Данные в ПЛИС
10	GND	Сигнальная земля	GND	Сигнальная земля

Режим последовательной пассивной конфигурации (PS Mode) применяется для загрузки конфигурации ПЛИС, выполненных по технологии SRAM, таких семейств, как FLEX6000, 8000, 10K, APEX, ACEX. Режим программирования по порту JTAG (JTAG mode) применяется для про-

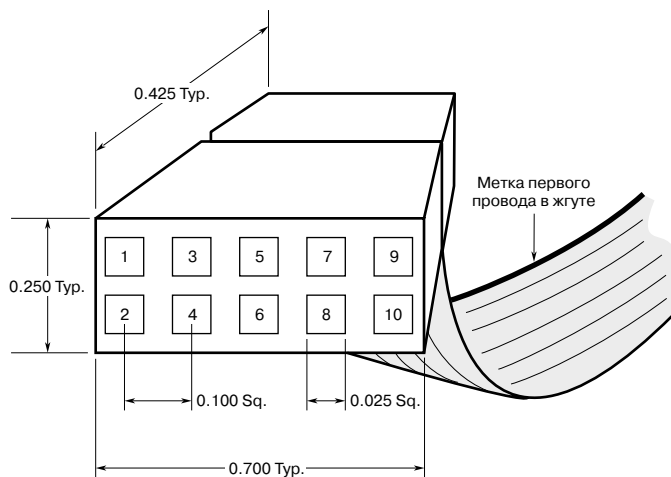


Рис. 1.81. Разъем кабеля ByteBlasterMV

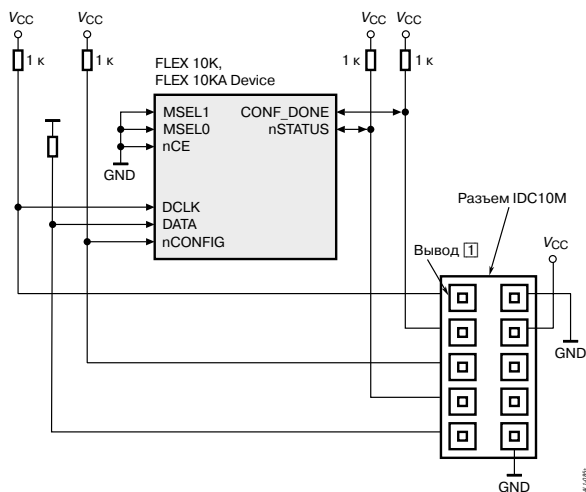


Рис. 1.82. Включение ПЛИС семейства FLEX10K в режиме последовательной пассивной конфигурации с помощью кабеля ByteBlasterMV

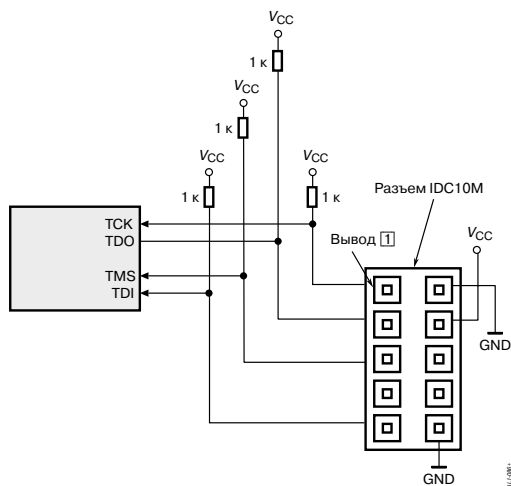


Рис. 1.83. Включение ПЛИС семейств MAX3000, 7000, 9000 в режиме программирования через интерфейс JTAG

граммирования в системе ПЛИС CPLD, а также конфигурационных ПЗУ EPC2 и готовящихся к выпуску EPC4, для загрузки SRAM устройств (правда, реже, чем PS Mode). При работе с устройством необходимо помнить, что все коммутации и подключение кабеля следует проводить при выключенном питании. Питание устройства осуществляется от источника питания системы, в которую установлена ПЛИС. Естественно, что земля должна быть общей. На **Рис. 1.82** и **1.83** приведены примеры включения ПЛИС для конфигурации ПЛИС и программирования.

Глава 2. Система проектирования MAX+PLUS II

2.1. Общие сведения

До последнего времени MAX+PLUS II являлся единственной системой проектирования устройств на ПЛИС фирмы «Altera». Только в 1999 году появилась система проектирования нового поколения Quartus, предназначенная для создания устройств на ПЛИС семейства APEX20K. Программное обеспечение (ПО) системы MAX+PLUS II, представляющее собой единое целое, обеспечивает пользователю управление средой логического проектирования и помогает достичь максимальной эффективности и производительности. Все пакеты работают как на платформе IBM PC, так и на платформах SUN, IBM RISC/6000 и HP9000. В дальнейшем мы будем рассматривать работу на платформе IBM PC.

Для нормальной инсталляции и работы САПР MAX+PLUS II (версия 9.4 вышла в декабре 1999 года) необходима IBM PC, совместимая ЭВМ с процессором не хуже Pentium, объемом ОЗУ не меньше 16 Мбит и свободным местом на жестком диске порядка 150...400 Мбит в зависимости от конфигурации системы. Из собственного опыта можем сказать, что для разработки больших кристаллов на ПЛИС семейства FLEX10K50 и выше желательно иметь не менее 64 Мбит ОЗУ (лучше 128, еще лучше 256, совсем хорошо 384 Мбит и выше) и процессор Pentium II (P-3 реально не дает особого выигрыша). Конечно, можно использовать и более слабые машины, но тогда возрастает время компиляции и увеличивается нагрузка на жесткий диск из-за свопинга. Увеличение объема оперативной памяти и кэша дает лучшие результаты по сравнению с увеличением тактовой частоты процессора. Если не предполагается трассировка больших кристаллов, то вполне хватает 32 Мбит ОЗУ при хорошей скорости компиляции проекта. Что ка-

сается выбора операционной системы, то, без сомнения, лучше использовать Windows NT, хуже — Windows 95 OSR2, плохо — Windows 98, особенно локализованную версию. Очевидно, это связано с тем, что изначально пакет был разработан под Unix и не полностью использует все механизмы Windows. Особенно это заметно при временном моделировании сложных устройств ЦОС, когда перерисовка экрана занимает основное время. Поскольку пакет не локализован, то лучше использовать не локализованные (американскую или паневропейскую) версии Windows.

Во время инсталляции системы MAX+PLUS II создаются два каталога: \maxplus2 и \max2work. Каталог \maxplus2 содержит системное ПО и файлы данных и разбит на подкаталоги, перечисленные в **Табл. 2.1**.

Таблица 2.1. Структура системного каталога \maxplus2 системы MAX+PLUS II

Подкаталог	Описание
\drivers	Содержит драйверы устройств для среды Windows NT (только для инсталляции на платформе PC в среде Windows NT)
\edc	Содержит поставляемые фирмой «Altera» командные файлы (.edc), которые генерируют выходные файлы (.edo) по заказу пользователя для заданных условий тестирования
\lmf	Содержит поставляемые фирмой «Altera» файлы макробиблиотек (.lmf), которые устанавливают соответствие между логическими функциями пользователя и эквивалентными логическими функциями MAX+PLUS II
\max2inc	Содержит Include-файлы (файлы «заголовков») с прототипами функций для разработанных фирмой «Altera» макрофункций. В прототипах функций перечисляются порты (выводы) для макрофункций, реализованных в текстовых файлах проекта (.tdf), написанных на языке AHDL
\max2lib\edif	Содержит примитивы и макрофункции, используемые для пользовательских интерфейсов EDIF
\max2lib\mega_lpm	Содержит мегафункции, в том числе библиотеку функций параметризованных модулей (LPM) и Include-файлы для них с соответствующими прототипами на языке AHDL
\max2lib\mf	Содержит макрофункции пользовательские и устаревшие (74-series)
\max2lib\prim	Содержит поставляемые фирмой «Altera» примитивы
\vhdl\lnn\Altera	Содержит библиотеку фирмы «Altera» с программным пакетом max-plus2. В этот пакет входят все примитивы, мегафункции и макрофункции системы MAX+PLUS II, поддерживаемые языком VHDL
\vhdl\lnn\ieee	Содержит библиотеку ieee пакетов VHDL, в том числе std_logic_1164, std_logic_arith, std_logic_signed и std_logic_unsigned
\vhdl\lnn\std	Содержит библиотеку std с пакетами стандартов и средств ввода/вывода текста, описанных в справочнике по стандартам института IEEE на языке VHDL IEEE Standard VHDL Language Reference Manual

Таблица 2.2. Структура рабочего каталога \max2work системы MAX+PLUS II

Подкаталог	Описание
.\ahdl	Содержит файлы примеров, иллюстрирующих тему «Как использовать язык AHDL» (How to use AHDL) в электронном справочнике (MAX+PLUS II Help) и в руководстве MAX+PLUS II AHDL
.\chiptrip	Содержит все файлы обучающего проекта chiptrip, описанного в руководстве MAX+PLUS II AHDL
.\edif	Содержит все файлы примеров, иллюстрирующих особенности EDIF в электронном справочнике (MAX+PLUS II Help)
.\tutorial	Содержит информационный файл read.me обучающего проекта chiptrip. Все файлы, создающиеся в проекте chiptrip, должны находиться в этом подкаталоге
.\vhdl	Содержит файлы примеров, иллюстрирующих тему «Как использовать язык VHDL» (How to use VHDL) в электронном справочнике (MAX+PLUS II Help) и в руководстве MAX+PLUS II VHDL
.\verilog	Содержит файлы примеров, иллюстрирующих тему «Как использовать язык верификационного протокола Verilog HDL» (How to use Verilog HDL) в электронном справочнике (MAX+PLUS II Help) и в руководстве MAX+PLUS II Verilog HDL

Каталог \max2work содержит файлы обучающей программы и примеры и разделяется на подкаталоги, описанные в **Табл. 2.2**.

Название системы MAX+PLUS II является аббревиатурой от Multiple Array MatriX Programmable Logic User System (Пользовательская Система Программирования Логики Упорядоченных Структур). Система MAX+PLUS II разработана фирмой «Altera» и обеспечивает многоплатформенную архитектурно-независимую среду создания дизайна, легко приспособляемую для конкретных требований пользователя. Система MAX+PLUS II имеет средства удобного ввода дизайна, быстрого прогона и непосредственного программирования устройств.

Представленный на **Рис. 2.1** состав ПО системы MAX+PLUS II является полным комплектом, обеспечивающим создание логических дизайнов для устройств фирмы «Altera» с программируемой логикой, в том числе семейства устройств Classic, MAX5000, MAX7000, MAX9000, FLEX6000, FLEX8000 и FLEX10K. Информация о других поддерживаемых семействах устройств фирмы «Altera» приведена в файле read.me в системе MAX+PLUS II.

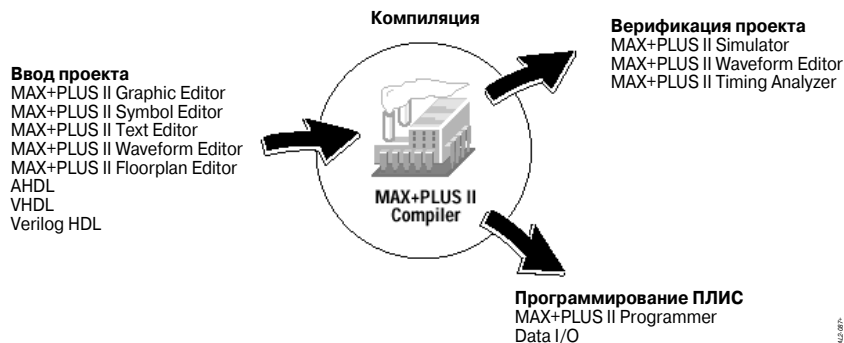


Рис. 2.1. Среда проектирования в системе MAX+PLUS II

Система MAX+PLUS II предлагает полный спектр возможностей логического дизайна: разнообразные средства описания проекта для создания проектов с иерархической структурой, мощный логический синтез, компиляция с заданными временными параметрами, разделение на части, функциональное и временное тестирование (симуляция), тестирование нескольких связанных устройств, анализ временных параметров системы, автоматическая локализация ошибок, а также программирование и верификация устройств. В системе MAX+PLUS II можно как читать, так и записывать файлы на языке AHDL и файлы трассировки в формате EDIF, файлы на языках описания аппаратуры Verilog HDL и VHDL, а также схемные файлы OrCAD. Кроме того, система MAX+PLUS II читает файлы трассировки, созданные с помощью ПО фирмы «Xilinx», и записывает файлы задержек в формате SDF для удобства взаимодействия с пакетами, работающими с другими промышленными стандартами.

Система MAX+PLUS II предлагает пользователю богатый графический интерфейс, дополненный иллюстрированной оперативной справочной системой. В полную систему MAX+PLUS II входят 11 полностью внедренных в систему приложений (Рис. 2.2). Логический дизайн (Design), включая все поддизайны (Subdesign), называется в системе MAX+PLUS II проектом (Project).

Возможно описание проекта (Design Entry) в виде файла на языке описания аппаратуры, созданного либо во внешнем редакторе, либо в текстовом редакторе MAX+PLUS II (Text Editor), в виде электрической

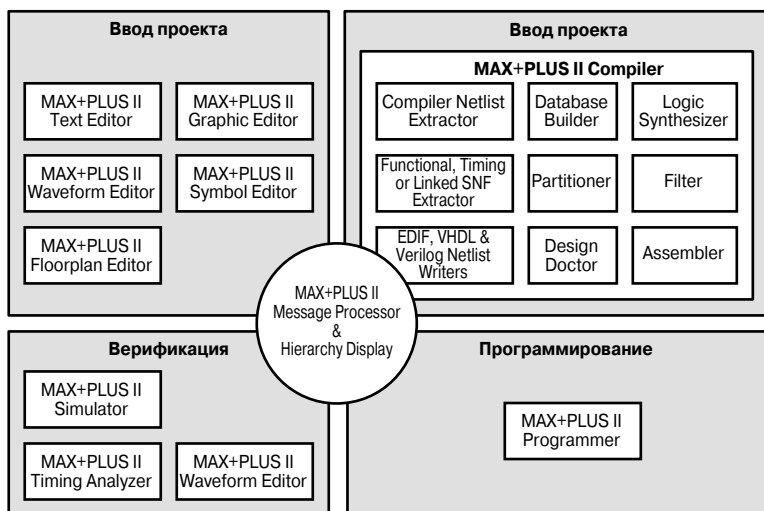


Рис. 2.2. Приложения в системе MAX+PLUS II

принципиальной схемы с помощью графического редактора Graphic Editor, в виде временной диаграммы, созданной в сигнальном редакторе Waveform Editor. Для удобства работы со сложными иерархическими проектами каждому поддизайну соответствует символ, редактирование которого производится с помощью графического редактора Symbol Editor. Размещение узлов по ЛБ и выводам ПЛИС выполняют с помощью поуровневого планировщика Floorplan Editor.

Верификация проекта (Project Verification) выполняется с помощью симулятора (Simulator), результаты работы которого удобно просмотреть в сигнальном редакторе Waveform Editor, в нем же создаются тестовые воздействия.

Компиляция проекта, включая извлечение списка соединений (Netlist Extractor), построение базы данных проекта (Data Base Builder), логический синтез (logic synthesis), извлечение временных, функциональных параметров проекта (SNF Extractor), разбиение на части (Partioner), трассировка (Fitter) и формирование файла программирования или загрузки (Assembler) выполняются с помощью компилятора системы (Compiler).

Непосредственно программирование или загрузка конфигурации устройств с использованием соответствующего аппаратного обеспечения выполняются с использованием модуля программатора (Programmer).

Многие характерные черты и команды, такие как открытие файлов, ввод назначений устройств, выводов и логических элементов, компиляция текущего проекта, похожи для многих приложений системы MAX+PLUS II. Редакторы для разработки проекта (графический, текстовый и сигнальный) имеют много общего со вспомогательными редакторами (поуровневого планирования и символьными). Каждый редактор разработки проекта позволяет выполнять похожие задачи (например поиск сигнала или символа) схожим способом. Можно легко комбинировать разные типы файлов в иерархическом проекте, выбирая для каждого функционального блока тот формат описания, который больше подходит. Поставляемая фирмой «Altera» большая библиотека мега- и макрофункций, в том числе функций из библиотеки параметризованных моделей (LPM), обеспечивает широкие возможности ввода дизайна.

Можно одновременно работать с разными приложениями системы MAX+PLUS II. Например можно открыть несколько файлов проекта и переносить информацию из одного в другой в процессе компиляции или тестирования другого проекта. Или, например, просматривать все дерево проекта и в окне просмотра перемещаться с одного уровня на другой, а в окне редактора будет появляться выбранный вами файл, причем вызывается автоматически соответствующий редактор для каждого файла (**Рис. 2.3**).

Основой системы MAX+PLUS II является компилятор, обеспечивающий мощные средства обработки проекта, при этом можно задавать нужные режимы работы компилятора. Автоматическая локализация ошибки, выдача сообщения и обширная документация об ошибках ускоряют и облегчают проведение изменений в дизайне. Можно создавать выходные файлы в разных форматах для разных целей, таких как работа функций, связь нескольких устройств, анализ временных параметров, программирование устройства.

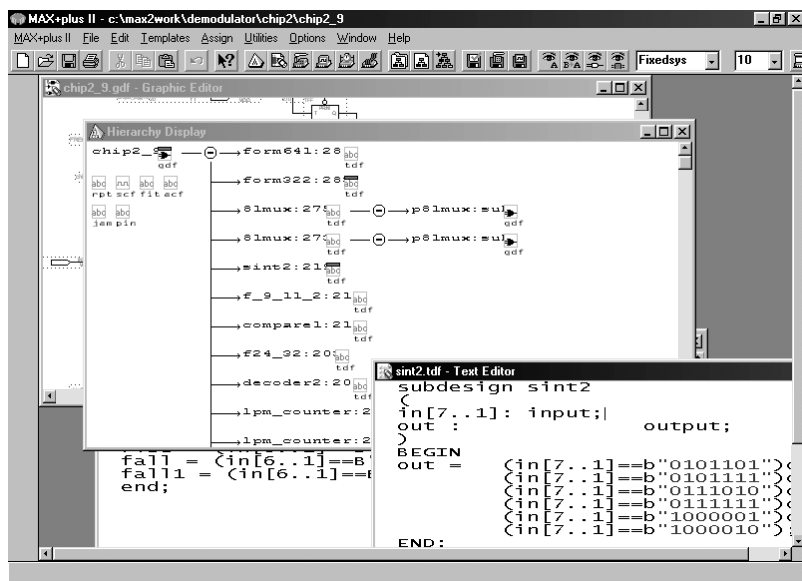


Рис. 2.3. Иерархический просмотр проекта

2.2. Процедура разработки проекта

Процедуру разработки нового проекта от концепции до завершения можно упрощенно представить следующим образом:

- создание нового файла (design file) проекта или иерархической структуры нескольких файлов проекта с использованием различных редакторов разработки проекта в системе MAX+PLUS II, т.е. графического, текстового и сигнального редакторов;
- задание имени файла проекта верхнего уровня (Top of Hierarchy) в качестве имени проекта (project name);
- назначение семейства ПЛИС для реализации проекта. Пользователь может сам назначить конкретное устройство или предоставить это компилятору для того, чтобы оценить требуемые ресурсы;
- открытие окна компилятора и его запуск нажатием кнопки Start для начала компиляции проекта. По желанию пользователя можно подключить

модуль извлечения временных задержек Timing SNF Extractor для создания файла разводки, используемого при тестировании временных параметров и анализе временных параметров;

— в случае успешной компиляции возможно тестирование и временной анализ, для проведения которого необходимо выполнить следующие действия:

- для проведения временного анализа открыть окно Timing Analyzer, выбрать режим анализа и нажать кнопку Start;

- для проведения тестирования нужно сначала создать тестовый вектор в файле канала тестирования (.scf), пользуясь сигнальным редактором, или в файле вектора (.vec), пользуясь текстовым редактором. Затем открыть окно отладчика-симулятора и нажать кнопку Start;

— программирование или загрузка конфигурации выполняется путем запуска модуля программатора с последующей вставкой устройства в программирующий адаптер программатора MPU (Master Programming Unit) или подключения устройств MasterBlaster, BitBlaster, ByteBlaster или кабеля загрузки FLEX (FLEX download cable) к устройству, программируемому в системе;

— выбор кнопки Program для программирования устройств с памятью типа EPROM или EEPROM (MAX, EPC) либо выбор кнопки Configure для загрузки конфигурации устройства с памятью типа SRAM (FLEX).

Далее будут подробно рассмотрены основные элементы разработки проекта в системе MAX+PLUS II.

Систему MAX+PLUS II можно запустить двумя способами, щелкнув дважды левой кнопкой мыши на пиктограмме MAX+PLUS II или набрав maxplus2 в командной строке.

При запуске системы MAX+PLUS II автоматически открывается ее Главное окно, меню которого охватывает все приложения системы MAX+PLUS II (**Рис. 2.4**). В верхней части окна отображается имя проекта и текущего файла проекта. Затем следует строка меню, под ней панель основных инструментов системы, обеспечивающая быстрый вызов ее компонентов. В нижней части экрана располагается строка подсказки. Вызов компонентов системы удобно производить с помощью окна меню MAX+PLUS II, представленном на **Рис. 2.5**.

Рассмотрим подробнее меню MAX+ PLUS II (**Рис. 2.5**). ПО системы MAX+ PLUS II содержит 11 приложений и главную управляющую оболочку. Различные приложения, обеспечивающие создание файлов про-

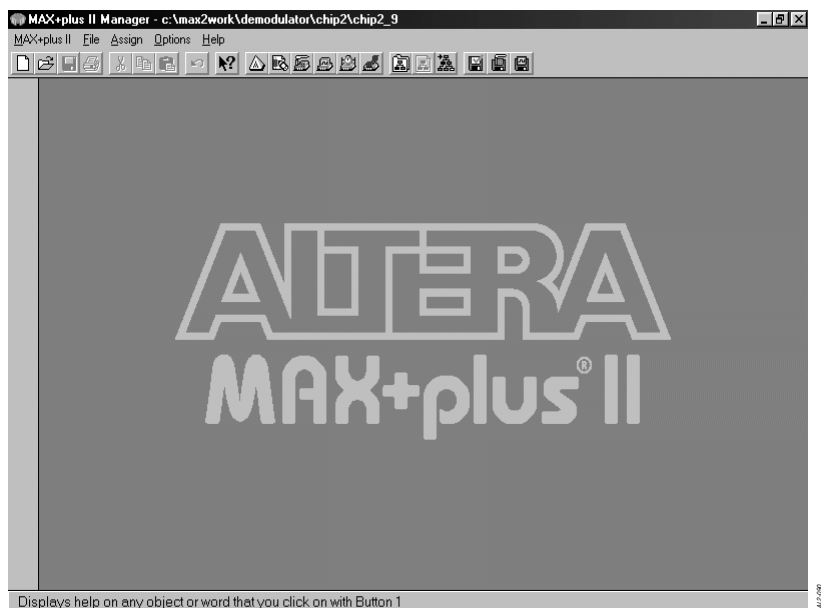


Рис. 2.4. Главное окно системы MAX+PLUS II

екта, могут быть активизированы одновременно, что позволяет пользователю переключаться между приложениями щелчком мыши или с помощью команд меню. В это же время может работать одно из фоновых приложений, например, компилятор, симулятор, временной анализатор и программатор. Одни и те же команды разных приложений работают одинаково, что облегчает задачу разработки проекта.

Окно любого приложения можно свернуть до пиктограммы, не закрывая приложения, а затем снова раз-

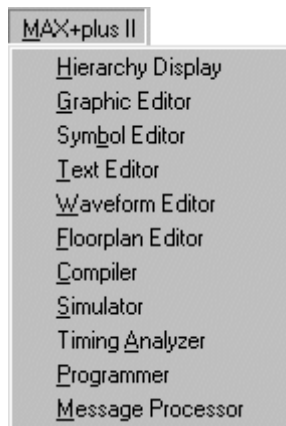


Рис. 2.5. Окно меню MAX+PLUS II

вернуть его. Это позволяет пользователю работать эффективно, не загромождая рабочий экран. В **Табл. 2.3** приведены пиктограммы и описание приложений.

Ранее на **Рис. 2.3** проиллюстрирован рабочий момент многооконной разработки дизайна, когда на экране одновременно открыты несколько окон обзора иерархии и текстового редактора.

Перед тем как начать работать в системе MAX+PLUS II, следует понять разницу между файлами проекта (design file), вспомогательными файлами и проектами.

Файл проекта — это графический, текстовый или сигнальный файл, созданный с помощью графического или сигнального редактора системы MAX+PLUS II или в любом другом схемном или текстовом редакторе, использующем промышленные стандарты, либо при помощи программы netlist writer, имеющейся в пакетах, поддерживающих EDIF, VHDL и Verilog HDL. Этот файл содержит логику для проекта MAX+PLUS II и обрабатывается компилятором. Компилятор может автоматически обрабатывать следующие файлы проекта:

Таблица 2.3. Приложения системы MAX+PLUS II


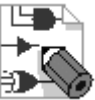



Приложение	Выполняемая функция
Hierarchy Display 	Обзор иерархии — отображает текущую иерархическую структуру файлов в виде дерева с ветвями, представляющими собой поддизайны. Можно визуально определить, является ли файл проекта схемным, текстовым или сигнальным; какие файлы открыты в данный момент; какие вспомогательные файлы в проекте доступны пользователю для редактирования. Можно также непосредственно открыть или закрыть один или несколько файлов дерева и ввести назначения ресурсов для них
Graphic Editor 	Графический редактор — позволяет разрабатывать схемный логический дизайн в формате реального отображения на экране WYSIWYG. Применяя разработанные фирмой «Altera» примитивы, мегафункции и макрофункции в качестве основных блоков разработки, пользователь может также использовать собственные символы
Symbol Editor 	Символьный редактор — позволяет редактировать существующие символы и создавать новые

Таблица 2.3 (окончание)

Приложение	Выполняемая функция
Text Editor 	<p>Текстовый редактор — позволяет создавать и редактировать текстовые файлы проекта, написанные на языках описания аппаратуры AHDL, VHDL и Verilog HDL. Кроме того, в этом редакторе можно создавать, просматривать и редактировать другие файлы формата ASCII, используемые другими приложениями MAX+PLUS II. Можно создавать файлы на языках HDL и в других текстовых редакторах, однако данный текстовый редактор системы MAX+PLUS II дает преимущества в виде контекстной справки, выделения цветом синтаксических конструкций и готовых шаблонов языков AHDL, VHDL и Verilog HDL</p>
Waveform Editor 	<p>Сигнальный редактор — выполняет двойную функцию: инструмент для разработки дизайна и инструмент для ввода тестовых сигналов и наблюдения за результатами тестирования</p>
Floorplan Editor 	<p>Поуровневый планировщик — позволяет графическими средствами делать назначения выводам устройства и ресурсов логических элементов и блоков. Можно редактировать расположение выводов на чертеже корпуса устройства и назначать сигналы отдельным логическим элементам на более подробной схеме логической структуры (LAB view). Можно также просматривать результаты последней компиляции</p>
Compiler 	<p>Компилятор — обрабатывает логические проекты, разработанные для семейств устройств «Altera» Classic, MAX5000, MAX7000, MAX9000, FLEX6000, FLEX8000 и FLEX10K. Большинство заданий выполняется автоматически. Однако пользователь может управлять процессом компиляции полностью или частично</p>
Simulator 	<p>Симулятор — позволяет тестировать логические операции и внутреннюю синхронизацию проектируемой логической схемы. Возможны три режима тестирования: функциональное, временное и тестирование нескольких соединенных между собой устройств</p>
Timing Analyzer 	<p>Анализатор временных параметров — анализирует работу проектируемой логической цепи после того, как она была синтезирована и оптимизирована компилятором, позволяет оценить задержки, возникающие в схеме</p>
Programmer 	<p>Программатор — позволяет программировать, конфигурировать, проводить верификацию и испытывать устройства фирмы «Altera»</p>
Message Processor 	<p>Генератор сообщений — выдает на экран сообщения об ошибках, предупреждающие и информационные сообщения о состоянии проекта пользователя и позволяет пользователю автоматически найти источник сообщения в исходном или вспомогательном файле (файлах) и в поуровневом плане назначений</p>

- графические файлы проекта (.gdf);
- текстовые файлы проекта на языке AHDL (.tdf);
- сигнальные файлы проекта (.wdf);
- файлы проекта на языке VHDL (.vhd);
- файлы проекта на языке Verilog (.v);
- схемные файлы OrCAD (.sch);
- входные файлы EDIF (.edf);
- файлы формата Xilinx Netlist (.xnf);
- файлы проекта «Altera» (.adf);
- файлы цифрового автомата (.smf).

Вспомогательные файлы — это файлы, связанные с проектом MAX+PLUS II, но не являющиеся частью иерархического дерева проекта. Большинство таких файлов не содержит логики дизайна. Некоторые из них создаются автоматически приложением системы MAX+PLUS II, другие — пользователем. Примерами вспомогательных файлов являются файлы назначений и конфигурации (.acf), символьные файлы (.sym), файлы отчета (.rpt) и файлы тестовых векторов (.vec).

Проект состоит из всех файлов иерархической структуры дизайна, в том числе вспомогательных и выходных файлов. Именем проекта является имя файла (ghjtrnf) верхнего уровня без расширения. Система MAX+PLUS II выполняет компиляцию, тестирование, временной анализ и программирование сразу всего проекта, хотя пользователь может в это время редактировать файлы этого проекта в рамках другого проекта. Например, во время компиляции проекта project1 пользователь может редактировать дизайн-файл TDF с помощью языка AHDL, который является также файлом проекта project2, и сохранять его; однако если он захочет компилировать его, нужно будет сначала дать имя project2 в качестве названия проекта. Для каждого проекта следует создавать отдельный подкаталог в рабочем каталоге системы MAX+PLUS II (\max2work).

В системе MAX+PLUS II легкодоступны все инструменты для создания логического проекта. Разработка проекта ускоряется за счет имеющихся стандартных логических функций, в том числе примитивов, мегафункций, библиотеки параметризованных модулей и макрофункций устаревшего типа микросхем 74-й серии. Крайне вредно использовать устаревшие библиотеки и переносить на ПЛИС схемотехнику стандартных ТТЛ-серий. Следует проектировать проект именно под архитектуру ПЛИС для получения более или менее разумных результатов.

Схемные файлы проекта создаются в графическом редакторе MAX+PLUS II. Можно также открыть, редактировать и сохранять схемы, созданные схемным редактором OrCAD.

Проекты на языках AHDL, VHDL и Verilog HDL создаются в текстовом редакторе MAX+PLUS II или любом другом текстовом редакторе. Сигнальные проекты создаются в сигнальном редакторе MAX+PLUS II.

Файлы форматов EDIF и «Xilinx», разработанные другими стандартными инструментами системы EDA, могут быть импортированы в среду MAX+PLUS II. Схемные и тестовые файлы, созданные в системе MAX+PLUS II (под DOS) и программных пакетах фирмы «Altera» A+PLUS и SAM+PLUS могут быть интегрированы в среде MAX+PLUS II.

Назначения физических ресурсов для любого узла или контакта в текущем проекте могут быть введены в графическую среду с помощью поуровневого планировщика. Он сохраняет для проекта назначения в файле с расширением .acf, в котором хранятся все типы назначений ресурсов, зондов (Probes) и устройств (Devices) так же, как и конфигурационные установки (Assign) для компилятора, симулятора и временного анализатора.

Графические символы, представляющие любой тип файла проекта, могут быть автоматически созданы в любом из редакторов MAX+PLUS II, предназначенных для разработки проектов с помощью команды File/Create Default Symbol Command. С помощью символьного редактора MAX+PLUS II можно редактировать символы или создавать собственные, а затем использовать их в любом схемном файле проекта.

В иерархической структуре проекта на любом уровне допускается смешанное использование файлов с расширениями .gdf, .tdf, .vhd, .v, .edf, .sch. Однако файлы с расширением .wdf, .xnf, .adf, .smf должны быть либо на самом нижнем иерархическом уровне проекта, либо быть единственным файлом. Способы задания файлов проекта показаны на **Рис. 2.6**.

Во всех приложениях MAX+PLUS II есть возможность с помощью команд из меню Assign (назначить) вводить, редактировать и удалять типы назначений ресурсов, устройств и параметров, которые управляют компиляцией проекта, в том числе логическим синтезом, разделением на части и подгонкой. На **Рис. 2.7** представлены команды меню Assign. Пользователь может делать назначения для текущего проекта независимо от того, открыты ли какой-нибудь файл проекта или окно приложений. Система MAX+PLUS II сохраняет информацию для проекта в файле с расширением .acf. Изменения назначений, сделанные в окне поуров-

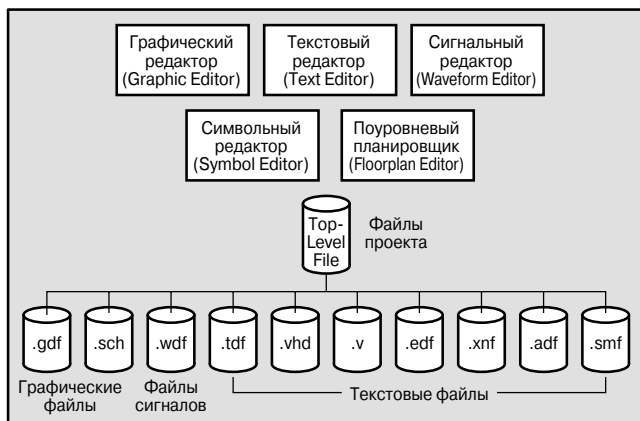


Рис. 2.6. Способы описания файлов проекта



Рис. 2.7. Меню назначений проекта Assign

нового планировщика, также сохраняются в файле ACF. Кроме того, можно редактировать файл ACF для проекта в текстовом редакторе.

Следующие функции являются общими для всех приложений MAX+PLUS II: назначения устройств, ресурсов и зондов; сохранение предыдущей версии; глобальные опции устройства в проекте; глобальные параметры проекта; глобальные требования к временным параметрам проекта; глобальный логический синтез проекта. Рассмотрим их подробнее.

Ресурс является частью устройства фирмы «Altera», как, например, контакт или логический элемент, который выполняет конкретное, определенное пользователем

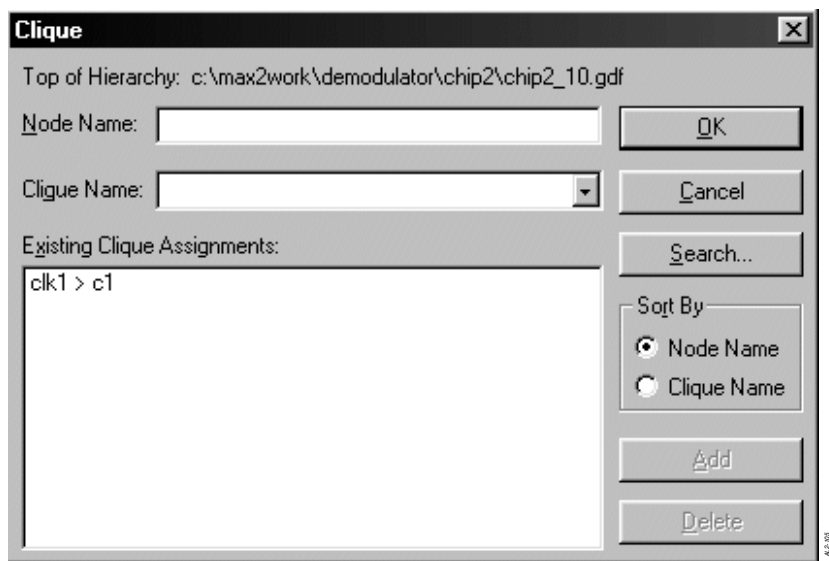


Рис. 2.8. Назначение клики (команда Assign/Clique)

задание. Пользователь может назначить логику ресурсам устройства для гарантии того, что компилятор MAX+PLUS II сделает подгонку в проекте так, как хочет пользователь. Есть следующие типы назначений.

Clique assignment (назначение клики) задает, какие логические функции должны оставаться вместе. Группировка логических функций в клики гарантирует, что они реализуются в одном и том же блоке логической структуры LAB, блоке ячеек памяти EAB, в одном ряду или устройстве. Для назначения клики используют команду Assign/Clique (Рис. 2.8).

В поле Clique Name задают имя клики. В поле Node Name задают имя узла. Узлы, объединенные в клики (иногда называют группы, но такое наименование приводит к путанице с понятием группы в AHDL), при компиляции будут размещаться в пределах одного ЛБ.

Chip assignment (назначение чипа) задает, какие логические функции должны быть реализованы в одном и том же устройстве в случае разделения проекта на части (на несколько устройств).

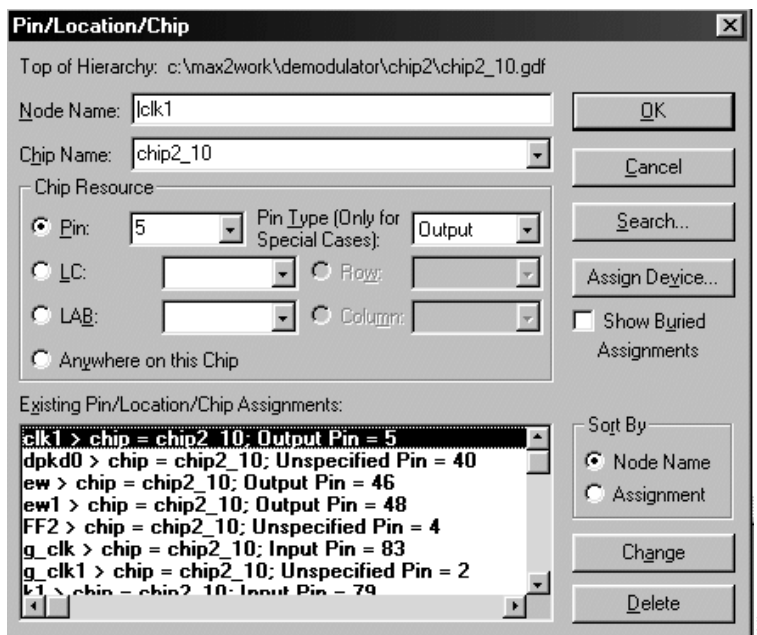


Рис. 2.9. Окно команды Assign/Pin/Location/Chip

Pin assignment (назначение вывода) назначает вход или выход одной логической функции, такой как примитив или мегафункция, конкретному контакту или вертикальному (горизонтальному) ряду выводов ПЛИС.

Location assignment (назначение ячейки) назначает единственную логическую функцию, такую как выход примитива или мегафункции, конкретной ячейке чипа, такой как логический элемент, ячейка ввода/вывода, ячейка памяти, блоки LAB и EAB, горизонтальные или вертикальные ряды.

Назначения вывода, чипа и ячейки выполняются с помощью команды Assign/Pin/Location/Chip, окно которой приведено на **Рис. 2.9**. В полях данного окна можно задать номер вывода (Pin), логическую ячейку или блок, а также, используя кнопки Change и Delete, изменить назначения.

Probe assignment (назначение зонда) присваивает легко запоминающееся уникальное имя входу или выходу логической функции. Данное назначение весьма полезно при моделировании системы. Для назначения зонда используют команду Assign/Probe (**Рис. 2.10**).

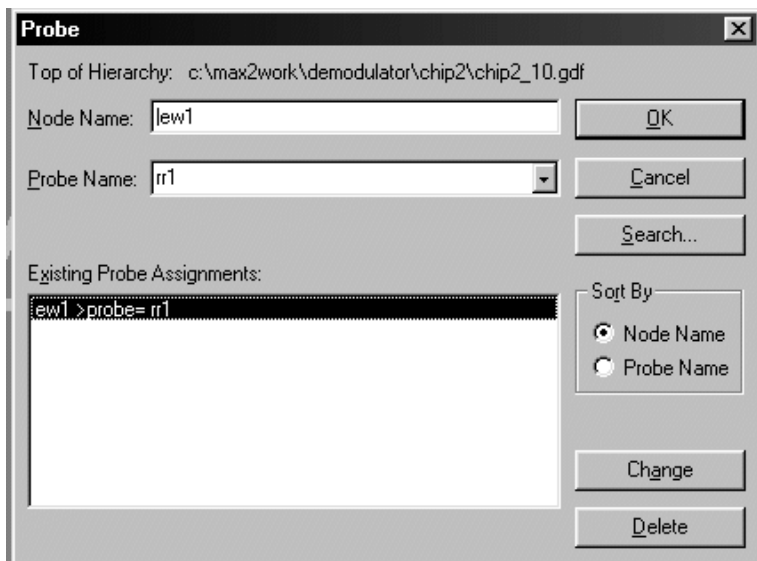


Рис. 2.10. Меню команды Assign/Probe

Connected pin assignment (назначение соединенных выводов) задает внешнее соединение двух или более выводов на схеме пользователя. Эта информация также полезна в режиме тестирования временных параметров и при тестировании нескольких скомпонованных проектов. Для выполнения назначения соединенных выводов используют команду Assign/Connected Pins (**Рис. 2.11**).

Local routing assignment (назначение местной трассировки) присваивает коэффициент разветвления по выходу узла логическому элементу, находящемуся в том же блоке LAB, что и узел, или же в соседнем LAB, смежном с выбранным узлом, с использованием общих местных связей. Местная трассировка также производится между узлом, помещенным в блок LAB на периферии устройства, и выходным контактом, с которым он соединен. Назначение местной трассировки производится с помощью команды Assign/Local routing (**Рис. 2.12**).

Device assignment (назначение устройства) назначает тип ПЛИС, в которой будет воплощен проект. Если проект состоит из нескольких устройств, данный тип назначения делает назначения чипов конкретным устройствам. Можно также выбрать опцию AUTO и предоставить компилятору выбрать

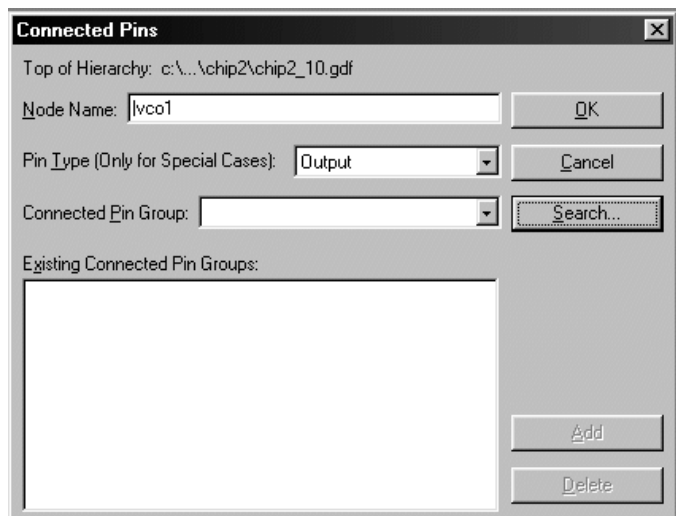


Рис. 2.11. Меню команды Assign/Connected Pins

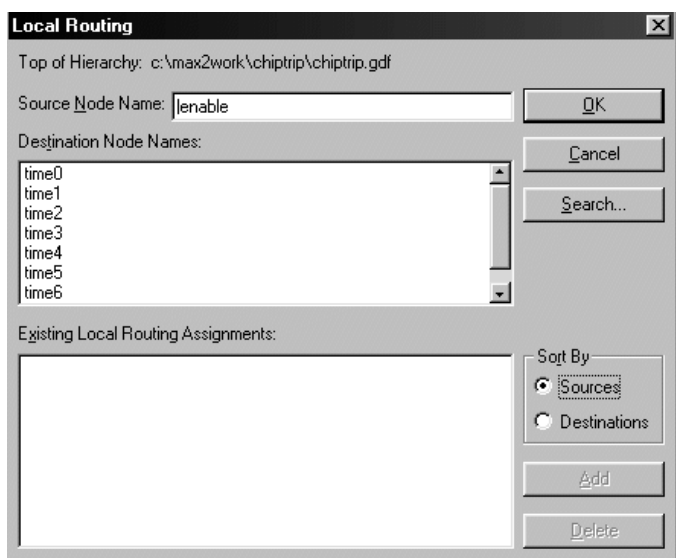


Рис. 2.12. Окно команды Assign/Local routing

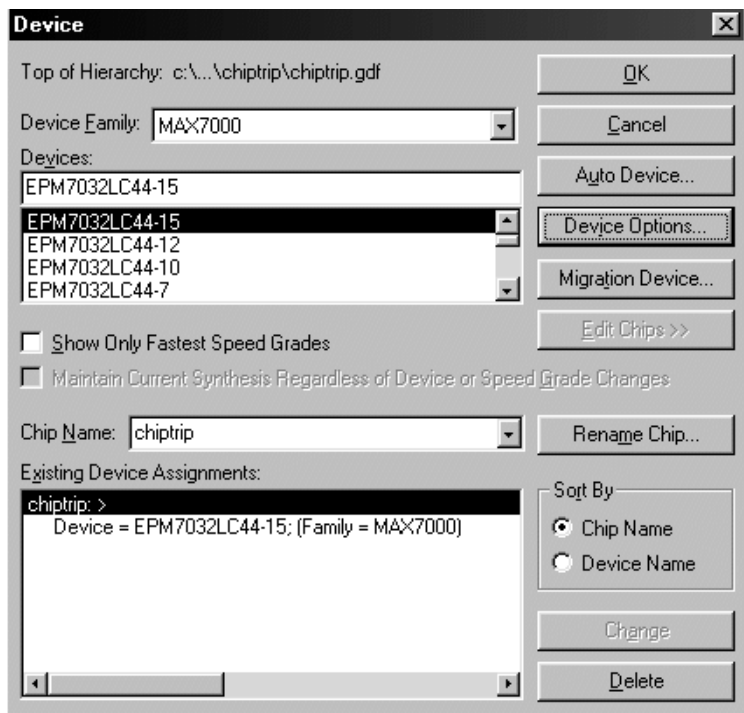


Рис. 2.13. Окно команды Assign/ Device

устройство из заданного семейства устройств. Процессом автоматического выбора устройства можно управлять, задавая диапазон и число устройств в семействе. Если проект оказался слишком большим для реализации в одном устройстве, можно задать тип и число дополнительных устройств. Для выбора устройства используется команда Assign/Device (Рис. 2.13).

Logic option assignment (назначение логической опции) управляет логическим синтезом отдельных логических функций во время компиляции с применением стиля логического синтеза и/или отдельных опций логического синтеза. Фирма «Altera» обеспечивает большое количество логических опций, а также готовых стилей, каждый из которых представляет собой собрание установок для логических опций, объединенных одним именем стиля синтеза (Synthesis Style). Пользователь может применять гото-

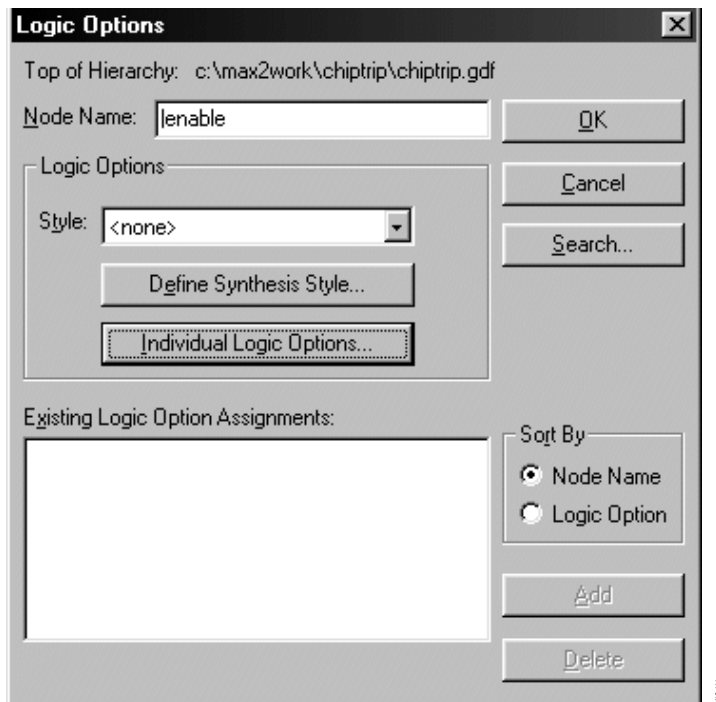


Рис. 2.14. Окно команды Assign/Logic Options

вые стили или создавать новые. Стили синтеза позволяют настраивать опции синтеза на определенное семейство устройств, учитывая при этом архитектуру семейства. Для настройки стилей синтеза применяется команда Assign/Logic Options (**Рис. 2.14**).

Timing assignment (назначение временных параметров) управляет логическим синтезом и подгонкой отдельных логических функций для получения требуемых характеристик для времени задержки t_{PD} (вход—неактивный выход), t_{CO} (синхросигнал—выход), t_{SU} (синхросигнал—время установки), f_{MAX} (частота синхросигнала). Пользователь может также вырезать соединения между путями распространения для конкретного сигнала, называемого «узлом» в системе MAX+PLUS II и другими узлами проекта. Назначение временных параметров узла производится по команде Assign/Timing Requirements (**Рис. 2.15**).

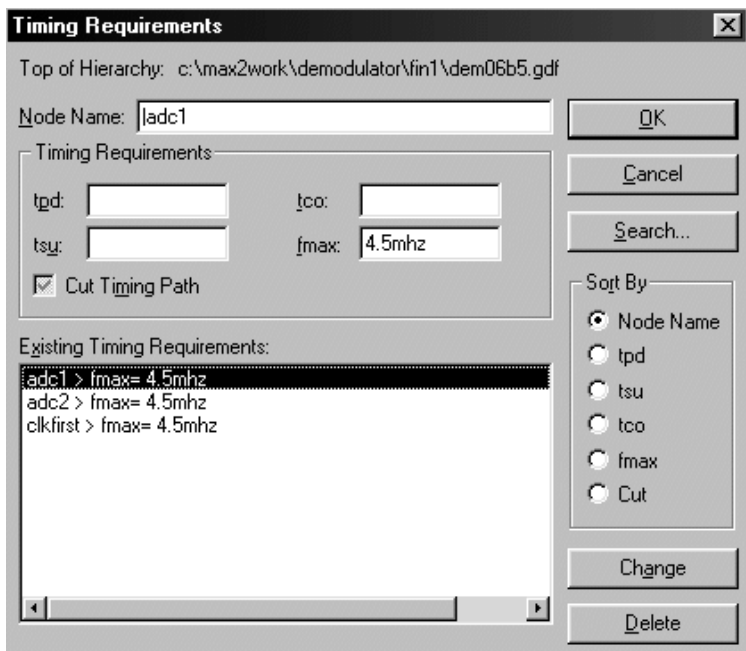


Рис. 2.15. Окно команды Assign/Timing Requirements

Кроме использования команд меню Assign, назначения можно выполнять щелчком правой кнопки мыши по выбранному узлу проекта и выбирая соответствующее назначение во всплывающем меню (Рис. 2.16).

Можно определить глобальные опции компилятора для того, чтобы он их использовал для всех устройств при обработке проекта. Для резервирования дополнительных возможностей логики на будущее можно задать процентное соотношение выводов и логических элементов, которые должны оставаться неиспользованными во время текущей компиляции. Можно также задать установки для битов опций устройств и выводов в конфигурации устройств, используемой для нескольких целей. Например, можно задать бит защиты от несанкционированного считывания (security bit) глобальным, что предотвратит пиратское копирование топологии устройств, базирующихся на памяти EPROM или EEPROM.

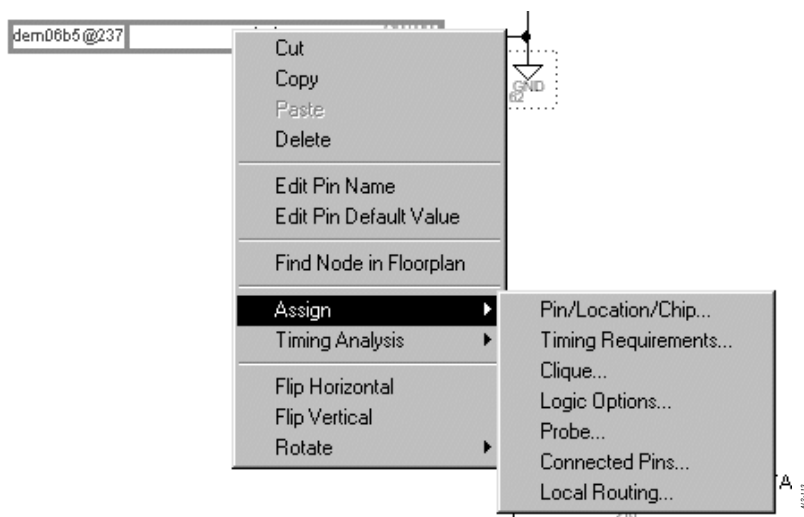


Рис. 2.16. Выбор команд Assign с помощью всплывающего меню

Можно задать имена и глобальные установки, которые будут использованы компилятором для параметров во всех параметризованных функциях в проекте.

Можно ввести глобальные временные требования для проекта, задавая общие характеристики для времени задержки t_{PD} (вход—нерегистрируемый выход), t_{CO} (синхросигнал—выход), t_{SU} (синхросигнал—время установки), f_{MAX} (частота синхросигнала). Можно также вырезать соединения между всеми двунаправленными контурами обратной связи, цепями прохождения сигналов Preset (установка 1) и Clear (установка 0) и другими цепями синхронизации в проекте. Для этих целей используется команда Assign/Global Project Timing Requirements (Рис. 2.17). Флажок Cut All Bidirectional Feedback Timing Paths позволяет исключить все цепи обратной связи для двунаправленных выводов. Флажок Cut All Clear & Preset Timing Paths позволяет удалить соединения между всеми цепями сброса и предустановки проекта. Флажок Ignore Timing Assignments During Fitting позволяет запустить трассировщик (Fitter) без учета временных ограничений проекта. Когда этот флажок сброшен и заданы временные параметры, осуществляется т.н. управляемый синтез (time driving synthesis).

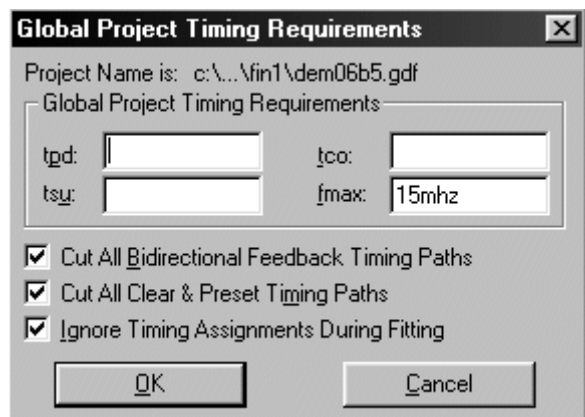


Рис. 2.17. Окно команды Assign/Global Project Timing Requirements

Из собственного опыта замечено, что вначале для ускорения компиляции следует выбрать этот флажок, а в дальнейшем при «вылизывании» проекта — сбросить. Следует помнить, что управляемый временной синтез возможен только для устройств FLEX, для устройств MAX времена задержек предопределены, и в случае назначения временных параметров происходит только лишь проверка соответствия полученных при синтезе параметров заданным.

Можно сделать глобальные установки для компилятора в части логического синтеза проекта. Можно задать используемый по умолчанию стиль логического синтеза, определить степень оптимизации по скорости и занимаемым ресурсам, дать указания компилятору по выбору автоматических глобальных сигналов управления, таких как Clock (тактовый), Clear (установка 0), Preset (установка 1) и Output Enable (разрешение выхода). Можно также выбрать для компилятора режим стандартного или многоуровневого синтеза, режим кодирования цифрового автомата с 1 при подключении питания, а также режим автоматической упаковки регистров. Кроме того, можно выбрать вариант автоматической реализации логики в быстрых входных или выходных логических элементах и ячейках входа/выхода, выводах с открытым стоком и блоках ячеек памяти. Для назначения глобальных параметров логического синтеза используют команду Assign/Global Project Logic Synthesis (Рис. 2.18).



Рис. 2.18. Окно команды Assign/Global Project Logic Synthesis

Кнопка Define Synthesis Style позволяет выбрать более тонкие параметры стиля синтеза (**Рис. 2.19**), такие как имя стиля, способ реализации и максимальную длину цепочек переноса и каскадирования, степень минимизации логических функций и другие параметры синтеза. Кнопка Advanced Options позволяет выбрать параметры синтеза в диалоговом окне, изображенном на **Рис. 2.20**.

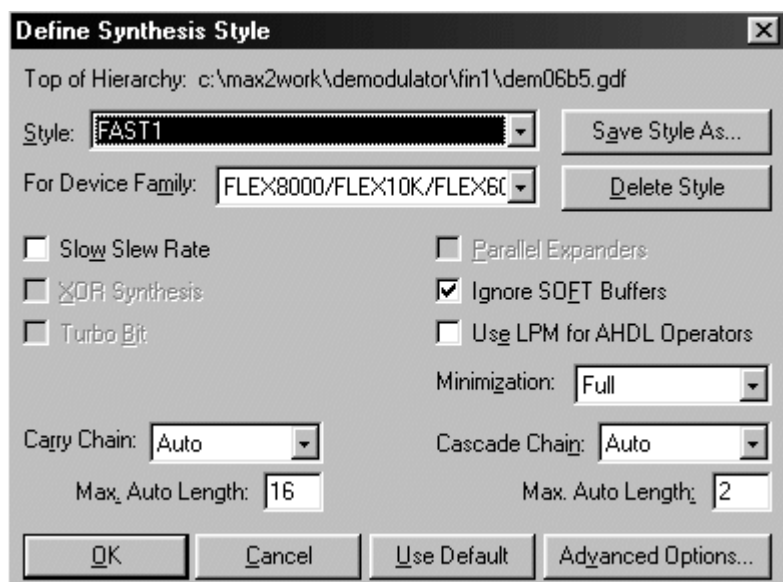


Рис. 2.19. Определение стиля синтеза

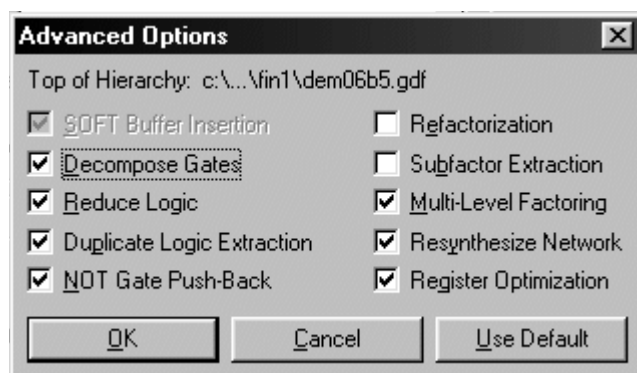


Рис. 2.20. Окно Advanced Options

2.3. Редакторы MAX+PLUS II

Все пять редакторов MAX+PLUS II и три редактора создания файла проекта (графический, текстовый и сигнальный) имеют общие функции, такие как, например, сохранение и вызов файла. Кроме того, приложения редактора MAX+PLUS II имеют следующие общие функции:

- создание файлов символов и файлов с прототипами функций (in-clude file symbol and include file generation);
- поиск узлов (node location);
- трассировка иерархического дерева (hierarchy traversal);
- всплывающие окна меню, зависящего от контекста (context-sensitive menu comands);
- временной анализ (timing analysis);
- поиск и замена фрагментов текста (find & replace text);
- отмена последнего шага редактирования, его возвращение, вырезка, копирование, вклеивание и удаление выбранных фрагментов, обмен фрагментами между приложениями MAX+PLUS II или приложениями Windows (undo, cut, copy, paste & delete);
- печать (print).

На **Рис. 2.21** показано окно графического редактора (Graphic Editor) MAX+PLUS II, обеспечивающего проектирование в реальном формате изображения (WYSIWIG). В нем можно создавать новые файлы (команда New из меню File). Вызывается графический редактор из меню MAX+PLUS II.

Графические файлы проекта (.gdf) или схемные файлы OrCAD (.sch), созданные в данном графическом редакторе, могут включать любую комбинацию символов примитивов, мегафункций и макрофункций. Символы могут представлять собой любой тип файла проекта (.gdf, .sch, .tdf, .vhd, .v, .wdf, .edf, .xnf, .adf, .smf). Универсальность графического редактора характеризуется следующими чертами:

- инструмент выбора («стрелка») облегчает разработку проекта. Он позволяет двигать и копировать объекты, а также вводить новые символы. Когда вы помещаете его на вывод или конец линии, он автоматически преобразуется в инструмент рисования ортогональных линий. Если им щелкнуть на тексте, он автоматически преобразуется в инструмент редактирования текста;
- символы соединяются сигнальными линиями, которые называют узлами (nodes) или линиями шин (bus), которые представляют собой не-

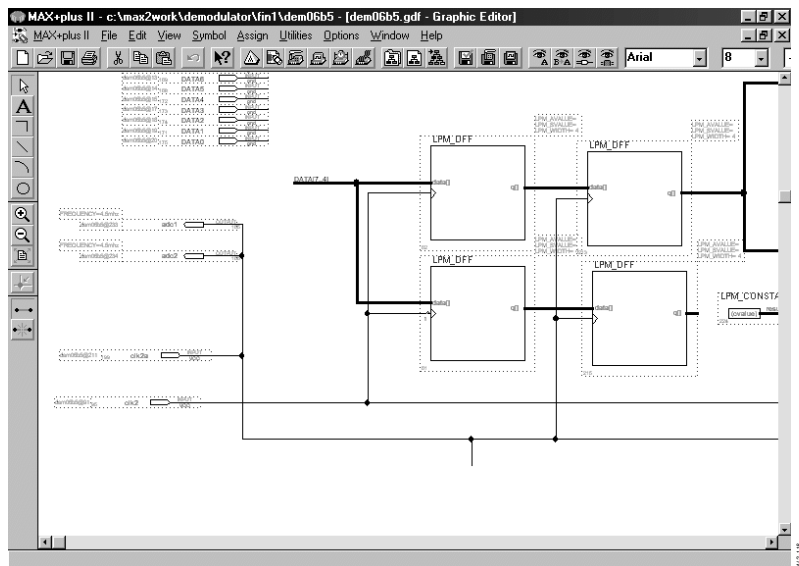


Рис. 2.21. Окно графического редактора MAX+PLUS II

сколько логически сгруппированных узлов. Когда вы присваиваете узлу имя, вы можете соединить его с другими узлами или символами только по имени. Шины соединяются по имени, но возможно и их графическое соединение;

- пользователь может переопределить порты, используемые в каждом отдельном примере символа мега- или макрофункции, а также инвертировать их. При этом для указания инвертированного порта появится кружок, обозначающий инверсию;

- можно выбрать несколько объектов в прямоугольной области и редактировать их вместе или по отдельности. При перемещении выбранной области сигнальные связи сохраняются;

- для каждого символа можно присвоить назначения зондов, выводов, расположения, чипов, клик, временных параметров, местную трассировку, логические опции и назначения параметров. Для облегчения тестирования можно также создать назначения групп выводов, которые будут определять соединения внешнего устройства между выводами;

— поставляемые фирмой «Altera» примитивы, мега- и макрофункции сокращают время разработки дизайна. Пользователь может также создавать свои собственные библиотеки функций. При редактировании символа или восстановлении его по умолчанию можно автоматически создавать выбранные примеры или все примеры этого символа в файле в графическом редакторе.

Графический редактор обеспечивает и много других возможностей. Например, можно увеличить или уменьшить масштаб отображения на экране и увидеть дизайн целиком или какую-либо его деталь. Можно выбирать гарнитуру и размер шрифта, задавать стили линий, устанавливать и отображать направляющие. Можно копировать, вырезать, вклеивать и удалять выбранные фрагменты; получать зеркальное отображение (вертикальное или горизонтальное); поворачивать выделенные фрагменты на 90, 180 или 270 градусов; задавать размер, размещение текущего листа схемы по вертикали или горизонтали.

На **Рис. 2.22** представлено окно символьного редактора системы MAX+PLUS II, с помощью которого можно просматривать, создавать и редактировать символ, представляющий собой логическую схему. В нем можно создавать новые файлы (команда New из меню File). Вызывается символьный редактор из меню MAX+PLUS II.

Символьный файл имеет то же имя, что и файл проекта, и расширение .sym. Команда Creat Default Suymbol меню File, которая есть в графическом, текстовом и сигнальном редакторах, создает символ для любого файла дизайна. Символьный редактор обладает следующими характеристиками:

- можно переопределить символ, представляющий файл проекта;
- можно создавать и редактировать выводы и их имена, разрабатывая входные, выходные и двунаправленные контакты, а также задавать варианты ввода символа в файл графического редактора с отображением на экране имен выводов или без отображения, с отображением полного или сокращенного имени. Таким образом, полное имя порта и имя, отображаемое в файле в окне графического редактор, могут быть разными;
- имена выводов автоматически дублируются за границу символа. Редактированию подлежат только имена внутри границы символа. Имена снаружи нельзя менять, они просто иллюстрируют соединение выводов;
- можно задать значения параметров и их значения по умолчанию;
- сетка и направляющие помогают выполнить точное выравнивание объектов;

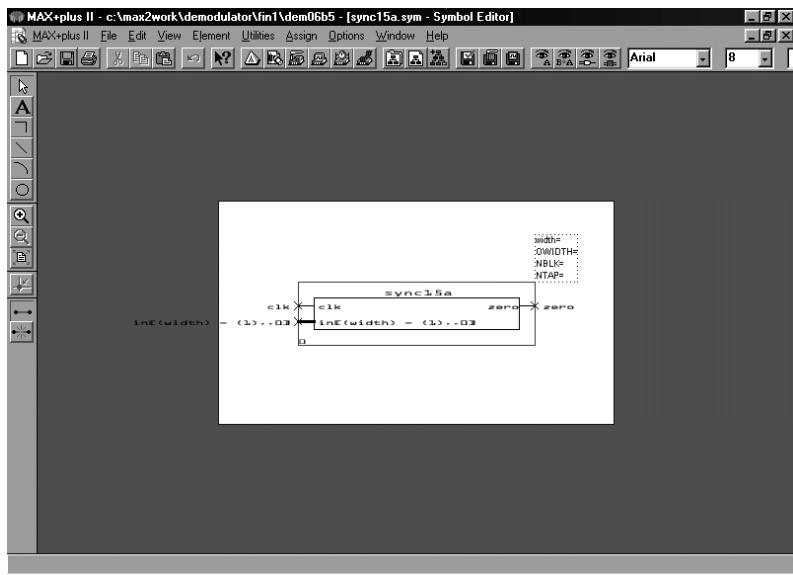


Рис. 2.22. Символьный редактор MAX+PLUS II

— в символах можно вводить комментарии или полезные замечания, которые также появятся при вводе символа в файл в графическом редакторе.

На **Рис. 2.23** показано окно текстового редактора MAX+PLUS II, который является гибким инструментом для создания текстовых файлов проекта на языках описания аппаратуры: AHDL (.tdf), VHDL (.vhd), Verilog HDL (.v). В этом текстовом редакторе можно работать также с произвольным файлом формата ASCII. В нем можно создавать новые файлы (команда New из меню File). Вызывается символьный редактор из меню MAX+PLUS II.

Все перечисленные файлы проекта можно создавать в любом текстовом редакторе, однако данный редактор имеет встроенные возможности удобного ввода файлов проекта, их компиляции и отладки с выдачей сообщений об ошибках и их локализацией в исходном тексте или в тексте вспомогательных файлов; кроме того, существуют шаблоны языковых конструкций для AHDL, VHDL и Verilog HDL, выполнено окрашивание синтаксических конструкций. В данном редакторе можно вручную редактировать файлы назначений и конфигурации (.acf), а также делать установки конфигурации для компилятора, симулятора и временного анализатора.

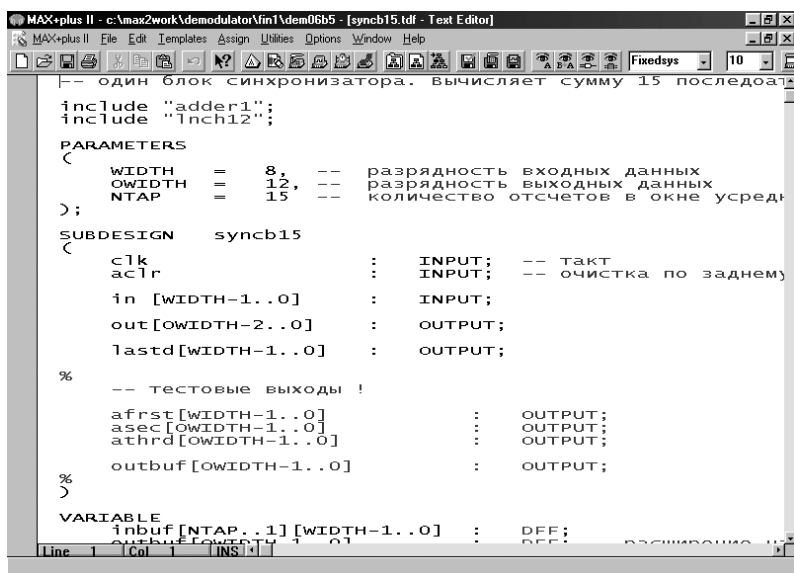


Рис. 2.23. Текстовый редактор MAX+PLUS II

Пользуясь данным текстовым редактором, можно создавать тестовые векторы (.vec), используемые для тестирования, отладки функций и при вводе сигнального проекта. Можно также создавать командные файлы (.cmd — для симулятора и .edc — для EDIF), а также макробιβотеки (.lmf). В текстовом редакторе MAX+PLUS II обеспечивается контекстуальная справка.

Сигнальный редактор (Рис. 2.24) выполняет две роли: служит инструментом создания дизайна и инструментом ввода тестовых векторов и просмотра результатов тестирования. Пользователь может создавать сигнальные файлы проекта (.wdf), которые содержат логику дизайна для проекта, а также файлы каналов тестирования (.scf), которые содержат входные векторы для тестирования и функциональной отладки. Новый файл создается командой New меню File. Вызывается сигнальный редактор из меню MAX+PLUS II.

Разработка дизайна в сигнальном редакторе является альтернативой созданию дизайна в графическом или текстовом редакторах. Здесь мож-

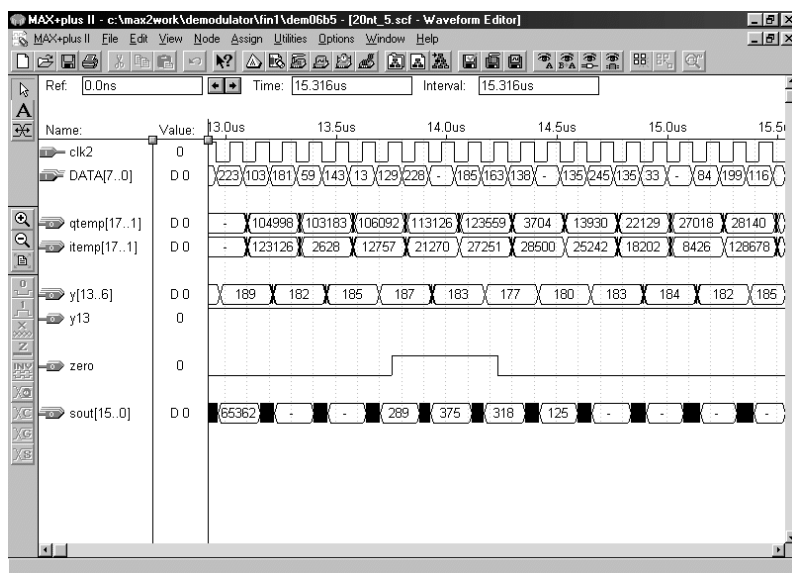


Рис. 2.24. Сигнальный редактор MAX+PLUS II

но графическим способом задавать комбинации входных логических уровней и требуемых выходов. Созданный таким образом файл с расширением WDF (Waveform Design File) может содержать как логические входы, так и входы цифрового автомата, а также выходы комбинаторной логики, счетчиков и цифровых автоматов. Можно использовать также «замурованные» (buried) узлы, которые помогают определить требуемые выходы.

Способ разработки проекта в сигнальном редакторе лучше подходит для цепей с четко определенными последовательными входами и выходами, т.е. для цифровых автоматов, счетчиков и регистров.

С помощью сигнального редактора можно легко преобразовывать формы сигналов целиком или частично, создавая и редактируя узлы и группы. Простыми командами можно создавать файл таблицы ASCII-символов (.tbl) или импортировать файл тестовых векторов в формате ASCII (.vec) для создания файлов тестируемых каналов SCF и сигнального дизайна WDF. Можно также сохранить файл WDF как SCF для проведения

тестирования или преобразовать SCF в WDF для использования его в качестве файла проекта.

Сигнальный редактор имеет следующие отличительные черты:

- можно создать или отредактировать узел для получения типа I/O (вход/выход), который представляет собой входной или выходной контакт или «замурованную» логику;

- при разработке WDF можно задать тип логики, которая делает каждый узел контактом, причем входным, регистровым, комбинаторным или цифровым автоматом;

- можно также задать значения по умолчанию в логическом узле для активного логического уровня: высокий ("1"), неопределенный (X) или с высоким импедансом (Z), а также имя состояния по умолчанию в узле типа цифрового автомата;

- для упрощения создания тестового вектора можно легко добавить в файл тестируемых каналов SCF несколько узлов или все из информационного файла симулятора (.snf), существующего для полностью откомпилированного и оптимизированного проекта;

- можно объединить от 2 до 256 узлов для создания новой группы (шины) или разгруппировать объединенные ранее в группу узлы. Можно также объединять группы с другими группами. Значение группы может быть отображено в двоичной, десятичной, шестнадцатеричной или восьмеричной системе счисления с преобразованием (или без) в код Грэя;

- можно копировать, вклеивать, перемещать или удалять выбранную часть («интервал») формы сигнала или всю форму сигнала, а также весь узел или группу (т.е. имя узла или группы плюс форма сигнала). Одной операцией можно отредактировать несколько интервалов, целые формы сигналов, а также целые узлы и группы. Копии целых узлов и групп связаны, так что редакционные правки одной копии отражаются во всех копиях. Можно также инвертировать, вставлять, переписывать, повторять, расширять или сжимать интервал формы сигнала любой длины с любым логическим уровнем, тактовым сигналом, последовательностью счета или именем состояния;

- можно задать и, по желанию, отображать на экране сетку для выравнивания переходов между логическими уровнями либо до их создания, либо после;

- в любом месте файла можно вводить комментарии между формами сигнала;

- можно менять масштаб отображения;

— для того, чтобы показать разницу между выходами при тестировании и выходами реального устройства, можно сделать наложение любых выходов в текущем файле или наложить второй файл сигнального редактора для сравнения форм сигналов его узлов и групп с соответствующими им из текущего файла.

Для отладки устройств ЦОС часто приходится тестировать алгоритм на реальных или смоделированных сигналах. Для этого удобно использовать векторный сигнальный файл (Vector File).

Vector File (формат текста ASCII) используется для определения входных условий моделирования и узлов, которые нужно моделировать. Vector File может также использоваться, чтобы создать Waveform Design File для входных данных проекта. Рассмотрим формат векторного файла подробнее.

Все разделы, используемые в Vector File, рассматриваются ниже в том порядке, в котором они обычно присутствуют в файле. Возможно повторение любого раздела с целью внесения дополнительных условий для входных данных в пределах одного Vector File.

Unit Section

Начинается с ключевого слова UNIT с дальнейшим указанием единиц измерения в файле. Параметр необязательный. По умолчанию единицы измерения ns. Возможные единицы измерения: ns (нс), ms (мс), us (мкс), s (с), mhz (МГц). Раздел заканчивается символом «;».

Пример: UNIT ms;

Start Section

Начинается с ключевого слова START с последующим указанием начального временного значения. Параметр необязательный. Значение по умолчанию нулевое. Если не указаны единицы измерения, то они принимаются из раздела Unit Section. Раздел заканчивается символом «;».

Пример: START 5ns;

Stop Section

Аналогичен разделу Start Section. По умолчанию принимается значение времени последнего вектора модели.

Пример: STOP 150ms;

Необходимо учитывать, что Vector File должен содержать кратное количество Start-Stop Section, представляющих собой временные интервалы. Недопустимо обращение к одному временному интервалу различных векторов модели.

Interval Section

Начинается с ключевого слова INTERVAL с последующим указанием временного значения. Определяет временной интервал ввода векторов. Параметр необязательный. Значение по умолчанию 1 нс. Раздел заканчивается символом «;».

Пример: INTERVAL 15ns;

Group Create Section

Начинается с ключевого слова GROUP CREATE. Данный раздел требуется не всегда для групп, шин или конечных автоматов, которые были созданы в исходных файлах проекта. Все узлы в группе должны иметь тип I/O. В Vector File, используемом для симуляции, узлы должны иметь имена, совпадающие с именами узлов, введенными в файл-проект, включая иерархический путь, если это необходимо. Раздел заканчивается символом «;».

Пример: GROUP CREATE groupABC = nodeA nodeB nodeC;

Radix Section

Начинается с ключевого слова RADIX с последующим указанием обозначения системы счисления. Параметр необязательный. По умолчанию принимается шестнадцатеричная система счисления. Различают четыре системы: BIN (двоичная), DEC (десятичная), HEX (шестнадцатеричная), OCT (восьмеричная). Раздел заканчивается символом «;».

Пример: RADIX DEC;

Inputs Section

Начинается с ключевого слова INPUTS. Далее следует список узлов и/или имен групп. В Vector File, используемом для симуляции, имена узлов должны совпадать с именами узлов в файле-проект, включая иерархический путь, если это необходимо. Раздел заканчивается символом «;».

Пример: INPUT databus clk OEN nodeA;

В приведенном ниже примере при задействовании следующей секции входных данных значения в предыдущей секции теряются.

Пример: INPUTS A1 A2;

```
                                START 0;
                                STOP 25;
                                PATTERN
%Секция модели 1 с входами A1 и A2%
                                0 0 0
                                0 0 1;
```

```
START 26;
STOP 50;
PATTERN
%Секция модели 2 с входами A1 и A2%
0 1 0
1 0 0;

INPUTS A1 B1;

START 51;
STOP 100;
PATTERN
%Секция модели 3 с входами A1 и B1%
1 1 0
0 1 1;
```

Outputs Section

Начинается с ключевого слова OUTPUTS. Используется для описания выходов. Аналогия с Inputs Section.

Пример: OUTPUTS RCO QA QB QC;

Пример: OUTPUTS A1 A2;

```
START 0;
STOP 25;
PATTERN
%Секция модели 1 с выходами A1 и A2%
0 0 0
0 0 1;

START 26;
STOP 50;
PATTERN
%Секция модели 2 с выходами A1 и A2%
0 1 0
1 0 0;

OUTPUTS A1 B1;
```

```
START 51;
STOP 100;
PATTERN
%Секция модели 3 с выходами A1 и B1%
1 1 0
0 1 1;
```

Buried Section

Начинается с ключевого слова BURIED. Используется для описания узлов. Аналогия с Inputs Section.

Пример: BURIED nodeQA0 nodeQA1 nodeQB0 nodeQB1;

Pattern Section

Начинается с ключевого слова PATTERN. В этом разделе используются данные предыдущих разделов Inputs, Outputs и Buried Section. Добавление новых Inputs, Outputs и Buried Section очищает все предыдущие данные этих типов, т.е. старые данные заменяются новыми (см. пример для Inputs Section). Раздел заканчивается символом «;».

Vector File, используемый, чтобы создать WDF, может содержать дополнительные разделы Combinatorial Section, Machine Section, Registered Section. Эти разделы игнорируются, если Vector File используется для моделирования.

Рассмотрим пример создания Vector File для восьмибитного сумматора. Описание на языке AHDL.

Пример программы sum8_.tdf:

```
SUBDESIGN SUM8_
%8 разрядный сумматор без знака%
(
  clk:INPUT;

  a[7..0]:INPUT = GND;
  b[7..0]:INPUT = GND;
  sum[7..0]:OUTPUT;
  cr:OUTPUT;
)
VARIABLE
  c[7..1]:NODE;
```

```
sum[7..0]:DFF;
cr:DFF;

BEGIN

sum[7..0].clk=clk;
sum[7..0].prn=VCC;
sum[7..0].clrn=VCC;
cr.clk=clk;
cr.prn=VCC;
cr.clrn=VCC;

If (a[0]&b[0])==VCC then
    sum[0]=GND;
    c[1]=VCC;
ElsIf (a[0]#b[0])==GND then
    sum[0]=GND;
    c[1]=GND;
Else sum[0]=VCC;
    c[1]=GND;
End If;

%%
FOR i IN 1 TO 6 GENERATE
    If (a[i]&b[i]&c[i])==VCC then
        sum[i]=VCC;
        c[i+1]=VCC;
    ElsIf (a[i]#b[i]#c[i])==GND then
        sum[i]=GND;
        c[i+1]=GND;
    ElsIf
        (((a[i]&b[i])#(a[i]&c[i])#(b[i]&c[i]))&!(a[i]&b[i]&c[
i]))==VCC then
        sum[i]=GND;
        c[i+1]=VCC;
    Else sum[i]=VCC;
        c[i+1]=GND;
```

```

        End If;
END GENERATE;
%%

    If (a[7]&b[7]&c[7])==VCC then
                                sum[7]=VCC;
                                cr=VCC;
    ElseIf (a[7]#b[7]#c[7])==GND then
                                sum[7]=GND;
                                cr=GND;
                                ElseIf
(( (a[7]&b[7])#(a[7]&c[7])#(b[7]&c[7]))&!(a[7]&b[7]&c[
7]))==VCC then
                                sum[7]=GND;
                                cr=VCC;
    Else sum[7]=VCC;
                                cr=GND;
    End If;

END;
```

Пример программы *sum8_vec*:

```

START 0;
STOP 500;
INTERVAL 25;
RADIX BIN;
INPUTS clk;
PATTERN
0 1;

START 0;
STOP 500;
INTERVAL 50;
RADIX BIN;
INPUTS a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2
b1 b0;
```



```

OUTPUTS sum[7..0] cr;
PATTERN
%a7 a6 a5 a4 a3 a2 a1 a0 b7 b6 b5 b4 b3 b2 b1 b0%
1 1 1 0 0 0 0 1 0 0 0 0 0 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1
0 0 0 0 0 0 0 0 1 1 0 0 1 1 0 0
1 1 1 1 1 1 0 0 1 0 1 1 0 0 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0;
```

В данном примере будут сложены 6 пар двоичных чисел. При этом будет задан тактирующий меандр `clk` (тактирующий импульс для D-триггеров) с периодом повторения 50 нс и скважностью 2. Начальным значением импульса будет 0. Преобразование будет проводиться от 0 до 500 нс.

При начале симуляции необходимо отсутствие в рабочей директории соответствующего файла `sum8_.scf`, так как иначе симулятор будет по умолчанию использовать файл не с расширением `.ves`, а файл с расширением `.scf`. После процесса симуляции с использованием `.ves` файла файл с расширением `.scf` будет создан автоматически.

На **Рис. 2.25** показано окно поуровневого планировщика (Floorplan Editor), с помощью которого пользователь назначает ресурсы физических устройств и просматривает результаты разветвлений и монтажа, сделанных компилятором. Окно поуровневого планировщика открывается при выборе опции Floorplan Editor в меню MAX+PLUS II.

В окне поуровневого планировщика могут быть представлены два типа изображения:

- Device View (вид устройства) показывает все выводы устройства в сборке и их функции;

- LAB View (вид логического блока) показывает внутреннюю структуру устройства, в том числе все логические блоки и отдельные логические элементы или макроячейки.

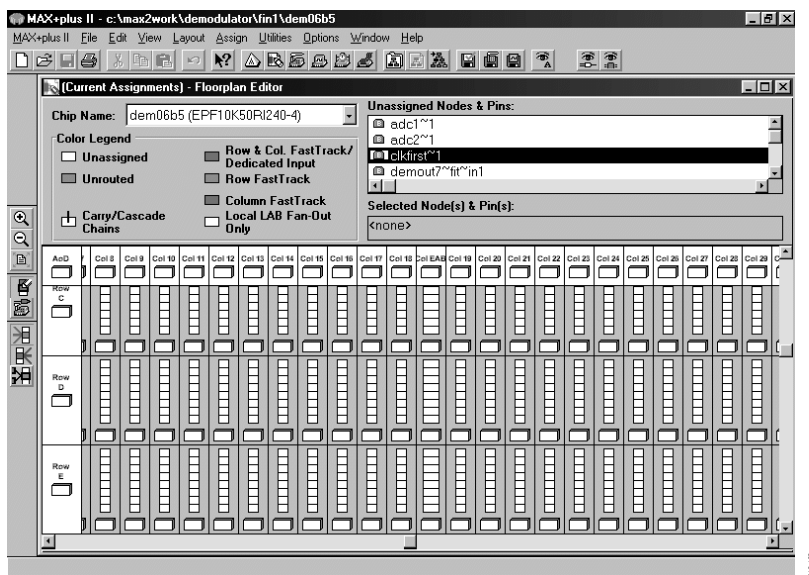


Рис. 2.25. Поуровневый планировщик MAX+PLUS II

2.4. Процесс компиляции

Сначала компилятор извлекает информацию об иерархических связях между файлами проекта и проверяет проект на простые ошибки ввода дизайнов. Он создает организационную карту проекта и затем, комбинируя все файлы проекта, превращает их в базу данных без иерархии, которую может эффективно обрабатывать.

Компилятор применяет разнообразные способы увеличения эффективности проекта и минимизации использования ресурсов устройства. Если проект слишком большой, чтобы быть реализованным в одной ПЛИС, компилятор может автоматически разбить его на части для реализации в нескольких устройствах того же самого семейства ПЛИС, при этом минимизируется число соединений между устройствами. В файле отчета (.rpt) затем будет отражено, как проект будет реализован в одном или нескольких устройствах.

Кроме того, компилятор создает файлы программирования или загрузки, которые программатор системы MAX+PLUS II или другой системы бу-

дет использовать для программирования одной или нескольких ПЛИС фирмы «Altera».

Несмотря на то что компилятор может автоматически компилировать проект, существует возможность задать обработку проекта в соответствии с точными указаниями разработчика. Например возможно задать стиль логического синтеза проекта по умолчанию и другие параметры логического синтеза в рамках всего проекта. Кроме того, удобно задать временные требования в рамках всего проекта, точно задать разбиение большого проекта на части для реализации в нескольких устройствах и выбрать варианты параметров устройств, которые будут применены для всего проекта в целом. Вы можете также выбрать, сколько выводов и логических элементов должно быть оставлено неиспользованными во время текущей компиляции, чтобы зарезервировать их для последующих модификаций проекта.

Компиляцию можно запустить из любого приложения MAX+PLUS II или из окна компилятора. Компилятор автоматически обрабатывает все входные файлы текущего проекта. Процесс компиляции можно наблюдать в окне компилятора в следующем виде (**Рис. 2.26**):

- опустошаются и переворачиваются песочные часы, что указывает на активность компилятора;
- высвечиваются прямоугольники модулей компилятора по очереди, по мере того как компилятор завершает каждый этап обработки;
- под прямоугольником модуля компилятора появляется пиктограмма выходного файла, сгенерированного данным модулем. Для открытия соответствующего файла следует дважды щелкнуть левой кнопкой мыши на пиктограмме, и он откроется;
- процент завершения компиляции постепенно увеличивается (до 100%), что отражается также растущим прямоугольником («градусник»);

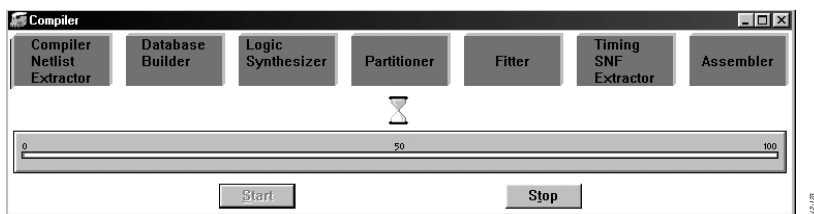


Рис. 2.26. Процесс компиляции

— во время разбиения и монтажа кнопка компилятора Stop (стоп) превращается в кнопку Stop/Show Status (стоп/показать состояние), которую вы можете выбрать для открытия диалогового окна, в котором отражается текущее состояние разбиения и монтажа проекта;

— при обнаружении в процессе компиляции каких-либо ошибок или возможных проблем автоматически открывается окно обработчика сообщений, в котором отображается список сообщений об ошибке, предупреждающих и информационных сообщений, а также сразу дается справка по исправлению ошибки. Кроме того, вы можете определить источники сообщений в файлах проекта или в его поуровневом плане назначений.

Компилятор может работать в фоновом режиме. Вы можете уменьшить до минимума окно компилятора, пока он обрабатывает проект, и продолжить работу над другими файлами проекта. Растущий прямоугольник («градусник») под пиктограммой уменьшенного окна компилятора позволяет вам наблюдать за продвижением процесса компиляции, и в то же время вы можете сосредоточить свое внимание на другой задаче. Однако следует помнить, что такая роскошь, как многозадачная работа, возможна только на высокопроизводительном ПК. Если существует дефицит оперативной памяти, то процесс компиляции займет много времени. Рекомендуемая конфигурация ПК дана в разделе 2.1.

Компилятор системы MAX+PLUS II обрабатывает проект, используя следующие модули и утилиты:

- Compiler Netlist Extractor (экстрактор форматов), включающий встроенные программы чтения форматов EDIF, VHDL, Verilog и XNF;
- Database Builder (построитель базы данных);
- Logic Synthesizer (логический синтезатор);
- Partitioner (разделитель);
- Fitter (трассировщик);
- Functional SNF Extractor (экстрактор для функционального тестирования);
- Timing SNF Extractor (экстрактор для тестирования временных параметров);
- Linked SNF Extractor (экстрактор для тестирования компоновки);
- EDIF Netlist Writer (программа записи выходного файла в формат EDIF);
- Verilog Netlist Writer (программа записи выходного файла в формат Verilog);
- VHDL Netlist Writer VHDL (программа записи выходного файла в формат VHDL);

- Assembler (модуль ассемблера);
- Design Doctor Utility (утилита диагностики проекта).

Модуль Compiler Netlist Extractor преобразует каждый файл проекта в один или несколько двоичных файлов с расширением .cnf. (compiler netlist file). Поскольку компилятор подставляет значения всех параметров, используемых в параметризованных функциях, содержимое файла CNF может меняться при последовательной компиляции, если значения параметров меняются. Данный модуль создает также файл иерархических взаимосвязей (.hif, hierarchy interconnect file), в котором документируются иерархические связи между файлами проекта, а также содержится информация, необходимая для показа иерархического дерева проекта в окне Hierarchy Display. Кроме того, данный модуль создает файл базы данных узлов (.ndb, node database), в котором содержатся имена узлов проекта для базы данных назначений ресурсов. Встроенные программы чтения форматов EDIF, VHDL, Verilog и XNF автоматически транслируют информацию проекта в файлы соответствующих форматов .edf, .vhd, .v, .xnf в формат, совместимый с системой MAX+PLUS II. Программа чтения формата EDIF обрабатывает входные файлы EDIF с помощью библиотечных файлов (.lmf, library mapping file), которые устанавливают соответствие между логическими функциями, разработанными в других САПР, и функциями системы MAX+PLUS II. Программа чтения формата XNF может создавать файл для экспорта текстового дизайна (.tdx, text design export file), который содержит информацию на языке AHDL, которая эквивалентна той, что содержится в файле формата XNF (.xnf). Это делается для того, чтобы редактировать проект на языке AHDL.

Модуль Database Builder использует файл иерархических связей HIF для компоновки созданных компилятором файлов CNF, в которых содержится описание проекта. На основании данных об иерархической структуре проекта данный модуль копирует каждый файл CNF в одну базу данных без иерархической структуры. Таким образом, эта база данных сохраняет электрическую связность проекта.

При создании базы данных модуль исследует логическую полноту и согласованность проекта, а также проверяет пограничную связность и наличие синтаксических ошибок (например узел без источника или места назначения). На этой стадии компиляции обнаруживается большинство ошибок, которые могут быть тут же легко исправлены. Каждый модуль компилятора последовательно обрабатывает и обновляет эту базу данных.

Первый раз, когда компилятор обрабатывает проект, все файлы проекта компилируются. Вы можете использовать возможность «быстрой повторной компиляции» (smart recompile) для создания расширенной базы данных проекта, которая помогает ускорить последующие компиляции. Эта база данных позволяет вам изменить назначения ресурсов физического устройства, такие как назначения выводов и логических элементов, а также повторно компилировать проект без повторного построения базы данных и повторного синтеза логики проекта. Используя возможность «полной повторной компиляции» (total recompile), возможно сделать выбор между повторной компиляцией только тех файлов, которые редактировались после последней компиляции, и полной повторной компиляцией всего проекта.

Модуль Logic Synthesizer применяет ряд алгоритмов, которые уменьшают использование ресурсов и убирают дублированную логику, обеспечивая тем самым максимально эффективное использование структуры логического элемента для архитектуры целевого семейства устройств. Данный модуль компилятора применяет также способы логического синтеза для требований пользователя по временным параметрам и др. Кроме того, логический синтезатор ищет логику для несоединенных узлов. Если находит несоединенный узел, он убирает примитивы, относящиеся к этому узлу.

Для управления логическим синтезом имеются три заранее описанных стиля и большое число логических опций. В любом приложении MAX+PLUS II можно ввести значения временных параметров проекта и выбрать логические опции, а также определить стили логического синтеза. Вы можете задать глобальный логический синтез по умолчанию и временные параметры проекта для всего проекта в целом, а также все дополнительные назначения логических опций и временных параметров проекта и для отдельных логических функций.

Если проект не помещается при монтаже в одно устройство, модуль Partitioner разделяет базу данных, обновленную логическим синтезатором, на несколько ПЛИС одного и того же семейства, стараясь при этом разделить проект на минимально возможное число устройств. Разбиение проекта происходит по границам логических элементов, а число выводов, используемое для сообщения между устройствами, минимизируется.

Разбиение может быть проведено полностью автоматически либо под частичным или полным управлением со стороны пользователя. Назначения устройств и установки для автоматического выбора устройств позволяют применить тот уровень управления, который наиболее подходит для конкретного проекта.

Когда работают модули Partitioner и Fitter, вы можете приостановить компиляцию. Компилятор отобразит информацию о текущем состоянии процессов разбиения и трассировки кристалла, в том числе сравнение требуемых и имеющихся ресурсов. Это нужно для того, чтобы принять решение, продолжать ли компиляцию или вносить кардинальные изменения в проект.

Используя базу данных, обновленную модулем разбиения, модуль Fitter приводит в соответствие требования проекта с известными ресурсами одного или нескольких устройств. Он назначает каждой логической функции расположение реализующего ее логического элемента и выбирает соответствующие пути взаимных соединений и назначения выводов. Данный модуль пытается согласовать назначения ресурсов, т.е. выводов, логических элементов, элементов ввода/вывода, ячеек памяти, чипов, клик, устройств, местной трассировки, временных параметров и назначения соединенных выводов из файла назначений и конфигурации (.acf, Assignment & Configuration file), с имеющимися ресурсами. Модуль имеет параметры, позволяющие определить способы трассировки, например автоматическое введение логических элементов или ограничение коэффициента объединения по входу. Если трассировка не может быть выполнена, модуль выдает сообщение и предлагает вам выбор, проигнорировать некоторые или все ваши назначения либо прекратить компиляцию.

Независимо от того, завершена ли полная трассировка проекта, данный модуль генерирует файл отчета (.rpt, report file), в котором документируется информация о разбиении проекта, именах входных и выходных контактов, временных параметрах проекта и неиспользованных ресурсах для каждого устройства в проекте. Вы можете включить в файл отчета разделы, показывающие назначения пользователя, файловую иерархию, взаимные соединения логических элементов и уравнения, реализованные в ЛЭ.

Компилятор также автоматически создает файл трассировки (.fit), в котором документируются назначения ресурсов и устройств для всего проекта, а также информация о трассировке. Независимо от того, успешно ли прошла трассировка, пользователь может просмотреть информацию о согласовании, разбиении и трассировке из файла согласования в окне поуровневого планировщика. Возможно также переписать назначения из файла согласования в файл назначений и конфигурации ACF для последующего редактирования.

Существует возможность дать указание модулю Fitter сгенерировать выходные текстовые файлы проекта на языке AHDL (.tdo). Поскольку в проекте с несколькими устройствами для одного устройства генерируется

один файл, следует разбить проект на несколько проектов, каждый для одного устройства, если требуется зафиксировать логику в некоторых устройствах, затем сохранить файл TDO для этого устройства как файл текстового дизайна (.tdf) и перекомпилировать логику для этого устройства, сохраняя при этом результаты логического синтеза, полученные в ходе предыдущей компиляции.

Functional SNF Extractor создает файл для функционального тестирования (.snf). Компилятор генерирует этот файл перед синтезом проекта, поэтому он содержит все узлы, присутствующие в первоначальных файлах проекта. Этот функциональный файл SNF не содержит информацию о временных параметрах. Его генерация возможна только в случае, если компиляция проекта прошла без ошибок.

Timing SNF Extractor создает (если компиляция проекта прошла без ошибок) файл для тестирования временных параметров (.snf), который содержит данные о временных параметрах проекта. Этот файл используется для тестирования и анализа временных параметров. Кроме того, эти файлы SNF используют также модули компилятора, содержащие программы записи в форматы EDIF, Verilog и VHDL, генерирующие выходные файлы этих форматов и также (по желанию пользователя) выходные файлы формата стандартных задержек (.sdo, standart delay format output file).

Можно задать компилятору сгенерировать оптимизированный файл SNF с помощью команды Processing/Timing SNF Extractor. Оптимизация файла SNF увеличивает время компиляции, но помогает сэкономить ваше время при тестировании и анализе временных параметров.

Linked SNF Extractor создает файл (.snf) для тестирования компоновки при тестировании нескольких проектов (на уровне платы). Такой файл SNF комбинирует информацию из файлов SNF двух типов: для тестирования временных параметров и функционального тестирования, которые были сгенерированы для этих нескольких проектов по отдельности. Скомпонованные проекты могут использовать устройства, принадлежащие разным семействам. Если файл для тестирования компоновки содержит только информацию о временных параметрах, его можно также использовать при прогоне анализа временных параметров.

EDIF Netlist Writer. Компилятор MAX+PLUS II может взаимодействовать с большинством стандартных программных средств САПР, которые могут читать файлы стандартного формата EDIF 200 или 300. Данный (необязательный) модуль компилятора, содержащий программу записи в фор-

мат EDIF, создает один или несколько выходных файлов в формате EDIF (.edo), содержащих информацию о функциях и (необязательно) временных параметрах, полученную после проведения синтеза. Информация о временных параметрах может быть также записана в отдельные выходные файлы формата стандартной задержки (.sdo).

Verilog Netlist Writer (необязательный модуль программы записи в формат Verilog) генерирует выходные файлы с расширением .vo, содержащие информацию о функциях и временных параметрах, полученную после проведения синтеза. Информация о временных параметрах может быть также записана в отдельные выходные файлы формата стандартной задержки (.sdo).

VHDL Netlist Writer VHDL (необязательный модуль компилятора с программой записи в формат VHDL) генерирует один или несколько выходных файлов (.vho) на языке VHDL с синтаксисом 1987 или 1993, содержащих информацию о функциях и (необязательно) временных параметрах, полученную после проведения синтеза. Информация о временных параметрах может быть также записана в отдельные выходные файлы формата стандартной задержки (.sdo).

Выходные файлы на языках описания аппаратуры можно использовать при верификации проекта с использованием внешнего симулятора. Эти файлы генерируются только в случае успешной компиляции проекта.

Модуль Assembler преобразует назначения логических элементов, выводов и устройства, сделанные модулем Fitter, в программный образ для устройства (устройств) в виде одного или нескольких двоичных объектных файлов для программатора (.prof) или объектных файлов SRAM (.sof); для некоторых устройств компилятор также генерирует ASCII-файлы JEDEC (.jed), содержащие информацию для программатора, конфигурационные ASCII-файлы (.tff) и ASCII-файлы формата Intel (.hex). Объектные файлы POF и SOF, а также конфигурационные файлы JEDEC затем обрабатываются программатором системы MAX+PLUS II и программирующей аппаратурой фирмы «Altera» (или другим программатором). Файлы HEX и TTF могут быть использованы для конфигурирования устройств FLEX6000, FLEX8000 и FLEX10K с помощью других средств. Модуль ассемблера создает файлы для программатора только в случае успешной компиляции проекта.

После завершения компиляции компилятор и программатор системы MAX+PLUS II позволяют сгенерировать дополнительные файлы для программирования устройства, которые можно использовать в других услови-

ях программирования. Например можно создать файлы с последовательным потоком битов (.bbs) и необработанные двоичные файлы (.rbf) для конфигурирования устройств FLEX6000, FLEX8000 и FLEX10K. Можно создать последовательные файлы тестовых векторов (.svf) или файлы на языке JAM (.jam) для программирования устройств в автоматизированной испытательной аппаратуре типа ATE.

Design Doctor Utility (необязательная утилита диагностики) проекта проверяет логику каждого файла проекта для выявления элементов, которые могут вызвать проблемы надежности на системном уровне. Эти проблемы обычно обнаруживаются только после запуска устройства «в железе».



Существует возможность выбора одного из трех предварительно определенных наборов правил разработки проекта с разными уровнями. Кроме того, вы можете разработать свой собственный свод правил разработки дизайна.

Правила разработки дизайна основываются на принципах надежности, которые охватывают логику, содержащую асинхронные входы, тактовые сигналы (Clock), многоуровневую логику на конфигурациях с сигналами Clock, Preset и Clear, а также в условиях состязаний. Установка правил проверки производится с помощью команды Design Doctor Settings (**Рис. 2.27**).

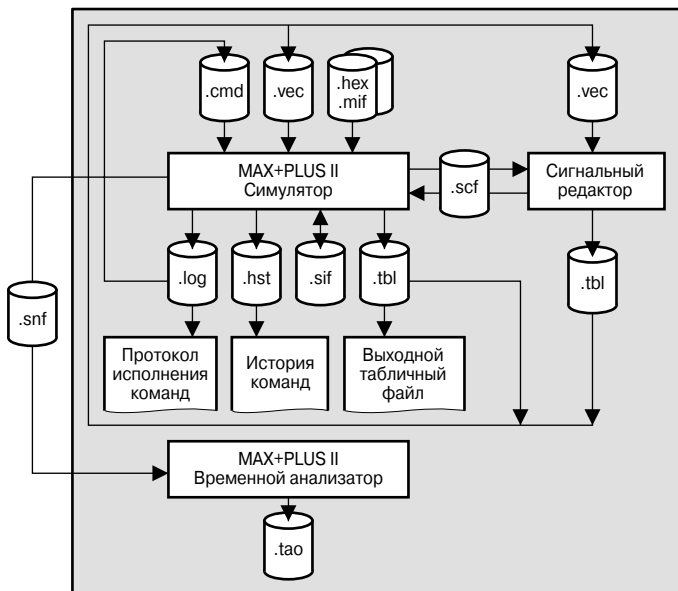
Рис. 2.27. Окно команды
Design Doctor Settings

2.5. Верификация проекта

Для верификации проекта (**Рис. 2.28**) в системе MAX+PLUS II используются три приложения: симулятор, анализатор временных параметров и сигнальный редактор.

Симулятор (Simulator)

Симулятор системы MAX+PLUS II тестирует логические операции и внутреннюю синхронизацию проекта, позволяя пользователю моделировать



А.В.ИП

Рис. 2.28. Верификация проекта в системе MAX+PLUS II

проект. Симулятор может работать или в диалоговом, или автоматическом (пакетном) режимах. Окно симулятора показано на **Рис. 2.29**.

Перед тестированием проект необходимо скомпилировать, задав компилятору опцию сгенерировать файл (.snf) для функционального тестирования, тестирования временных параметров либо тестирования компоновки нескольких проектов (устройств). Затем полученный для текущего проекта файл SNF загружается автоматически при открытии симулятора.

В качестве источника входных векторов используются либо графический сигнальный файл каналов тестирования (.scf), либо текстовый ASCII-файл (.vec). Для проектов, работающих с памятью, можно задать некое исходное содержимое памяти в файлах шестнадцатеричного формата («Intel») с расширением .hex или в файлах инициализации памяти с расширением .mif. Сигнальный редактор может автоматически создавать файл SCF по умолчанию, который пользователь может редактировать с целью получения нужных ему тестовых входных векторов. Если вместо этого используется

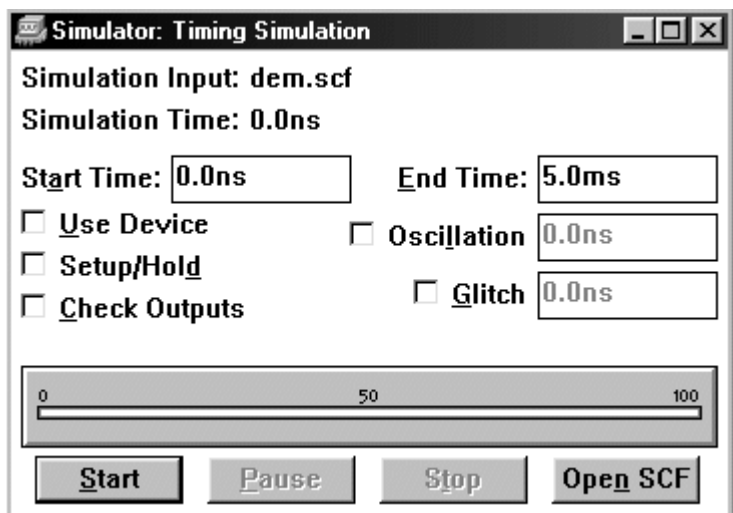


Рис. 2.29. Симулятор MAX+PLUS II

текстовый ASCII-файл векторов, сигнальный редактор автоматически генерирует из него файл каналов тестирования SCF (Simulator Chanel File).

Симулятор позволяет проверить выходные значения, получаемые в ходе тестирования, по выходам, содержащимся в файле SCF (заданным пользователем прогнозируемым значениям или результатам предыдущих тестов). С помощью соответствующей аппаратуры для программирования можно также выполнить функциональное тестирование для проверки действительных значений выходов программируемого устройства по результатам тестирования.

Используя различные опции симулятора, можно контролировать проект на появление сбоев (glitches), а также нарушение установочных параметров и временных задержек. После завершения тестирования можно открыть сигнальный редактор для просмотра обновленного файла SCF или сохранить полученные выходные значения в табличном файле с расширением .tbl, а затем просматривать результаты в текстовом редакторе.

Функциональное тестирование. Если компилятору «дано задание» сгенерировать файл SNF для функционального тестирования, он создает его

перед синтезом проекта. Следовательно, при функциональном тестировании можно смоделировать все узлы проекта.

Во время функционального тестирования симулятор игнорирует все задержки распространения. Поэтому в файле SNF для функционального тестирования нет задержек, выходные логические уровни изменяются одновременно со входными векторами.

Тестирование временных параметров. Файл SNF для тестирования временных параметров компилятор генерирует, после того как проведены полный синтез и оптимизация проекта. Поэтому этот файл содержит только те узлы, которые не были уничтожены в процессе логического синтеза. Из этого файла симулятор берет информацию об аппаратной части, которая была собрана из файлов моделей устройств (.dmf), имеющихся в комплекте системы MAX+PLUS II.

Если проект был разбит на несколько устройств, компилятор создает файл SNF для проекта в целом и для каждого устройства. Однако тестирование временных параметров осуществляется только для проекта в целом.

Можно ускорить тестирование временных параметров, выдав компилятору указание сгенерировать оптимизированный файл SNF, содержащий динамические модели, которые представляют собой разные типы комбинаторной логики. Процессорное время компилятора при этом увеличивается, однако полученный оптимизированный SNF может уменьшить время тестирования, поскольку симулятор может работать с динамическими моделями, вместо того чтобы интерпретировать всю логику в комбинаторной схеме.

При создании файла SNF для тестирования компоновки нескольких проектов компилятор комбинирует файлы SNF для функционального тестирования и/или файлы для тестирования временных параметров нескольких отдельных проектов. Отдельные подпроекты в компоновочном SNF могут быть предназначены для устройств разных семейств. Кроме того, поскольку файлы SNF для функционального тестирования создаются до окончания полной компиляции, можно ввести подпроекты, которые представляют логику, не реализованную в устройстве фирмы «Altera».

Компоновочный файл SNF можно использовать для тестирования на уровне платы. Кроме того, если он содержит только информацию о временных параметрах, его можно использовать для прогона временного анализатора системы MAX+PLUS II.

Вместе с другими приложениями системы MAX+PLUS II симулятор позволяет вам выполнять следующие задачи:

- задать ожидаемые логические уровни на выходе, которые можно будет сравнить с результатами тестирования;

- смоделировать отдельные узлы или узлы, объединенные в группы. Можно комбинировать биты цифрового автомата в проекте, моделировать их как группу и обращаться к ним по имени состояния;

- определить временной интервал, представляющий собой дрожание фронта импульса или сбой, и проанализировать проект на наличие обоих этих условий или одного из них;

- контролировать наличие в проекте нарушений начальных установок регистров и временных задержек;

- регистрировать действительные значения выходов устройств вместо моделированных;

- проводить функциональное тестирование. Можно проверить, являются ли смоделированные выходные значения функционально эквивалентными реальным выходам устройств;

- задавать условия точки прерывания, которая заставляет симулятор делать паузу при их реализации в процессе тестирования;

- составлять перечень имен и логических уровней любой комбинации узлов и групп и инициализировать логические уровни узла или группы перед тестированием;

- инициализировать содержимое блоков памяти RAM (ОЗУ) или ROM (ПЗУ) перед тестированием;

- сохранять инициализированные значения узлов и групп, в том числе инициализированное содержимое памяти, в файле инициализации симулятора (.sif) или перезагружать инициализированные значения, хранящиеся в файле;

- регистрировать команды симулятора в текстовом файле протокола испытаний (.log) того же формата, что и командный файл (.cmd), используемый при тестировании в автоматическом (пакетном) режиме, а затем использовать этот файл LOG для повторения этого цикла тестирования. Команды симулятора и полученные результаты можно также записать в тестовый файл истории тестирования с расширением .hst.

Таким образом, мы рассмотрели основные приемы работы пакета MAX+PLUS II. Конечно, в рамках одной главы практически невозможно подробно рассмотреть все приемы работы с таким сложным и разнообразным программным средством, однако заинтересованный пользователь в состоянии самостоятельно освоить пакет, используя данную книгу и фирменное руководство пользователя.

Глава 3. **Язык описания аппаратуры AHDL**

3.1. Общие сведения

Язык описания аппаратуры AHDL разработан фирмой «Altera» и предназначен для описания комбинационных и последовательностных логических устройств, групповых операций, цифровых автоматов (state machine) и таблиц истинности с учетом архитектурных особенностей ПЛИС фирмы «Altera». Он полностью интегрируется с системой автоматизированного проектирования ПЛИС MAX+PLUS II. Файлы описания аппаратуры, написанные на языке AHDL, имеют расширение .TDF (text design file). Для создания TDF-файла можно использовать как текстовый редактор системы MAX+PLUS II, так и любой другой. Проект, выполненный в виде TDF-файла, компилируется, отлаживается и используется для формирования файла программирования или загрузки ПЛИС фирмы «Altera».

Операторы и элементы языка AHDL являются достаточно мощным и универсальным средством описания алгоритмов функционирования цифровых устройств, удобным в использовании. Язык описания аппаратуры AHDL дает возможность создавать иерархические проекты в рамках одного этого языка или же в иерархическом проекте использовать как TDF-файлы, разработанные на языке AHDL, так и другие типы файлов. Для создания проектов на AHDL можно, конечно, пользоваться любым текстовым редактором, но текстовый редактор системы MAX+PLUS II предоставляет ряд дополнительных возможностей для ввода, компиляции и отладки проектов (см. главу 2).

Проекты, созданные на языке AHDL, легко внедряются в иерархическую структуру. Система MAX+PLUS II позволяет автоматически создать символ компонента, алгоритм функционирования которого описывается

TDF-файлом, и затем вставить его в файл схемного описания (GDF-файл). Подобным же образом можно вводить собственные функции разработчика, помимо порядка 300 макрофункций, разработанных фирмой «Altera», в любой TDF-файл. Для всех функций, включенных в макробиблиотеку системы MAX+PLUS II, фирма «Altera» предоставляет файлы с расширением .inc, которые используются в операторе включения INCLUDE.

При распределении ресурсов устройств разработчик может пользоваться командами текстового редактора или операторами языка AHDL для того, чтобы сделать назначения ресурсов и устройств. Кроме того, разработчик может только проверить синтаксис или выполнить полную компиляцию для отладки и запуска проекта. Любые ошибки автоматически обнаруживаются обработчиком сообщений и высвечиваются в окне текстового редактора.

При работе с AHDL следует соблюдать так называемые «золотые правила» (golden rules). Выполнение этих правил позволит эффективно применять язык AHDL и избежать многих ошибок:

- Соблюдайте форматы и правила присвоения имен, описанные в руководстве по стилям AHDL, чтобы программа была читаемой и содержала меньше ошибок.
- Несмотря на то, что язык AHDL не различает прописные и строчные буквы, фирма «Altera» рекомендует для улучшения читаемости использовать прописные буквы для ключевых слов.
- Не применяйте вложенные конструкции условного оператора IF, если можно использовать оператор выбора CASE.
- Строка в TDF-файле может быть длиной до 255 символов. Однако следует стремиться к длине строки, уместящейся на экране. Строки заканчиваются нажатием клавиши Enter.
- Новую строку можно начинать в любом свободном месте, т.е. на местах пустых строк, табуляций, пробелов. Основные конструкции языка отделяются пустым пространством.
- Ключевые слова, имена и числа должны разделяться соответствующими символами или операторами и/или одним или более пробелами.
- Комментарии должны быть заключены в символы процента (%). Комментарий может включать любой символ, кроме символа %, поскольку компилятор системы MAX+PLUS II игнорирует все, заключенное в символы процента. Комментарии не могут быть вложенными.
- При соединении одного примитива с другим используйте только «раз-

решенные» связи между ними, не все примитивы могут соединяться друг с другом.

- Используйте только макрофункции EXPDFF, EXPLATCH, NANDLTCH и NORLTCH, входящие в макробибблиотеку системы MAX+PLUS II. Не создавайте свои собственные структуры перекрестных связей. Избегайте многократного связывания вместе EXPDFF, EXPLATCH, NANDLTCH и NORLTCH. Многочисленные примеры этих макрофункций должны всегда разделяться примитивами LCELL.

Общие «золотые правила» ввода проекта:

- Если многочисленные двунаправленные или выходные выводы связаны вместе, разработчик не может использовать оператор Pin Connection для соединения выводов при функциональном моделировании с аппаратной поддержкой или функциональном тестировании.
- Нет необходимости создавать прототипы функций для примитивов. Однако разработчик может переопределить примитивы в объявлениях прототипов функций для изменения порядка вызова входов в вашем TDF-файле.
- Не редактируйте файл FIT. Если разработчик желает отредактировать назначения для проекта, необходимо сохранить сначала файл FIT как TDF-файл или сделать обратное назначение с помощью команды Project Back-Annotate и отредактировать их с помощью команд Chip to Device, Pin/LC/Chip и Enter Assignments.
- Если разработчик хочет загрузить регистр по определенному фронту глобального тактового сигнала Clock, фирма «Altera» рекомендует, когда регистр загружен, использовать для управления вход Clock Enable одного из триггеров типа Enable: DFFE, TFFE, JKFFE или SRFFE;
- Когда разработчик начинает работать с новым файлом проекта, сразу же необходимо задать семейство ПЛИС, на которое ориентирован проект, с помощью конструкции Family для того, чтобы в дальнейшем иметь возможность воспользоваться макрофункциями, специфичными для данного семейства. Если разработчик не задаст семейство, оно будет считаться таким же, как и в текущем проекте.
- Используйте опцию Design Doctor для проверки надежности логики проекта во время компиляции.
- Предоставляемые по умолчанию фирмой «Altera» стили для логического синтеза имеют разные установки для разных семейств устройств, что обеспечивает более эффективное использование архитектуры каж-

дого устройства. Когда разработчик использует какой-нибудь из этих стилей, его установки изменятся при переходе к другому семейству устройств. После смены семейства необходимо проверить новые установки стиля.

Общие «золотые правила» системы MAX+PLUS II:

- В начале работы над новым проектом рекомендуется сразу же задать его имя как имя нового проекта с помощью меню Project Name для того, чтобы в дальнейшем его было легко компилировать. Позже разработчик всегда может изменить имя проекта.
- Используйте иерархические возможности системы MAX+PLUS II для перемещения между файлами проекта. Для открытия файла нижнего уровня иерархии откройте файл верхнего уровня, затем используйте окно Hierarchy Display или Hierarchy Down и откройте файл более низкого уровня. Если разработчик выбирает Open или Retrieve для открытия файла нижнего уровня, считается, что он является файлом верхнего уровня нового дерева, и все назначения ресурсов, устройств и зондов сохраняются только для этой новой иерархии, а не для текущего проекта.
- Если разработчик создает вспомогательный файл для какого-нибудь проекта, пиктограмма этого файла появится в окне Hierarchy Display в том случае, если используется то же имя, что и для проекта.
- Не рекомендуется редактировать системные файлы MAX+PLUS II, в том числе файлы с расширением .prb, файлы HIF или maxplus2.ini либо файлы <имя проекта>.ini.
- Если разработчик желает переименовать файл проекта или вспомогательный файл, следует использовать команду Save As... Не рекомендуется переименовывать файлы проекта вне среды MAX+PLUS II (например из MS-DOS или Windows File Manager).
- После завершения проекта рекомендуется выполнить команду архивирования проекта Project Archive для создания резервной копии всего проекта (всех его файлов). На архивированную копию не повлияют никакие последующие редактирования.

3.2. Использование чисел и констант в языке AHDL

3.2.1. Использование чисел

Числа используются для представления констант в булевых выражениях и уравнениях. Язык AHDL поддерживает все комбинации десятичных, двоичных, восьмеричных и шестнадцатеричных чисел.

Ниже приведен файл `decode1.tdf`, который представляет собой дешифратор адреса, генерирующий высокий активный уровень сигнала разрешения доступа к шине, если адрес равен шестнадцатеричному числу `370h`.

```
SUBDESIGN decode1
(
    address[15..0] : INPUT;
    chip_enable : OUTPUT;
)
BEGIN
    chip_enable = (address[15..0] == H"0370");
END;
```

В этом примере десятичные числа использованы для указания размерности массива бит, которым записывается адрес шины. Шестнадцатеричным числом `H"0370"` записано значение адреса, при котором обеспечивается высокий уровень сигнала.

3.2.2. Использование констант

В файле AHDL можно использовать константы для описательных имен разных чисел. Такое имя, используемое на протяжении всего файла, может быть более информативным, чем число; например имя `UPPER_LI` несет больше информации, чем число `103`. В языке AHDL константы вводятся объявлением `CONSTANT`. Приведенный выше файл можно записать по-другому, используя вместо числа `H"0370"` константу `IO_ADDRESS`.

```
CONSTANT IO_ADDRESS = H"0370";
SUBDESIGN decode2
(
```

```
    a[15..0] : INPUT;  
    ce : OUTPUT;  
  )  
BEGIN  
    ce = (a[15..0] == IO_ADDRESS);  
END;
```

Преимущество использования констант особенно заметно, если одно и то же число используется в файле несколько раз. Тогда, если его нужно изменить, меняют его только один раз в объявлении константы.

3.3. Комбинационная логика

Как известно, логическая схема называется комбинационной, если в заданный момент времени выходы являются только функциями входов в этот момент времени. Комбинационная логика в языке AHDL реализована булевыми выражениями и уравнениями, таблицами истинности и большим количеством макрофункций. В число примеров комбинаторных логических функций входят дешифраторы, мультиплексоры и сумматоры.

3.3.1. Реализация булевых выражений и уравнений

Булевы выражения — это множество узлов, чисел, констант и других булевых выражений, выделяемых операторами, компараторами, и, возможно, сгруппированные в заключающих круглых скобках. Булево уравнение устанавливает равенство между узлом или группой и булевым выражением. В качестве примера приведен файл `boole1.tdf`, в котором даны два простых булевых выражения, представляющие два логических элемента.

```
SUBDESIGN boole1  
(  
    a0, a1, b : INPUT;  
    out1, out2 : OUTPUT;  
)  
BEGIN  
    out1 = a1 & !a0;  
    out2 = out1 # b;  
END;
```

Здесь выход `out1` получается в результате логической операции «И», примененной ко входу `a1` и инвертированному входу `a0`, а выход `out2` — в результате применения логической операции «ИЛИ» к выходу `out1` и входу `b`. Поскольку эти уравнения обрабатываются одновременно, последовательность их следования в файле не важна.

3.3.2. Объявление NODE (узел)

Узел, который объявляется в секции переменных `VARIABLE` в объявлении `NODE`, можно использовать для хранения промежуточных выражений. Это полезно делать, если булево выражение повторяется несколько раз и его целесообразно заменить именем узла. Приведенный выше файл `boole1.tdf` можно переписать по-другому:

```
SUBDESIGN boole2
(
    a0, a1, b : INPUT;
    out : OUTPUT;
)
VARIABLE
    a_equals_2 : NODE;
BEGIN
    a_equals_2 = a1 & !a0;
    out = a_equals_2 # b;
END;
```

Здесь объявляется узел `a_equals_2`, и ему присваивается значение выражения `a1 & !a0`. Использование узлов помогает экономить ресурсы устройств, если узел используется в нескольких выражениях.

3.3.3. Определение групп

Группа может включать в себя до 256 элементов (бит), рассматривается как совокупность узлов и участвует в различных действиях как единое целое. В булевых уравнениях группа может быть приравнена булевому выражению, другой группой, одному узлу, `VCC`, `GND`, 1 или 0. В каждом случае значения группы разные.

Если группа определена, для краткого указания всего диапазона ставят две квадратные скобки `[]`. Например, группу `a[4..1]` можно кратко записать как `a[]`.

Примеры определения групп:

- При приравнении двух групп обе должны иметь одинаковое число бит. В приводимом ниже примере каждый бит первой группы соединяется с соответствующим битом второй группы, а именно d2 соединяется с q8, d1 с q7 и d0 с q6:
 $d[2..0] = q[8..6]$
- При приравнении группы одному узлу все биты группы присоединяются к узлу. В приводимом ниже примере d1, d2 и d0 присоединяются все вместе к n:
 $d[2..0] = n$
- При приравнении группы значению V_{CC} или GND все биты группы присоединяются к этому значению. В приводимом ниже примере d1, d2 и d0 присоединяются все вместе к V_{CC} :
 $d[2..0] = V_{CC}$
- При приравнении группы к 1 младший бит группы присоединяется к V_{CC} , а все другие биты — к значению GND. В приводимом ниже примере только d0 присоединяется к V_{CC} , значение 1 расширяется до представления B"001".
 $d[2..0] = 1$

3.3.4. Реализация условной логики

Условная логика делает выбор между режимами в зависимости от логических входов. Для реализации условной логики используются операторы IF или CASE.

В операторе IF оцениваются одно или несколько булевых выражений, и затем описываются режимы для разных значений этих выражений. В операторе CASE дается список альтернатив, которые имеются для каждого возможного значения некоторого выражения. Оператор оценивает значение выражения и по нему выбирает режим в соответствии со списком.

3.3.4.1. Логика оператора IF

В качестве примера рассмотрим файл priority.tdf, в котором описан кодировщик приоритета, который преобразует уровень самого приоритетного активного входа в значение. Он генерирует двухразрядный код, показывающий вход с наивысшим приоритетом, запускаемый V_{CC} .

```

SUBDESIGN priority
(
    low, middle, high : INPUT;
    highest_level[1..0] : OUTPUT;
)

BEGIN
    IF high THEN
        highest_level[] = 3;
    ELSIF middle THEN
        highest_level[] = 2;
    ELSIF low THEN
        highest_level[] = 1;
    ELSE
        highest_level[] = 0;
    END IF;
END;

```

Здесь оцениваются входы low, middle и high, чтобы определить, запущены ли они V_{CC} . На выходе получится код, соответствующий приоритету того входа, который был запущен V_{CC} . Если ни один вход не запущен, значение кода станет 0.

3.3.4.2. Логика оператора CASE

В качестве примера рассмотрим файл decoder.tdf, реализующий функции дешифратора, преобразующего код из двухразрядного в четырехразрядный. В результате его работы два двухразрядных двоичных входа преобразуются в один «горячий код», который так называется потому, что четыре его допустимых значения содержат по одной единице: 0001, 0010, 0100, 1000.

```

SUBDESIGN decoder
(
    code[1..0] : INPUT;
    out[3..0] : OUTPUT;
)
BEGIN

```

```

CASE code[] IS
WHEN 0 => out[] = B"0001";
WHEN 1 => out[] = B"0010";
WHEN 2 => out[] = B"0100";
WHEN 3 => out[] = B"1000";
END CASE;
END;
```

Здесь группа входа code [1..2] может принимать значения 0, 1, 2, 3. В зависимости от реального кода активизируется соответствующая ветвь оператора, и только она одна в данный момент времени. Например, если на входе code [] равен 1, на выходе out устанавливается значение B"0010".

3.3.4.3. Сравнение операторов IF и CASE

Операторы IF и CASE похожи. Иногда использование любого из них приводит к одним и тем же результатам:

IF THEN	CASE
IF a[] == 0 THEN y = c & d; ELSIF a[] == 1 THEN y = e & f; ELSIF a[] == 2 THEN y = g & h; ELSIF a[] == 3 THEN y = i; ELSE y = GND; END IF;	CASE a[] IS WHEN 0 => y = c & d; WHEN 1 => y = e & f; WHEN 2 => y = g & h; WHEN 3 => y = i; WHEN OTHERS => y = GND; END CASE;

Однако между этими двумя операторами существуют несколько важных различий:

- В операторе IF можно использовать любое булево выражение. Каждое выражение, записываемое в предложении, следующем за IF или ELSEIF, может не быть связанным с другими выражениями в операторе. В операторе CASE одно-единственное выражение сравнивается с проверяемым значением.

- В результате интерпретации оператора IF может быть сгенерирована логика, слишком сложная для компилятора системы MAX+PLUS II. В приведенном ниже примере показано, как компилятор интерпретирует оператор IF. Если а и b сложные выражения, то инверсия каждого из них даст еще более сложное выражение.

IF a THEN	IF a THEN
c = d;	c = d;
	END IF;
ELSIF b THEN	IF !a & b THEN
c = e;	c = e;
	END IF;
ELSE	IF !a & !b THEN
c = f;	c = f;
END IF;	END IF;

3.3.5. Описание дешифраторов

Дешифратор содержит комбинаторную логику, которая преобразует входные схемы в выходные значения или задает выходные значения для входных схем. Для создания дешифратора в языке AHDL используется объявление таблицы истинности TABLE.

Ниже приведен файл 7segment.tdf, представляющий собой дешифратор, который задает логику схемы светодиодов. Светодиоды светятся на семисегментном дисплее для индикации шестнадцатеричной цифры (от 0 до 9 и буквы от A до F).

```
% -a- %
% f| |b %
% -g- %
% e| |c %
% -d- %
% %
% 0 1 2 3 4 5 6 7 8 9 A b C d E F %
% %

SUBDESIGN 7segment
```

```
(
  i[3..0] : INPUT;
  a, b, c, d, e, f, g : OUTPUT;
)
BEGIN
  TABLE
    i[3..0] => a, b, c, d, e, f, g;

    H"0" => 1, 1, 1, 1, 1, 1, 0;
    H"1" => 0, 1, 1, 0, 0, 0, 0;
    H"2" => 1, 1, 0, 1, 1, 0, 1;
    H"3" => 1, 1, 1, 1, 0, 0, 1;
    H"4" => 0, 1, 1, 0, 0, 1, 1;
    H"5" => 1, 0, 1, 1, 0, 1, 1;
    H"6" => 1, 0, 1, 1, 1, 1, 1;
    H"7" => 1, 1, 1, 0, 0, 0, 0;
    H"8" => 1, 1, 1, 1, 1, 1, 1;
    H"9" => 1, 1, 1, 1, 0, 1, 1;
    H"A" => 1, 1, 1, 0, 1, 1, 1;
    H"B" => 0, 0, 1, 1, 1, 1, 1;
    H"C" => 1, 0, 0, 1, 1, 1, 0;
    H"D" => 0, 1, 1, 1, 1, 0, 1;
    H"E" => 1, 0, 0, 1, 1, 1, 1;
    H"F" => 1, 0, 0, 0, 1, 1, 1;
  END TABLE;
END;
```

В этом примере перечислены все возможные шестнадцатеричные цифры (i[3..0]) и соответствующие им состояния светодиодов (a, b, c, d, e, f, g), которые обеспечивают «начертание» цифры на дисплее. Изображение светодиодов на дисплее дано в виде комментария перед файлом.

Ниже приведен файл дешифратора адреса decode3.tdf для шестнадцатизрядной микропроцессорной системы.

```
SUBDESIGN decode3
(
  addr[15..0], m/io : INPUT;
  rom, ram, print, sp[2..1] : OUTPUT;
```

```

)
BEGIN
  TABLE
    m/io, addr[15..0] => rom, ram, print, sp[];
    1, B"00XXXXXXXXXXXXXXXX" => 1, 0, 0, B"00";
    1, B"100XXXXXXXXXXXXXXXX" => 0, 1, 0, B"00";
    0, B"00000001010101110" => 0, 0, 1, B"00";
    0, B"00000001011011110" => 0, 0, 0, B"01";
    0, B"00000001101110000" => 0, 0, 0, B"10";
  END TABLE;
END;
```

В данном примере существуют тысячи возможных вариантов входа (адреса), поэтому было бы непрактично сводить их все в таблицу. Вместо этого можно пометить символом X незначительные, не влияющие на выход разряды. Например, выходной сигнал rom (ПЗУ) будет высоким для всех 16384 вариантов адреса addr[15..0], которые начинаются с 00. Поэтому вам нужно только указать общую для всех вариантов часть кода, т.е. 00, а в остальных разрядах кода поставить X. Такой прием позволит сделать проект, требующий меньше устройств и ресурсов.

Приведенный ниже пример decode4.tdf показывает использование стандартной параметризуемой функции lpm_decode в задаче разработки дешифратора, аналогичная примеру decode1.tdf.

```

INCLUDE "lpm_decode.inc";

SUBDESIGN decode4
(
  address[15..0] : INPUT;
  chip_enable : OUTPUT;
)
BEGIN
  chip_enable = lpm_decode(.data[]=address[])
  WITH (LPM_WIDTH=16, LPM_DECODES=2^10)
  RETURNS (.eq[H"0370"]);
END;
```

3.3.6. Использование для переменных значений по умолчанию

Можно определить значения по умолчанию для узла или группы, которые будут автоматически использоваться для них, если в файле их значения не будут заданы. Язык AHDL позволяет присваивать значение узлу или группе в файле неоднократно. Если при этом произойдет конфликт, система автоматически будет использовать значения по умолчанию. Если значения по умолчанию не были заданы, используется значение GND. Объявление значений по умолчанию DEFAULTS можно использовать для задания переменных в таблице истинности, операторах IF и CASE.

Примечание. Не следует путать значения по умолчанию для переменных со значениями по умолчанию для портов, которые задаются в секции SUBDESIGN.

Ниже приводится файл default1.tdf, в котором происходит оценка входов и выбор соответствующего ASCII кода.

```
SUBDESIGN default1
(
    i[3..0] : INPUT;
    ascii_code[7..0] : OUTPUT;
)
BEGIN
    DEFAULTS
    ascii_code[] = B"00111111";% ASCII question
    mark"? " % END DEFAULTS;
    TABLE
    i[3..0] => ascii_code[];
    B"1000" => B"01100001"; % "a" %
    B"0100" => B"01100010"; % "b" %
    B"0010" => B"01100011"; % "c" %
    B"0001" => B"01100100"; % "d" %
    END TABLE;
END;
```

Если значение входа совпадает с одним из значений в левой части таблицы, код на выходе приобретает соответствующее значение ASCII кода в правой части таблицы. Если входное значение не совпадает ни с одним из (в левой колонке) табличных, выходу будет присвоено значение по умолчанию B"00111111" (вопросительный знак).

В приведенном ниже файле default2.tdf показано, как при многократном присваивании узлу разных значений возникает конфликт и как он решается средствами AHDL.

```
SUBDESIGN default2
(
  a, b, c : INPUT;
  select_a, select_b, select_c : INPUT;
  wire_or, wire_and : OUTPUT;
)
BEGIN
    DEFAULTS
        wire_or = GND;
        wire_and = VCC;
    END DEFAULTS;
    IF select_a THEN
        wire_or = a;
        wire_and = a;
    END IF;
    IF select_b THEN
        wire_or = b;
        wire_and = b;
    END IF;
    IF select_c THEN
        wire_or = c;
        wire_and = c;
    END IF;
END;
```

В данном примере выход `wire_or` устанавливается равным `a`, `b` или `c` в зависимости от входных сигналов `select_a`, `select_b` и `select_c`. Если ни один из них не равен V_{CC} , то выход `wire_or` принимает значение по умолчанию, равное GND .

Если более одного сигнала (`select_a`, `select_b` или `select_c`) равны V_{CC} , то `wire_or` равно результату логической операции «ИЛИ» над соответствующими входными сигналами. Например, если `select_a` и `select_b` равны V_{CC} , то `wire_or` равно `a` или `b`.

С сигналом `wire_and` производятся аналогичные действия, но он становится равным V_{CC} , когда входные сигналы `select` равны V_{CC} , и равен ло-

гическому «И» от соответствующих входных сигналов, если более чем один из них равен V_{CC} .

3.3.7. Реализация логики с активным низким уровнем

Значение сигнала с низким активным уровнем равно GND. Сигналы с низким активным уровнем могут быть использованы для управления памятью, периферийными устройствами и микропроцессорными чипами.

Ниже приводится файл daisy.tdf, который представляет модуль арбитражной схемы для соединения «гирляндой» (daisy chain). Данный модуль делает запрос на доступ к шине для предыдущего (в «гирлянде») модуля. Он получает запрос на доступ к шине от самого себя и от следующего (в цепочке) модуля. Доступ к шине предоставляется тому модулю, у которого приоритет выше.

```
SUBDESIGN daisy
(
  /local_request : INPUT;
  /local_grant   : OUTPUT;
  /request_in    : INPUT; % from lower priority %
  /request_out   : OUTPUT; % to higher priority %
  /grant_in      : INPUT; % from higher priority %
  /grant_out     : OUTPUT; % to lower priority %
)
BEGIN
  DEFAULTS
  /local_grant = VCC; % active-low output %
  /request_out = VCC; % signals should default %
  /grant_out   = VCC; % to VCC %
  END DEFAULTS;
  IF /request_in == GND # /local_request == GND THEN
    /request_out = GND;
  END IF;
  IF /grant_in == GND THEN
    IF /local_request == GND THEN
      /local_grant = GND;
    ELSIF /request_in == GND THEN
      /grant_out = GND;
```

```

        END IF;
    END IF;
END;
```

Все сигналы в данном файле имеют активный низкий уровень. Фирма «Altera» рекомендует помечать как-либо имена сигналов с низким активным уровнем, например первым символом в имени ставить символ “/”, который не является оператором, и использовать его постоянно.

В операторе IF проверяется, является ли модуль активным, т.е. равен ли он GND. Если модуль оказывается активным, реализуются действия, записанные в операторе IF. В объявлении по умолчанию DEFAULTS считается, что сигналу присваивается значение V_{CC} , если он неактивен.

3.3.8. Реализация двунаправленных выводов

Система MAX+PLUS II позволяет конфигурировать порты ввода/вывода в устройствах «Altera» как двунаправленные порты. Двунаправленный вывод задается как порт BIDIR, который подсоединяется к выходу примитива TRI. Сигнал между этим выводом и буфером с тремя состояниями является двунаправленным, и его можно использовать в проекте для запуска других логических схем.

Приводимый ниже файл bus_reg2.tdf реализует регистр, который делает выборку значения, найденного на шине с тремя состояниями, а также может передать обратно на шину хранимое значение.

```

SUBDESIGN bus_reg2
(
    clk : INPUT;
    oe  : INPUT;
    io  : BIDIR;
)
BEGIN
    io = TRI(DFF(io, clk,, ), oe);
END;
```

Двунаправленный сигнал io, запускаемый примитивом TRI, используется в качестве входа d для D-триггера (DFF). Запятые в конце списка параметров отделяют места для сигналов триггера clrn и prn. Эти сигналы по умолчанию установлены в неактивное состояние.

Двунаправленный вывод можно также использовать для подсоединения TDF-файла более низкого уровня к выводу с высоким уровнем. Прототип функции для TDF-файла более низкого уровня должен содержать двунаправленный вывод в предложении RETURNS. В приведенном ниже файле `bidir1.tdf` даны четыре примера использования макрофункции `bus_reg2`.

```
FUNCTION bus_reg2 (clk, oe) RETURNS (io);
  SUBDESIGN bidir1
    (
      clk, oe : INPUT;
      io[3..0] : BIDIR;
    )
  BEGIN
    io0 = bus_reg2(clk, oe);
    io1 = bus_reg2(clk, oe);
    io2 = bus_reg2(clk, oe);
    io3 = bus_reg2(clk, oe);
  END;
```

3.4. Последовательностная логика

Логическая схема называется последовательностной, если выходы в заданный момент времени являются функцией входов не только в тот же момент, но и во все предыдущие моменты времени. Таким образом, в последовательностную схему должны входить некоторые элементы памяти (триггеры). В языке AHDL последовательностная логика реализована цифровыми автоматами с памятью (state machines), регистрами и триггерами. При этом средства описания цифровых автоматов занимают особое место. Кроме того, к последовательностным логическим схемам относятся различные счетчики и контроллеры.

3.4.1. Объявление регистров

Регистры используются для хранения значений данных и промежуточных результатов счетчика, тактирование осуществляется синхросигналом. Регистр создается его объявлением в секции VARIABLE.

Для подсоединения примера примитива, макрофункции или цифрового автомата к другой логике в TDF-файл можно использовать порты. Порт примера описывается в следующем формате: <имя примера>.<имя порта>.

Имя порта — это вход или выход примитива, макрофункции или цифрового автомата, что является синонимом имени вывода в файлах проектов GDF-файл, WDF и других.

Ниже приводится файл `bur_reg.tdf`, содержащий байтовый регистр, который фиксирует значения на входах `d` по выходам `q` на фронте синхросигнала, когда уровень загрузки высокий.

```
SUBDESIGN bur_reg
(
    clk, load, d[7..0] : INPUT;
    q[7..0] : OUTPUT;
)
VARIABLE
    ff[7..0] : DFFE;
BEGIN
    ff[].clk = clk;
    ff[].ena = load;
    ff[].d = d[];
    q[] = ff[].q;
END;
```

Как видно из файла, регистр объявлен в секции `VARIABLE` как D-триггер с разрешением (DFFE). В первом булевом уравнении в логической секции происходит соединение входа тактового сигнала подпроекта к портам тактового сигнала триггеров `ff [7..0]`. Во втором уравнении отпирающий тактовый сигнал соединяется с загрузкой. В третьем уравнении входы данных подпроекта соединяются с портами данных триггеров `ff [7..0]`. В четвертом уравнении выходы подпроекта соединяются с выходами триггеров. Все четыре уравнения оцениваются одновременно.

Можно также в секции `VARIABLE` объявить T-, JK- и SR-триггеры и затем использовать их в логической секции. При использовании T-триггеров придется в третьем уравнении изменить порт `d` на `t`. При использовании JK- и SR-триггеров вместо третьего уравнения придется записать два уравнения, в которых происходит соединение портов `j` и `k` или `s` и `r` с сигналами.

Примечание. При загрузке регистра по заданному фронту глобального тактового сигнала фирма «Altera» рекомендует использовать вход разре-

шения одного из регистров: DFFE, TFFE, JKFFE или SRFFE, чтобы контролировать загрузку регистра.

3.4.2. Объявление регистровых выходов

Можно объявить регистровые выходы, если объявить выходы подпроцекта как D-триггеры в секции VARIABLE. Приведенный ниже файл `reg_out.tdf` обеспечивает те же функции, что и предыдущий файл `bur_reg.tdf`, но имеет регистровые выходы.

```
SUBDESIGN reg_out
(
    clk, load, d[7..0] : INPUT;
    q[7..0] : OUTPUT;
)
VARIABLE
    q[7..0] : DFFE;
BEGIN
    q[].clk = clk;
    q[].ena = load;
    q[] = d[];
END;
```

При присвоении значения регистровому выходу в логической секции это значение формирует входные d-сигналы регистров. Выход регистра не изменяется до тех пор, пока не придет фронт синхросигнала. Для определения тактирующего сигнала регистра нужно использовать описание в следующем формате: <имя выходного вывода>.clk для входа тактирующего сигнала регистра в логической секции. Глобальный синхросигнал можно реализовать примитивом GLOBAL или выбором в диалоговом окне компилятора logic synthesis (логический синтез) опции Automatic Global Clock.

Каждый отпирающий D-триггер, объявленный в секции VARIABLE, возбуждает выход с таким же именем, поэтому можно обращаться к q-выходам объявленных триггеров без использования q-порта этих триггеров.

3.4.3. Создание счетчиков

Счетчиками называются последовательностные логические схемы для счета тактовых импульсов. В некоторых счетчиках реализован счет на сложение и

вычитание (реверсивные счетчики), в некоторые счетчики можно загружать данные, а также обнулять их. Счетчики обычно определяют как D-триггеры (DFF и DFFE) и используют операторы IF. Ниже приведен файл `ahdlcnt.tdf`, который реализует 16-разрядный загружаемый счетчик со сбросом.

```
SUBDESIGN ahdlcnt
(
    clk, load, ena, clr, d[15..0] : INPUT;
    q[15..0] : OUTPUT;
)
VARIABLE
    count[15..0] : DFF;
BEGIN
    count[].clk = clk;
    count[].clrn = !clr;
    IF load THEN
        count[].d = d[];
    ELSIF ena THEN
        count[].d = count[].q + 1;
    ELSE
        count[].d = count[].q;
    END IF;
    q[] = count[];
END;
```

В данном файле в секции `VARIABLE` объявлены 16 D-триггеров, и им присвоены имена от `count0` до `count15`. В операторе `IF` определяется значение, загружаемое в триггеры по фронту синхросигнала (например, если загрузка запускается V_{CC} , то триггерам присваивается значение `d[]`).

3.5. Цифровые автоматы с памятью (state mashine)

Цифровые автоматы так же, как таблицы истинности и булевы уравнения, легко реализуются в языке AHDL. Язык структурирован, поэтому пользователь может либо сам назначить биты и значения состояний, либо предоставить эту работу компилятору системы MAX+PLUS II.

Компилятор, по уверениям производителя, «использует патентованные перспективные эвристические алгоритмы», позволяющие сделать такие ав-

томатические назначения состояний, которые минимизируют логические ресурсы, нужные для реализации цифрового автомата.

Пользователю просто нужно нарисовать диаграмму состояний и построить таблицу состояний. Затем компилятор выполняет автоматически следующие функции:

- назначает биты, выбирая для каждого бита либо Т-триггер, либо D-триггер;
- присваивает значения состояний;
- применяет сложные методы логического синтеза для получения уравнений возбуждения.

По желанию пользователя, можно задать в TDF-файле переходы в машине состояний с помощью объявления таблицы истинности. В языке AHDL для задания цифрового автомата нужно включить в TDF-файл следующие элементы:

- объявление цифрового автомата (в секции VARIABLE);
- булевы уравнения управления (в логической секции);
- переходы между состояниями (в логической секции).

Цифровые автоматы в языке AHDL можно также экспортировать и импортировать, совершая обмен между файлами типа TDF и GDF или WDF, при этом входной или выходной сигнал задается как порт цифрового автомата в секции SUBDESIGN.

3.5.1. Реализация цифровых автоматов (state machine)

Цифровой автомат задают в секции VARIABLE путем объявления имени цифрового автомата, его состояний и, возможно, выходных битов. Ниже приведен файл simple.tdf, который реализует функцию D-триггера.

```
SUBDESIGN simple
(
    clk  : INPUT;
    reset : INPUT;
    d     : INPUT;
    q     : OUTPUT;
)
VARIABLE
    ss : MACHINE WITH STATES (s0, s1);
BEGIN
    ss.clk = clk;
```

```

ss.reset = reset;
CASE ss IS
    WHEN s0 =>
        q = GND;
        IF d THEN
            ss = s1;
        END IF;
    WHEN s1 =>
        q = VCC;
        IF !d THEN
            ss = s0;
        END IF;
END CASE;
END;
```

В данном файле в секции VARIABLE объявлен цифровой автомат (state machine) ss. Состояния автомата определяются как s0 и s1. Биты состояний не определены.

3.5.2. Установка сигналов Clock, Reset и Enable

Сигналы Clock, Reset и Enable управляют триггерами регистра состояний в цифровом автомате. Эти сигналы задаются булевыми уравнениями управления в логической секции.

В предыдущем примере (файл simple.tdf) синхросигнал цифрового автомата (Clock) формируется входом clk. Асинхронный сигнал сброса цифрового автомата (Reset) формируется входом reset, имеющим высокий активный уровень. Для подключения сигнала отпираания (Enable) нужно добавить в данный файл проекта строку «ena : INPUT;» в секцию SUBDESIGN, а также добавить в логическую секцию булево уравнение «ss.ena = ena;».

3.5.3. Задание выходных значений для состояний

Для задания выходных значений можно использовать операторы IF и CASE. В приведенном выше примере (файл simple.tdf) значение выхода q устанавливается равным GND, если цифровой автомат ss находится в состоянии s0, и равным V_{CC}, когда он находится в состоянии s1. Эти присваивания делаются в предложениях WHEN оператора CASE. Выходные зна-

чения можно также задавать в таблицах истинности, как будет описано в разделе 3.5.5.

3.5.4. Задание переходов между состояниями

Переходы между состояниями определяют условия, при которых машина переходит в новое состояние. Переходы в машине состояний задаются путем условного присвоения состояния в рамках одной конструкции, описывающей режим. Для этой цели рекомендуется использовать оператор CASE или таблицу истинности. В приведенном выше примере (файл simple.tdf) переходы для каждого состояния определены в предложениях WHEN оператора CASE.

3.5.5. Присвоение битов и значений в цифровом автомате

Бит состояния — это выход триггера, используемый для хранения одного бита значений цифрового автомата. В большинстве случаев для минимизации логических ресурсов следует предоставить компилятору системы MAX+PLUS II присвоение битов и значений состояния. Однако пользователь может сделать это самостоятельно в объявлении цифрового автомата, если, например, он хочет, чтобы определенные биты были выходами цифрового автомата. Ниже приведен файл stepper.tdf, реализующий функцию контроллера шагового двигателя.

```
SUBDESIGN stepper
(
    clk, reset : INPUT;
    ccw, cw    : INPUT;
    phase[3..0] : OUTPUT;
)
VARIABLE
ss      : MACHINE OF BITS (phase[3..0])
        WITH STATES (
s0 = B"0001", s1 = B"0010", s2 = B"0100", s3 = B"1000");
BEGIN
    ss.clk = clk;
    ss.reset = reset;
    TABLE
```

```

        ss,    ccw,          cw    =>          ss;
        s0,    1,           x      =>          s3;
        s0,    x,           1      =>          s1;
        s1,    1,           x      =>          s0;
        s1,    x,           1      =>          s2;
        s2,    1,           x      =>          s1;
        s2,    x,           1      =>          s3;
        s3,    1,           x      =>          s2;
        s3,    x,           1      =>          s0;
    END TABLE;
END;
```

В данном примере выходы `phase[3..0]`, объявленные в секции `SUBDESIGN`, объявляются так же, как биты цифрового автомата `ss` в объявлении цифрового автомата.

3.5.6. Цифровые автоматы с синхронными выходами

Если выходы цифрового автомата зависят только от его состояния, их можно задать в предложении `WITH STATES` объявления цифрового автомата. Это сделает их менее подверженными ошибкам. Кроме того, в некоторых случаях для логических операций потребуется меньше логических ячеек. Ниже приведен пример (файл `moorel.tdf`), в котором реализован автомат Мура с четырьмя состояниями.

```

SUBDESIGN moorel
(
    clk : INPUT;
    reset : INPUT;
    y : INPUT;
    z : OUTPUT;
)
VARIABLE          % current current %
                  % state output %
    ss :MACHINE OF BITS (z)
        WITH STATES (s0 = 0,
                      s1 = 1,
                      s2 = 1,
                      s3 = 0);
```

```

BEGIN

    ss.clk = clk;
    ss.reset = reset;
    TABLE
    % current current next %
    % state input state %
        ss,    Y      =>    ss;
        s0,    0      =>    s0;
        s0,    1      =>    s2;
        s1,    0      =>    s0;
        s1,    1      =>    s2;
        s2,    0      =>    s2;
        s2,    1      =>    s3;
        s3,    0      =>    s3;
        s3,    1      =>    s1;
    END TABLE;

END;
```

В данном примере состояния определены в объявлении цифрового автомата. Переходы между состояниями определены в таблице next_state, которая задана в объявлении таблицы истинности. В данном примере машина имеет четыре состояния и только один бит состояния Z. Компилятор системы MAX+PLUS II автоматически добавляет еще один бит и делает соответствующие присвоения этой синтезированной переменной для того, чтобы получилась машина с четырьмя состояниями. Такой цифровой автомат (state machine) требует, по крайней мере, двух битов.

Если значения состояний используются как выходы (как в файле moorel.tdf), для проекта потребуется меньше логических ячеек, но, возможно, логические ячейки потребуют больше логики, чтобы возбудить входы триггера. В этом случае модуль логического синтезатора компилятора, возможно, не сможет полностью минимизировать автомат.

Другой способ построения цифрового автомата заключается в том, чтобы не делать присвоения состояний и явно объявить выходные триггеры. Этот альтернативный метод использован в приведенном ниже файле moore2.tdf.

```

SUBDESIGN moore2
(
```



```

        clk : INPUT;
        reset : INPUT;
        y : INPUT;
        z : OUTPUT;
    )
VARIABLE
ss : MACHINE WITH STATES (s0, s1, s2, s3);
zd : NODE;
BEGIN
    ss.clk = clk;
    ss.reset = reset;
    z = DFF(zd, clk, Vcc, Vcc);
TABLE
%current      current      next  next %
%state        input        state output %
ss,           y            =>    ss,   zd;
s0,           0            =>    s0,   0;
s0,           1            =>    s2,   1;
s1,           0            =>    s0,   0;
s1,           1            =>    s2,   1;
s2,           0            =>    s2,   1;
s2,           1            =>    s3,   0;
s3,           0            =>    s3,   0;
s3,           1            =>    s1,   1;
END TABLE;
END;
```

В данном примере вместо того, чтобы задать выходы присвоением значений состояниям в объявлении цифрового автомата, в объявление таблицы истинности добавлен один столбец под названием "next output" (следующий выход). В этом методе для синхронизации выходов синхросигналом используется D-триггер, вызов которого записан с помощью непосредственной ссылки.

3.5.7. Цифровые автоматы с асинхронными выходами

В языке AHDL возможна реализация цифрового автомата с асинхронными выходами. Выходы такого типа автоматов всегда изменяются, когда изменяются входы, независимо от состояния синхросигнала. В приведен-

ном ниже файле mealy.tdf реализован автомат Мили с четырьмя состояниями и асинхронными выходами.

```

SUBDESIGN mealy
(
    clk : INPUT;
    reset : INPUT;
    y : INPUT;
    z : OUTPUT;
)
VARIABLE
ss : MACHINE WITH STATES (s0, s1, s2, s3); BEGIN
    ss.clk = clk;
    ss.reset = reset;
TABLE
%      current current  current      next %
%      state  input   outputstate %
ss,      y    =>    z,      ss;
s0,      0    =>    0,      s0;
s0,      1    =>    1,      s1;
s1,      0    =>    1,      s1;
s1,      1    =>    0,      s2;
s2,      0    =>    0,      s2;
s2,      1    =>    1,      s3;
s3,      0    =>    0,      s3;
s3,      1    =>    1,      s0;
END TABLE;
END;
```

3.5.8. Восстановление после неправильных состояний

Логика, сгенерированная для цифрового автомата компилятором системы MAX+PLUS II, будет работать так, как определено в TDF-файле. Однако проекты с использованием цифровых автоматов часто допускают значения битов состояний, которые не присваиваются правильным состояниям. Эти значения с не присвоенными битами состояний называются неправильными состояниями. Проект, который переходит в неправильное состояние, например, в результате нарушений временных требований к установке или задержке,

может реализовать ошибочные выходы. Несмотря на рекомендации фирмы «Altera» по соблюдению временных требований к установке и задержке, пользователь может сделать восстановление цифрового автомата после неправильного состояния путем принудительного преобразования неправильного состояния к известному правильному в рамках оператора CASE.

Для восстановления из неправильных состояний следует объявить их поименно для данного автомата. Предложение WHEN OTHERS в операторе CASE, которое принудительно преобразует состояния, применяется только к состояниям, которые были объявлены, а не упомянуты в предложении WHEN. Данный метод работает, только если все неправильные состояния определены в объявлении цифрового автомата.

Для n -битового цифрового автомата существуют 2^n возможных состояний. Поэтому нужно добавить воображаемые имена состояний, чтобы получилось такое их число. Ниже приведен файл recover.tdf, в котором реализован цифровой автомат, который может восстанавливаться из неправильных состояний.

```
SUBDESIGN recover
(
    clk : INPUT;
    go  : INPUT;
    ok  : OUTPUT;
)
VARIABLE
sequence : MACHINE
OF BITS (q[2..0]) WITH STATES (idle, one, two,
three, four, illegal1, illegal2, illegal3);
BEGIN
    sequence.clk = clk;
    CASE sequence IS
        WHEN idle =>
            IF go THEN
                sequence = one;
            END IF;
        WHEN one =>
            sequence = two;
        WHEN two =>
```

```
sequence = three;
WHEN three =>
sequence = four;
WHEN OTHERS =>
sequence = idle;
END CASE;
ok = (sequence == four);
END;
```

В данном примере цифровой автомат имеет три бита, поэтому он должен иметь 2^3 или 8 состояний. В объявлении заданы только 5 состояний. Следовательно, нужно туда добавить еще три воображаемых состояния illegal1, illegal2, illegal3.

3.6. Реализация иерархического проекта

В иерархической структуре проекта TDF-файлы, написанные на языке AHDL, можно использовать вместе с другими файлами проектов. На нижнем уровне проекта могут быть макрофункции, поставляемые фирмой «Altera» или разработанные пользователями.

3.6.1. Использование макрофункций системы MAX+PLUS II фирмы «Altera»

В системе MAX+PLUS II есть большая библиотека, в которую входят 74 стандартные макрофункции, реализующие шины с последовательным опросом, оптимизацию архитектуры и конкретные приложения. Библиотека представляет собой собрание блоков высокого уровня, используемых для создания проекта с иерархической логикой. Во время инсталляции системы эти макрофункции автоматически записываются в каталог \maxplus2\max2lib и его подкаталоги, создаваемые в процессе инсталляции.

В языке AHDL существуют два способа вызова (т. е. вставки в качестве примера) макрофункции:

— объявить переменную типа <macrofunction> в объявлении примеров INSTANCE в секции VARIABLE и использовать порты примера макрофункции в логической секции. В этом способе важное значение имеют имена портов;

— использовать для макрофункции непосредственную ссылку в логической секции файла TDF. В этом способе важен порядок портов.

Входы и выходы макрофункций перечисляются в описании прототипов функций (FUNCTION PROTOTYPE). Прототипы функций можно записать в отдельный файл и указать его в своем файле с помощью директивы INCLUDE. Такие Include-файлы создаются автоматически для данного проекта с помощью команды Create Default Include File. Include-файл вставляется вместо вызывающей его директивы INCLUDE. Для всех макрофункций системы MAX+PLUS II Include-файлы должны находиться в каталоге \maxplus2\max2inc.

Ниже приведен файл macro1.tdf, который реализует четырехразрядный счетчик, подсоединенный своими выходами к дешифратору 4 в 16. Соответствующие макрофункции вызываются объявлениями примеров в секции VARIABLE.

```
INCLUDE "4count";
INCLUDE "16dmux";

SUBDESIGN macro1
(
    clk          : INPUT;
    out[15..0]   : OUTPUT;
)
VARIABLE
    counter : 4count;
    decoder : 16dmux;
BEGIN
    counter.clk = clk;
    counter.dnup = GND;
    decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
    out[15..0] = decoder.q[15..0];
END;
```

В данном файле используются директивы INCLUDE для импортирования прототипов функций для двух макрофункций фирмы «Altera»: 4count и 16dmux. В секции VARIABLE объявляются две переменные counter и decoder как примеры этих макрофункций. В логической секции определяются входные порты для обеих макрофункций в формате <имя переменной-примера>.<имя порта> (они ставятся в левой части булевых уравне-

ний, а выходные порты — справа.) Порядок портов в прототипе функции не важен, так как имена портов в логической секции перечисляются явно.

Ниже приведен файл macro2.tdf, выполняющий те же функции, что и предыдущий, но макрофункции в нем вызываются непосредственной ссылкой.

```
INCLUDE "4count";
INCLUDE "16dmux";
SUBDESIGN macro2
(
    clk          : INPUT;
    out[15..0]   : OUTPUT;
)
VARIABLE
    q[3..0]      : NODE;
BEGIN
    (q[3..0], ) = 4count (clk, , , , GND, , , , );
    out[15..0] = 16dmux (.(d, c, b, a)=q[3..0]);
    % equivalent in-line ref. with positional port association %
    % out[15..0] = 16dmux (q[3..0]);                                %
END;
```

Вызов макрофункций 4count и 16dmux осуществляется в логической секции непосредственной ссылкой (в правой части булевых уравнений).

Ниже приведены прототипы этих макрофункций, записанные в файлах 4count.inc, 16dmux.inc:

```
FUNCTION 4count (clk, clrn, setn, ldn, cin, dnup, d, c, b, a)
    RETURNS (qd, qc, qb, qa, cout);
FUNCTION 16dmux (d, c, b, a)
    RETURNS (q[15..0]);
```

Соединение портов показано в логической секции файла macro2.tdf. Порядок портов важен, так как должно быть однозначное соответствие между портами, описанными в прототипе функции и при ее реализации в логической секции. В данном примере запятыми отделяются (но не перечисляются) порты, для которых не делается явное подключение.

3.6.2. Создание и применение пользовательских макрофункций

В файлах, написанных на AHDL, можно легко создавать и использовать пользовательские макрофункции, выполняя следующие действия:

- Создать логику для макрофункции в файле проекта.
- Определить порты макрофункции в объявлении прототипа функции.

Прототип функции дает краткое описание функции: ее имя, а также входные, выходные и двунаправленные порты. Можно также использовать машинные порты для макрофункций, которые импортируют или экспортируют цифровой автомат.

Объявление прототипов функций может быть размещено в Include-файле, который вызывается в пользовательском файле. Используя команду Create Default Include File, можно автоматически создавать Include-файл с прототипом функции для любого файла проекта:

- вставить в файл пример макрофункции с помощью объявления примера в секции VARIABLE или с помощью непосредственной ссылки в тексте;
- использовать макрофункцию в файле.

3.6.3. Определение пользовательской макрофункции

Для использования макрофункции ее нужно либо включить в описание прототипа функции в TDF-файле, либо указать в директиве INCLUDE файла TDF имя Include-файла, содержащего прототип этой макрофункции. Как уже упоминалось выше, Include-файлы можно создавать автоматически.

Ниже приводится файл keyboard.tdf, в котором реализован кодировщик для 16-клавишной клавиатуры.

```
TITLE "Keyboard Encoder";
  INCLUDE "74151";
  INCLUDE "74154";
  INCLUDE "4count";
  FUNCTION debounce (clk, key_pressed)
    RETURNS (pulse);
  SUBDESIGN keyboard
  (
    clk : INPUT; % 50-KHz clock %
    col[3..0] : INPUT; % keyboard columns %
    row[3..0], d[3..0] : OUTPUT; % keyboard rows, key code %
```

```
strobe : OUTPUT; % key code is valid %
    )
    VARIABLE
key_pressed : NODE; % VCC when key d[3..0] is pressed %
mux : 74151;
decoder : 74154;
counter : 4count;
opencol[3..0] : TRI;
    BEGIN
% Drive keyboard rows with a decoder and
open-collector outputs %
        row[] = opencol[].out;
        opencol[].in = GND;
opencol[].oe = decoder.(o0n,o1n,o2n,o3n);
decoder.(b,a) = counter.(qd,qc);
% Sense keyboard columns with a multiplexer %
        mux.d[3..0] = col[3..0];
        mux.(b,a) = counter.(qb,qa);
        key_pressed = !mux.y;
% Scan keyboard until a key is pressed. %
% Drive key's code onto d[] outputs %
        counter.clk = clk;
        counter.cin = !key_pressed;
        d[] = counter.(qd,qc,qb,qa);
% Generate strobe when key has settled %
        strobe = debounce(clk, key_pressed);
    END;
```

В данном примере в директивах INCLUDE указываются файлы прототипов функции для стандартных макрофункций фирмы «Altera»: 4count, 74151 и 74154. Отдельно в файле дан прототип функции для макрофункции debounce, в котором описаны входы clk и key_pressed и выход pulse. Примеры макрофункций 4count, 74151 и 74154 вызываются объявлением примеров в секции VARIABLE. Пример макрофункции debounce вызывается непосредственной ссылкой в тексте логической секции.

3.6.4. Импорт и экспорт цифровых автоматов (state machine)

Операции импорта и экспорта цифровых автоматов осуществляются между файлами TDF и другими файлами проекта путем задания входного или выходного порта как входа автомата (MACHINE INPUT) и его выхода (MACHINE OUTPUT) в секции SUBDESIGN. Прототип функции, который представляет собой файл, содержащий машину состояний, должен указывать, какие входы и выходы принадлежат машине состояний. Это осуществляется снабжением имен сигналов префиксом — ключевым словом MACHINE.

Замечание. Порты типа MACHINE INPUT и MACHINE OUTPUT нельзя использовать в файле проекта верхнего уровня.

Можно переименовать автомат, дав ему временное имя (псевдоним) в объявлении MACHINE в секции VARIABLE. Псевдоним цифрового автомата можно использовать в том файле, где создается цифровой автомат, или в том файле, где используется порт MACHINE INPUT для импортирования цифрового автомата. Это имя можно потом использовать вместо первоначального имени цифрового автомата. Ниже приводится файл ss_def.tdf, который определяет и экспортирует цифровой автомат ss с портом ss_out типа MACHINE OUTPUT.

```
SUBDESIGN ss_def
(
    clk, reset, count : INPUT;
    ss_out : MACHINE OUTPUT;
)
VARIABLE
ss : MACHINE WITH STATES (s1, s2, s3, s4, s5);
BEGIN
    ss_out = ss;
    CASE ss IS
    WHEN s1=>
    IF count THEN ss = s2; ELSE ss = s1; END IF;
    WHEN s2=>
    IF count THEN ss = s3; ELSE ss = s2; END IF;
    WHEN s3=>
    IF count THEN ss = s4; ELSE ss = s3; END IF;
    WHEN s4=>
```

```

IF count THEN ss = s5; ELSE ss = s4; END IF;
      WHEN s5=>
IF count THEN ss = s1; ELSE ss = s5; END IF;
      END CASE;
      ss.(clk, reset) = (clk, reset);

END;
```

Ниже приводится файл `ss_use.tdf`, который импортирует цифровой автомат с портом `ss_in` типа `MACHINE INPUT`.

```

SUBDESIGN ss_use
(
    ss_in : MACHINE INPUT;
    out : OUTPUT;
)
BEGIN
    out = (ss_in == s2) OR (ss_in == s4);
END;
```

Ниже приведен файл `top1.tdf`, в котором используются непосредственные ссылки в тексте для вставки примеров функций `ss_def` и `ss_use`. В прототипах этих функций содержится ключевое слово `MACHINE` для указания, какие входы и выходы являются цифровыми автоматами с памятью.

```

FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);
DESIGN IS «top1» DEVICE IS «AUTO»;
SUBDESIGN top1
(
    sys_clk, /reset, hold : INPUT;
    sync_out : OUTPUT;
)
VARIABLE
    ss_ref: MACHINE; % Machine Alias Declaration %
BEGIN
    ss_ref = ss_def(sys_clk, !/reset, !hold);
```

```

    sync_out = ss_use(ss_ref);
END;
```

Внешний цифровой автомат можно также реализовать в TDF-файле верхнего уровня с объявлением примера в секции VARIABLE. Ниже приведен файл top2.tdf, который имеет ту же функцию, что и файл top1.tdf, но для вызова макрофункций использует объявление примеров.

```

FUNCTION ss_def (clk, reset, count) RETURNS (MACHINE ss_out);
FUNCTION ss_use (MACHINE ss_in) RETURNS (out);
DESIGN IS "top2" DEVICE IS "AUTO";
SUBDESIGN top2
    (
        sys_clk, /reset, hold : INPUT;
sync_out : OUTPUT;
    )
    VARIABLE
sm_macro : ss_def;
sync : ss_use;
    BEGIN
sm_macro.(clk, reset, count) = (sys_clk, !/reset, !hold);
sync.ss_in = sm_macro.ss_out;
sync_out = sync.out;
    END;
```

3.7. Управление синтезом

3.7.1. Реализация примитивов LCELL и SOFT

Можно ограничить логический синтез с помощью замены переменных типа узел (NODE) примитивами SOFT и LCELL. Переменные NODE и примитивы LCELL обеспечивают наилучшее управление логическим синтезом. Примитивы SOFT обеспечивают более слабое управление логическим синтезом.

Переменные NODE, которые объявляются в секции VARIABLE, накладывают слабые ограничения на логический синтез. Во время синтеза модуль логического синтеза компилятора системы MAX+PLUS II заменяет каждый пример использования переменной NODE логикой, которую

она представляет. Затем происходит минимизация логики до одной логической ячейки. Этот метод обычно приводит к ускорению работы схемы, но в результате может получиться слишком сложная логика или же ее трудно свести к одной ячейке.

Буферы SOFT обеспечивают лучшее управление использованием ресурсов, чем переменные NODE. Модуль логического синтезатора выбирает, когда заменить примеры использования примитивов SOFT примитивами LCELL.

Буферы SOFT могут помочь уничтожить логику, которая оказалась слишком сложной, и сделать проект проще. Однако при этом может быть увеличено число логических операций и скорость выполнения программы соответственно уменьшится.

Наиболее сильное управление процессом логического синтеза обеспечивается примитивами LCELL. Модуль логического синтезатора минимизирует всю логику, которая запускает примитив LCELL, таким образом, чтобы можно было свести ее к одной логической ячейке. Примитивы LCELL реализуются в виде одной логической ячейки (их нельзя убрать из проекта, даже если они имеют единственный вход). Если проект минимизирован до такой степени, что один примитив LCELL имеет единственный вход, то в этом случае вместо примитивов LCELL можно использовать примитивы SOFT, которые убираются в процессе логического синтеза.

Примечание. При многоуровневом синтезе компилятор системы MAX+PLUS II автоматически помещает буферы SOFT в более выгодное место проекта, если включить опцию SOFT Buffer Insertion logic.

Ниже приводятся две версии файла TDF — с переменными NODE и с примитивами SOFT. В версии nodevar переменная odd_parity объявлена как NODE, затем ей присваивается булево выражение d0 \$ d1 \$... \$ d8. В версии softbuf компилятор заменит некоторые примитивы SOFT примитивами LCELL во время обработки данных для лучшего использования ресурсов устройства.

```
TDF with NODE
Primitives:
SUBDESIGN nodevar
(
:
)
```

```
TDF with SOFT Variables:
SUBDESIGN softbuf
(
:
)
```

```
VARIABLE
odd_parity : NODE;
BEGIN
odd_parity = d0 $ d1 $
d2
$ d3 $ d4 $ d5
$ d6 $ d7 $ d8;
END;
```

```
VARIABLE
odd_parity : NODE;
BEGIN
odd_parity = SOFT(d0 $
d1 $ d2)
$ SOFT(d3 $ d4 $ d5)
$ SOFT(d6 $ d7 $ d8);
END;
```

3.7.2. Значения констант по умолчанию

Логический синтезатор автоматически выполняет подключение к GND всех выходов таблицы истинности, если не удовлетворяется ни одно из условий входа таблицы. Для присвоения выходам таблицы истинности значения V_{CC} можно использовать одно или несколько объявлений языка AHDL по умолчанию. С помощью этих объявлений можно задать значения по умолчанию для соответствующих выходов. Например, если большинство выходов таблицы истинности равно "1", можно задать значение по умолчанию V_{CC} .

Примечание. Не следует путать значения по умолчанию для переменных и портов, которые присваиваются в секции SUBDESIGN.

3.7.3. Присвоение битов и значений в цифровом автомате

Логический синтезатор автоматически минимизирует число битов состояния, требуемое для цифрового автомата. При этом оптимизируются как использование устройства, так и характеристики его работы. Однако некоторые устройства цифрового автомата могут работать быстрее при значениях состояния, использующих число битов больше минимального. Для контроля этих случаев пользователь сам объявляет биты и значения для цифрового автомата.

3.8. Элементы языка AHDL

3.8.1. Зарезервированные ключевые слова

Зарезервированные ключевые слова используются для следующих целей:

— для обозначения начала, конца и переходов в объявлениях языка AHDL;

— для обозначения предопределенных констант, т.е. GND и V_{CC}.

Ключевые слова можно использовать как символические имена, только если они заключены в символы одинарных кавычек ('). Их можно также использовать в комментариях.

Для того чтобы получить контекстовую помощь по ключевому слову, убедитесь, что ваш файл сохранен с расширением .tdf, затем нажмите одновременно две кнопки Shift+F1 в окне текстового редактора Text Editor и щелкните кнопкой мыши Button 1 на ключевом слове.

Фирма «Altera» рекомендует все ключевые слова набирать прописными буквами. Список всех зарезервированных ключевых слов языка AHDL:

FUNCTION	OTHERS	JKFFE
CASE	TABLE	NCLUDE
BITS	SRFFE	NODE
DFF	VCC	NOR
DFFE	WHEN	NOT
ELSE	WITH	OPTIONS
END	XNOR	OR
EXP	XOR	OUTPUT
AND	GLOBAL	RETURNS
BEGIN	GND	SOFT
BURIED	INPUT	SRFF
BIDIR	IF	STATES
CARRY	IS	SUBDESIGN
CASCADE	JKFF	TFF
CLIQUE	LATCH	TFFE
CONNECTED_PINS	LCELL	THEN
CONSTANT	MACHINE	TITLE
DEFAULTS	MACRO	TRI
DESIGN	MCELL	VARIABLE
DEVICE	NAND	X
ELSIF	OF	

3.8.2. Символы

В Табл. 3.1 приведены символы, имеющие определенное значение в языке AHDL. В этот перечень не включены символы, используемые в булевых выражениях как операторы и для операций сравнения.

Таблица 3.1. Символы, используемые в языке AHDL

Символ	Функция
<u> </u> (подчеркивание)	Используемые пользователем идентификаторы
<u> </u> (тире)	Символы в символических именах
<u> </u> <u> </u> (два тире)	Начинает комментарий в стиле VHDL, который продолжается до конца строки
% (процент)	Заключает с двух сторон комментарий стиля AHDL
() (круглые скобки)	Закljučают и определяют последовательные имена групп. Закljučают имена выводов в секции подпроекта (Subdesign Section) и в прототипах функций. Закljučают (необязательно) входы и выходы таблиц в объявлении Truth Table. Закljučают состояния в объявлении цифрового автомата State Machine.
[] (квадратные скобки)	Закljučают более приоритетные операции в булевых выражениях. Закljučают необязательные варианты в секции проекта Design Section (внутри объявления назначения ресурсов Assignment)
'...' (одинарные кавычки)	Закljučают диапазон значений в десятичном имени группы
'...' (одинарные кавычки)	Закljučают символические имена
"..." (двойные кавычки)	Закljučают строку в объявлении названия Title. Закljučают цифры в недесятичных номерах. Закljučают путь в объявлении Include. Могут (необязательно) заключать имя проекта и устройства в секции проекта Design Section. Могут (необязательно) заключать имя в объявлении назначения клики Clique Assignment
. (точка)	Отделяет символические имена переменных в макрофункции или примитиве от имен портов. Отделяет имя файла от расширения
... (многоточие)	Разделяет наименьшее и наибольшее значение в диапазонах
; (точка с запятой)	Заканчивает объявления и секции в языке AHDL
, (запятая)	Разделяет элементы последовательных групп и списков

Таблица 3.1 (окончание)

Символ	Функция
: (двоеточие)	Отделяет символические имена от типов в объявлениях и назначениях ресурсов
@ «собака»	Присваивает символические узлы выводам устройства и логическим ячейкам в объявлениях назначения ресурсов Resource Assignment
= (равенство)	Присваивает значения по умолчанию GND и V _{CC} входам в секции подпроекта Subdesign. Присваивает установочные значения в вариантах. Присваивает значения состояниям в машине состояний. Присваивает значения в булевых уравнениях
=> (стрелка)	Отделяет входы от выходов в объявлениях таблицы истинности Truth Table. Отделяет предложения с WHEN от булевых выражений в операторе Case

3.8.3. Имена в кавычках и без кавычек

В языке AHDL есть три типа имен:

Символические имена — это определяемые пользователем идентификаторы. Они используются для обозначения следующих частей TDF:

- внутренних и внешних узлов (вершин);
- констант;
- переменных цифрового автомата, битов состояний, имен состояний;
- примеров (Instance).

Имена подпроекта — это определяемые пользователем имена для файлов проекта более низкого уровня. Имя подпроекта должно быть таким же, как имя файла TDF.

Имена портов — это символические имена, идентифицирующие вход или выход примитива или макрофункции.

В файле FIT вашего проекта могут появиться генерируемые компилятором имена выводов с символом «тильда» (~). Этот символ зарезервирован для имен, генерируемых компилятором, пользователю запрещается его использовать для обозначения имен выводов, узлов (вершин), групп (шин).

Существуют две формы записи для всех трех типов имен (символических, подпроекта и портов): в кавычках (') и без кавычек. Если разработчик создает символ по умолчанию для файла TDF, который включает в себя имена портов в кавычках, собственно кавычки не входят в имена выводов. Ниже в **Табл. 3.2** указаны все возможные варианты записи имен в языке AHDL.

Таблица 3.2. Варианты записи имен в языке AHDL

Разрешенные символы	Имя подпроекта		Символическое имя		Имя порта	
	Без кавычек	В кавычках	Без кавычек	В кавычках	Без кавычек	В кавычках
A-Z	+	+	+	+	+	+
A-z	+	+	+	+	+	+
0-9	+	+	+	+	+	+
Подчеркивание (_)	+	+	+	+	+	+
Косая черта (/)	—	—	+	+	+	+
Тире (—)	—	+	—	+	—	+
Только цифры (0-9)	+	+	—	+	+	+
Ключевое слово	—	+	—	+	—	+
Максимальное число символов	8	8	32	32	32	32

Примеры:

Символические имена без кавычек:

a, /a: разрешенные

-foo, node, 55: неразрешенные

Символические имена в кавычках:

'bar', 'table', '1221': разрешенные

'bowling4\$', 'has a space', 'a_name_with_more_than_32_characters': неразрешенные

3.8.4. Группы

Символические имена и порты одного и того же типа можно объявить и использовать как группы в булевых выражениях и уравнениях. Группа, в которую может входить до 256 элементов (или «битов»), обрабатывается как коллекция узлов (вершин) и считается единым целым.

Группа в логической секции файла TDF может состоять из узлов или чисел. Одиночные узлы и константы GND и V_{CC} могут дублироваться для образования групп в булевых выражениях и уравнениях.

В файле FIT могут появиться генерируемые компилятором имена выводов с символом «тильда» (~). Этот символ зарезервирован для имен, генерируемых компилятором, пользователю запрещается его использовать для обозначения имен выводов, узлов (вершин), групп (шин).

3.8.4.1. Форма записи групп

Существуют две формы записи при объявлении групп:

1. Десятичное имя группы состоит из символического имени (или имени порта), за которым следует диапазон десятичных чисел, заключенный в квадратные скобки, например `a[4..1]`. После этого имени группы указывается только один диапазон. Всего символическое имя (или имя порта) вместе с самым длинным (по написанию) номером в диапазоне может содержать до 32 символов.

Если группа определена, квадратные скобки обеспечивают самый краткий путь задания всего диапазона. Вместо диапазона можно также указывать только одно десятичное число, например `a[5]`. Однако такая форма записи означает единственное символическое имя, а не имя группы и эквивалентно имени `a5`.

2. Последовательное имя группы состоит из заключенного в скобки перечня символических имен, имен портов или чисел, разделенных запятыми, например `(a, b, c)`. В этом перечне могут быть также десятичные имена групп, например `(a, b, c[5..1])`. Такая запись используется для задания имен портов. Например:

`reg. (d, clk, clrn, prn).`

В следующем примере показаны варианты записи одной и той же группы:

`b[5..0]`

`(b5, b4, b3, b2, b1, b0)`

`b[]`

3.8.4.2. Диапазон и поддиапазон групп

Диапазоны в десятичных именах групп обозначаются десятичными номерами, обычно перечисляемыми в порядке убывания. Для того чтобы указать диапазон в порядке возрастания или в обоих порядках, разработчик должен установить опцию `BIT0` в объявлении опций `Options`, тогда компилятор не будет выдавать предупреждающие сообщения.

Поддиапазоны включают подмножество узлов, заданных в объявлении группы, их можно указывать различными способами. Например, если разработчик объявил группу `c[5..1]`, он может использовать следующие поддиапазоны этой группы: `c[3..1]`, `c4`, `c[5]`, `(c2,,c4)`.

В левой части булевых уравнений или в ссылках (reference) по тексту программы (in-line) на макрофункции или примитивы в написании имени группы можно использовать запятые вместо перечисления имен.

При указании диапазона вместо чисел можно использовать константы. Например `q[MAX...0]` является разрешенной формой записи, если константа `MAX` определена в объявлении `CONSTANT`.

3.8.5. Числа в языке AHDL

В языке AHDL можно использовать десятичные, двоичные, восьмеричные и шестнадцатеричные числа в любой комбинации. В **Табл. 3.3** приведен синтаксис записи чисел в языке AHDL для каждой системы счисления.

Таблица 3.3. Синтаксис записи чисел в языке AHDL

Система счисления	Значения
Десятичная	<последовательность цифр 0—9>
Двоичная	B"<последовательность из 0, 1, X>", где символ X обозначает безразличное значение
Восьмеричная	O"<последовательность цифр 0—7>" или Q"<последовательность цифр 0—7>"
Шестнадцатеричная	X"<последовательность цифр 0—9, букв A—F>" или H"<последовательность цифр 0—9, букв A—F>"

Примеры разрешенной записи чисел в языке AHDL:

`B"0110X1X10"`

`Q"4671223"`

`H"123AECF"`

К числам в языке AHDL применяются следующие правила:

- Компилятор системы `MAX+PLUS II` всегда интерпретирует числа как группы двоичных цифр.
- В булевых выражениях числа нельзя присваивать одиночным узлам (вершинам). Вместо этого нужно использовать константы `VCC` и `GND`.

3.8.6. Булевы выражения

Булевы выражения состоят из операндов, разделенных логическими и арифметическими операторами и компараторами и (необязательно) сгруппированных с помощью круглых скобок. Выражения используются в булевых уравнениях, а также в других конструкциях языка, таких как операторы `CASE` и `IF`.

Существуют следующие применения булевых выражений:

- Операнд.

Пример: `a, b[5..1], 7, VCC`

- Встроенная в текст (in-line) ссылка (reference) на примитив или макрофункцию.
- Префиксный оператор (! или -), примененный к булеву выражению.
Пример: !c
- Два булевых выражения, разделенные двоичным (не префиксным) оператором.
Пример: d1 \$ d3
- Заключенное в круглые скобки булево выражение.
Пример: (!foo & bar)

Результат каждого булева выражения должен иметь ту же ширину, что и узел или группа (в левой стороне уравнения), которому он, в конечном счете, присваивается.

3.8.7 Логические операторы

В Табл. 3.4 приведены логические операторы для булевых выражений.

Таблица 3.4. Логические операторы для булевых выражений

Оператор	Пример	Описание
!	!tob	Дополнение (префиксное обращение)
NOT	NOT tob	
&	bread & butter	Логическое «И»
AND	bread AND butter	
!&	a[3..1] !& b[5..3]	Обращение логического «И»
NAND	a[3..1] NAND b[5..3]	
#	trick # treat	Логическое «ИЛИ»
OR	trick OR treat	
!#	c[8..5] !# d[7..4]	Обращение логического «ИЛИ»
NOR	c[8..5] NOR d[7..4]	
\$	foo \$ bar	«Исключающее ИЛИ»
XOR	foo XOR bar	
!\$	x2 !\$ x4	Обращение «исключающего логического ИЛИ»
XNOR	x2 XNOR x4	

Каждый оператор представляет собой логический вентиль с двумя входами; исключение составляет оператор NOT, являющийся префиксным инвертором. Для записи логического оператора можно использовать его имя или символ.

Выражения, в которых используются эти операторы, интерпретируются по-разному в зависимости от того, что представляют собой операнды: одиночные узлы (вершины), группы или числа. Кроме того, выражения с оператором NOT интерпретируются не так как другие логические операторы.

3.8.8. Выражения с оператором NOT

С оператором NOT можно использовать три типа операндов:

- Если операнд представляет собой одиночный узел, константы GND или V_{CC} , выполняется одна операция обращения.

Пример: !a.

- Если операнд представляет собой группу узлов, каждый элемент группы проходит через инвертор.

Пример: !a[4..1] интерпретируется как (!a4, !a3, !a2, !a1).

- Если операнд представляет собой число, он обрабатывается как двоичное число, т.е. как группа соответствующего числа битов, где обращается каждый бит.

Пример: !9 операнд интерпретируется как двоичное число !B"1001" (группа из четырех элементов), результат B"0110".

3.8.9. Выражения с операторами AND, NAND, OR, NOR, XOR, & XNOR

Существуют четыре комбинации операндов с двоичными (не префиксными) операторами, и каждая из них интерпретируется по-особому:

- Если операнд представляет собой одиночный узел, константы GND или V_{CC} , оператор выполняет логическую операцию над двумя элементами.

Пример: (a&b).

- Если оба оператора являются группами узлов, оператор применяется к соответствующим узлам каждой группы, производя ряд операций на битовом уровне между группами. Группы должны быть одинакового размера.

Пример: (a,b) # (c,d) интерпретируется как (a#c, b#d).

- Если один оператор представляет собой одиночный узел, константы GND или V_{CC} , а другой операнд – группу узлов, то одиночный узел или константа дублируются для образования группы такого же размера, что и второй операнд. Затем выражение интерпретируется как групповая операция.

Пример: a & b[2..1] интерпретируется как (a&b2, a&b1).

- Если оба операнда представляют собой числа, более короткое (в смысле числа битов в двоичном представлении) число дополняется незна-

чащими нулями, чтобы сравняться по числу битов с другим операндом. Затем выражение интерпретируется как групповая операция.

Пример: в выражении (3#8) 3 и 8 преобразуются в двоичные числа В"0011" и В"1000". Результатом является В"1011".

• Если один операнд представляет собой число, а другой – узел или группу узлов, то число усекается или расширяется до размера группы. При усечении значащих битов генерируется сообщение об ошибке. Выражение затем интерпретируется как групповая операция.

Пример: (a,b,c)&1 интерпретируется как (a&0, b&0, c&1).

Выражение с константой V_{CC} интерпретируется не так как выражение с операндом 1. В первом выражении приведенного ниже примера число 1 расширяется по числу битов двоичного представления, чтобы соответствовать размеру группы. Во втором уравнении узел V_{CC} дублируется для образования группы того же размера. Затем каждое уравнение интерпретируется как групповая операция.

! operation.

$$(a, b, c) \& 1 = (0, 0, c)$$

$$(a, b, c) \& V_{CC} = (a, b, c).$$

3.8.10. Арифметические операторы

Арифметические операторы используются для выполнения арифметических операций сложения и вычитания над группами и числами. В языке AHDL применяются следующие арифметические операторы (Табл. 3.5).

Таблица 3.5. Арифметические операторы, применяемые в языке AHDL

Оператор	Пример	Описание
+ (унарный)	+1	Положительное значение
- (унарный)	-a[4..1]	Отрицательное значение
+	count[7..0] + delta[7..0]	Сложение
-	rightmost_x[] - leftmost_x[]	Вычитание

Унарные плюс (+) и минус (-) являются префиксными операторами. Оператор + не влияет на свой операнд, и разработчик может использовать его для иллюстративных целей (т.е. для указания положительного числа). Оператор интерпретирует свой операнд в двоичном представлении (если он таковым не является с самого начала). Затем он выполняет операцию унарного минуса, т.е. получает дополнение операнда как двоичного числа.

- К другим арифметическим операторам применяются следующие правила:
- Операции выполняются между двумя операндами, которые должны быть группами узлов или числами.
 - Если оба операнда представляют собой группы узлов, то они должны иметь одинаковый размер.
 - Если оба операнда представляют собой числа, то более короткое (в двоичном представлении) число расширяется (дополняется незначащими нулями), чтобы сравняться по числу битов с другим операндом.
 - Если один операнд представляет собой число, а другой является группой узлов, то число усекается или расширяется в двоичном представлении до размера группы. Если при этом усекаются значащие биты, компилятор системы MAX+PLUS II генерирует сообщение об ошибке.

3.8.11. Компараторы (операторы сравнения)

Компараторы используются для сравнения одиночных узлов или групп. Существуют два типа компараторов: логические и арифметические. В языке AHDL реализованы следующие компараторы (**Табл. 3.6**).

Таблица 3.6. Компараторы, реализованные в языке AHDL

Компаратор	Пример	Описание
<code>==</code> (логический)	<code>addr[19..4] == B"B800"</code>	Равны
<code>!=</code> (логический)	<code>b1 != b3</code>	Не равны
<code><</code> (арифметический)	<code>fame[] < power[]</code>	Меньше чем
<code><=</code> (арифметический)	<code>money[] <= power[]</code>	Меньше либо равно
<code>></code> (арифметический с)	<code>love[] > money[]</code>	Больше чем
<code>>=</code> (арифметический)	<code>delta[] >= 0</code>	Больше либо равно

Оператор (`==`) используется исключительно в булевых выражениях. Логические компараторы могут сравнивать одиночные узлы, группы узлов и числа. При сравнении групп узлов или чисел размер групп должен быть одинаковым. Компилятор системы MAX+PLUS II выполняет побитовое сравнение групп и возвращает значение V_{CC} , если результат истина, и GND , если результат — ложь.

Арифметические компараторы могут сравнивать только группы узлов и числа, и группы должны иметь одинаковый размер. Компилятор выполняет над группами операции беззнакового сравнения значений, т.е. каждая группа интерпретируется как положительное двоичное число и сравнивается с другой группой.

3.8.12. Приоритеты в булевых уравнениях

Операнды, разделенные логическими и арифметическими операторами и компараторами, оцениваются по правилам приоритетов, перечисленным ниже (Табл. 3.7) (приоритет 1 является наивысшим). Равноприоритетные операции выполняются по очереди слева направо. Порядок выполнения может быть изменен с помощью заключения в круглые скобки.

Таблица 3.7. Операнды и их приоритеты

Приоритет	Оператор	Компаратор
1	–	Отрицание
1	!	Логическое «НЕ»
2	+	Сложение
2	–	Вычитание
3	==	Равно
3	!=	Не равно
3	<	Меньше чем ...
3	<=	Меньше либо равно ...
3	>	Больше чем ...
3	>=	Больше либо равно ...
4	&	AND
4	!&	NAND
5	\$	XOR
5	!\$	XNOR
6	#	OR
6	!#	NOR

3.8.13. Примитивы

В системе MAX+PLUS II есть разнообразные примитивные функции для проектирования электрических схем. Поскольку логические операторы, порты и некоторые объявления языка AHDL заменяют примитивы в файлах TDF, написанных на языке AHDL, примитивы языка AHDL являются подмножеством примитивов, которые имеются для файлов графического проекта (GDF-файл), как будет показано ниже.

В файлах TDF нет необходимости использования прототипов функций языка AHDL для примитивов. Однако путем включения в ваш TDF-файл

прототипа функции разработчик можете переопределить порядок вызова входов примитива.

3.8.13.1. Примитивы буферов

CARRY LCELL (MCELL)

CASCADE SOFT

EXP TRI

GLOBAL (SCLK)

Примитив CARRY (перенос)

Прототип функции: FUNCTION CARRY (in)

RETURNS (out);

Примитив CARRY определяет логику «переноса на выход» (carry-out) для одной функции и действует как «перенос на вход» (carry-in) для другой функции. Функция переноса осуществляет логику быстрого переноса по цепочке для таких функций как сумматоры и счетчики. Примитив CARRY поддерживается только для устройств семейства FLEX8000. Для других устройств он игнорируется. Если разработчик использует примитив CARRY некорректно, он игнорируется и компилятор выдает соответствующее предупреждающее сообщение.

Вы можете позволить компилятору автоматически вставлять и убирать CARRY во время синтеза логики, для этого нужно выбрать опцию логики с переносом по цепочке Carry Chain или стиль синтеза логики, который включает эту опцию.

Примитив CASCADE (каскад)

Прототип функции: FUNCTION CASCADE (in)

RETURNS (out);

Буфер CASCADE определяет функцию каскадного выхода из логической схемы «И» (AND) или «ИЛИ» (OR) и действует как входной каскад для другой логической схемы «И» или «ИЛИ». Функция входного каскада позволяет организовать каскадный процесс, который представляет собой быстрый выход, расположенный на каждой комбинационной логической ячейке и участвующий в операции «ИЛИ» или «И» с выходом соседней комбинационной логической ячейки в устройстве. С помощью примитива CASCADE логическая ячейка «И» или «ИЛИ», которая питает примитив CASCADE, и логическая ячейка «И» или «ИЛИ», которая запитывается примитивом CASCADE, помещаются в одном устройстве, причем первый символ после проведения над ним логической операции «И» или «ИЛИ» переходит во второй символ. Примитив CASCADE под-

При использовании примитива CASCADE нужно соблюдать следующие правила:

- Примитив CASCADE может формировать сигнал для логической ячейки «И» или «ИЛИ» или быть запитанным одной-единственной схемой «И» или «ИЛИ».
- Обращенная логическая схема «ИЛИ» рассматривается как схема «И», и наоборот. Логическими эквивалентами схемы «И» являются BAND, BNAND и NOR. Логическими эквивалентами схемы «ИЛИ» являются BOR, BNOR and NAND.
- Два примитива CASCADE не могут формировать сигнал для одной и той же логической схемы.
- Примитив CASCADE не может формировать сигнал для логической схемы XOR.
- Примитив CASCADE не может формировать сигнал для примитива вывода OUTPUT или OUTPUTC, или регистра.
- В соответствии с правилами Де Моргана (De Morgan) об инверсии для каскадированных логических схем «И» или «ИЛИ» нужно, чтобы все примитивы в каскадированной цепочке были одного и того же типа. Каскадированная логическая схема «И» не может формировать сигнал для каскадированной схемы «ИЛИ», и наоборот.

Вы можете позволить компилятору автоматически вставлять и убирать примитивы CASCADE во время синтеза логики, для этого нужно выбрать опцию логики с переносом по цепочке Cascade Chain или стиль синтеза логики, который включает эту опцию.

Прототип функции: FUNCTION EXP (in)

Расширительный буфер EXP определяет необходимость дополнительного места при вычислении произведения в проекте. Произведение из расширителя в устройстве инвертируется.

Примитив EXP поддерживается только семейством устройств MAX5000/EP464 и MAX7000. В других устройствах он рассматривается как логическая схема «НЕ» (NOT). Для каждого отдельного устройства

нужно ознакомиться с его техническим паспортом, чтобы узнать, как конкретное устройство использует логические ячейки и расширительные буферы для вычисления произведений.

Будет или нет использоваться расширитель для произведения, зависит от полярности логики, требуемой вычисляемыми функциями. Например, если буфер EXP питает две логические схемы «И» (т.е. логическое умножение), а вторая схема «И» имеет инвертированный вход, то примитив EXP, питающий инвертированный вход, во время логического синтеза убирается, создавая, таким образом, положительную логику. Примитив EXP, питающий неинвертированный вход, не убирается, и для реализации логики используется расширитель для произведения. Обычно логический синтезатор определяет, куда вставлять и откуда убирать буферы. Фирма «Altera» рекомендует, чтобы только опытные разработчики использовали примитив EXP в своих проектах.

В устройствах, содержащих много логических блоков (LAB), выход буфера EXP может формировать сигнал только в пределах одного LAB. Примитив EXP дублируется для каждого LAB, где он требуется. Если проект содержит много расширительных буферов, логический синтезатор может преобразовать их в буферы LCELL, чтобы сбалансировать использование расширителя для произведения и логической ячейки.

Не используйте примитивы EXP для создания намеренной задержки или асинхронного импульса. Задержка этих элементов изменяется с температурой, напряжением питания и при процессах изготовления устройства, поэтому может возникнуть режим состязания и получится ненадежная схема.

Примитив GLOBAL (глобальный)

Прототип функции: FUNCTION GLOBAL (in)
RETURNS (out);

Буфер GLOBAL указывает, что сигнал (signal) должен использовать глобальный синхронный сигнал Clock, Clear, Preset или Output Enable вместо сигналов, сгенерированных внутренней логикой или поступающих через обычные выходы вход/выход. Глобальные сигналы используются в различных семействах устройств в соответствии с их синхронизацией.

Если входной вывод формирует сигнал непосредственно на вход примитива GLOBAL, то выход примитива GLOBAL может быть использован, чтобы формировать сигнал на входы Clock, Clear, Preset или Output Enable к примитиву. Должно существовать непосредственное соединение от выхода GLOBAL ко входу регистра или буфера TRI. Если буфер GLOBAL

формирует сигнал, являющийся входным сигналом Output Enable буфера TRI, то может потребоваться логическая схема «НЕ».

Один вход может проходить через GLOBAL, перед тем как формировать сигнал для входа регистра сигналов Clock, Clear или Preset или вход буфера TRI для сигнала Output Enable.

Глобальные синхросигналы распространяются быстрее, чем асинхронные (атау) сигналы, и могут освобождать ресурсы устройства для реализации другой логики. Для осуществления синхронизации части или всего проекта следует использовать примитив GLOBAL. Для того чтобы проверить, имеют ли регистры глобальную синхронизацию, нужно просмотреть файл отчета Report File обрабатываемого проекта.

Если ваш проект устройства MAX5000 сочетает асинхронный режим работы и режим с глобальной синхронизацией, а модуль компилятора Fitter не может выбрать подходящий, проблема может быть решена удалением буфера GLOBAL. Если аналогичная проблема встретится вам в проекте MAX7000, замените асинхронный режим режимом с глобальной синхронизацией. В качестве альтернативы использованию примитива GLOBAL разработчик может предоставить компилятору автоматически выбирать, каким будет существующий сигнал проекта, глобальным Clock, Clear, Preset или Output Enable, посредством команды Logic Synthesis.

Примитив LCELL

Прототип функции: FUNCTION LCELL (in)

RETURNS (out);

Буфер LCELL выделяется для логической ячейки проекта. Буфер LCELL вырабатывает истинное значение и дополнение логической функции и делает их доступными для всей логики устройства.

Для обратной совместимости с более ранними версиями системы MAX+PLUS II имеется буфер MCELL, функционально такой же, как и буфер LCELL. В новых проектах следует использовать только буфер LCELL.

Буфер LCELL всегда занимает одну логическую ячейку. Он не удаляется из проекта во время логического синтеза.

Не используйте примитивы LCELL для создания задержки или асинхронного импульса. Задержка этих элементов изменяется с температурой, напряжением питания и в процессе изготовления изделия, поэтому могут возникнуть режимы состязаний.

С помощью команды компилятора Automatic LCELL Insertion разработчик может дать указание компилятору автоматически вставлять буферы LCELL в проект, если это необходимо для трассировки проекта в том случае, когда назначения определенных пользователем ресурсов и имеющиеся у устройства не обеспечивают подгонки монтажа в проекте.

Примитив SOFT

Прототип функции: FUNCTION SOFT (in)
RETURNS (out);

Буфер SOFT устанавливает, что логическая ячейка может понадобиться в проекте. При обработке проекта процессором логический синтезатор исследует логику, питающую примитив, и определяет, нужна ли логическая ячейка. Если нужна, буфер SOFT преобразуется в LCELL, в противном случае SOFT удаляется.

Если компилятор показывает, что проект слишком сложный, разработчик может редактировать проект, вставляя буферы SOFT для предотвращения расширения логики. Например, разработчик может добавить буфер SOFT в комбинаторный выход макрофункции для развязки двух комбинаторных цепей. Разработчик может также включить логическую опцию SOFT Buffer Insertion в стили логического синтеза по умолчанию в проект, чтобы компилятор вставлял буферы SOFT автоматически.

Примитив TRI

Прототип функции: FUNCTION TRI (in, oe)
RETURNS (out);

Примитив TRI представляет собой трехстабильный буфер с сигналами входным, выходным и Output Enable (отпирание выхода). Если уровень сигнала Output Enable ВЫСОКИЙ, выход будет управляться входом. При НИЗКОМ уровне сигнала Output Enable выход переходит в состояние высокого импеданса, что позволяет использовать I/O как входной вывод. По умолчанию сигнал Output Enable устанавливается в значение V_{CC} .

Если сигнал Output Enable буфера TRI подсоединяется к V_{CC} или логической функции, которая будет минимизировать до истины, буфер TRI может быть преобразован в буфер SOFT во время логического синтеза.

При использовании буфера TRI нужно соблюдать следующие правила:

- Буфер TRI может управлять только одним выводом BIDIR или BIDIRC. Разработчик должен использовать вывод BIDIR или BIDIRC, если после буфера TRI применяется обратная связь. Если буфер TRI формирует сигнал на логику, он должен также формировать сигнал для BIDIR

- Если сигнал Output Enable не привязывается к V_{CC} , буфер TRI должен формировать сигнал для выводов OUTPUT, OUTPUTC, BIDIR или BIDIRC. Внутренние сигналы могут не быть трехстабильными.

DFF	SRFF
DFFE	SRFFE
JKFF	TFF
JKFFE	TFFE
LATCH	

Для устройств, не поддерживающих сигнал Latch Enable, логический синтез генерирует логические уравнения, содержащие D-триггеры с сигналами Latch Enable. Эти логические уравнения корректно эмулируют логику, заданную в вашем проекте.

Примитив LATCH
Прототип функции: FUNCTION LATCH (D, ENA)
RETURNS (O);

Входы		Выход
ENa	D	Q
L	X	Q ₀ *
H	L	L
H	H	H

Примитив DFF (D-триггер типа Flipflop)
Прототип функции: FUNCTION DFF (D, CLK, CLRN, PRN)
 RETURNS (Q);

Входы				Выход
PRN	CLRN	CLK	D	Q
L	H	X	X	H
H	L	X	X	L
L	L	X	X	Запрещенное состояние
H	H	┐	L	L
H	H	┐	H	H
H	H	L	X	Q ₀ *
H	H	H	X	Q ₀

* Q₀ = уровень Q перед тактовым импульсом.

Примитив DFFE

Прототип функции: FUNCTION DFFE (D, CLK, CLRN, PRN, ENA)
RETURNS (Q);

Входы					Выход
CLRN	PRN	ENA	D	CLK	Q
L	H	X	X	X	L
H	L	X	X	X	H
L	L	X	X	X	Запрещенное состояние
H	H	L	X	X	Q ₀ *
H	H	H	L	┐	L
H	H	H	H	┐	H
H	H	X	X	L	Q ₀ *

* Q₀ = уровень Q перед тактовым импульсом.

Примитив TFF (Триггер Т-типа, Flipflop)

Прототип функции: FUNCTION TFF (T, CLK, CLRN, PRN)
RETURNS (Q);

Входы				Выход
PRN	CLRN	CLK	T	Q
L	H	X	X	H
H	L	X	X	L
L	L	X	X	Запрещенное состояние
H	H	┐	L	Q ₀ *
H	H	┐	H	Бистабильное состояние
H	H	L	X	Q ₀ *

* Q₀ = уровень Q перед тактовым импульсом.

Прототип функции: FUNCTION TFFE (T, CLK, CLRN, PRN, ENA)
RETURNS (Q);

Входы					Выход
CLRN	PRN	ENA	T	CLK	Q
L	H	X	X	X	L
H	L	X	X	X	H
L	L	X	X	X	Запрещенное состояние
H	H	L	X	X	Q_0^*
H	H	H	L	\neg	Q_0^*
H	H	H	H	\neg	Бистабильное состояние
H	H	X	X	L	Q_0^*

* Q_0 = уровень Q перед тактовым импульсом.

Прототип функции: FUNCTION JKFF (J, K, CLK, CLRN, PRN)
RETURNS (Q);

Входы					Выход
PRN	CLRN	CLK	J	K	Q
L	H	X	X	X	H
H	L	X	X	X	L
L	L	X	X	X	Запрещенное состояние
H	H	L	X	X	Q_0^*
H	H	\bar{L}	L	L	Q_0^*
H	H	\bar{L}	H	L	H
H	H	\bar{L}	L	H	L
H	H	\bar{L}	H	H	Бистабильное состояние

* Q_0 = уровень Q перед тактовым импульсом.

```
Прототип функции: FUNCTION JKFFE
                    (J, K, CLK, CLRN, PRN, ENA)
                    RETURNS (Q);
```


Входы						Выход
ENA	PRN	CLRN	CLK	J	K	Q
X	L	H	X	X	X	H
X	H	L	X	X	X	L
X	L	L	X	X	X	Запрещенное состояние
X	H	H	L	X	X	Q_0^*
H	H	H	┐	L	L	Q_0^*
H	H	H	┐	H	L	H
H	H	H	┐	L	H	L
H	H	H	┐	H	H	Бистабильное состояние
L	H	H	X	X	X	Q_0^*

* Q_0 = уровень Q перед тактовым импульсом.

Примитив SRFF (Триггер SR-типа, Flipflop)

Прототип функции: FUNCTION SRFF (S, R, CLK, CLRN, PRN)
RETURNS (Q);

Входы					Выход
PRN	CLRN	CLK	S	R	Q
L	H	X	X	X	H
H	L	X	X	X	L
L	L	X	X	X	Запрещенное состояние
H	H	L	X	X	Q_0^*
H	H	┐	L	L	Q_0^*
H	H	┐	H	L	H
H	H	┐	L	H	L
H	H	┐	H	H	Бистабильное состояние

* Q_0 = уровень Q перед тактовым импульсом.

Примитив SRFFE

Прототип функции: FUNCTION SRFFE (S, R, CLK, CLRN, PRN, ENA)
RETURNS (Q);

Входы						Выход
ENA	PRN	CLRN	CLK	S	R	Q
X	L	H	X	X	X	H
X	H	L	X	X	X	L
X	L	L	X	X	X	Запрещенное состояние
X	H	H	L	X	X	Q ₀ *
H	H	H	┐	L	L	Q ₀ *
H	H	H	┐	H	L	H
H	H	H	┐	L	H	L
H	H	H	┐	H	H	Бистабильное состояние
L	H	H	X	X	X	Q ₀ *

* Q₀ = уровень Q перед тактовым импульсом.

3.8.14. Порты

Порт представляет собой вход или выход примитива, макрофункции или цифрового автомата. Порт может появляться в трех местах файла:

- порт, являющийся входом или выходом текущего файла, объявляется в секции подпроекта Subdesign;
- порт, являющийся входом или выходом текущего файла, может быть назначен (присвоен) выводу, логическому элементу или чипу в секции проекта Design;
- порт, являющийся входом или выходом примера (Instance) примитива или файл проекта более низкого уровня, используется в логической секции Logic.

3.8.14.1. Порты текущего файла

Порт, являющийся входом или выходом текущего файла, объявляется в секции подпроекта Subdesign следующим образом:

<имя порта> : <тип порта> [= <значение порта по умолчанию>]

Есть следующие типы портов:

INPUT

MACHINE INPUT

OUTPUT

MACHINE OUTPUT

BIDIR

Если TDF-файл является файлом верхнего уровня иерархии, имя порта является синонимом имени вывода. Для портов типа INPUT и BIDIR мо-

жет быть определено по умолчанию значение V_{CC} или GND. В случае использования примера данного TDF в файле более высокого уровня это значение используется только если порт остается неподключенным.

Входные и выходные порты, объявленные в секции подпроекта Subdesign, могут быть назначены выводам, логическим элементам, чипам, кликам и логическим опциям при назначении ресурсов Resource Assignment в секции проекта Design. Назначения выводов могут быть сделаны только на верхнем уровне иерархии. Любые назначения выводов на более низких иерархических уровнях игнорируются.

В приводимом ниже примере в секции подпроекта Subdesign объявлены входные, выходные и двунаправленные порты, а в секции проекта Design два входных порта назначаются (присваиваются) выводам.

```
DESIGN IS top
BEGIN

                                DEVICE IS AUTO
                                BEGIN
                                    foo @ 1, bar @ 2 : INPUT;
                                END;

END;
SUBDESIGN top
(
    foo, bar, clk1, clk2 : INPUT = VCC;
    % VCC - это значение порта по умолчанию %
    a0, a1, a2, a3, a4 : OUTPUT;
    b[7..0] : BIDIR;
)
```

Между файлами TDF, (GDF-файл) или WDF можно осуществлять импортирование и экспортирование цифровых автоматов, если разработчик определит входной или выходной порт как MACHINE INPUT или MACHINE OUTPUT в секции подпроекта Subdesign. В прототипе функции (Function Prototype), описывающем этот файл, должно быть указано, какие порты являются цифровыми автоматами с памятью. Порты MACHINE INPUT и MACHINE OUTPUT можно использовать только в файлах более низких уровней в иерархии проекта.

3.8.15.2. Порты примеров (INSTANCE)

Соединение порта, являющегося входом или выходом примера (Instance) в файл проекта более низкого уровня или примитива, осуществляется в логической секции Logic. Для соединения примитива, макрофункции или цифрового автомата с другими частями файла TDF нужно вставить пример (Instance) примитива или макрофункции как ссылку с ключевым словом in-line или дать в конструкции объявления примера Instance, или объявить цифровой автомат в конструкции, а затем использовать порты функции в логической секции.

Если разработчик для вставки примера примитива или макрофункции использует ссылку с ключевым словом in-line, важен порядок перечисления портов, но не их имена. Этот порядок определяется в прототипе функции для примитива или макрофункции.

Если для вставки примера разработчик использует конструкцию объявления примера Instance, важны имена портов, а не порядок их перечисления. Имена портов даются в следующем формате:

<имя примера>.<имя порта>

Здесь <имя примера> представляет собой определенное пользователем имя функции, а <имя порта> совпадает с именем порта, объявленного в логической секции файла TDF более низкого уровня или совпадает с именем вывода в файле проекта другого типа. Это <имя порта> является синонимом имени штыревого вывода для символа, который представляет пример данного файл проекта в файле (GDF-файл).

В приводимом ниже примере триггер типа D объявлен как переменная reg в секции объявления переменных VARIABLE и затем используется в логической секции.

```
VARIABLE
                                reg : DFF;
BEGIN
                                reg.clk = clk
                                reg.d  = d
                                out  = reg.q
END;
```

Все поставляемые фирмой «Altera» примитивы и макрофункции имеют предопределенные имена портов (штыревые выводы), которые описываются в прототипе функции. Наиболее широко используемые имена портов приведены в **Табл. 3.8**.

Таблица 3.8. Имена портов

Имя порта	Описание
.q	Выход бистабильного мультивибратора (Flipflop) или триггера-защелки (Latch)
.d	Информационный вход в триггер D-типа (Flipflop) или триггер-защелку (Latch)
.t	Вход переключения (Toggle) триггера Т-типа (Flipflop)
.j	Вход J триггера J-K-типа (Flipflop)
.k	Вход K триггера J-K-типа (Flipflop)
.s	Вход S триггера S-R-типа (Flipflop)
.r	Вход R триггера S-R-типа (Flipflop)
.clk	Вход тактового сигнала (Clock) триггера (Flipflop)
.ena	Вход отпираания тактового сигнала (Clock Enable) для триггера, отпираания защелки (Latch Enable) и отпираания цифрового автомата (Enable)
.prn	Вход установки триггера (Flipflop) Preset с низким активным уровнем
.clrn	Вход сброса триггера (Flipflop) Clear с низким активным уровнем
.reset	Вход цифрового автомата Reset с высоким активным уровнем
.oe	Вход Output Enable (отпираание выхода) примитива TRI
.in	Основной вход примитивов TRI, SOFT, GLOBAL и LCELL
.out	Выход примитивов TRI, SOFT, GLOBAL и LCELL

3.9. Синтаксис языка AHDL

3.9.1. Лексические элементы

Синтаксис лексических элементов языка AHDL описан ниже с использованием формул Бэкуса—Наура.

```

<основание системы счисления> ::= В | Q | Н | О | X | b | q | h | o | x
<число с основанием> ::= <основание системы счисления> > " <цифра>
{ <цифра> } "
<цифра> ::= <буква> | <десятичная цифра>
<столбец> ::= <десятичная цифра> : 1 : 2
<символ комментария> ::= <любой символ, кроме %> | %%
<комментарий> ::= % { <символ комментария> } %

```

| — { < символ комментария > } <конец строки>
 <десятичное число> ::= <десятичная цифра>:1:10
 <имя проекта> ::= <имя файла>
 <вывод устройства> ::= <символ имени>:1:3
 <десятичная цифра> ::= 0 | 1 | ... | 9
 <имя файла> ::= (<символ имени>):1:8 | ' (<символ имени в кавычках>):1:8 '
 <лабиринт> ::= <строка матрицы> <столбец>
 <буква> ::= A | ... | Z | a | ... | z
 <логический элемент> ::= LC <десятичная цифра>:1:3
 | LC <десятичная цифра>:1:2 <лабиринт>
 | MC <десятичная цифра>:1:3
 | MC <десятичная цифра>:1:2 <лабиринт>
 <символ имени> ::= <буква> | <десятичная цифра> | _
 <число> ::= <десятичное число> | <число с основанием>
 <имя порта> ::= (<символ имени> | /):1:32 | ' (<символ имени в кавычках> | /):1:32 '
 <символ имени в кавычках> ::= <символ имени> | -
 <строка матрицы> ::= <буква>
 <символ строки> ::= <любой символ, кроме " и символа перехода на новую строку> | ""
 <строка> ::= " { <символ строки> } "
 <символическое имя> ::= (<имя символа> | /):1:32 | ' (<символ имени в кавычках> | /):1:32 '

Символическое имя без кавычек не может состоять из одних цифр.

3.9.2. Основные конструкции языка AHDL

С помощью формул Бэкуса—Наура синтаксис файла TDF можно описать следующим образом:

```

<файл AHDL > ::=
<оператор AHDL >
{ < оператор AHDL > }

< оператор AHDL > ::=
    
```

```

<название>
| <задание константы>
| <прототип>
| <оператор включения >
| < варианты>
| < секция проекта >
| <секция подпроекта>

```

```

<операторы> ::=
<оператор>
{ <оператор> }

```

```

<оператор> ::=
<булево уравнение>
| <оператор выбора>
| <условный оператор>
| <оператор таблицы>
| <присвоение по умолчанию>

```

```

<название> ::=
TITLE <строка>;

```

```

<задание константы> ::=
CONSTANT <символическое имя> = <число>;

```

```

<прототип> ::=
FUNCTION <макрофункция> ( <список входов> )
RETURNS ( <список выходов> );
| FUNCTION <примитив> ( <список входов> )
RETURNS ( <список выходов> );

```

```

<макрофункция> ::=
<имя проекта>

```

```

<примитив> ::=
<символическое имя>

```

```

<список входов> ::=

```

<группа прототипа> {, <группа прототипа> }

<список выходов> ::=

<группа прототипа> {, <группа прототипа> }

<оператор включения> ::=

INCLUDE <строка> ;

<вариант> ::=

<вариант устройства>

| <вариант 0-го бита AHDL >

| <логическая опция>

<вариант устройства> ::=

| SECURITY = (ON | OFF)

| TURBO = (ON | OFF | DEFAULT)

<вариант 0-го бита AHDL > ::=

BIT0 = (ANY-произвольный | LSB-младший | MSB-старший)

<логическая опция> ::=

CARRY_CHAIN = (AUTO | IGNORE | MANUAL | DEFAULT)

| CASCADE_CHAIN = (AUTO | IGNORE | MANUAL | DEFAULT)

| DECOMPOSE_GATES = (ON | OFF | DEFAULT)

| DUPLICATE_LOGIC_EXTRACTION = (ON | OFF | DEFAULT)

| EXPANDER_FACTORING = (ON | OFF | DEFAULT)

| MINIMIZATION = (FULL | PARTIAL | NONE | DEFAULT)

| MULTI-LEVEL_FACTORING = (ON | OFF | DEFAULT)

| NOT_GATE_PUSH-BACK = (ON | OFF | DEFAULT)

| OPTIMIZE = (AREA | DELAY | ROUTING | DEFAULT)

| PARALLEL_EXPANDERS = (ON | OFF | DEFAULT)

| PERIPHERAL_REGISTER = (ON | OFF | DEFAULT)

| REDUCE_LOGIC = (ON | OFF | DEFAULT)

| REFACTORIZATION = (ON | OFF | DEFAULT)

| REGISTER_OPTIMIZATION = (ON | OFF | DEFAULT)

| RESYNTHESIZE_NETWORK = (ON | OFF | DEFAULT)

| SLOW_SLEW_RATE = (ON | OFF | DEFAULT)

| SOFT_BUFFER_INSERTION = (ON | OFF | DEFAULT)


```
| STYLE = <имя стиля>
| SUBFACTOR_EXTRACTION = (ON | OFF | DEFAULT)
| TURBO = (ON | OFF | DEFAULT)
| XOR_SYNTHESIS = (ON | OFF | DEFAULT)
```

```
<имя стиля> ::=
FAST
| WYSIWYG
| NORMAL
| <стиль пользователя>
```

```
<стиль пользователя> ::= <символическое имя>
```

Каждый <вариант> за исключением BIT0 и имени стиля может быть сокращен до 3 первых символов по обе стороны от знака равенства (=). Однако фирма «Altera» рекомендует писать полные имена для удобства документирования.

3.9.3. Синтаксис объявления названия

```
<название> ::=
TITLE <строка>;
```

3.9.4. Синтаксис оператора включения

```
<оператор включения> ::=
INCLUDE <имя файла>;
```

Описываемый в операторе включения файл должен иметь расширение .inc, а <имя файла> не должно содержать путь.

3.9.5. Синтаксис задания константы

```
<задание константы> ::=
CONSTANT <символьное имя> = <число>;
```

3.9.6. Синтаксис прототипа функции

```
<прототип> ::=
FUNCTION <макрофункция> ( <список входов> )
```

```

RETURNS ( <список выходов> );
| FUNCTION <примитив> ( <список входов> )
RETURNS ( <список выходов> );

```

```

<макрофункция> ::=
<имя проекта>

```

```

<примитив> ::=
<символьное имя>

```

```

<список входов> ::=
<группа прототипа> {, <группа прототипа> }

```

```

<список выходов> ::=
<группа прототипа> {, <группа прототипа> }

```

3.9.7. Синтаксис оператора вариантов

```

<варианты> ::=
OPTIONS <вариант> {, <вариант> };

```

```

<вариант> ::=
<вариант устройства>
| <вариант 0-го бита AHDL>
| <логическая опция>

```

```

<вариант устройства> ::=
| SECURITY = (ON-вкл | OFF-откл)
| TURBO = (ON-вкл | OFF-откл | DEFAULT-по умолчанию)

```

```

<вариант 0-го бита AHDL> ::=
BIT0 = (ANY-произвольный | LSB-младший | MSB-старший)

```

```

<логическая опция> ::=
CARRY_CHAIN = (AUTO-авто | IGNORE-игнорировать | MANUAL-
ручной | DEFAULT-по умолчанию)
| CASCADE_CHAIN = (AUTO | IGNORE | MANUAL | DEFAULT)
| DECOMPOSE_GATES = (ON-вкл | OFF-откл | DEFAULT-по умолча-
нию)

```

```

| DUPLICATE_LOGIC_EXTRACTION = (ON | OFF | DEFAULT)
| EXPANDER_FACTORIZING = (ON | OFF | DEFAULT)
| MINIMIZATION = (FULL-полная | PARTIAL-частичная | NONE-нет |
DEFAULT-по умолчанию)
| MULTI-LEVEL_FACTORIZING = (ON | OFF | DEFAULT)
| NOT_GATE_PUSH-BACK = (ON | OFF | DEFAULT)
| OPTIMIZE = (AREA-площадь | DELAY-задержка | ROUTING-марш-
рутизация | DEFAULT-по умолчанию)
| PARALLEL_EXPANDERS = (ON | OFF | DEFAULT)
| PERIPHERAL_REGISTER = (ON | OFF | DEFAULT)
| REDUCE_LOGIC = (ON | OFF | DEFAULT)
| REFACTORIZATION = (ON | OFF | DEFAULT)
| REGISTER_OPTIMIZATION = (ON | OFF | DEFAULT)
| RESYNTHESIZE_NETWORK = (ON | OFF | DEFAULT)
| SLOW_SLEW_RATE = (ON | OFF | DEFAULT)
| SOFT_BUFFER_INSERTION = (ON | OFF | DEFAULT)
| STYLE = <имя стиля>
| SUBFACTOR_EXTRACTION = (ON | OFF | DEFAULT)
| TURBO = (ON | OFF | DEFAULT)
| XOR_SYNTHESIS = (ON | OFF | DEFAULT)

```

```

<имя стиля> ::=

```

```

FAST

```

```

| WYSIWYG

```

```

| NORMAL

```

```

| <стиль пользователя>

```

```

<стиль пользователя> ::= <символьное имя>

```

1. Каждый <вариант>, кроме BIT0 и имени стиля, может быть сокращен до трех символов по обеим сторонам знака равенства (=). Однако фирма «Altera» рекомендует использовать полное имя для удобства документации.

2. Типы вариантов, которые могут быть включены в оператор OPTIONS, определяются местоположением этого оператора и правилами построения содержимого файла.

3.9.8. Синтаксис секции подпроекта Subdesign

```
<секция подпроекта> ::=
SUBDESIGN <имя проекта>
(
  { <список сигналов> ; }
  <список сигналов> [ ; ]
)
[ <переменные> ]
BEGIN
  <операторы>
END;

<список сигналов> ::=
<список портов> : <тип порта>

<тип порта> ::=
INPUT [ = VCC | = GND ]
| OUTPUT
| BIDIR [ = VCC | = GND ]
| MACHINE INPUT
| MACHINE OUTPUT
```

Ключевые слова BEGIN и END обрамляют логическую секцию, которая является телом секции подпроекта.

3.9.9. Синтаксис секции переменных

```
<переменные> ::=
VARIABLE
<список портов> : <тип переменной> ;
{ <список портов> : <тип переменной> ; }

<тип переменной> ::=
NODE
| <макрофункция>
| <примитив>
```

```
| <цифровой автомат (state machine)>
| <псевдоним цифрового автомата>
```

```
<макрофункция> ::=
<имя проекта>
```

```
<примитив> ::=
<символьное имя>
```

3.9.10. Синтаксис объявления цифрового автомата

```
<символьное имя> : <цифровой автомат (state machine)>;
```

```
<цифровой автомат (state machine)> ::=
MACHINE [ <биты> ] <состояния>
```

```
<биты> ::=
OF BITS ( <список портов> )
```

```
<состояния> ::=
WITH STATES ( <состояние> {, <состояние> } )
```

```
<состояние> ::=
<символьное имя> [ = <значение состояния> ]
```

```
<значение состояния> ::=
<число>
| <символьное имя>
```

3.9.11. Синтаксис объявления псевдонима цифрового автомата

```
<символьное имя> : MACHINE;
```

Объявление псевдонима цифрового автомата не задает биты или имена состояний. Эта информация импортируется из секции подпроекта через порт MACHINE INPUT файла более высокого уровня в иерархии проекта или же через прототип функции из файла более низкого уровня в иерархии проекта.

3.9.12. Синтаксис логической секции

Логическая секция, заключенная между ключевыми словами BEGIN и END, представляет собой тело секции подпроекта. Информация о синтаксисе приводится отдельно для каждого варианта логической секции.

3.9.13. Синтаксис булевых уравнений

```
<булево уравнение> ::=  
<левая часть> = <правая часть>;
```

3.9.14. Синтаксис булевых уравнений управления

```
<символьное имя>.clk = <правая часть>;  
[<символьное имя>.reset = <правая часть>;]  
[<символьное имя>.ena = <правая часть>;]
```

Элемент <символьное имя> в булевом уравнении управления должен быть объявлен так же, как в объявлении цифрового автомата в секции переменных.

3.9.15. Синтаксис оператора CASE

```
<оператор выбора> ::=  
CASE <правая часть> IS  
WHEN <выбор> => <операторы>  
{ WHEN <выбор> => <операторы> }  
[ WHEN OTHERS => <операторы> ]  
END CASE;  
  
<выбор> ::=  
<группа констант> {, <группа констант> }
```

3.9.16. Объявление по умолчанию

```
<объявление по умолчанию> ::=  
DEFAULTS  
<задание значения> ;  
{ <задание значения> ; }  
END DEFAULTS;  
<задание значения> ::=  
<левая часть> = <группа констант>
```

3.9.17. Синтаксис условного оператора IF

```

<условный оператор> ::=
IF <правая часть> THEN
<операторы>
{ ELSIF <правая часть> THEN
<операторы> }
[ ELSE <правая часть> THEN
<операторы> ]
END IF;

```

3.9.18. Синтаксис встроенных (in-line) ссылок на макрофункцию или примитив

```

<макрофункция> ( <список правых частей> )
<примитив> ( <список правых частей> )

```

3.9.19. Синтаксис объявления таблицы истинности

```

<оператор таблицы> ::=
TABLE <входы> => <выходы> ;
<входные значения> => <выходные значения> ;
{ <входные значения> => <выходные значения> ; }
END TABLE;

<входы> ::=
<правая часть> {, <правая часть> }

<выходы> ::=
<левая часть> {, <левая часть> }

<входные значения> ::=
<группа констант> {, <группа констант> }

<выходные значения> ::=
<группа констант> {, <группа констант> }

```

3.9.20. Синтаксис порта

Порт, являющийся в текущем файле входом или выходом, объявляется в секции подпроекта в следующем формате:

```

<имя порта> : <тип порта> [ = <значение порта по умолчанию> ]

```

Имеются следующие типы портов:

INPUT	MACHINE INPUT
OUTPUT	MACHINE OUTPUT
	BIDIR

Порты, являющиеся входами и выходами примера (Instance) примитива или макрофункции, используются в следующем формате:

<имя примера>.<имя порта>

<имя примера> ::=

<символьное имя>

3.9.21. Синтаксис группы

Группы могут быть записаны в следующих двух нотациях:

1. Нотация <десятичное имя группы> состоит из символьного имени, за которым следует диапазон десятичных чисел, заключенный в квадратные скобки, например a[4..1]. Допускается указание только одного диапазона после символьного имени.

<десятичное имя группы> ::=

<символьное имя> [<диапазон>]

<диапазон> ::=

<десятичное число>

| <десятичное число>..<десятичное число>

Если группа уже определена, можно для краткости вместо диапазона указывать только пустые скобки []. Вместо диапазона можно также использовать одно только число, например a[5]. Однако в такой нотации обозначается только одно символьное имя, а не имя группы, и это эквивалентно записи a5.

2. Нотация <последовательное имя группы> состоит из списка символьных имен, портов или чисел, разделенных запятой и заключенных в круглые скобки, например (a, b, c). Десятичные имена группы также можно перечислять в круглых скобках.

<последовательное имя группы> ::= (<список правых частей>)

<список правых частей> ::=

[<правая часть>] {, [<правая часть>] }

Данная нотация полезна для задания портов примера (Instance) функции, который объявляется в объявлении примера.

3.9.22. Синтаксические группы и списки

```

<группа констант> ::=
<символьное имя>
| <число>
| VCC
| GND
| ( <список группы констант> )

<список группы констант> ::=
[ <группа констант> ] {, [ <группа констант> ] }

<группа точек> ::=
<имя порта>
| <символьное имя> [ ]
| <символьное имя> [ <диапазон> ]
| ( <список группы точек> )

<список группы точек> ::=
[ <группа точек> ] {, [ <группа точек> ] }

<левая часть> ::=
< левая часть >. <группа точек>
| <символьное имя>
| <символьное имя> [ ]
| <символьное имя> [ <диапазон> ]
| ! < левая часть >
| ( <список левых частей> )

<список левых частей> ::=
[ < левая часть > ] {, [ < левая часть > ] }

<группа портов> ::=
<имя порта>
| <символьное имя> [ <диапазон> ]
| ( <список портов> )

<список портов> ::=
<группа портов> {, <группа портов> }

```

```

<группа прототипа> ::=
<группа портов>
| MACHINE <символьное имя>

<диапазон> ::=
<десятичное число> [.. <десятичное число> ]

<правая часть> ::=
<правая часть>. <группа точек>
| ! <правая часть>
| <число>
| <символьное имя>
| <символьное имя> [ ]
| <символьное имя> [ <диапазон> ]
| VCC
| GND
| <правая часть> == <правая часть>
| <правая часть> >= <правая часть>
| <правая часть> > <правая часть>
| <правая часть> <= <правая часть>
| <правая часть> < <правая часть>
| <правая часть> != <правая часть>
| <правая часть> # <правая часть>
| <правая часть> !# <правая часть>
| <правая часть> & <правая часть>
| <правая часть> !& <правая часть>
| <правая часть> $ <правая часть>
| <правая часть> !$ <правая часть>
| <правая часть> + <правая часть>
| <правая часть> - <правая часть>
| - <правая часть>
| + <правая часть>
| <макрофункция> ( <список правых частей> )
| <примитив> ( <список правых частей> )
| ( <список правых частей> )

<список правых частей> ::=
[ <правая часть> ] {, [ <правая часть> ] }
```

Глава 4. **Язык описания аппаратуры VHDL**

4.1. Общие сведения

Языки описания аппаратуры (Hardware Description Language) являются формальной записью, которая может быть использована на всех этапах разработки цифровых электронных систем. Это возможно вследствие того, что язык легко воспринимается как машиной, так и человеком, он может использоваться на этапах проектирования, верификации, синтеза и тестирования аппаратуры так же, как и для передачи данных о проекте, модификации и сопровождении. Наиболее универсальным и распространенным языком описания аппаратуры является VHDL. На этом языке возможно как поведенческое, так структурное и потоковое описание цифровых схем.

Язык VHDL используется во многих системах для моделирования цифровых схем, проектирования программируемых логических интегральных микросхем, базовых матричных кристаллов, заказных интегральных микросхем. С точки зрения программиста, язык VHDL состоит как бы из двух компонент — общеалгоритмической и проблемно-ориентированной.

Общеалгоритмическая компонента VHDL — это язык, близкий по синтаксису и семантике к современным языкам программирования типа «Паскаль», «С» и др. Язык относится к классу строго типизированных. Помимо встроенных (пакет STANDART), простых (скалярных) типов данных: целый, вещественный, булевский, битовый, данных типа время, данных типа ссылка (указатель), пользователь может вводить свои типы данных (перечислимый, диапазонный и др.).

Помимо скалярных данных, можно использовать агрегаты: массивы ARRAY, в том числе и битовые векторы BIT_VECTOR, и символьные строки STRING, записи RECORD, файлы FILE.

Последовательно выполняемые (последовательные) операторы VHDL могут использоваться в описании процессов, процедур и функций. Их состав включает:

- оператор присваивания переменной (`:=`);
- последовательный оператор назначения сигналу (`<=`);
- последовательный оператор утверждения (`ASSERT`);
- условный оператор (`IF`);
- оператор выбора (`CASE`);
- оператор цикла (`LOOP`);
- пустой оператор (`NULL`);
- оператор возврата процедуры-функции (`RETURN`);
- оператор последовательного вызова процедуры.

Язык поддерживает концепцию пакетного и структурного программирования. Сложные операторы заключены в операторные скобки: `IF- END IF`; `PROCESS- END PROCESS`; `CASE- END CASE`; `LOOP- END LOOP` и т. д.

Различаются локальные и глобальные переменные. Область «видимости» локальных переменных ограничена пределами блока (процессного, процедурного, оператора блока, оператора описания архитектуры).

Фрагменты описаний, которые могут независимо анализироваться компилятором и при отсутствии ошибок помещаться в библиотеку проекта (рабочую библиотеку `WORK`), называются проектными пакетами (`Design Unit`). Такими пакетами могут быть объявление интерфейса объекта проекта (`Entity`), объявление архитектуры (`Architecture`), объявление конфигурации (`Configuration`), объявление интерфейса пакета (`Package`) и объявление тела пакета (`Package body`).

Модули проекта, в свою очередь, можно разбить на две категории: первичные и вторичные. К первичным пакетам относятся объявления пакета, объекта проекта, конфигурации. К вторичным — объявление архитектуры, тела пакета. Один или несколько модулей проекта могут быть помещены в один файл, называемый файлом проекта.

Каждый проанализированный модуль проекта помещается в библиотеку проекта (`Design Library`) и становится библиотечным модулем (`Library Unit`). Каждая библиотека проекта в языке VHDL имеет логическое имя (идентификатор). По отношению к сеансу работы с VHDL-системой существуют два класса рабочих библиотек проекта: рабочие библиотеки и библиотеки ресурсов.

Рабочая библиотека — это библиотека WORK, с которой в данном сеансе работает пользователь и в которую помещается пакет, полученный в результате анализа пакета проекта.

Библиотека ресурсов — это библиотека, содержащая библиотечные модули, ссылка на которые имеется в анализируемом модуле проекта.

В каждый конкретный момент времени пользователь работает с одной рабочей библиотекой и произвольным количеством библиотек ресурсов.

Модули, как и в обычных алгоритмических языках, — это средство выделения из ряда программ и подпрограмм общих типов данных, переменных, процедур и функций, позволяющее упростить, в частности, процесс их замены.

Так же, как в описаниях проектируемых систем разделяются описания интерфейсов и тел, в VHDL у пакета разделяются описание интерфейса и тела пакета. По умолчанию предусмотрено подключение стандартных пакетов STANDART и TEXT 10. Пакет STANDART, в частности, содержит описание булевских операций над битовыми данными и битовыми векторами. Нестандартные пакеты реализуются пользователями, желающими более точно отобразить свойства описываемых ими объектов. Например, можно в пользовательском пакете переопределить логические операции «И», «ИЛИ» и «НЕ» и перейти от булева ("0", "1") к многозначному ("1", "0", "X", "Z") алфавиту моделирования.

Проблемно-ориентированная компонента позволяет описывать цифровые системы в привычных разработчику понятиях и терминах. Сюда можно отнести:

- понятие модельного времени (Now);
- данные типа (Time), позволяющие указывать время задержки в физических единицах;
- данные вида сигнал (Signal), значение которых изменяется не мгновенно, как у обычных переменных, а с указанной задержкой, а также специальные операции и функции над ними;
- средства объявления объектов и их архитектур.

Если говорить про операторную часть проблемно-ориентированной компоненты, то условно ее можно разделить на: средства поведенческого описания аппаратуры (параллельные процессы и средства их взаимодействия); средства потокового описания (описание на уровне межрегистровых передач) — параллельные операторы назначения сигнала (\leq) с транспортной или инерциальной задержкой передачи сигналов и средства

структурного описания объектов (операторы конкретизации компонент с заданием карт портов (PORT MAP) и карт настройки (GENERIC MAP), объявление конфигурации и т. д.).

Параллельные операторы VHDL включают:

- оператор процесса (PROCESS);
- оператор блока (BLOCK);
- параллельный оператор назначения сигналу (\leq);
- оператор условного назначения сигналу (WHEN);
- оператор селективного назначения сигналу (SELECT);
- параллельный оператор утверждения (ASSERT);
- параллельный оператор вызова процедуры;
- оператор конкретизации компоненты (PORT MAP);
- оператор генерации конкретизации (GENERATE).

Как видно из этого перечня, последовательные и параллельные операции назначения, вызова процедуры и утверждения различаются контекстно, т.е. внутри процессов и процедур они последовательные, вне — параллельные.

Базовым элементом описания систем на языке VHDL является блок. Блок содержит раздел описаний данных и раздел параллельно исполняемых операторов. Частным случаем блока является описание архитектуры объекта. В рамках описания архитектуры могут использоваться внутренние, вложенные блоки. Наряду со всеми преимуществами блочной структуры программы и ее соответствием естественному иерархическому представлению структуры проекта операторы блока языка VHDL позволяют устанавливать условия охраны (запрета) входа в блок. Только при истинности значения охранного выражения управление передается в блок и инициирует выполнение операторов его тела.

4.2. Алфавит языка

Как и любой другой язык программирования, VHDL имеет свой алфавит — набор символов, разрешенных к использованию и воспринимаемых компилятором. В алфавит языка входят:

1. Латинские строчные и прописные буквы:
A, B, . . . , Z и a, b, . . . , z.
2. Цифры от 0 до 9.
3. Символ подчеркивания «_» (код ASCII номер 95).

Из символов, перечисленных в пп.1—3 (и только из них!), могут конструироваться идентификаторы в программе. Кроме того, написание идентификаторов должно подчиняться следующим правилам:

- идентификатор не может быть зарезервированным словом языка;
- идентификатор должен начинаться с буквы;
- идентификатор не может заканчиваться символом подчеркивания «_»;
- идентификатор не может содержать два последовательных символа подчеркивания «__».

Примеры корректных идентификаторов:

`cont, clock2, full_add`

Примеры некорректных идентификаторов:

`1clock, _adder, add__sub, entity`

Следует отметить что прописные и строчные буквы не различаются, т.е. идентификаторы `clock` и `CLOCK` являются эквивалентными.

4. Символ «пробел» (код 32), символ табуляции (код 9), символ новой строки (коды 10 и 13).

Данные символы являются разделителями слов в конструкциях языка. Количество разделителей не имеет значения. Таким образом, следующие выражения для компилятора будут эквивалентны:

```
count:=2+2;
count := 2 + 2;
count :=      2
+
2;
```

5. Специальные символы, участвующие в построении конструкций языка:

`+ - * / = < > . , () : ; # ' « |`

6. Составные символы, воспринимаемые компилятором как один символ:

`<= >= => := /=`

Разделители между элементами составных символов недопустимы.

4.2.1. Комментарии

Признаком комментария являются два символа тире («—»). Компилятор игнорирует текст начиная с символов «—» до конца строки, т.е. комментарий может включать в себя символы, не входящие в алфавит языка (в частности русские буквы).

4.2.2. Числа

В стандарте языка определены числа как целого, так и вещественного типа. Однако средства синтеза ПЛИС допускают применение только целых чисел. Целое число в VHDL может быть представлено в одной из четырех систем счисления: двоичной, десятичной, восьмеричной и шестнадцатеричной. Конкретные форматы написания числовых значений будут описаны далее при рассмотрении различных типов языка. К разновидности числовых значений можно отнести также битовые строки.

4.2.3. Символы

Запись символа представляет собой собственно символ, заключенный в одиночные кавычки. Например:

```
'A', '*', ' '
```

В средствах синтеза ПЛИС область применения символов ограничена использованием их в качестве элементов перечислимых типов.

4.2.4. Строки

Строки представляют собой набор символов, заключенных в двойные кавычки. Чтобы включить двойную кавычку в строку, необходимо ввести две двойные кавычки. Например:

```
"A string"
```

```
"A string in a string "A string""
```

4.3. Типы данных

Подобно высокоуровневым языкам программирования, VHDL является языком со строгой типизацией. Каждый тип данных в VHDL имеет определенный набор принимаемых значений и набор допустимых операций. В языке предопределено достаточное количество простых и сложных типов, а также имеются средства для образования типов, определяемых пользователем.

Необходимо отметить, что в данном пособии рассматриваются не все типы данных, определенные в стандарте, а только те, которые поддерживаются средствами синтеза ПЛИС.

4.3.1. Простые типы

Следующие простые типы являются предопределенными:

1. BOOLEAN (логический) — объекты данного типа могут принимать значения FALSE (ложь) и TRUE (истина).

2. INTEGER (целый) — значения данного типа представляют собой 32-разрядные числа со знаком. Объекты типа INTEGER могут содержать значения из диапазона $-(2^{31}-1) \dots 2^{31}-1$ ($-2147483647 \dots 2147483647$).

3. BIT (битовый) — представляет один логический бит. Объекты данного типа могут содержать значение '0' или '1'.

4. STD_LOGIC (битовый) — представляет один бит данных. Объекты данного типа могут принимать 9 состояний. Данный тип определен стандартом IEEE 1164 для замены типа BIT.

5. STD_ULOGIC (битовый) — представляет один бит данных. Объекты данного типа могут принимать 9 состояний. Данный тип определен стандартом IEEE 1164 для замены типа BIT (см. примечание).

6. ENUMERATED (перечислимый) — используется для задания пользовательских типов.

7. SEVERITY_LEVEL — перечислимый тип, используется только в операторе ASSERT.

8. CHARACTER — символьный тип.

Примечание. В действительности тип STD_ULOGIC является базовым типом для типа STD_LOGIC, т.е. объекты обоих типов могут принимать одно и то же множество значений и имеют одинаковый набор допустимых операций. Единственное различие между типами заключается в том, что для типа STD_ULOGIC не определена функция разрешения (resolving function). В языке VHDL функция разрешения используется для определения значения сигнала, имеющего несколько источников (драйверов).

Пример: Выходы двух буферов с тремя состояниями подключены к одной цепи X. Пусть выход одного буфера установился в состояние 'Z', а выход другого — в состояние '1'. Функция разрешения определяет, что в этом случае значение сигнала X будет равно '1'. Поскольку для типа STD_ULOGIC не определена функция разрешения, сигналы этого типа могут иметь только один источник.

Далее рассматривается только тип STD_LOGIC, однако все сказанное будет справедливо и для типа STD_ULOGIC. На практике в подавляющем большинстве случаев достаточно использования типа STD_LOGIC.

4.3.2. Сложные типы

Из всей совокупности сложных типов, определенных в стандарте языка, для синтеза логических схем используются только массивы (тип ARRAY) и записи (тип RECORD). Однако тип RECORD поддерживается не всеми средствами синтеза и в данной книге рассмотрен не будет.

Следующие типы-массивы являются предопределенными:

1. BIT_VECTOR — одномерный массив элементов типа BIT;
2. STD_LOGIC_VECTOR — одномерный массив элементов типа STD_LOGIC;
3. STD_ULOGIC_VECTOR — одномерный массив элементов типа STD_ULOGIC;
4. STRING — одномерный массив элементов типа CHARACTER.

Направление и границы диапазона индексов не содержатся в определении указанных типов и должны быть указаны непосредственно при объявлении объектов данных типов.

4.3.3. Описание простых типов

Тип BOOLEAN. Тип BOOLEAN является перечислимым типом. Объект данного типа может принимать значения FALSE (ложь) и TRUE (истина), причем FALSE эквивалентно "0", а TRUE эквивалентно "1".

Все три типа объектов в VHDL (константы, переменные и сигналы) могут иметь тип BOOLEAN. Таким объектам может быть присвоено только значение типа BOOLEAN.

Пример:

```
PROCESS (a, b)
    VARIABLE cond : BOOLEAN;
BEGIN
    cond := a > b;
    IF cond THEN
        output <= '1';
    ELSE
        output <= '0';
    END IF;
END PROCESS;
```

Операторы отношения. Значения типа BOOLEAN могут участвовать в выражениях. Операторы отношения (=, /=, <, <=, >, >=) определены для операндов типа BOOLEAN и одномерных массивов, содержащих элементы типа BOOLEAN. Результат выражения также имеет тип BOOLEAN. (Как для всех перечислимых типов, операции сравнения над одномерными массивами типа BOOLEAN производятся поэлементно, начиная с крайнего левого элемента).

Логические операторы. Для операндов типа BOOLEAN и одномерных массивов, содержащих элементы типа BOOLEAN, определены все логические операции (AND, OR, NAND, NOR, XOR и NOT). Тип и размер операндов должны быть одинаковыми. Тип и размер результата такой же, как тип и размер операндов.

Оператор конкатенации. Оператор конкатенации также определен для операндов типа BOOLEAN и одномерных массивов, содержащих элементы типа BOOLEAN. Результат выражения представляет собой одномерный массив, содержащий элементы типа BOOLEAN. Размер массива равен сумме размеров операндов.

Другие операторы. Другие операции над операндами типа BOOLEAN не определены.

Тип INTEGER. Стандарт VHDL определяет тип INTEGER для использования в арифметических выражениях. По умолчанию объекты типа INTEGER имеют размерность 32 бита и представляют целое число в интервале — $(2^{31}-1) \dots 2^{31}-1$ ($-2147483647 \dots 2147483647$). Стандарт языка позволяет также объявлять объекты типа INTEGER, имеющие размер меньше 32 бит, используя ключевое слово RANGE, ограничивающее диапазон возможных значений:

SIGNAL X : INTEGER RANGE -127 TO 127

Данная конструкция определяет X как 8-битное число. Кроме того, можно определить ограниченный целый тип, используя следующую конструкцию:

TYPE имя_типа IS RANGE диапазон_индексов;

диапазон_индексов определяется следующим образом:

m TO n
n DOWNTO m,

где m, n — целочисленные константы, $m \leq n$.

Пример:

TYPE byte_int 0 TO 255;

TYPE signed_word_int is range -32768 TO 32768;

TYPE bit_index is range 31 DOWNT0 0;

Значения типа INTEGER записываются в следующей форме:

[основание #] разряд { [_] разряд } [#]

По умолчанию «основание» принимается равным 10. Допустимыми также являются значения 2, 8, 16. При записи числа допускается использование одиночных символов подчеркивания, которые не влияют на результирующее значение.

Пример:

CONSTANT min : INTEGER := 0;

CONSTANT group : INTEGER := 13_452; — эквивалентно 13452

CONSTANT max : INTEGER := 16#FF#;

Допустимое использование

Операторы отношения. Значения типа INTEGER могут участвовать в выражениях. Операторы отношения (=, /=, <, <=, >, >=) определены для операндов типа INTEGER и одномерных массивов, содержащих элементы типа INTEGER. Результат выражения имеет тип BOOLEAN.

Арифметические операторы. Операторы +, -, ABS допустимы для операндов типа INTEGER. Результат выражения имеет тип INTEGER.

Операторы *, /, MOD, REM допустимы в следующих случаях:

— если оба операнда являются константами (CONSTANT);

— если второй операнд является константой, и его значение равно $2n$, где $n = 0, 1, 2, 3, \dots$

Применение операторов *, /, MOD, REM недопустимо, если оба операнда являются сигналами (SIGNAL) или переменными (VARIABLE). Оператор возведения в степень (**), как правило, не поддерживается средствами синтеза.

Другие операторы. Другие операции над операндами типа INTEGER не определены.

Тип BIT. Объект данного типа может принимать значение '0' (лог. "0") или '1' (лог. "1").

Примечание. Стандартом IEEE 1164 определена замена типа BIT на более гибкий тип STD_LOGIC. Поэтому использование типа BIT в новых разработках не рекомендуется.

Допустимое использование

Операторы отношения. Значения типа BIT могут участвовать в выражениях. Операторы отношения (=, /=, <, <=, >, >=) определены для операндов типа BIT и одномерных массивов, содержащих элементы типа BIT. Результат выражения имеет тип BOOLEAN. (Как для всех перечислимых типов, операции сравнения над одномерными массивами типа BIT производятся поэлементно, начиная с крайнего левого элемента).

Логические операторы. Для операндов типа BIT и одномерных массивов, содержащих элементы типа BIT, определены все логические операции (AND, OR, NAND, NOR, XOR и NOT). Тип и размер операндов должны быть одинаковыми. Тип и размер результата такой же, как тип и размер операндов.

Оператор конкатенации. Оператор конкатенации также определен для операндов типа BIT и одномерных массивов, содержащих элементы типа BIT. Результат выражения представляет собой одномерный массив, содержащий элементы типа BIT. Размер массива равен сумме размеров операндов.

Другие операторы. Другие операции над операндами типа BIT не определены.

Тип STD_LOGIC. Тип STD_LOGIC является перечислимым типом. Объекты типа STD_LOGIC могут принимать 9 значений: '0', '1', 'Z', '-', 'L', 'H', 'U', 'X', 'W'.

Для синтеза логических схем используются только первые четыре:

'0' — логический "0";

'1' — логическая "1";

'Z' — третье состояние;

'-' — не подключен.

Допустимое использование

Операторы отношения. Значения типа STD_LOGIC могут участвовать в выражениях. Операторы отношения (=, /=, <, <=, >, >=) определены для операндов типа STD_LOGIC и одномерных массивов, содержащих элементы типа STD_LOGIC. Результат выражения имеет тип BOOLEAN. (Как для всех перечислимых типов, операции сравнения над одномерными массивами типа STD_LOGIC производятся поэлементно, начиная с крайнего левого элемента).

Логические операторы. Для операндов типа STD_LOGIC и одномерных массивов, содержащих элементы типа STD_LOGIC определены все логические операции (AND, OR, NAND, NOR, XOR и NOT). Тип и размер операндов должны быть одинаковыми. Тип и размер результата такой же, как тип и размер операндов.

Оператор конкатенации. Оператор конкатенации также определен для операндов типа STD_LOGIC и одномерных массивов, содержащих элементы типа STD_LOGIC. Результат выражения представляет собой одномерный массив, содержащий элементы типа STD_LOGIC. Размер массива равен сумме размеров операндов.

Другие операторы. Другие операции над операндами типа STD_LOGIC не определены.

Перечислимый тип. Перечислимый тип — это такой тип данных, при котором количество всех возможных значений конечно. Строго говоря, все описанные выше типы являются перечислимыми.

Применение перечислимых типов преследует две цели:

- улучшение смысловой читаемости программы;
- более четкий и простой визуальный контроль значений.

Наиболее часто перечислимый тип используется для обозначения состояний конечных автоматов. Перечислимый тип объявляется путем перечисления названий элементов-значений. Объекты, тип которых объявлен как перечислимый, могут содержать только те значения, которые указаны при перечислении.

Элементы перечислимого типа должны быть идентификаторами или символами, которые должны быть уникальными в пределах одного типа. Повторное использование названий элементов в других перечислимых типах разрешается.

Объявление перечислимого типа имеет вид:

TYPE имя_типа IS (название_элемента [, название_элемента]).

Пример:

Type State_type IS (stateA, stateB, stateC);

VARIABLE State : State_type;

.

.

.

State := stateB

В данном примере объявляется переменная `state`, допустимыми значениями которой являются `stateA`, `stateB`, `stateC`.

Примеры predefined перечислимых типов:

```
TYPE SEVERITY_LEVEL IS (NOTE, WARNING, ERROR, FAILURE);  
TYPE BOOLEAN IS (FALSE, TRUE);  
TYPE BIT IS ('0', '1');  
TYPE STD_LOGIC IS ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

Любой перечислимый тип имеет внутреннюю нумерацию: первый элемент всегда имеет номер 0, второй – номер 1 и т.д. Порядок нумерации соответствует порядку перечисления.

Допустимое использование

Операторы отношения. Значения определенных пользователем перечислимых типов могут участвовать в выражениях. Операторы отношения (`=`, `/=`, `<`, `<=`, `>`, `>=`) определены как для перечислимых типов, так и для одномерных массивов, содержащих элементы этих типов. Результат выражения имеет тип `BOOLEAN`.

Оператор конкатенации. Оператор конкатенации определен для операндов, имеющих перечислимый тип, и одномерных массивов, содержащих элементы перечислимого типа. При этом оба операнда должны быть одного типа. Результат выражения представляет собой одномерный массив, тип элементов которого равен типу операндов. Размер массива равен сумме размеров операндов.

Другие операторы. К операндам перечислимых типов применим оператор указания типа. Данный оператор используется для уточнения типа объекта в случае, если одно и то же название элемента используется различными типами.

Пример:

```
TYPE COLOR IS (green, red, tellow, orange);  
TYPE FRUIT IS (orange, apple, pear);  
VARIABLE VC : COLOR;  
VARIABLE VF : FRUIT;  
VC := COLOR'orange;  
VF := FRUIT'orange;
```

Другие операции над операндами перечислимых типов, определенных пользователем, не определены.

Тип SEVERITY_LEVEL. Переменные этого типа используются только в операторе ASSERT и игнорируются при синтезе логических схем. Переменные типа SEVERITY_LEVEL могут принимать следующие значения: NOTE, WARNING, ERROR и FAILURE.

Тип CHARACTER. Перечислимый тип. Значением объекта данного типа может быть любой символ из набора ASCII (128 первых символов).

Массивы. Массив (тип «массив») является сложным типом. Массив представляет собой упорядоченную структуру однотипных данных. Массив имеет диапазон индексов, который может быть возрастающим либо убывающим. На любой элемент массива можно сослаться, используя его индекс. Несмотря на то что стандартом языка допускается использование массивов любой размерности, для синтеза ПЛИС используются только одномерные (поддерживаются всеми средствами синтеза) и двухмерные (поддерживаются ограниченным числом средств синтеза) массивы. Также можно сослаться на часть одномерного массива, используя вместо индекса диапазон индексов.

Существуют две разновидности типа «массив»: ограниченный (constrained) и неограниченный (unconstrained):

— объявление ограниченного типа определяет границы диапазона индексов (число элементов массива) в каждом измерении при определении типа.

— объявление неограниченного типа не определяет границы диапазона индексов. В этом случае границы диапазона устанавливаются при объявлении конкретного экземпляра объекта данного типа.

Объявление ограниченного типа «массив» имеет вид:

TYPE имя_типа IS

ARRAY (диапазон_индексов [, диапазон_индексов])

OF тип_элемента;

диапазон_индексов может определяться двумя способами:

1) явным заданием границ диапазона

m TO n

n DOWNTO m,

где m, n — целочисленные константы, $m \leq n$;

2) с использованием идентификатора ограниченного подтипа. В этом случае значения границ подтипа являются значениями границ индекса массива. Описание подтипов см. далее.

Объявление неограниченного типа «массив» имеет вид:

```
TYPE имя_типа IS  
    ARRAY (тип_индекса  
           [, тип_индекса])  
    OF тип_элемента;
```

тип_индекса определяется следующим образом:

подтип RANGE \diamond ,

где подтип может быть:

INTEGER – индекс находится в диапазоне $-(2^{31}-1) \dots 2^{31}-1$;

NATURAL – индекс находится в диапазоне $0 \dots 2^{31}-1$;

POSITIVE – индекс находится в диапазоне $1 \dots 2^{31}-1$.

Примеры:

1) Объявление ограниченного массивного типа:

```
TYPE word IS ARRAY (31 DOWNTO 0) OF STD_LOGIC;
```

```
TYPE register_bank IS ARRAY (byte RANGE 0 TO 132) OF INTEGER;
```

```
TYPE memory IS ARRAY (address) OF word; — двумерный массив
```

2) Объявление неограниченного массивного типа:

```
TYPE logic IS ARRAY (INTEGER RANGE  $\diamond$ ) OF BOOLEAN;
```

3) Объявление двухмерного массива:

```
TYPE Reg IS ARRAY (3 DOWNTO 0) OF STD_LOGIC_VECTOR(7  
DOWNTO 0);
```

```
TYPE transform IS ARRAY (1 TO 4, 1 TO 4) OF BIT;
```

Как было сказано, в языке имеется несколько предопределенных типов «массив». Их объявления выглядят следующим образом:

```
TYPE STRING IS ARRAY (POSITIVE RANGE  $\diamond$ ) OF CHARACTER;
```

```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE  $\diamond$ ) OF BIT;
```

```
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE  $\diamond$ ) OF  
STD_LOGIC;
```

```
TYPE STD_ULOGIC_VECTOR IS ARRAY (NATURAL RANGE  $\diamond$ ) OF  
STD_ULOGIC;
```

Объявление объекта типа «неограниченный массив» должно содержать ограничения на индекс. Диапазон изменения индексов может быть ограничен:

- 1) с использованием ключевых слов TO или DOWNTO:

```
TYPE data_memory_type IS ARRAY (INTEGER RANGE <>) OF
BIT;
```

...

```
VARIABLE data_memory : data_memory_type(0 TO 255);
```

- 2) путем использования диапазона начального значения:

```
CONSTANT part_id : STRING := "M38006";
```

Строки, битовые строки и агрегаты. Строки, битовые строки, агрегаты (STRINGS, BIT STRINGS, AGGREGATES) используются для конструирования значений объектов массивных типов. Они могут использоваться в любом месте, где допускается значение типа «массив», например как начальное значение константы или операнд в выражении.

Строковая запись может быть использована для представления значений как объектов некоторых предопределенных типов (STRING, BIT_VECTOR, STD_LOGIC_VECTOR), так и для любого одномерного массива, элементы которого имеют тип CHARACTER; например:

```
TYPE bit6 IS ('U', '0', '1', 'F', 'R', 'X');
```

```
TYPE bit6_data IS ARRAY (POSITIVE RANGE<>) OF bit6;
```

...

```
SIGNAL data_bus : bit6_data(15 DOWNTO 0);
```

.

.

```
data_bus <= "UUUUUUUUFFFFFFFF";
```

VHDL позволяет компактно описывать битовые строки (значение типа BIT_VECTOR или STD_LOGIC_VECTOR) в базе 2, 8 и 16. Например:

```
CONSTANT clear : bit_vector := B"00_101_010";
```

```
CONSTANT empty : bit_vector := O"052";
```

```
CONSTANT null : bit_vector := X"2A";
```

Все три константы имеют одно и то же значение. Отметим, что символы подчеркивания могут использоваться в любом месте битовой строки

для облегчения чтения. Расширенными цифрами (extended digits) для шестнадцатеричного представления являются буквы от A до F, причем могут использоваться как прописные, так и строчные буквы.

Массивные агрегаты используются для присваивания значений объектам типа «массив». Массивные агрегаты формируются при помощи позиционной (positional) записи, поименованной (named) записи или комбинации этих двух форм. Рассмотрим пример.

Предположим, что имеются следующие описания:

```
TYPE ArrayType IS ARRAY (1 TO 4) OF CHARACTER;  
...  
VARIABLE Test : ArrayType,
```

и требуется, чтобы переменная Test содержала элементы 'f', 'o', 'o', 'd' в указанном порядке.

Позиционная запись имеет вид:

```
Test := ('f', 'o', 'o', 'd').
```

Агрегат в данном случае записывается как список значений элементов, разделенных запятыми. Первое значение назначается элементу с наименьшим значением индекса (крайнему левому).

Поименованная запись имеет вид:

```
Test := (1=>'f', 3=>'o', 4=>'d', 2=>'o').
```

В этом случае агрегат также является списком, элементы которого разделены запятыми, однако элементы списка имеют формат:

позиция => значение.

Порядок перечисления элементов при поименованной записи не имеет значения.

Комбинированная запись имеет вид:

```
Test := ('f', 'o', 4=>'d', 3=>'o').
```

В этом случае сначала записываются элементы, присваиваемые с использованием позиционной записи, а оставшиеся элементы присваиваются с использованием поименованной записи.

При формировании агрегата с использованием поименованной (или комбинированной) записи вместо номера позиции можно указывать ключевое слово OTHERS, которое определяет значение для всех элементов, которые еще не были описаны в агрегате. Например:

```
Test := ('f', 4=>'d', OTHERS=>'o').
```

Подтипы. Использование подтипов позволяет объявлять объекты, принимающие ограниченный набор значений из диапазона, допустимого для базового типа.

Подтипы применяются в двух случаях:

1) Подтип может ограничить диапазон значений базового скалярного типа (ограничение по диапазону). В этом случае объявление подтипа выглядит следующим образом:

SUBTYPE имя_подтипа IS имя_базового_типа RANGE диапазон_индексов;

диапазон_индексов определяется следующим образом:

m TO n
n DOWNTO m,

где m, n — целочисленные константы, m ≤ n.

Пример:

Предположим что разработчик желает создать сигнал A типа SEVERITY и что A может принимать значения только OKAY, NOTE и WARNING.

TYPE severity IS (OKAY, NOTE, WARNING, ERROR, FAILURE);

SUBTYPE go_status IS severity RANGE OKAY TO WARNING;

SIGNAL A : go_status;

Базовый тип и ограничение диапазона могут быть включены непосредственно в объявление объекта. Объявление сигнала A, эквивалентное приведенному выше, будет выглядеть следующим образом:

SIGNAL A : severity RANGE OKAY TO WARNING;

Другие примеры:

SUBTYPE pin_count IS INTEGER RANGE 0 TO 400;

SUBTYPE digits IS character range '0' TO '9'.

Подтипы, объявленные таким образом, могут также участвовать в описании ограниченных массивных типов (см. «Массивы»).

2) Подтип может определить границы диапазона индексов для неограниченного (unconstrained) массивного типа. В этом случае объявление подтипа выглядит следующим образом:

SUBTYPE имя_подтипа IS имя_базового_типа (диапазон_индексов [,
диапазон_индексов]);

диапазон_индексов определяется следующим образом:

m TO n
n DOWNTO m,

где m, n — целочисленные константы, $m \leq n$.

Такое использование подтипа может быть удобно при наличии большого числа объектов некоторого типа с одинаковыми ограничениями на индексы.

Пример:

```
TYPE bit6_data IS ARRAY (POSITIVE RANGE <>) OF bit6;  
SUBTYPE data_store IS bit6_data (63 DOWNTO 0);  
SIGNAL A_reg, B_reg, C_reg : data_store;  
VARIABLE temp : data_store.
```

В языке имеются два предопределенных числовых подтипа NATURAL и POSITIVE, которые определены как:

```
SUBTYPE NATURAL IS INTEGER RANGE 0 TO highest_integer;  
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO highest_integer;
```

4.4. Операторы VHDL

4.4.1. Основы синтаксиса

Исходный текст программы на VHDL состоит из последовательностей операторов, записанных с учетом следующих правил:

- каждый оператор — это последовательность слов, содержащих буквы английского алфавита, цифры и знаки пунктуации;
- слова разделяются произвольным количеством пробелов, табуляций и переводов строки;
- операторы разделяются символами «;»;
- в некоторых операторах могут встречаться списки объектов, разделяемые символами «,» или «;».

Комментарии могут быть включены в текст программы с помощью двух подряд идущих символов «—». После появления этих символов весь текст до конца строки считается комментарием.

Для указания системы счисления для констант могут быть применены спецификаторы:

- В — двоичная система счисления, например В"0011";
- О — восьмеричная система счисления, например О"3760";
- Н — шестнадцатеричная система счисления, например Н"F6A0".

4.4.2. Объекты

Объекты являются контейнерами для хранения различных значений в рамках модели. Идентификаторы объектов содержат буквы, цифры и знаки подчеркивания. Идентификаторы должны начинаться с буквы, не должны заканчиваться знаком подчеркивания, и знаки подчеркивания не могут идти подряд. Прописные и строчные буквы в VHDL не различаются. Все объекты должны быть явно объявлены перед использованием, за исключением переменной цикла в операторе FOR, которая объявляется по умолчанию.

Каждый объект характеризуется типом и классом. Типы разделяются на предопределенные в VHDL и определяемые пользователем. Тип показывает, какого рода данные может содержать объект. Класс показывает, что можно сделать с данными, содержащимися в объекте. В VHDL определены следующие классы объектов:

- Constant — константы. Значение константы определяется при ее объявлении и не может быть изменено. Константы могут иметь любой из поддерживаемых типов данных.
- Variable — переменные. Значение, хранимое в переменной, меняется везде, где встречается присваивание данной переменной. Переменные могут иметь любой из поддерживаемых типов данных.
- Signal — сигналы. Сигналы представляют значения, передаваемые по проводам и определяемые присвоением сигналов (отличным от присвоения переменных). Сигналы могут иметь ограниченный набор типов (обычно BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, INTEGER и, возможно, другие, в зависимости от среды разработки).

Повторное использование присваивания сигналов в наборе параллельных операторов не допускается. В наборе последовательных операторов такое присваивание допустимо и даст значение сигнала, соответствующее последнему по порядку присваиванию.

Синтаксис объявления объектов:

```
Constant { name [, name] } : Type [ ( index_range [ , index_range ] ) ] :=
initial_value;
```

```
Variable { name [, name] } : Type [ ( index_range [ , index_range ] ) ] [ :=
initial_value ];
```

```
Signal { name [, name] } : Type [ ( index_range ) ];
```

Диапазон значений индексов задается в виде int_value to int_value или int_value downto int_value.

4.4.3. Атрибуты

Атрибуты (или иначе свойства) определяют характеристики объектов, к которым они относятся. Стандарт VHDL предусматривает как предопределенные, так и определяемые пользователем атрибуты, однако современные инструментальные средства в большинстве своем поддерживают только предопределенные атрибуты. Для обращения к атрибутам объекта используется символ «'» (например A1'left).

В VHDL определены следующие атрибуты:

- 'left — левая граница диапазона индексов массива;
- 'right — правая граница диапазона индексов массива;
- 'low — нижняя граница диапазона индексов массива;
- 'high — верхняя граница диапазона индексов массива;
- 'range — диапазон индексов массива;
- 'reverse-range — обращенный диапазон индексов массива;
- 'length — ширина диапазона индексов массива.

4.4.4. Компоненты

Объявление компоненты определяет интерфейс к модели на VHDL (Entity и Architecture), описанной в другом файле. Обычно объявление компоненты совпадает с соответствующим объявлением Entity. Они могут различаться только значениями по умолчанию. Эти значения используются, когда какой-либо из отводов компоненты остается не присоединенным (ключевое слово Open) при установке компоненты в схему.

Оператор объявления компонента может находиться внутри объявления Architecture или в заголовке пакета (Package). Соответствующие компоненте объявления entity и Architecture не обязательно должны существовать в момент анализа схемы. В момент моделирования или синтеза должны существовать объявления Entity и Architecture для компонент, которые не только объявлены, но и установлены в схему. Это позволяет, например, конструктору задать объявления библиотечных элементов, а реальное их описание (объявления Entity и Architecture) задавать по мере использования этих элементов в конструкции.

Объявление компоненты записывается следующим образом:

```
Component name  
  [ port( port_list ); ]  
end component;
```

4.4.5. Выражения

Выражения могут содержать следующие операторы: преобразование типа AND, OR, NAND, NOR, XOR, =, /=, <, <=, >, >=, +, -, &, *, /, MOD, REM, ABS, NOT.

В зависимости от избранной САПР при синтезе может поддерживаться подмножество приведенных выше операторов. Порядок вычисления выражений определяется приоритетом операторов:

AND, OR, NAND, NOR, XOR — самый низкий приоритет

=, /=, <, <=, >, >=

+, -, & (бинарные)

+, - (унарные)

*, /, MOD, REM

ABS, NOT — высший приоритет

Операторы с более высоким приоритетом выполняются раньше. Чтобы изменить такой порядок, используются скобки.

При моделировании (но не при синтезе) схемы возможно также описание формы сигнала в виде выражения. Записывается оно следующим образом:

```
Value_expression [ after time_expression ]
{ , value_expression [ after time_expression ] }
```

4.4.6. Операторы

С помощью операторов описывается алгоритм, определяющий функционирование схемы. Они могут находиться в теле функции, процедуры или процесса.

WAIT UNTIL condition;

Приостанавливает выполнение процесса, содержащего данный оператор до момента выполнения условия.

Signal <= expression

Оператор присваивания сигнала устанавливает его значение равным выражению справа.

Variable := expression

Оператор присваивания устанавливает значение переменной равным выражению справа.

Procedure_name (parameter {, parameter})

Оператор вызова процедуры состоит из имени процедуры и списка фактических параметров.

If condition then

Sequence_of_statements

{ Elself condition then

Sequence_of_statements }

[else

sequence_of_statements]

end if ;

Оператор IF используется для ветвления алгоритма по различным условиям.

Case expression is

When choices_list => sequence_of_statements;

{ When choices_list => sequence_of_statements; }

When others => sequence_of_statements;

End case;

Оператор CASE, подобно оператору IF, задает ветвление алгоритма. Значения в списках разделяются символом «|». Когда значение выражения встречается в одном из списков значений, выполняется соответствующая последовательность операторов. Если значение выражения не присутствует ни в одном из списков, то выполняется список операторов, соответствующий ветви WHEN OTHERS.

[loop_label :]

for loop_index_variable in range loop

sequence_of_statements

end loop [loop_label] ;

Оператор цикла позволяет многократно выполнять последовательность операторов. Диапазон значений задается в виде value1 to value2 или value1 downto value2. Переменная цикла последовательно принимает значения из

заданного диапазона. Количество итераций равно количеству значений в диапазоне.

RETURN expression ;

Этот оператор возвращает значение из функции.

NULL

Пустой оператор, не выполняет никаких действий.

4.5. Интерфейс и тело объекта

Полное VHDL-описание объекта состоит как минимум из двух отдельных описаний: описание интерфейса объекта и описание тела объекта (описание архитектуры).

Интерфейс описывается в объявлении объекта Entity Declaration и определяет входы и выходы объекта, его входные и выходные порты Ports и параметры настройки Generic. Параметры настройки отражают тот факт, что некоторые объекты могут иметь управляющие входы, с помощью которых может производиться настройка экземпляров объектов, в частности, задаваться временем задержки.

Например у объекта Q1 три входных порта X1, X2, X3 и два выхода Y1, Y2. Описание его интерфейса на VHDL имеет вид:

Entity Q1 is

Port (X1, X2, X3: in real; Y1, Y2: out real);

End Q1.

Порты объекта характеризуются направлением потока информации.

Они могут быть:

- входными (in);
- выходными (out);
- двунаправленными (inout);
- двунаправленными буферными (buffer);
- связными (linkage).

А также имеют тип, характеризующий значения поступающих на них сигналов:

- целый (INTEGER);
- вещественный (REAL);
- битовый (BIT);
- символьный (CHARACTER).

Тело объекта специфицирует его структуру или поведение. Его описание по терминологии VHDL содержится в описании его архитектуры.

VHDL позволяет отождествлять с одним и тем же интерфейсом несколько архитектур. Это связано с тем, что в процессе проектирования происходит проработка архитектуры объекта: переход от структурной схемы к электрической принципиальной, от поведенческого к структурному описанию.

Средства VHDL для отображения структур цифровых систем базируются на представлении о том, что описываемый объект Entity представляет собой структуру из компонент Component, соединяемых друг с другом линиями связи. Каждая компонента, в свою очередь, является объектом и может состоять из компонент низшего уровня (иерархия объектов). Взаимодействуют объекты путем передачи сигналов Signal по линиям связи. Линии связи подключаются к входным и выходным портам компонент. В VHDL сигналы отождествляются с линиями связи.

Имена сигналов и имена линий связи совпадают (они отождествляются). Для сигналов (линий), связывающих компоненты друг с другом, необходимо указывать индивидуальные имена.

Описание структуры объекта строится как описание связей конкретных компонент, каждая из которых имеет имя, тип и карты портов. Карта портов PORT MAP определяет соответствие портов компонент поступающим на них сигналам, можно интерпретировать карту портов как разъем, на который приходят сигналы и в который вставляется объект-компонента.

Принятая в VHDL форма описания связей конкретных компонент имеет следующий вид:

Имя: тип связи (сигнал, порт)

Например, описание связей некоторого объекта Q1, состоящего из трех компонент K1, K2, K3 и сигналов между компонентами X1, X2, X3, S, Y1, Y2 имеет следующий вид:

K1: SM port map (X1, X2, S);

K3: M port map (S, Y1);

K2: SM port map (S, X3, Y2);

Здесь K1, K2, K3 — имена компонент; SM, M — типы компонент; X1, X2, X3, S, Y1, Y2 — имена сигналов, связанных с портами.

Полное VHDL-описание архитектуры STRUCTURA объекта Q1 имеет вид:

```
Architecture STRUCTURA of Q1 is
Component SM port (A, B: in real; C: out real);
End component;
Component M port (E: in real; D: out real);
End component;
Signal S: real;
Begin
K1: SM port map (X1, X2, S);
K3: M port map (S, Y1);
K2: SM port map (S, X3, Y2);
End STRUCTURA;
```

Средства VHDL для отображения поведения описываемых архитектур строятся на представлении их как совокупности параллельно взаимодействующих процессов. Понятие процесса PROCESS относится к базовым понятиям языка VHDL.

Архитектура включает в себя описание одного или нескольких параллельных процессов. Описание процесса состоит из последовательности операторов, отображающих действия по переработке информации. Все операторы внутри процесса выполняются последовательно. Процесс может находиться в одном из двух состояний: либо пассивном, когда процесс ожидает прихода сигналов запуска или наступления соответствующего момента времени, либо активном, когда процесс исполняется. Процессы взаимодействуют путем обмена сигналами.

В общем случае в поведенческом описании состав процессов не обязательно соответствует составу компонент, как это имеет место в структурном описании.

Поведение VHDL-объектов воспроизводится на ЭВМ, и приходится учитывать особенности воспроизведения параллельных процессов на однопроцессорной ЭВМ. Особая роль в синхронизации процессов отводится механизму событийного воспроизведения модельного времени Now.

Когда процесс вырабатывает новое значение сигнала перед его посылкой на линию связи, говорят, что он вырабатывает будущее сообщение

Transaction. С каждой линией связи (сигналом) может быть связано множество будущих сообщений. Множество сообщений для сигнала называется его драйвером Driver. Таким образом, драйвер сигнала — это множество пар время-значение (множество планируемых событий).

VHDL реализует механизм воспроизведения модельного времени, состоящий из циклов. На первой стадии цикла вырабатываются новые значения сигналов. На второй стадии процессы реагируют на изменения сигналов и переходят в активную фазу. Эта стадия завершается, когда все процессы перейдут снова в состояние ожидания. После этого модельное время становится равным времени ближайшего запланированного события, и все повторяется.

Особый случай представляет ситуация, когда в процессах отсутствуют операторы задержки. Для этого в VHDL предусмотрен механизм так называемой дельта-задержки.

В случае дельта-задержек новый цикл моделирования не связан с увеличением модельного времени. В приведенном выше примере новое значение сигнала Y1 вырабатывается через дельта-задержку после изменения сигнала S.

Другая способность VHDL-процессов связана с так называемыми разрешенными (Resolved) сигналами. Если несколько процессов изменяют один и тот же сигнал (сигнал имеет несколько драйверов), то в описании объектов может указываться функция разрешения. Эта функция объединяет значения из разных драйверов и вырабатывает одно. Это позволяет, например, особенности работы нескольких элементов подать на общую шину.

В языке VHDL для наиболее часто используемых видов процессов (процессов межрегистровых передач) введена компактная форма записи.

Полное описание архитектуры POVEDENIE объекта Q1 в этом случае имеет следующий вид:

```
Architecture POVEDENIE of Q1 is
Signal S: real;
Begin
Y1<=S;
Y2<=S+X3 after 10 ns;
S<=X3+X2 after 10 ns;
End POVEDENIE;
```

4.5.1. Описание простого объекта

Для иллюстрации возможностей VHDL рассмотрим пример проектирования простой комбинационной схемы, назовем ее объект F. Объект проекта F имеет два входа A1 и A2 и два выхода B1 и B2.

4.5.2. Объявление объекта проекта F

Entity F is
Port (A1, A2: in BIT; B1, B2: out BIT)

Сигналы принимают значения '1' или '0' в соответствии с таблицей истинности.

Входы		Выходы	
A1	A2	B1	B2
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

4.5.3. Поведенческое описание архитектуры

Вариант описания архитектуры BEHAVIOR объекта F использует условный оператор IF языка VHDL и учитывает, что только при обоих входах A1 и A2, равных '1', выходы B1 = '1' и B2 = '0'. В остальных случаях наоборот B1 = '0' и B2 = '1'.

```
Architecture BEHAVIOR of F is
Begin
Process
Begin
Wait on (A1' A2)
If (A1='1') and (A2='1')
Then B1<='1'; B2<='0';
End if;
End process;
End;
```

В каждом процессе может быть только один оператор WAIT ON. Второй вариант поведенческого описания архитектуры объекта F, назовем его BEHAVIOR_F, использует выбор CASE языка VHDL и учитывает то свой-

ство функции F, что для первых трех строк ее значение не меняется. В заголовке процесса указан список чувствительности процесса PROCESS (A1, A2). Это указание эквивалентно оператору WAIT ON (A1, A2) в начале описания процесса.

```
Architecture BEHAVIOR_F of F is
Begin
Process (A1,A2);
Begin
-&-операция
case (A1& A2) is
-первые три строки таблицы
when "00"/ "01"/ "10"=> B1<='0'; B2<='1'
-последняя строка таблицы
when "11" => B1<='1'; B2<='0'
end case
end process
end BEHAVIOR_F;
```

4.5.4. Потокковая форма

В процессе проектирования объекта F могут быть предложены различные варианты его функциональных схем. Описание архитектуры объекта F может быть таким:

```
Architecture F_A of F is
Begin
-каждому вентилю сопоставлен оператор назначения
сигнала
B1<= A1 and A2;
B2<= not (A1 and A2);
End;
```

Здесь каждому элементу задан процесс, отображающий последовательность преобразования входной информации и передачи ее на выход. Процесс представлен в форме оператора параллельного назначения сигнала. Операторы назначения сигнала (<=) срабатывают параллельно при изменении хотя бы одного из сигналов в своих правых частях.

Другой вариант описания архитектуры F_B. Здесь вентили включены последовательно.

```
Architecture F_B of F is
Signal X: bit
Begin
B2<= not (X);
X <= A1 and A2;
B1 <= X;
End;
```

Промежуточный сигнал X введен в описание архитектуры F_B объекта F потому, что в описании интерфейса объекта F порт B1 объявлен выходным, т. е. с него нельзя считывать сигнал и запись B2 <= not (B1) была бы некорректной.

Сигнал B2 вырабатывается только после изменения сигнала X. Оператор B2 <= not (X) сработает только тогда, когда изменится сигнал X, т. е. после оператора X <= A1 and A2, так как он реагирует только на изменение сигнала в своей правой части. С учетом задержки E1 = 10 нс, а E2 = 5 нс описание архитектуры будет иметь вид:

```
Architecture F_B_TIME of F is
Signal X: bit
Begin
—задержка на B1 - 10 нс.
—задержка на B2 - 5 нс.
B1<=X;
B2<= not (X) after 5 ns;
X<= A1 and A2 after 10 ns;
End;
```

Через 10 нс после изменения одного из входных сигналов (A1 или A2) может измениться выходной сигнал B1, и с задержкой 5 нс после него изменится B2.

4.5.5. Структурное описание архитектуры

Описание архитектуры представляет собой структуру объекта как композицию компонент, соединенных между собой и обменивающихся сиг-

налами. Функции, реализуемые компонентами в явном виде, в отличие от предыдущих примеров в структурном описании не указываются. Структурное описание включает описание интерфейсов компонент, из которых состоит схема, и их связей. Полные (интерфейс + архитектура) описания объектов-компонент должны быть ранее помещены в проектируемую библиотеку, подключенную к структурному описанию архитектуры.

```
Library WORK;
  use WORK.all
—подключение рабочей библиотеки WORK, содержащей
описание объекта соответствующего компоненте INE2
Architecture CXEM_F_C of F is
—ниже интерфейсы компоненты INE2
Component INE2
Port (X1, X2: in bit; Y: out bit);
End component;
—ниже описание связей экземпляров компонент
Signal X: bit;
Begin
E1: INE2 port map (A1, A2, X);
E2: INE2 port map (X, X, B1);
B2<= X;
End;
```

В описании архитектуры CXEM_F_C объекта F сначала указан интерфейс компонент, из которых строится схема. Это компоненты типа INE2 с двумя входными и одним выходным портом. Затем после begin идут операторы конкретизации компонент. Для каждого экземпляра компоненты задано ее имя, карта портов, указывающая соответствие портов экземпляра компоненты поступающим на них сигналам. Например, для компоненты по имени E1 типа INE2 на порт X1 подан сигнал A1, на порт X2 — сигнал A2. Порядок конкретизации безразличен, так как это параллельные операторы. Для того чтобы описание F было полным, в данном случае в рабочей библиотеке проекта WORK необходимо иметь описание интерфейса и архитектуры некоторого объекта, сопоставляемого компоненте INE2. Обозначим этот объект в библиотеке как LA3. Его описание:

```
Entity LA3 is
Port (X, Y: in bit; Z: out bit);
End LA3;
Architecture DF_LA3 of LA3 is
Begin
Z<= not (X and Y) after 10ns;
End;
```

У объекта LA3 может быть несколько архитектур. В примерах дан вариант потокового описания архитектуры DF_LA3 объекта LA3, который содержит оператор назначения сигналу Z инверсного значения конъюнкции сигналов X и Y с задержкой 10 нс.

4.6. Описание конфигурации

Конфигурацию можно рассматривать как аналог списка компонентов для проекта. Оператор конфигурации (идентифицируемый ключевым словом «Configuration»), определяет, какие реализации должны быть использованы, и позволяет изменять связи компонентов в вашем проекте во время моделирования и синтеза.

Конфигурации не являются обязательными, независимо от того, насколько сложен описываемый проект. При отсутствии конфигурации стандарт VHDL определяет набор правил, который обеспечивает конфигурацию по умолчанию; например в случае, когда предусмотрено более одной реализации для блока, последняя скомпилированная реализация получит приоритет и будет связана с объектом.

Обозначение типа компоненты в описании архитектуры CXEM_F_C объекта F и обозначение соответствующего объекта проекта в библиотеке могут не соответствовать друг другу. Связывание обозначений осуществляется в форме объявления конфигурации. Для того чтобы задать информацию о том, что использованная при описании архитектуры CXEM_F_C объекта F компонента INE2 соответствует библиотечному объекту LA3 и варианту его архитектуры под названием DF_LA3, необходимо объявить конфигурацию Configuration. Конфигурация V1 указывает, что из рабочей библиотеки проекта Library WORK для архитектуры CXEM_F_C объекта F для компонент с именами E1 и E2 типа INE2 следует использовать архитектуру DF_LA3 объекта LA3.

```
Library WORK
```

```
—подключается рабочая библиотека проекта  
configuration V1 of F is
```

```
—конфигурация по имени V1 объекта F  
use WORK. all;
```

```
—используются все (all) компоненты библиотеки WORK  
for CXEM_F_C
```

```
—для архитектуры CXEM_F_C
```

```
—компоненты E1, E2 соответствуют объекту LA3  
с архитектурой DF_LA3 из библиотеки WORK
```

```
for E1,E2: INE2
```

```
use entity LA3 (DF_LA3);
```

```
end for;
```

```
end for;
```

```
end V1;
```

4.7. Векторные сигналы и регулярные структуры

Одним из средств повышения компактности описаний цифровых устройств является использование векторных представлений сигналов и операций над ними. Например, пусть некоторый объект FV выполняет ту же функцию, что и объект F, но над 20-разрядными двоичными векторами AV1 и AV2. Его описание определяет порты как двоичные векторы:

```
Entity FV is
```

```
Port (AV1, AV2: in bit_vector (1 to 20);
```

```
      BV1, BV2: out bit_vector (1 to 20));
```

```
End FV1;
```

Поведенческое описание архитектуры FV в потоковой форме использует операции над битовыми векторами.

```
Architecture BECHAV_FV of FV is
```

```
Begin
```

```
BV2<= not (AV1 and AV2);
```

```
BV1<= AV1 and AV2;
```

```
End BECHAV_FV;
```

Структурное описание архитектуры FV для варианта реализации объекта FV как совокупности объектов F, представленное ниже, выполнено с использованием оператора генерации конкретизации. Это позволяет повысить компактность описаний регулярных фрагментов схем.

```
Architecture STRUCT_FV of FV is
Component F port (X1, X2: in bit; Y1, Y2: out bit);
End component;
Begin
— первая компонента конкретизирована обычным
  способом с использованием позиционного
  соответствия сигналов портам
K1: F port map (AV1 (1), AV2 (1), BV1 (1), BV2 (1));
— вторая компонента конкретизирована с
  использованием ключевого способа указания
  соответствия сигналов ее портам
K2: F port map (AV1 ( 2)=>X1, BV1 ( 2)=>Y1, AV2
(2)=>X2, BV2 (2) =>Y2 );
— компоненты K3 - K20 конкретизированы с
  использованием оператора генерации, позволяющего
  компактно описывать регулярные фрагменты схем
for I in 3 to 20 generate
K( I ): F port map ( AV1 ( 1 ), AV2 (1), BV1 (1),
BV2 (1) );
End generate;
End STRUCT_FV;
```

4.8. Задержки сигналов и параметры настройки

Объект с задержкой можно представить состоящим как бы из двух элементов — идеального элемента и элемента задержки. В языке VHDL встроены две модели задержек — инерциальная и транспортная.

Инерциальная модель предполагает, что элемент не реагирует на сигналы, длительность которых меньше порога, равного времени задержки элемента. Транспортная модель лишена этого ограничения.

Инерциальная модель по умолчанию встроена в оператор назначения сигнала языка VHDL. Например, оператор назначения $Y \leq X1 \text{ and } X2 \text{ after}$

10 ns; описывает работу вентиля «2И» и соответствует инерциальной модели. Указание на использование транспортной модели обеспечивается ключевым словом `Transport` в правой части оператора назначения. Например, оператор `YT<=transport X1 and X2 after 10 ns` отображает транспортную модель задержки вентиля.

Задержка может быть задана не константой, а выражением, значение которого может конкретизироваться для каждого экземпляра объекта, используемого как компонента. Для этого ее следует задать как параметр настройки в описании интерфейса объекта. Приведенное ниже описание объекта 12 включает описание интерфейса и тела 12 с инерциальной задержкой, заданной как параметр настройки.

```
Entity 12 is
--параметр настройки T по умолчанию равен 10 нс
Generic (T: time = 10 ns);
Port ( X1, X2: in bit; Y: out bit );
End 12;
Architecture A1_inert of 12 is
Begin
Y<= X1 and X2 after T;
End A1_inert;
Architecture A1_transport of 12 is
Begin
Y<= transport X1and X2 after 10 ns;
End;
```

Ниже представлен вариант описания архитектуры, иллюстрирующей возможность использования параметра настройки (задержка E1 равна 5 нс, E2 — 20 нс) и возможность совмещения структурного и поведенческого описаний в одной архитектуре.

```
Architecture MIX_8_a of F is
Component 12
Generic (T: time);
Port (X1, X2: in bit; Y: out bit );
End component;
Begin
```

```
E1: 12 generic map (5 ns );
  Port map ( A1, A2, B1);
E2: B2<= not ( A1 and A2 ) after 20 ns;
End;
```

Более сложной представляется ситуация, когда необходимо отобразить в описании архитектуры объекта тот факт, что задержки фронта и среза сигналов не совпадают или зависят от путей прохождения сигналов в схеме и ее предыдущего состояния.

Одним из вариантов описания инерциального поведения вентиля «2И» с разными задержками фронта и среза может быть следующим:

```
Architecture INERT of 12 is
Begin
  Process ( X1, X2 );
  Variable Z: bit;
  Begin
    —выход идеального вентиля
    Z=X1 and X2;
    if Z='1' and Z'DELAYED='0' then
    —срез
    Y<='0' after 3 ns
    End if;
  End process;
End;
```

Атрибут Z'DELAYED дает предыдущее значение сигнала.

При описании более сложных ситуаций следует учитывать особенности реализации механизма учета задержек сигналов в VHDL. С сигналом ассоциируется драйвер — множество сообщений о планируемых событиях в форме пар время-значение сигнала.

В случае транспортной задержки, если новое сообщение имеет время, большее, чем все ранее запланированные, оно включается в драйвер последним. В противном случае предварительно уничтожаются все сообщения, запланированные на большее время.

В случае инерциальной задержки также происходит уничтожение всех сообщений, запланированных на большее время. Однако разница в том, что происходит анализ событий, запланированных на меньшее время, и если значение сигнала отличается от нового, то они уничтожаются.

4.9. Атрибуты сигналов и контроль запрещенных ситуаций

Описания систем могут содержать информацию о запрещенных ситуациях, например, недопустимых комбинациях сигналов на входах объектов, рекомендуемых длительностях или частотах импульсов и т. п. Например в вентиле «2И» возникает риск сбоя в ситуациях, когда фронт одного сигнала перекрывает срез другого. Входные сигналы X1 и X2 изменяются в противоположном направлении, и время их перекрытия меньше необходимого, допустим, равного 10 нс.

Средством отображения информации о запрещенных ситуациях в языке VHDL является оператор утверждения (оператор контроля, оператор аномалии) ASSERT. В нем, помимо контролируемого условия, которое не должно быть нарушено, т. е. должно быть истинным, записывается сообщение REPORT о нарушении и уровень серьезности ошибки SEVERITY.

Для приведенного примера в описании архитектуры вентиля «2И» может быть вставлено утверждение о том, что все будет нормально, если внутренний сигнал Z будет равен '1' не менее чем 10 нс, иначе идет сообщение об ошибке. Для этого необходимо, чтобы в момент среза сигнала его задержанное на 10 нс. значение было равно '1'. Если оно равно '0', то длительность сигнала меньше 10 нс. Время предыдущего события в сигнале Z можно получить атрибутом LAST_EVENT.

```
Architecture C1 of 12 is
Signal Z: bit = '0';
Begin
Process ( X1, X2 );
Z<= X1 and X2;
Assert not (Z='0' and not Z'STABLE and Z'DELAYED
(10 ns)= '0')
Report «риск сбоя в 1 в вентиле 12»
Severity warning;
Y<= transport Z after 10 ns;
End C1;
```

Более полное представление о предопределенных атрибутах сигналов можно получить из **Табл. 4.1**. Помимо предопределенных, пользователь может вводить дополнительные атрибуты для сигналов и других типов данных.

Таблица 4.1. Предопределенные атрибуты сигналов

Пример	Тип результата	Пояснения
s'quiet(t)	BOOLEAN	TRUE, если сигнал s пассивен на интервале T
s'transaction	BIT	Инвертируется s каждый раз, когда s активен (изменяется)
s'stable(t)	BOOLEAN	TRUE, если не было событий за интервал T
s'delayed(t)	Signal	Предыдущее значение s в момент NOW-T
s'active	Boolean	TRUE, если сигнал активен
s'last_active	Time	Время, когда сигнал последний раз был активен
s'event	BOOLEAN	TRUE, если происходит событие в s
s'last_value	Signal	Значение сигнала перед последним событием в нем
s'last_event	Time	Время последнего события в s

4.10. Алфавит моделирования и пакеты

Описание пакета VHDL задается ключевым словом `Package` и используется, чтобы собирать часто используемые элементы конструкции для глобального использования в других проектах. Пакет можно рассматривать как общую область хранения, используемую для того, чтобы хранить такие вещи, как описания типов, констант и глобальные подпрограммы. Объекты, определенные в пределах пакета, можно использовать в любом другом проекте на VHDL и можно откомпилировать в библиотеки для дальнейшего повторного использования.

Пакет может состоять из двух основных частей: описания пакета и дополнительного тела пакета. Описание пакета может содержать следующие элементы:

- Объявления типов и подтипов.
- Объявления констант.
- Глобальные описания сигналов.
- Объявления процедур и функций.
- Спецификация атрибутов.

- Объявления файлов.
- Объявления компонентов.
- Объявления псевдонимов.
- Операторы включения.

Пункты, появляющиеся в пределах описания пакета, могут стать видимыми в других проектах с помощью оператора включения.

Если пакет содержит описания подпрограмм (функций или процедур) или определяет одну или более задерживаемых констант (константы, чья величина не задана), то в дополнение к описанию необходимо тело пакета. Тело пакета (которое определяется с использованием комбинации ключевых слов «Package body») должно иметь то же имя, как соответствующее описание пакета, но может располагаться в любом месте проекта (оно не обязано располагаться немедленно после описания пакета).

Отношение между описанием и телом пакета отчасти напоминает отношение между описанием и реализацией элемента (тем не менее, может быть только одно тело пакета для каждого описания пакета). В то время как описание пакета обеспечивает информацию, необходимую для использования элементов, определенных в пределах этого пакета (список параметров для глобальной процедуры или имя определенного типа или подтипа), фактическое поведение таких объектов, как процедуры и функции, должно определяться в пределах тела пакета.

Приведенные выше описания объекта F базировались на стандартных средствах языка VHDL: сигналы представлялись в алфавите '1', '0', логические операции «И», «ИЛИ», «НЕ» также определялись в этом алфавите. Во многих случаях приходится описывать поведение объектов в других алфавитах. Например, в реальных схемах сигнал, кроме значений '1' и '0', может принимать значение высокого импеданса 'Z' (на выходе буферных элементов) и неопределенное значение 'X', например, отражая неизвестное начальное состояние триггеров.

Переход к другим алфавитам осуществляется в VHDL с помощью пакетов. У пакета, как и у объекта проекта, различают объявление интерфейса и объявление тела объекта.

Ниже приводится пример пакета P4 для описания объектов в четырехзначном алфавите представления сигналов ('X', '0', '1', 'Z'), X — значение не определено, Z — высокий импеданс. В этом пакете приводится тип КОНТАКТ для представления сигналов в четырехзначном алфавите, и опреде-

ляются функции NOT и AND над ними. В ТТЛ-логике высокий импеданс на входе воспринимается как '1', что учитывается в таблице функции AND.

Ниже следует объявление пакета P4:

```
Package P4 is
--перечислимый тип
type КОНТАКТ is ('X', '0', '1', 'Z' );
function «NOT» (X: in КОНТАКТ) return КОНТАКТ;
function «AND» (X1, X2: in КОНТАКТ) return КОНТАКТ;
end P4;
```

Ниже следует объявление тела пакета P4:

```
Package body P4 is
Function «NOT» (A: in КОНТАКТ) return КОНТАКТ is
Begin
    If A='X' then return 'X'
    Else if A='1' then return '0'
    Else if A='0' then return '1'
    Else return 'Z'
    End if;
End «NOT»;
Function «AND» ( A1, A2: in КОНТАКТ ) return
КОНТАКТ is
Begin
    If (A1='0' ) or (A2='X' ) then return to '0'
    Else if (A1= 'X' ) or (A2='0' ) or (A2= 'X') and
    (A1='0' ) then return 'X'
    Else return to'1'
    End if;
End «AND»;
End P4;
```

Пример использования пакета P4 при описании объекта F_P4. Этот объект отличается от F, так как у него другой интерфейс.

```
--подключается (use) пакет P4, все его функции (ALL)
use P4.ALL;
```

```
entity F_P4 is
port (A1, A2: in KONTAKT; B1, B2: out KONTAKT)
end F_P4;
Описание архитектуры F_P4_a
Architecture F_P4_a of F is
Begin
B2<= not (A1 and A2);
B1<= A1 and A2;
End F_P4_a;
```

Из этого примера видно, что в ряде случаев изменение алфавита моделирования не требует внесения изменений в описания объектов. Например, переход к семизначному алфавиту ('0', '1', 'X', 'Z', 'F', 'S', 'R'), где тип KONTAKT имеет дополнительные значения: F-фронт, S-срез, R-риск сбоя, потребует только создания нового пакета и подключения его к объявлению объекта F_P4. Изменение в других частях описаний объекта проекта не потребуется.

4.11. Описание монтажного «ИЛИ» и общей шины

В цифровой аппаратуре используются монтажные «ИЛИ» («И») и двунаправленные шины на элементах с тремя состояниями выхода.

Монтажное «ИЛИ»:

```
Signal Y: WIRED_OR bit;
K1: process
Begin
Y<= X1 and X2
End process K1;
K2: process
Y<= X3 and X4;
End process K2;
```

Общая шина на элементах с трехстабильными выходами:

```
Type A3 is ('0', '1', 'Z');
Signal D: CHIN A3;
```

```
C1: process
Begin
  If E1='0' then D<='Z'
Else D<= X1 and X2
End if;
End process C1;
C2: process
  If E2='0' then D<='Z'
Else D<= X3 and X4
End process C2;
```

Если каждой компоненте (K1, K2) схемы задать процесс, то получим два параллельных процесса, каждый из которых вырабатывает свой выходной сигнал. В языке VHDL предусмотрен механизм разрешения конфликтов, возможных в подобных ситуациях, когда сигнал имеет несколько драйверов. Функция разрешения обычно описывается в пакете, а ее имя указывается при описании соответствующего сигнала. Например, тело функции разрешения WIRED_OR (монтажное «ИЛИ») имеет следующий вид:

```
Function WIRED_OR (INPUTS: bit_vector) return bit is
Begin
For I in INPUTS'RANGE loop
If INPUTS( I ) = '1' then
Return '1' ;
End if;
End loop;
Return '0';
End;
```

Драйверы сигнала INPUTS неявно рассматриваются как массив, границы которого определяются атрибутом 'RANGE'. Функция сканирует драйверы сигнала и, если хоть один из них равен '1', возвращает значение '1', иначе '0'.

Функция разрешения SHIN для шины на элементах с тремя состояниями выходов может быть такой:

```
Type A3 is ('0', '1', 'Z');
Type VA3 is array ( integer range <> of A3 );
Function SHIN (signal X: VA3 ) return A3 is
Variable VIXOD: A3:= 'Z';
Begin
For I in X'RANGE loop
  I f X( I) /= 'Z' then
VIXOD:= X ( I);
Exit;
End if;
End loop;
Return VIXOD;
End SHIN;
```

Предполагается, что может быть включен (т. е. не равен 'Z'), только один из драйверов входных сигналов.

4.12. Синтезируемое подмножество VHDL

4.12.1. Общие сведения

Как известно, изначально язык описания аппаратуры VHDL создавался как средство моделирования цифровых систем. Однако его популярность у разработчиков и определенные удобства при использовании привели к тому, что модели на языке VHDL стали средством описания алгоритмов, синтезируемых специальными программными средствами в файлы прошивки (межсоединений) ПЛИС. В то же время для каждого программного продукта набор поддерживаемых конструкций языка различается, и порой довольно существенно. В настоящее время сделана попытка в стандарте IEEE P1076.6 определить синтезируемое подмножество языка VHDL. В этом стандарте определяются те элементы языка, которые могли бы быть синтезированы средствами синтеза (компиляторами) на уровне регистровых передач (register transfer level). В этом случае средства синтеза (Synthesis tools), удовлетворяющие стандарту IEEE P1076.6, могли бы обеспечить подлинную переносимость проекта и возможность единообразного описания. Рассмотрим элементы синтезируемого подмножества языка VHDL.

4.12.2. Переопределенные типы (Redefined types)

Средства синтеза, которые соответствуют стандарту IEEE P1076.6 должны поддерживать следующие переопределенные типы:

- BIT, BOOLEAN и BIT_VECTOR в соответствии со стандартом IEEE Std 1076-1993.
- INTEGER в соответствии со стандартом IEEE Std 1076-1993.
- STD_ULOGIC, STD_ULOGIC_VECTOR, STD_LOGIC и STD_LOGIC_ в соответствии с пакетом STD_LOGIC_1164 (стандарт IEEE Std 1164-1993).
- SIGNED и UNSIGNED в соответствии с пакетом NUMERIC_BIT, являющимся частью стандарта IEEE Std 1076.3-1997.
- SIGNED и UNSIGNED в соответствии с пакетом NUMERIC_STD, являющимся частью стандарта IEEE Std 1076.3-1997.

Кроме того, должна быть обеспечена поддержка средствами синтеза типов, определенных пользователем (User-defined types).

4.12.3. Методика верификации синтезируемого описания (Verification methodology)

Схемы, полученные в результате процедуры синтеза, могут быть как последовательностные (sequential), так и комбинационные (combinational). Как известно, последовательностные схемы содержат некоторые элементы памяти (internal storage), такие как защелки, регистры и т.п. (latch, register, memory), которые учитываются при определении выходного значения сигнала, в то время как выход комбинационной схемы зависит только от входных переменных.

Процесс верификации синтезированного проекта состоит в подаче тестовых воздействий как на поведенческую модель, так и на синтезированную с последующим сравнением результатов их функционирования. Нетрудно понять, что такой подход к процессу проектирования позволяет по заранее обкатанной «чисто поведенческой» модели отработать методику тестирования системы.

Как правило, входной тестовый сигнал должен удовлетворять следующим критериям:

- входные данные не содержат неизвестных величин;
- при верификации последовательностных устройств следует иметь в виду, что триггеры имеют время установления и тестовые сигналы должны иметь необходимую для завершения переходных процессов длительность;

- тактовые сигналы и входные данные должны изменяться спустя достаточное время после асинхронного сброса или установки;
- для синхронных последовательностных устройств первичные входные сигналы должны измениться значительно раньше, чем активный фронт синхроимпульса. Также входные данные должны оставаться неизменными достаточно долго, чтобы быть удержанными относительно активного фронта синхроимпульса;
- аналогичные требования — к времени удержания сигнала и на входах последовательностной схемы, управляемой потенциалами.

Таким образом, входные данные должны оставаться неизменными достаточно долго, чтобы обеспечить требуемое время удержания (hold times) сигнала на входе.

Средства синтеза могут доопределять неопределенные значения, появляющиеся на магистральных выходах в одной модели как эквивалент к логическим значениям в другой модели. По этой причине следует запоминать входной задающий сигнал до того, как проведено сравнение логических значений на выходах обеих моделей.

4.12.3.1. Верификация комбинационных устройств (Combinational verification)

При верификации комбинационной логической схемы после подачи входной задающего тестового сигнала должно пройти некоторое время, прежде чем начать анализ сигнала на выходах схемы. Как правило, это осуществляется в цикле таким образом, что выходы анализируются непосредственно перед подачей следующего набора входных воздействий, т. е. по завершении переходных процессов. Каждая итерация цикла должна включать достаточную задержку, превышающую задержки распространения (transient delays) и паузы (timeout clause delays). Если не придерживаться этих правил, то тестируемое устройство может никогда не достигнуть установившегося состояния (т. е. демонстрировать колебательное поведение), например:

$A \leq \text{not } A \text{ after } 5 \text{ ns};$

4.12.3.2. Верификация последовательностных устройств (Sequential verification)

При верификации последовательностных устройств обычно к входам схемы прикладываются внешние воздействия, после оценивается состояние выходов. Затем процесс повторяется. Но поскольку последовательно-

стное устройство управляется либо фронтом, либо уровнем тактового сигнала, процесс верификации имеет некоторые особенности.

При верификации устройств, тактируемых фронтом (Edge-sensitive models), подача входных данных и проверка выхода тактируются синхросигналом. Проверка выхода должна выполняться до подачи следующего активного фронта тактового сигнала. Таким образом, время переходных процессов в триггерах и период синхросигнала должны соответствовать друг другу.

При верификации устройств, тактируемых уровнем (Level-sensitive models), следует учитывать, что их поведение гораздо сложнее предсказать, чем поведение устройств, тактируемых фронтом, из-за асинхронной природы взаимодействий сигналов. Проверка результатов синтеза зависит от приложения. Общее правило — входные данные должны установиться, пока активен сигнал разрешения, проверка выходов осуществляется при неактивном сигнале разрешения.

4.12.4. Моделирование элементов аппаратуры (Modeling hardware elements)

Рассмотрим особенности описания таких устройств, как тактируемые фронтом последовательностные схемы (Edge-sensitive storage elements), последовательностные схемы, тактируемые уровнем (Level-sensitive storage elements) и схемы с тремя состояниями (Three-state drivers).

4.12.4.1. Синхронные последовательностные схемы (Edge-sensitive sequential logic)

Типы тактового сигнала (Clock signal type). Для задания тактового сигнала могут быть использованы типы BIT, STD_ULOGIC и их подтипы (например STD_LOGIC) с минимальным набором значений '0' и '1'. Рассмотрим пример описания тактового сигнала.

```
signal BUS8: std_logic_vector(7 downto 0);
...
process (BUS8(0))
begin
  if BUS8(0) = '1' and BUS8(0)'EVENT then
    ...
    ...
    — BUS8(0) используется как тактовый сигнал.
```


Определение фронта тактового сигнала. Функция `RISING_EDGE` представляет передний фронт импульса, а функция `FALLING_EDGE` используется для описания заднего фронта тактового импульса. Функции `RISING_EDGE` и `FALLING_EDGE` объявляются в пакете `STD_LOGIC_1164` (стандарт IEEE Std 1164-1993) или `NUMERIC_BIT`, определенном в стандарте IEEE Std 1076.3-1997. Синтаксис определения тактового сигнала имеет вид:

```
clock_edge ::=
RISING_EDGE(clk_signal_name)
| FALLING_EDGE(clk_signal_name)
clock_level and event_expr
event_expr and clock_level
clock_level ::=
clk_signal_name = '0' | clk_signal_name = '1'
event_expr ::=
clk_signal_name'EVENT
| not clk_signal_name'STABLE
```

Передний фронт. Ниже показано задание переднего фронта тактового сигнала для использования в качестве условия в условном операторе (`IF statement`).

```
(positive <clock_edge>):
RISING_EDGE(clk_signal_name)
clk_signal_name'EVENT and clk_signal_name = '1'
clk_signal_name = '1' and clk_signal_name'EVENT
not clk_signal_name'STABLE and clk_signal_name = '1'
clk_signal_name = '1' and not clk_signal_name'STABLE
```

Для использования в качестве условия в операторе `WAIT UNTIL` применяются следующие конструкции для задания переднего фронта:

```
RISING_EDGE(clk_signal_name)
clk_signal_name = '1'
clk_signal_name'EVENT and clk_signal_name = '1'
clk_signal_name = '1' and clk_signal_name'EVENT
```

```
not clk_signal_name'STABLE and clk_signal_name = '1'  
clk_signal_name = '1' and not clk_signal_name'STABLE
```

Задний фронт. Ниже показано задание заднего фронта тактового сигнала для использования в качестве условия в операторе (IF statement).

```
FALLING_EDGE(clk_signal_name)  
clk_signal_name'EVENT and clk_signal_name = '0'  
clk_signal_name = '0' and clk_signal_name'EVENT  
not clk_signal_name'STABLE and clk_signal_name = '0'  
clk_signal_name = '0' and not clk_signal_name'STABLE
```

Для использования в качестве условия в операторе WAIT UNTIL применяются следующие конструкции для задания заднего фронта:

```
FALLING_EDGE(clk_signal_name)  
clk_signal_name = '0'  
clk_signal_name'EVENT and clk_signal_name = '0'  
clk_signal_name = '0' and clk_signal_name'EVENT  
not clk_signal_name'STABLE and clk_signal_name = '0'  
clk_signal_name = '0' and not clk_signal_name'STABLE
```

4.12.4.2. Описание синхронных последовательностных устройств

При синхронном назначении (Synchronous assignment) переменных или сигналов их изменение происходит по фронту тактового импульса (Clock edge).

Сигнал, модифицированный в синхронном назначении, описывает один или несколько синхронных триггеров (Edge-sensitive storage elements). Переменная, модифицированная в синхронном назначении, может быть использована для описания синхронного триггера. В одном процессе может быть использован только один фронт тактового сигнала.

Использование оператора IF. Для описания синхронных последовательностных устройств удобно использовать условный оператор IF, задавая фронт сигнала в качестве условия (см. выше). Приведем типовый шаблон (template) для описания синхронного последовательностного устройства:

```
process (<clock_signal>)
<declarations>
begin
if <clock_edge> then
<sequential_statements>
end if;
end process;
```

Тактовый сигнал в секции <clock_edge> должен задаваться в списке сигналов возбуждения процесса (Process sensitivity list).

Последовательностные операторы, предшествующие или следующие за условным оператором, не поддерживаются средствами синтеза. Например:

```
DFF: process (CLOCK)
    <declarations>
begin
    if CLOCK'EVENT and CLOCK = '1' then
        Q <= D; — тактирование передним фронтом
    end if;
end process;
```

Использование конструкции WAIT. Синхронные последовательностные устройства (Edge-sensitive storage element) могут быть описаны с использованием тактового сигнала (Clock edge) как условия в операторе WAIT UNTIL.

Оператор ожидания (WAIT UNTIL statement) всегда является первым оператором процесса. В каждом процессе может быть только один оператор WAIT UNTIL. Ниже приводится типовой шаблон для создания описания синхронных последовательностных устройств с помощью оператора WAIT.

```
process
    <declarations>
begin
    wait until <clock_edge>;
        — должен быть первым оператором в процес-
        ce
    <sequential_statements>
end process;
```

Поскольку оператор WAIT UNTIL является первым оператором процесса, асинхронные сброс или установка не могут быть описаны с его помощью.

Сигналы, назначенные с помощью условного оператора или оператора выбора, не могут быть использованы для описания синхронных последовательностных устройств.

Приведем пример D-триггера:

```
DFF: process
begin
wait until CLOCK = '0';
Q <= D; — синхронизация задним фронтом
end process;
```

Еще один пример (счетчик):

```
process
variable VAR: UNSIGNED(3 downto 0);
begin
wait until CLOCK = '1';
VAR := VAR + 1;
COUNT <= VAR;
end process;
```

Переменная VAR может описывать четыре триггера, синхронизированных по переднему фронту.

4.12.4.3. Асинхронные сброс и установка (*Asynchronous set/reset*)

В схемах, тактируемых фронтом, возможны асинхронные сброс и установка (*Asynchronous set/reset*). Ниже приводится шаблон асинхронного сброса и установки.

```
process (<clock_signal>, <asynchronous_signals>)
<declarations>
begin
    if <condition1> then
        <sequential_statements>
    elsif <condition2> then
        <sequential_statements>
```

```
    elsif <condition3> then
    ...
    elsif <clock_edge> then
        <sequential_statements>
    end if;
end process;
```

В ветвях оператора IF проверяется условие сброса или установки. Фронт тактового сигнала может появиться только при выполнении условия в операторе ELSIF. Последовательностные операторы (sequential statements) не должны использовать тактовый импульс в условиях операторов IF.

Список сигналов возбуждения процесса (sensitivity list) должен содержать следующие сигналы:

- тактовый сигнал, задаваемый с помощью соответствующего выражения;
- все сигналы, проверяемые в условиях оператора IF, который содержит выражение для фронта синхроимпульса;
- все сигналы, использующиеся в последовательностных операторах, управляемых оператором IF.

Последовательностные операторы, предшествующие или следующие за условным оператором, не поддерживаются средствами синтеза.

Следует помнить, что условия асинхронного сброса и установки не должны содержать выражений для фронта синхросигнала. Асинхронные сброс или установка имеют более высокий приоритет, чем синхросигнал. Очевидно, что не обязательно описывать как сброс, так и установку. В 70% случаев достаточно асинхронного сброса.

Приведем пример описания D-триггера с асинхронной установкой:

```
AS_DFF: process (CLOCK, RESET, SET,
SET_OR_RESET, A)
begin
    if RESET = '1' then
        Q <= '0';
    elsif SET = '1' then
        Q <= '1';
    elsif SET_OR_RESET = '1' then
        Q <= A;
    elsif CLOCK'EVENT and CLOCK = '1' then
```

```
        Q <= D;  
    end if;  
end process;
```

В этом примере сигнал Q сбрасывается по сигналу RESET и устанавливается по сигналу SET, сигнал SET_OR_RESET может либо сбрасывать, либо устанавливать триггер в зависимости от величины A.

4.12.4.4. Последовательностные узлы с потенциальным управлением (*Level-sensitive sequential logic*)

Последовательностные узлы (Storage element), управляемые уровнями, моделируются с помощью сигналов или переменных, при этом внутри процесса не должны присутствовать конструкции, работающие по фронтам импульсов, а также в процессе не должно быть явных назначений сигналов с использованием оператора назначения (Assignment statement). Все сигналы и переменные, участвующие в процессе, должны иметь четко определенные значения. Сигналы или переменные, назначаемые в процессе, не должны использовать фронты синхроимпульса. В списке сигналов возбуждения процесса (process sensitivity list) должны быть отражены все сигналы, участвующие в процессе.

Переменные в подпрограммах никогда не используются при описании последовательностных узлов с потенциальным управлением, потому что переменные в подпрограммах всегда инициализируются на каждом запросе. Назначения сигнала с использованием операторов IF или CASE не следует использовать при описании последовательностных узлов с потенциальным управлением. Рекомендуется избегать стиля моделирования, в котором значение сигнала или переменной читается перед его назначением.

Пример последовательностного узла с потенциальным управлением:

```
LEV_SENS:process (ENABLE, D)  
begin  
    if ENABLE = '1' then  
        Q <= D;  
    end if;  
end process;
```

4.12.4.5. Логика с третьим состоянием и моделирование шин (Three-state and bus modeling)

Трехстабильная шина моделируется, когда сигналу назначается значение 'Z', в соответствии со стандартом IEEE Std 1164-1993. Назначение третьего состояния может быть условным (conditional) или безусловным (unconditional). Для сигналов, имеющих несколько источников-драйверов (drivers), если хотя бы один драйвер имеет третье состояние, то остальные сигналы-драйверы должны иметь по крайней мере одно назначение в 'Z'-состояние.

Следует заметить, что, если объект назначен в 'Z'-состояние в процессе, инициализируемом фронтом или уровнем, средства синтеза могут ввести триггеры на всех входах устройства в третье состояние.

4.12.4.6. Описание комбинационных логических схем (Modeling combinational logic)

Любой процесс, не содержащий тактовых импульсов или оператора ожидания, описывается или с использованием конструкций комбинационной логики, или последовательностных схем, управляемых уровнями.

Если назначение переменной или сигнала происходит во всех возможных вариантах выполнения процесса и все переменные и сигналы имеют определенные значения, то переменная или сигнал описывают комбинационную логику.

Если сигнал или переменная считываются до их изменения, то они могут быть использованы для описания комбинационной логики. Оператор параллельного назначения сигналов (Concurrent signal assignment statements) всегда описывает комбинационную схему.

Список чувствительности (сигналов возбуждения) процесса (process sensitivity list) должен содержать все сигналы, влияющие на процесс.

4.12.5. Директивы компилятора (псевдокомментарии, Pragmas)

Псевдокомментарии влияют на способ синтеза модели. В VHDL приняты следующие псевдокомментарии (Pragmas) — атрибуты (Attributes) и метакомментарии (Metacomments).

4.12.5.1. Атрибуты компилятора (*Attributes*)

Атрибуты компилятора не следует путать с атрибутами сигналов — это «две большие разницы». Первые используются для кодирования представлений перечислимых типов, а атрибуты сигналов определяют свойства сигнала — его изменение и т.п.

Атрибут компилятора ENUM_ENCODING. Атрибут ENUM_ENCODING определяет способ кодирования значений перечислимых типов. В определении этого атрибута задается представление литералов перечислимых типов в форме строки. Строка состоит из символов, разделенных одним или несколькими пробелами. Число символов должно соответствовать числу литералов в перечислимом типе. Каждый символ состоит из последовательности '0' и '1'. Символ '0', очевидно, означает НИЗКИЙ логический уровень, символ '1' — ВЫСОКИЙ.

Рассмотрим пример объявления атрибута:

```
type <enumeration_type> is (<enum_lit1>, <enum_lit2>, ... <enum_litN>);
attribute ENUM_ENCODING: STRING;
```

Спецификация атрибута определяет кодирование для перечисления литералов.

```
attribute ENUM_ENCODING of <enumeration_type>: type is
«<token1><space><token2><space>... <tokenN>»;
```

Символ <token1> задает представление <enum_lit1>, и т.д. Следует помнить, что использование этого атрибута приводит к несоответствиям моделирования, например, при использовании операторов отношения.

Например:

```
attribute ENUM_ENCODING: string;
type COLOR is (RED, GREEN, BLUE, YELLOW, ORANGE);
attribute ENUM_ENCODING of COLOR: type is «10000 01000 00100
00010 00001»;
```

- перечислимый литерал RED представляется как 10000,
- GREEN как 01000, и т.д.

4.12.5.2. Метакommenтaрии (Metacomments)

Метакommenтaрии используются для управления компиляцией. Существуют два типа метакommenтaриев:

- RTL_SYNTHESIS OFF
- RTL_SYNTHESIS ON

Средства синтеза игнорируют любое описание на VHDL после директивы «RTL_SYNTHESIS OFF» и возобновляют процесс синтеза по директиве «RTL_SYNTHESIS ON».

4.12.6. Синтаксис синтезируемого подмножества VHDL

4.12.6.1. Описание интерфейса (Entity declarations)

```
entity_declaration ::=
entity identifier is
entity_header
entity_declarative_part
[ begin
entity_statement_part ]
end [ entity ] [ entity_simple_name ] ;
```

Поддерживаемые средствами синтеза конструкции

- entity_declaration

Игнорируемые при синтезе конструкции

- entity_statement_part

Не поддерживаемые средствами синтеза конструкции

- entity_declarative_part
- Резервированное слово entity после end.

Например:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity E is
generic(DEPTH : Integer := 8);
port (CLOCK : in std_logic;
RESET : in std_logic;
A :in std_logic_vector(7 downto 0);
```

```
B : inout std_logic_vector(7 downto 0);  
C : out std_logic_vector(7 downto 0));  
end E;
```

4.12.6.2. Заголовок интерфейса (Entity header)

```
entity_header ::=  
[ formal_generic_clause ]  
[ formal_port_clause ]  
generic_clause ::= generic(generic_list);  
port_clause ::= port(port_list);
```

Поддерживаемые средствами синтеза конструкции

- entity_header
- generic_clause
- port_clause

4.12.6.3. Настраиваемые типы (Generics)

```
generic_list ::= generic_interface_list
```

Поддерживаемые средствами синтеза конструкции

- generic_list

4.12.6.4. Порты (Ports)

```
port_list ::= port_interface_list
```

Поддерживаемые средствами синтеза конструкции

- port_list

Игнорируемые при синтезе конструкции

- начальные значения в списке портов (port_list)

4.12.6.5. Описание интерфейсной части (Entity declarative part)

```
entity_declarative_part ::=  
{ entity_declarative_item }  
entity_declarative_item ::  
subprogram_declaration  
| subprogram_body  
| type_declaration
```

```
| subtype_declaration  
| constant_declaration  
| signal_declaration  
| shared_variable_declaration  
| file_declaration  
| alias_declaration  
| attribute_declaration  
| attribute_specification  
| disconnection_specification  
| use_clause  
| group_template_declaration  
| group_declaration
```

Не поддерживаемые средствами синтеза конструкции

- entity_declarative_part
- entity_declarative_item

4.12.6.6. Описание операторов интерфейса (Entity statement part)

```
entity_statement_part ::=  
{ entity_statement }  
entity_statement ::=  
concurrent_assertion_statement  
| passive_concurrent_procedure_call  
| passive_process_statement
```

Игнорируемые при синтезе конструкции

- entity_statement_part
- entity_statement

Описание операторов интерфейса (Entity statement part) используется для контроля моделирования. Оно не может задавать сигналы в архитектурном теле и соответственно не оказывает влияние на поведение архитектурного тела.

4.12.6.7. Архитектурные тела (Architecture bodies)

```
architecture_body ::=  
architecture identifier of entity_name is  
architecture_declarative_part  
begin
```

```
[ architecture_statement_part ]  
end [ architecture ] [ architecture_simple_name ] ;
```

Поддерживаемые средствами синтеза конструкции

- architecture_body
- множественные архитектуры (Multiple Architectures)

Не поддерживаемые средствами синтеза конструкции

- глобальные взаимодействия сигналов между синтезируемыми архитектурными телами
- зарезервированное слово architecture после зарезервированного слова end

4.12.6.8. Объявление архитектуры (Architecture declarative part)

```
architecture_declarative_part ::=  
{ block_declarative_item }  
block_declarative_item ::=  
subprogram_declaration  
| subprogram_body  
| type_declaration  
| subtype_declaration  
| constant_declaration  
| signal_declaration  
| shared_variable_declaration  
| file_declaration  
| alias_declaration  
| component_declaration  
| attribute_declaration  
| attribute_specification  
| configuration_specification  
| disconnection_specification  
| use_clause  
| group_template_declaration  
| group_declaration
```

Поддерживаемые конструкции:

- architecture_declarative_part
- block_declarative_item

Игнорируемые при синтезе конструкции

- `alias_declaration`
- `configuration_specification`
- `disconnection_specification`
- определенные пользователем атрибуты

Не поддерживаемые средствами синтеза конструкции

- `shared_variable_declaration`
- `file_declaration`
- `group_template_declaration`
- `group_declaration`

4.12.6.9. Операторы архитектуры (*Architecture statement part*)

```
architecture_statement_part ::=  
{ concurrent_statement }
```

Поддерживаемые средствами синтеза конструкции

- `architecture_statement_part`

4.12.6.10. Объявление конфигурации (*Configuration declaration*)

```
configuration_declaration ::=  
configuration identifier of entity_name is  
configuration_declarative_part  
block_configuration  
end [configuration] [ configuration_simple_name];  
configuration_declarative_part ::=  
{ configuration_declarative_item }  
configuration_declarative_item ::=  
use_clause  
| attribute_specification  
| group_declaration
```

Поддерживаемые средствами синтеза конструкции

- `configuration_declaration`

Не поддерживаемые средствами синтеза конструкции

- `configuration_declarative_part`
- `configuration_declarative_item`

- зарезервированное слово `configuration` после зарезервированного слова `end`

Объявление конфигурации поддерживается для расширения возможностей описания архитектуры в многоуровневых иерархических проектах.

4.12.6.11. Конфигурация блока (*Block configuration*)

```
block_configuration ::=
for block_specification
{ use_clause }
{ configuration_item }
end for ;
block_specification ::=
architecture_name
| block_statement_label
| generate_statement_label [ (index_specification) ]
index_specification ::=
discrete_range
| static_expresion
configuration_item ::=
block_configuration
| component_configuration
```

Поддерживаемые конструкции:

- `block_configuration`
- `block_specification`

Не поддерживаемые средствами синтеза конструкции

- `use_clause`
- `index_specification`
- `configuration_item`
- `block_statement_label`
- `generate_statement_label`

4.12.6.12. Конфигурация компонента (*Component configuration*)

```
component_configuration ::=
for component_specification
[ binding_indication ; ]
[ block_configuration ]
end for ;
```

Не поддерживаемые средствами синтеза конструкции

- component_configuration

4.12.6.13. Объявление подпрограмм (Subprogram declarations)

```
subprogram_declaration ::=  
subprogram_specification ;  
subprogram_specification ::=  
procedure designator [ (formal_parameter_list) ]  
| [ pure | impure ] function designator [ (formal_parameter_list) ]  
return type_mark  
designator ::= identifier | operator_symbol  
operator_symbol ::= string_literal
```

Поддерживаемые средствами синтеза конструкции

- subprogram_declaration
- subprogram_specification
- designator
- operator_symbol

Не поддерживаемые средствами синтеза конструкции

- зарезервированные слова pure и impure

4.12.6.14. Формальные параметры (Formal parameters)

```
formal_parameter_list ::= parameter_interface_list
```

Поддерживаемые средствами синтеза конструкции

- formal_parameter_list

Значения по умолчанию для формальных параметров игнорируются средствами синтеза. Переменные, константы и сигналы как параметры подпрограмм поддерживаются средствами синтеза, в то время как файлы не могут быть параметрами синтезируемых подпрограмм.

4.12.6.15. Тело подпрограммы (Subprogram bodies)

```
subprogram_body ::=  
subprogram_specification is  
subprogram_declarative_part  
begin
```

```
[ subprogram_statement_part ]
end [ subprogram_kind ] [ designator ] ;
subprogram_declarative_part ::=
{ subprogram_declarative_item }
subprogram_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
subprogram_statement_part ::=
{ sequential_statement }
subprogram_kind ::= procedure | function
```

Поддерживаемые средствами синтеза конструкции

- subprogram_body
- subprogram_declarative_part
- subprogram_declarative_item
- subprogram_statement_part

Игнорируемые при синтезе конструкции

- file_declaration
- alias_declaration

Не поддерживаемые средствами синтеза конструкции

- subprogram_kind
- group_template_declaration
- group_declaration

Другие объявления атрибутов игнорируются при синтезе. Рекурсивный вызов подпрограмм поддерживается в том случае, если он статичен.

4.12.6.16. Перегрузка подпрограмм (*Subprogram overloading*)

Оператор перегрузки подпрограмм (Operator overloading) поддерживается средствами синтеза. Сигнатуры (signatures) не синтезируются.

4.12.6.17. Разрешающие функции (*Resolution functions*)

Функции разрешения игнорируются компилятором при синтезе. Функция разрешения RESOLVED поддерживается в подтипе STD_LOGIC.

4.12.6.18. Объявление пакета (*Package declarations*)

```
package_declaration ::=
package identifier is
package_declarative_part
end [ package ] [ package_simple_name ] ;
package_declarative_part ::=
{ package_declarative_item }
package_declarative_item ::=
subprogram_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
```

Поддерживаемые средствами синтеза конструкции

- package_declaration
- package_declarative_part
- package_declarative_item

Игнорируемые при синтезе конструкции

- file_declaration
- alias_declaration
- disconnection_specification
- определенные пользователем объявления атрибутов и их описания

Не поддерживаемые средствами синтеза конструкции

- зарезервированное слово package после end
- shared_variable_declaration
- group_template_declaration
- group_declaration

Объявления сигналов должны содержать его начальное значение.

Кроме того, сигнал, объявленный в пакете, не должен иметь никаких источников. Описание константы должно включать начальное значение.

4.12.6.19. Тело пакета (*Package bodies*)

```
package_body ::=
package body package_simple_name is
package_body_declarative_part
end [ package body ] [ package_simple_name ] ;
package_body_declarative_part ::=
{ package_body_declarative_item }
package_body_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| use_clause
| group_template_declaration
| group_declaration
```

Поддерживаемые конструкции:

- package_body
- package_body_declarative_part

- package_body_declarative_item

Игнорируемые при синтезе конструкции

- alias_declaration
- file_declaration

Не поддерживаемые средствами синтеза конструкции

- shared_variable_declaration
- group_template_declaration
- group_declaration

• зарезервированное слово package body после зарезервированного слова end

4.12.6.20. Скалярные типы (Scalar types)

```
scalar_type_definition ::=
enumeration_type_definition
| integer_type_definition
| physical_type_definition
| floating_type_definition
range_constraint ::= range range
range ::=
range_attribute_name
| simple_expression direction simple_expression
direction ::= to | downto
```

Поддерживаемые средствами синтеза конструкции

- scalar_type_definition
- range_constraint
- range
- direction

Игнорируемые при синтезе конструкции

- physical_type_definition
- floating_type_definition

Пустой диапазон (null ranges) не поддерживается средствами синтеза.

4.12.6.21. Перечислимые типы (Enumeration types)

```
enumeration_type_definition ::=
(enumeration_literal {, enumeration_literal })
enumeration_literal ::= identifier |
character_literal
```

Поддерживаемые средствами синтеза конструкции

- enumeration_type_definition
- enumeration_literal

Элементы типов BIT, BOOLEAN и STD_ULOGIC должны быть определены как однобитные в соответствии со стандартом IEEE Std 1076.3-1997. Средства синтеза могут иметь предопределенные значения и для других перечислимых типов. Пользователь также может переопределить их с использованием атрибута ENUM_ENCODING.

4.12.6.22. *Предопределенные перечислимые типы (Predefined enumeration types)*

Поддерживаемые средствами синтеза конструкции

- character

Игнорируемые при синтезе конструкции

- severity_level

Не поддерживаемые средствами синтеза конструкции

- file_open_kind
- file_open_status

4.12.6.23. *Целочисленные типы (Integer types)*

```
integer_type_definition ::= range_constraint
```

Поддерживаемые средствами синтеза конструкции

- integer_type_definition

Рекомендуется преобразовывать соответствующий целый сигнал или переменную в бит-вектор (vector of bits) соответствующей длины. Если не используются отрицательные числа, то сигнал или переменная должны иметь беззнаковое двоичное представление (unsigned binary representation), в противном случае используется дополнительный код (twos-complement implementation). Естественно, следует стремиться к тому, чтобы размерность вектора была минимально необходимой.

Средства синтеза должны поддерживать положительные, отрицательные целые числа в диапазоне от -2 147 483 648 до +2 147 483 647, что соответствует 32-битной целочисленной арифметике в дополнительном коде. Подтипы NATURAL и POSITIVE поддерживаются средствами синтеза.

4.12.6.24. Физические типы (*Physical types*)

```
physical_type_definition ::=  
range_constraint  
units  
primary_unit_declaration  
{ secondary_unit_declaration }  
end units [ physical_type_simple_name ]  
primary_unit_declaration ::= identifier  
secondary_unit_declaration ::= identifier = physical_literal ;  
physical_literal ::= [ abstract_literal ] unit_name
```

Игнорируемые при синтезе конструкции

- physical_type_definition
- physical_literal

Поддерживается только тип TIME, при этом конструкции типа after игнорируются.

4.12.6.25. Типы с плавающей точкой (*Floating point types*)

```
floating_type_definition ::= range_constraint
```

Игнорируемые при синтезе конструкции

- floating_type_definition

Конструкции, использующие вещественные типы данных, игнорируются компилятором.

4.12.6.26. Смешанные типы (*Composite types*)

```
composite_type_definition ::=  
array_type_definition  
| record_type_definition
```

Поддерживаемые средствами синтеза конструкции

- composite_type_definition

4.12.6.27. Векторные типы (*Array types*)

```
array_type_definition ::=  
unconstrained_array_definition  
| constrained_array_definition
```

```

unconstrained_array_definition ::=
array (index_subtype_definition {,
index_subtype_definition })
of element_subtype_indication
constrained_array_definition ::=
array index_constraint of element_subtype_indica
tion
index_subtype_definition ::= type_mark range <>
index_constraint ::= (discrete_range {,
discrete_range })
discrete_range ::= discrete_subtype_indication |
range
range ::= range_attribute_name |
simple_expression direction simple_expression

```

Поддерживаемые средствами синтеза конструкции

- array_type_definition
- unconstrained_array_definition
- constrained_array_definition
- index_subtype_definition
- index_constraint
- discrete_range

Синтезируются только одномерные массивы. Диапазон векторного типа задается целым числом. Пустой диапазон (null range) не синтезируется. Средства синтеза поддерживают только одно индексное определение подтипа. Выражение для диапазона должно возвратить целочисленные значения.

4.12.6.28. Ограничения индекса и дискретные диапазоны (*Index constraints and discrete ranges*)

Поддерживаются средствами синтеза.

4.12.6.29. Предопределенный вектор (*Predefined array types*)

Поддерживается средствами синтеза.

4.12.6.30. Записи (*Record types*)

```

record_type_definition ::=
record

```

```
element_declaration
{ element_declaration }
end record [ record_type_simple_name ]
element_declaration ::= identifier_list : element_subtype_definition ;
identifier_list ::= identifier {, identifier }
element_subtype_definition ::= subtype_indication
```

Поддерживаемые средствами синтеза конструкции

- record_type_definition
- element_declaration
- identifier_list
- element_subtype_definition

4.12.6.31. Типы доступа (*Access types*)

```
access_type_definition ::= access subtype_indication
```

Игнорируемые при синтезе конструкции

- access_type_definition

Средства синтеза не поддерживают типы доступа.

4.12.6.32. Неполное объявление типов (*Incomplete type declarations*)

```
incomplete_type_declaration ::= type identifier ;
```

Игнорируемые при синтезе конструкции

- incomplete_type_declaration

Неполное объявление типов не поддерживается средствами синтеза.

4.12.6.33. Распределение объектов (*Allocation and deallocation of objects*)

Распределение объектов не поддерживается средствами синтеза.

4.12.6.34. Файловые типы (*File types*)

```
file_type_definition ::= file of type_mark
```

Файловые типы не поддерживаются средствами синтеза.

4.12.6.35. Операции над файлами (*File operations*)

Не поддерживаются средствами синтеза конструкции.

4.12.6.36. Объявления (*declarations*)

```
declaration ::=
type_declaration
| subtype_declaration
| object_declaration
| interface_declaration
| alias_declaration
| attribute_declaration
| component_declaration
| group_template_declaration
| group_declaration
| entity_declaration
| configuration_declaration
| subprogram_declaration
| package_declaration
```

Поддерживаемые средствами синтеза конструкции

- declaration

Игнорируемые при синтезе конструкции

- alias_declaration

Не поддерживаемые средствами синтеза конструкции

- group_template_declaration
- group_declaration

4.12.6.37. Объявление типов (*type declarations*)

```
type_declaration ::=
full_type_declaration
| incomplete_type_declaration
full_type_declaration ::=
type identifier is type_definition ;
type_definition ::=
scalar_type_definition
| composite_type_definition
```



```
| access_type_definition  
| file_type_definition
```

Поддерживаемые средствами синтеза конструкции

- type_declaration
- full_type_declaration
- type_definition

Игнорируемые при синтезе конструкции

- incomplete_type_declaration
- access_type_definition
- file_type_definition

4.12.6.38. Объявление подтипов (subtype declarations)

```
subtype_declaration ::=  
subtype identifier is subtype_indication ;  
subtype_indication ::=  
[ resolution_function_name ] type_mark [ constraint ]  
type_mark ::=  
type_name  
| subtype_name  
constraint ::=  
range_constraint  
| index_constraint
```

Поддерживаемые средствами синтеза конструкции

- subtype_declaration
- subtype_indication
- type_mark
- constraint

Игнорируемые при синтезе конструкции

- определенные пользователем функции разрешения (user-defined resolution functions)

4.12.6.39. Объявление объектов (Object declarations)

```
object_declaration ::=  
constant_declaration  
| signal_declaration
```

```
| variable_declaration
| file_declaration
```

Поддерживаемые средствами синтеза конструкции

- object_declaration

Игнорируемые при синтезе конструкции

- file_declaration

4.12.6.40. Объявление констант (Constant declarations)

```
constant_declaration ::=
constant identifier_list : subtype_indication :=
expression ;
```

Поддерживаемые средствами синтеза конструкции

- constant_declaration

Косвенное объявление констант не поддерживается средствами синтеза, поэтому выражения должны быть включены в объявление константы.

4.12.6.41. Объявление сигналов (Signal declarations)

```
signal_declaration ::=
signal identifier_list : subtype_indication [sig
nal_kind] [:= expression];
signal_kind ::= register | bus
```

Поддерживаемые средствами синтеза конструкции

- signal_declaration

Игнорируемые при синтезе конструкции

- expression

Не поддерживаемые средствами синтеза конструкции

- signal_kind

Начальное значение выражения игнорируется, за исключением объявления пакета (package). Назначение сигнала, объявленного в пакете (package), не поддерживается при синтезе.

4.12.6.42. Объявление переменных (Variable declarations)

```
variable_declaration ::=
```

```
[shared] variable identifier_list :  
subtype_indication [:= expression] ;
```

Поддерживаемые средствами синтеза конструкции

- variable_declaration

Игнорируемые при синтезе конструкции

- expression

Не поддерживаемые средствами синтеза конструкции

- зарезервированное слово shared

Зарезервированное слово shared не поддерживается. Начальные значения игнорируются. Использование объектов доступа также не поддерживается синтезаторами.

4.12.6.43. Объявление файла (File declarations)

```
file_declaration ::=  
file identifier_list : subtype_indication  
[ file_open_information ] ;  
file_open_information ::=  
[ open file_open_kind_expression ]  
is file_logical_name  
file_logical_name ::= string_expression
```

Игнорируемые при синтезе конструкции

- file_declaration

Использование файлов не поддерживается средствами синтеза.

4.12.6.44. Объявление интерфейса (Interface declarations)

```
interface_declaration ::=  
interface_constant_declaration  
| interface_signal_declaration  
| interface_variable_declaration  
| interface_file_declaration  
interface_constant_declaration ::=  
[constant] identifier_list : [in]  
subtype_indication [:= static_expression]  
interface_signal_declaration ::=  
[signal] identifier_list : [mode]  
subtype_indication [bus]
```

```

[:= static_expression]
interface_variable_declaration ::=
[variable] identifier_list : [mode]
subtype_indication
[:= static_expression]
interface_file_declaration ::=
file identifier_list : subtype_indication
mode ::= in | out | inout | buffer | linkage

```

Поддерживаемые средствами синтеза конструкции

- interface_declaration
- interface_constant_declaration
- interface_signal_declaration
- interface_variable_declaration

Игнорируемые при синтезе конструкции

- static_expression

Не поддерживаемые средствами синтеза конструкции

- interface_file_declaration
- режим linkage
- зарезервированное слово bus

4.12.6.45. Список интерфейса (*Interface lists*)

```

interface_list ::=
interface_element {; interface_element}
interface_element ::= interface_declaration

```

Поддерживаемые средствами синтеза конструкции

- interface_list
- interface_element

4.12.6.46. Список ассоциаций (*Association lists*)

```

association_list ::=
association_element {, association_element}
association_element ::=
[formal_part =>] actual_part
formal_part ::=
formal_designator

```

```
| function_name(formal_designator)
| type_mark(formal_designator)
formal_designator ::=
generic_name
| port_name
| parameter_name
actual_part ::=
actual_designator
| function_name(actual_designator)
| type_mark(actual_designator)
actual_designator ::=
expression
| signal_name
| variable_name
| file_name
| open
```

Поддерживаемые средствами синтеза конструкции

- association_list
- association_element
- formal_part
- formal_designator
- actual_part
- actual_designator

Не поддерживаемые средствами синтеза конструкции

- function_name
- type_mark
- file_name

4.12.6.47. Объявление псевдонимов (*Alias declarations*)

```
alias_declaration ::=
alias alias_designator [: subtype_indication] is
name [signature];
alias_designator ::= identifier | character_literal
| operator_symbol
```

Игнорируемые при синтезе конструкции

- alias_declaration
- alias_designator

Не поддерживаемые средствами синтеза конструкции

- signature

Использование псевдонимов не поддерживается средствами синтеза.

4.12.6.48. Объявление атрибутов (*Attribute declarations*)

```
attribute_declaration ::=
attribute identifier : type_mark ;
```

Игнорируемые при синтезе конструкции

- attribute_declaration

Поддерживаются не все атрибуты, определенные в языке.

4.12.6.49. Объявление компонентов (*Component declarations*)

```
component_declaration ::=
component identifier [is]
[local_generic_clause]
[local_port_clause]
end component [component_simple_name];
```

Поддерживаемые средствами синтеза конструкции

- component_declaration

Не поддерживаемые средствами синтеза конструкции

- зарезервированное слово is
- component_simple_name

4.12.6.50. Объявление шаблона группы (*Group template declarations*)

```
group_template_declaration ::=
group identifier is (entity_class_entry_list) ;
entity_class_entry_list ::=
entity_class_entry {, entity_class_entry }
entity_class_entry ::= entity_class [<>]
```

Не поддерживаемые средствами синтеза конструкции

- group_template_declaration
- entity_class_entry_list
- entity_class_entry

4.12.6.51. Объявление группы (Group declarations)

```
group_declarataion ::=
group identifier : group_template_name(group_con
sistent_list);
group_constituent_list ::= group_constituent {,
group_constituent }
group_constituent ::= name | character_literal
```

Не поддерживаемые средствами синтеза конструкции

- group_declaration
- group_constituent_list
- group_constituent

4.12.6.52. Определение атрибутов (Attribute specification)

```
attribute_specification ::=
attribute attribute_designator of entity_specifica
tion is expression;
entity_specification ::=
entity_name_list : entity_class
entity_class ::=
entity| architecture| configuration
| procedure| function| package
| type| subtype| constant
| signal| variable| component
| label| literal| units
| group| file
entity_name_list ::=
entity_designator {, entity_designator}
| others
| all
entity_designator ::= entity_tag [signature]
entity_tag ::= simple_name | character_literal |
operator_symbol
```

Поддерживаемые средствами синтеза конструкции

- attribute_specification
- entity_specification
- entity_class
- entity_name_list
- entity_designator
- entity_tag

Игнорируемые при синтезе конструкции

- объявление атрибутов пользователя

Не поддерживаемые средствами синтеза конструкции

- signature
- интерфейсы group и file
- использование атрибутов пользователя
- зарезервированные слова other и all в списке интерфейса

4.12.6.53. Определение конфигурации (*Configuration specification*)

```
configuration_specification ::=  
for component_specification binding_indication;  
component_specification ::=  
instantiation_list : component_name  
instantiation_list ::=  
instantiation_label {, instantiation_label}  
| others  
| all
```

Игнорируемые при синтезе конструкции

- configuration_specification
- component_specification
- instantiation_list

4.12.6.54. Индикация связей (*Binding indication*)

```
binding_indication ::=  
[ use entity_aspect ]  
[ port_map_aspect ]
```

Игнорируемые при синтезе конструкции

- binding_indication

Не поддерживаемые средствами синтеза конструкции

- generic_map_aspect
- port_map_aspect

4.12.6.55. Индикация связей по умолчанию (Default binding indication)

Поддерживается компилятором.

4.12.6.56. Определение разрыва связи (Disconnection specification)

Не поддерживается при синтезе.

4.12.6.57. Имена (names)

```
name ::=
simple_name
| operator_symbol
| selected_name
| indexed_name
| slice_name
| attribute_name
prefix ::=
name
| function_call
```

Поддерживаемые средствами синтеза конструкции

- name
- prefix

4.12.6.58. Простые имена (Simple names)

```
simple_name ::= identifier;
```

Поддерживаемые средствами синтеза конструкции

- simple_name

4.12.6.59. Выбранные имена (Selected names)

```
selected_name ::= prefix.suffix
suffix ::=
```

```
simple_name
| character_literal
| operator_symbol
| all
```

Поддерживаемые средствами синтеза конструкции

- selected_name
- suffix

4.12.6.60. Индексированные имена (*Indexed names*)

```
indexed_name ::= prefix (expression {, expression })
```

Поддерживаемые средствами синтеза конструкции

- indexed_name

Использование индексированных имен в процедурах не синтезируется.

4.12.6.61. Имена части массива (*Slice names*)

```
slice_name ::= prefix (discrete_range)
```

Поддерживаемые средствами синтеза конструкции

- slice_name

Использование имен части массива (slice name) при неограниченных параметрах не поддерживается. Нулевой диапазон индексов части массива (Null slices) также не может быть синтезирован.

4.12.6.62. Имена атрибутов (*Attribute names*)

```
attribute_name ::=
prefix [signature] 'attribute_designator [ (expression) ]
attribute_designator ::= attribute_simple_name
```

Поддерживаются следующие идентификаторы атрибутов:

- 'base
- 'left
- 'right
- 'high
- 'low
- 'range

- 'reverse_range
- 'length
- 'event
- 'stable

Поддерживаемые средствами синтеза конструкции

- attribute_name
- attribute_designator

Не поддерживаемые средствами синтеза конструкции

- signature
- expression

4.12.6.63. Выражения (Expressions)

```
expression ::=
relation { and relation }
| relation { or relation }
| relation { xor relation }
| relation [ nand relation ]
| relation [ nor relation ]
| relation { xnor relation }
relation ::=
shift_expression [ relational_operator
shift_expression ]
shift_expression ::=
simple_expression [ shift_operator simple_expres
sion ]
simple_expression ::=
[ sign ] term { adding_operator term }
term ::=
factor { multiplying_operator factor }
factor ::=
primary [ ** primary ]
| abs primary
| not primary
primary ::=
name
| literal
| aggregate
```

```
| function_call  
| qualified_expression  
| type_conversion  
| allocator  
| (expression):
```

Поддерживаемые средствами синтеза конструкции

- expression
- relation
- shift_expression
- simple_expression
- term
- factor
- primary

Не поддерживаемые средствами синтеза конструкции

- оператор xnor
- все операторы сдвига

4.12.6.64. Операторы (operators)

```
logical_operator ::= and | or | nand | nor | xor |  
xnor  
relational_operator ::= = | /= | < | <= | > | >=  
shift_operator ::= sll | srl | sla | sra | rol | ror  
adding_operator ::= + | - | &  
sign ::= + | -multiplying_  
operator ::= * | / | mod | rem  
miscellaneous_operator ::= ** | abs | not
```

Поддерживаемые средствами синтеза конструкции

- logical_operator
- relational_operator
- adding_operator
- sign
- multiplying_operator
- miscellaneous_operator

Не поддерживаемые средствами синтеза конструкции

- оператор xnor
- операторы сдвига (SHIFT operator)

4.12.6.65. Логические операторы (*LOGICAL operators*)

Не поддерживаемые средствами синтеза конструкции

- оператор `xnor`

4.12.6.66. Операторы отношения (*RELATIONAL operators*)

Поддерживаются без ограничения всеми средствами синтеза.

Использование логических операторов для перечислимых типов возможно в случае применения атрибута компилятора `ENUM_ENCODING`.

4.12.6.67. Операторы сдвига (*SHIFT operators*)

Операторы сдвига не поддерживаются средствами синтеза.

4.12.6.68. Операторы сложения (*ADDING operators*)

Поддерживаются без ограничения всеми средствами синтеза.

4.12.6.69. Операторы смены знака (*SIGN operators*)

Поддерживаются без ограничения всеми средствами синтеза.

4.12.6.70. Операторы умножения (*MULTIPLYING operators*)

Поддерживаемые средствами синтеза конструкции

- `*` (multiply)
- `/` (division), `mod`, and `rem` operators
- все операторы умножения, определенные в стандарте IEEE Std 1076.3-1997.

Деление (division), остаток (`mod`) и оператор (`rem`) не поддерживаются в том случае, если оба операнда статические. Вообще говоря, результат синтеза может быть ужасен — используйте быстрые умножители.

4.12.6.71. Остальные операторы (*Miscellaneous operators*)

Поддерживаемые средствами синтеза конструкции

- `**` — возведение в степень
- `abs` — модуль.

Возведение в степень `**` не поддерживается в том случае, если оба операнда статические.

4.12.6.72. Литералы (*Literals*)

```
literal ::=
```

```
numeric_literal  
| enumeration_literal  
| string_literal  
| bit_string_literal  
| null  
numeric_literal ::=  
abstract_literal  
| physical_literal
```

Поддерживаемые средствами синтеза конструкции

- literal
- numeric_literal

Не поддерживаемые средствами синтеза конструкции

- null

Физические литералы в типе Time и литералы с плавающей точкой (Floating point literals) могут быть использованы только в конструкции after, которая игнорируется средствами синтеза.

4.12.6.73. Агрегаты (Aggregates)

```
aggregate ::=  
(element_association {, element_association})  
element_association ::=  
[ choices => ] expression  
choices ::= choice { | choice }  
choice ::=  
simple_expression  
| discrete_range  
| element_simple_name  
| others
```

Поддерживаемые средствами синтеза конструкции

- aggregate
- element_association
- choices
- choice

Пример:

```
subtype Src_Typ is Integer range 7 downto 4;
subtype Dest_Typ is Integer range 3 downto 0;
— Constant definition with aggregates
constant Data_c : Std_Logic_Vector(7 downto 0) :=
  (Src_Typ => '1', Dest_Typ => '0');
```

Объединения записей (Record aggregates) не поддерживаются средствами синтеза. Объединения массивов (векторов, Array aggregates) поддерживаются без ограничения всеми средствами синтеза.

4.12.6.74. Обращение к функции (Function calls)

```
function_call ::=
function_name [ (actual_parameter_part) ]
actual_parameter_part ::= parameter_association_list
```

Поддерживаемые средствами синтеза конструкции

- function_call
- actual_parameter_part

4.12.6.75. Составные выражения (Qualified expressions)

```
qualified_expression ::=
type_mark' (expression)
| type_mark' aggregate
```

Поддерживаются средствами синтеза.

4.12.6.76. Преобразование типов (type conversions)

```
type_conversion ::= type_mark(expression)
```

Преобразование типов поддерживается средствами синтеза.

4.12.6.77. Распределение ресурсов (Allocators)

```
allocator ::=
new subtype_indication
| new qualified_expression
```

Распределение ресурсов не поддерживается средствами синтеза.

4.12.6.78. Статические выражения (*Static expressions*)

Локальные и глобальные статические выражения поддерживаются средствами синтеза. Выражения с плавающей точкой (*Floating-point expressions*) не поддерживаются синтезатором. Бесконечная точность (*Infinite-precision expressions*) недопустима. Максимальная точность представления — 32 бита.

4.12.6.79. Последовательностные операторы (*Sequential statements*)

```
sequence_of_statements ::=
{ sequential_statement }
sequential_statement ::=
wait_statement
| assertion_statement
| report_statement
| signal_assignment_statement
| variable_assignment
| procedure_call_statement
| if_statement
| case_statement
| loop_statement
| next_statement
| exit_statement
| return_statement
| null_statement
```

Поддерживаемые средствами синтеза конструкции

- sequence_of_statements
- sequential_statement

4.12.6.80. Оператор ожидания (*WAIT statement*)

```
wait_statement ::=
[label:] wait [sensitivity_clause]
[condition_clause] [timeout_clause];
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name {, signal_name}
```



```
condition_clause ::= until condition
condition ::= boolean_expression
timeout_clause ::= for time_expression
```

Поддерживаемые средствами синтеза конструкции

- wait_statement
- condition_clause
- condition

Игнорируемые при синтезе конструкции

- timeout_clause
- sensitivity_list

Не поддерживаемые средствами синтеза конструкции

- label
- sensitivity_clause

В каждом процессе может быть не более одного оператора WAIT UNTIL, который всегда является первым оператором процесса.

4.12.6.81. Оператор контроля (*ASSERTION statement*)

```
assertion_statement ::= [ label: ] assertion ;
assertion ::=
assert condition
[ report expression ]
[ severity expression ]
```

Игнорируемые при синтезе конструкции

- assertion_statement
- assertion

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.82. Оператор отчета (*REPORT statement*)

```
report_statement ::=
[label:] report expression
[severity expression] ;
```

Оператор отчета относится к конструкциям, не поддерживаемым средствами синтеза.

4.12.6.83. Оператор назначения сигнала (*Signal assignment statement*)

```
signal_assignment_statement ::=
[ label: ] target <= [ delay_mechanism ] waveform ;
delay_mechanism ::= transport
| [ reject time_expression ] inertial
target ::= name
| aggregate
waveform ::= waveform_element {, waveform_element}
| unaffected
```

Поддерживаемые средствами синтеза конструкции

- signal_assignment_statement
- target
- waveform

Игнорируемые при синтезе конструкции

- delay_mechanism

Не поддерживаемые средствами синтеза конструкции

- метки
- зарезервированные слова reject, inertial и unaffected
- time_expression
- множественные элементы waveform_elements

Не следует забывать, что назначение сигнала в пакете не синтезируется.

4.12.6.84. Модификация выходного сигнала (*Updating a projected output waveform*)

```
waveform_element ::=
value_expression [after time_expression]
| null [after time_expression]
```

Поддерживаемые средствами синтеза конструкции

- waveform_element

Игнорируемые при синтезе конструкции

- time_expression после зарезервированного слова after

Не поддерживаемые средствами синтеза конструкции

- null_waveform_elements

4.12.6.85. Оператор назначения переменной (*Variable assignment statement*)

```
variable_assignment_statement ::=  
[ label: ] target := expression ;
```

Поддерживаемые средствами синтеза конструкции

- variable_assignment_statement

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.86. Назначение векторных переменных (*Array variable assignments*)

Назначение векторных переменных поддерживается при синтезе.

4.12.6.87. Оператор вызова процедуры (*Procedure call statement*)

```
procedure_call_statement ::= [ label: ] procedure_call ;  
procedure_call ::= procedure_name [ (actual_parameter_part) ]
```

Поддерживаемые средствами синтеза конструкции

- procedure_call_statement
- procedure_call

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.88. Оператор IF (*IF statement*)

```
if_statement ::=  
[ if_label: ]  
if condition then  
sequence_of_statements  
{ elsif condition then  
sequence_of_statements }  
[ else  
sequence_of_statements ]  
end if [ if_label ] ;
```

Поддерживаемые средствами синтеза конструкции

- if_statement

Не поддерживаемые средствами синтеза конструкции

- if_label

4.12.6.89. Оператор выбора CASE (CASE statement)

```
case_statement ::=
[ case_label: ]
case expression is
case_statement_alternative
{ case_statement_alternative }
end case [ case_label ] ;
case_statement_alternative ::=
when choices =>
sequence_of_statements
```

Поддерживаемые средствами синтеза конструкции

- case_statement
- case_statement_alternative

Не поддерживаемые средствами синтеза конструкции

- label

Неопределенные величины компилятор может синтезировать наиболее простым способом.

4.12.6.90. Операторы цикла (LOOP statement)

```
loop_statement ::=
[ loop_label: ]
[ iteration_scheme ] loop
sequence_of_statements
end loop [ loop_label ] ;
iteration_scheme ::=
while condition
| for loop_parameter_specification
parameter_specification ::=
identifier in discrete_range
discrete_range ::= discrete_subtype_indication |
range
```

Поддерживаемые средствами синтеза конструкции

- loop_statement
- iteration_scheme
- parameter_specification
- discrete_range

Не поддерживаемые средствами синтеза конструкции

- while

В определении параметра цикла границы дискретного диапазона должны быть определены прямо или косвенно как статические значения, принадлежащие к целочисленному типу.

4.12.6.91. Оператор NEXT (NEXT statement)

```
next_statement ::=
[ label: ] next [ loop_label ] [ when condition ] ;
```

Поддерживаемые средствами синтеза конструкции

- next_statement

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.92. Оператор EXIT (EXIT statement)

```
exit_statement ::=
[ label: ] exit [ loop_label ] [ when condition ] ;
```

Поддерживаемые средствами синтеза конструкции

- exit_statement

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.93. Оператор возврата (RETURN statement)

```
return_statement ::=
[ label: ] return [ expression ] ;
```

Поддерживаемые средствами синтеза конструкции

- return_statement

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.94. Пустой оператор (NULL statement)

```
null_statement ::=
[ label: ] null ;
```

Поддерживаемые средствами синтеза конструкции

- null_statement

Не поддерживаемые средствами синтеза конструкции

- label

4.12.6.95. Параллельные операторы (Concurrent statements)

```
concurrent_statement ::=
block_statement
| process_statement
| concurrent_procedure_call_statement
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement
```

Параллельные операторы поддерживаются средствами синтеза.

4.12.6.96. Оператор блока (BLOCK statement)

```
block_statement ::=
block_label:
block [ (guard_expression) ] [ is ]
block_header
block_declarative_part
begin
block_statement_part
end block [ block_label ] ;
block_header ::=
[ generic_clause
[ generic_map_clause ;] ]
[ port_clause
[ port_map_clause ;] ]
block_declarative_part ::=
{ block_declarative_item }
```

```
block_statement_part ::=
{ concurrent_statement }
```

Поддерживаемые средствами синтеза конструкции

- block_statement
- block_declarative_part
- block_statement_part

Не поддерживаемые средствами синтеза конструкции

- block_header
- guard_expression
- зарезервированное слово is

4.12.6.97. Операторы процесса (*PROCESS statement*)

```
process_statement ::=
[ process_label: ]
[ postponed ] process [ (sensitivity_list) ] [ is ]
process_declarative_part
begin
process_statement_part
end process [ process_label] ;
process_declarative_part ::=
{ process_declarative_item }
process_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
process_statement_part ::=
{ sequential_statement }
```

Поддерживаемые средствами синтеза конструкции

- process_statement
- sensitivity_list
- process_declarative_part
- process_declarative_item
- process_statement_part

Игнорируемые при синтезе конструкции

- file_declaration
- alias_declaration
- предопределенное пользователем объявление

Не поддерживаемые средствами синтеза конструкции

- зарезервированные слова postponed и is
- group_template_declaration
- group_declaration

Список чувствительности (sensitivity list) должен включать те сигналы или элементы сигналов, которые читаются процессом, за исключением сигналов, доступных только для чтения по фронту синхроимпульса. Использование файлов, объектов доступа (переменных доступа) и условных названий (псевдонимов) в процессе не поддерживается компиляторами.

4.12.6.98. Параллельный вызов процедур (Concurrent procedure call statement)

```
concurrent_procedure_call_statement ::=  
[ label: ] [ postponed ] procedure_call ;
```

Поддерживаемые средствами синтеза конструкции

- concurrent_procedure_call_statement

Не поддерживаемые средствами синтеза конструкции

- зарезервированное слово postponed

4.12.6.99. Параллельный контроль (Concurrent assertion statement)

```
concurrent_assertion_statement ::=  
[ label: ] [ postponed ] assertion ;
```

Не поддерживается средствами синтеза.

4.12.6.100. Параллельное назначение сигналов (*Concurrent signal assignment statement*)

```
concurrent_signal_assignment_statement ::=  
[ label: ] [ postponed ] conditional_signal_assignment  
| [ label: ] [ postponed ] selected_signal_assignment  
options ::= [ guarded ] [delay_mechanism]
```

Поддерживаемые средствами синтеза конструкции

- concurrent_signal_assignment_statement

Игнорируемые при синтезе конструкции

- options

Не поддерживаемые средствами синтеза конструкции

- зарезервированное слово postponed and guarded

Конструкция after игнорируется. Многократные элементы сигналов не поддерживаются. Значение unaffected не поддерживается. Определение фронта тактового импульса (<clock_edge> or <clock_level>) недопустимо при параллельном назначении сигналов. *Например:*

```
architecture A of E is  
begin  
B(7) <= A(6);  
B(3 downto 0) <= A(7 downto 4);  
C <= not A;  
end A;
```

4.12.6.101. Условное назначение сигналов (*Conditional signal assignment*)

```
conditional_signal_assignment ::=  
target <= options conditional_waveforms ;  
conditional_waveforms ::=  
{ waveform when condition else }  
waveform [ when condition ]
```

Поддерживаемые средствами синтеза конструкции

- conditional_signal_assignment
- conditional_waveforms

Игнорируемые при синтезе конструкции

- options

Не поддерживаемые средствами синтеза конструкции

- last when condition

Условное назначение сигналов содержит ссылку на один или несколько элементов целевого сигнала (target signal). Пример условного назначения сигнала:

```
architecture A of E is
begin
C <= B when A(0) = '1' else
not B when A(1) = '1' else
<00000000> when A(2) = '1' and RESET = '1' else
(others => ('1'));
end A;
```

4.12.6.102. Выборочное назначение сигналов (Selected signal assignments)

```
selected_signal_assignment ::=
with expression select
target <= options selected_waveforms ;
select_waveforms ::=
{ waveform when choices, }
waveform when choices
```

Поддерживаемые средствами синтеза конструкции

- selected_signal_assignment
- select_waveforms

Игнорируемые при синтезе конструкции

- options

При выборочном назначении сигнала выбранный сигнал содержит ссылку на один или несколько целевых сигналов.

Например:

```
architecture A of E is
begin
with A select
```

```
C <= B when "00000000",
not B when "10101010",
(others => ('1')) when "11110001",
not A when others;
end A;
```

4.12.6.103. Оператор реализации компонента (*Component instantiation statement*)

```
component_instantiation_statement ::=
instantiation_label:
instantiated_unit
[ generic_map_aspect ]
[ port_map_aspect ] ;
instantiated_unit ::=
[component] component_name
| entity entity_name [(architecture_name)]
| configuration configuration_name
```

Поддерживаемые средствами синтеза конструкции

- component_instantiation_statement
- instantiated_unit

Не поддерживаемые средствами синтеза конструкции

- формы entity и configuration
- зарезервированное слово component

4.12.6.104. Задание экземпляра компоненты (*Instantiation of a component*)

Полностью поддерживается средствами синтеза.

4.12.6.105. Экземпляр интерфейса проекта (*Instantiation of a design entity*)

Не поддерживается средствами синтеза.

4.12.6.106. Оператор GENERATE (*GENERATE statement*)

```
generate_statement ::=
generate_label:
generation_scheme generate
```

```
[ { block_declarative_item }  
begin ]  
{ concurrent_statement }  
end generate [ generate_label ] ;  
generation_scheme ::=  
for generate_parameter_specification  
| if condition  
label ::= identifier
```

Поддерживаемые средствами синтеза конструкции

- generate_statement
- generate_scheme
- label

Не поддерживаемые средствами синтеза конструкции

- block_declarative_item (the declarative region)
- зарезервированное слово begin

4.12.6.107. Область объявления (Declarative region)

Поддерживается средствами синтеза.

4.12.6.108. Контекст объявлений (Scope of declarations)

Поддерживается средствами синтеза.

4.12.6.109. Видимость (Visibility)

Правила видимости переменных поддерживаются средствами синтеза.

4.12.6.110. Использование предложений (Use clause)

```
use_clause ::=  
use selected_name {, selected_name} ;
```

Поддерживаемые средствами синтеза конструкции

- use_clause

4.12.6.111. Контекст перезагруженного разрешения (The context of overloaded resolution)

Поддерживается средствами синтеза.

4.12.6.112. Части проекта и их анализ *(Design units and their analysis)*

```
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
library_unit ::=
primary_unit
| secondary_unit
primary_unit ::=
entity_declaration
| configuration_declaration
| package_declaration
secondary_unit ::=
architecture_body
| package_body
```

Поддерживаемые средствами синтеза конструкции

- design_file
- design_unit
- library_unit
- primary_unit
- secondary_unit

4.12.6.113. Библиотеки проекта (Design libraries)

```
library_clause ::= library logical_name_list ;
logical_name_list ::= logical_name {, logical_name}
logical_name ::= identifier
```

Поддерживаемые средствами синтеза конструкции

- library_clause
- logical_name_list
- logical_name

4.12.6.114. Контекстные предложения (Context clauses)

```
context_clause ::= { context_item }
context_item ::=
library_clause
| use_clause
```

Поддерживаемые средствами синтеза конструкции

- context_clause
- context_item

4.12.6.115. *Предопределенные атрибуты (Predefined attributes)*

Атрибуты с префиксом t (Attributes whose prefix is a type t)

t'base

t'left

t'right

t'high

t'low

t'ascending

t'image

t'value(x)

t'pos(x)

t'val(x)

t'succ(x)

t'pred(x)

t'leftof(x)

t'rightof(x)

Атрибуты массивов (Attributes whose prefix is an array object a)

a'left[(n)]

a'right[(n)]

a'high[(n)]

a'low[(n)]

a'range[(n)]

a'reverse_range[(n)]

a'length[(n)]

a'ascending[(n)]

Атрибуты с префиксом сигнала s (Attributes whose prefix is a signal s)

s'delayed[(t)]

s'stable[(t)]

s'quiet

s'transaction

s'event

s'ective

s'last_event

s'last_active

s'last_value

s'driving

s'driving_value

Атрибуты именованного объекта с префиксом e (Attributes whose prefix is a named object e)

e'simple_name

e'instance_name

e'path_name

4.12.6.116. Пакет STANDARD

Поддерживаемые средствами синтеза конструкции

- функции одного или нескольких аргументов типа CHARACTER
- функции одного или нескольких аргументов типа STRING
- все функции аргументов типа BOOLEAN
- все функции аргументов типа BIT
- все функции аргументов типа BIT_VECTOR

Игнорируемые при синтезе конструкции

- Атрибут 'FOREIGN

Не поддерживаемые средствами синтеза конструкции

- функции одного или нескольких аргументов типа SEVERITY_LEVEL
- функции одного или нескольких аргументов типа Time
- функция NOW
- функции одного или нескольких аргументов типа FILE_OPEN_KIND
- функции одного или нескольких аргументов типа FILE_OPEN_STATUS

4.12.6.117. Пакет TEXTIO

Подпрограммы, определенные в пакете TEXTIO, не поддерживаются средствами синтеза.

4.13. Краткое описание синтаксиса синтезируемого подмножества VHDL

```
abstract_literal ::= decimal_literal | based_literal
access_type_definition ::= access subtype_indication
actual_designator ::=
expression
```

```

| signal_name
| variable_name
| file_name
| open
actual_parameter_part ::= parameter_association_list
actual_part ::=
actual_designator
| function_name(actual_designator)
| type_mark(actual_designator)
adding_operator ::= + | - | &
aggregate ::=
(element_association {, element_association})
alias_declaration ::=
alias alias_designator [: subtype_indication] is
name [signature];
alias_designator ::= identifier | character_literal
| operator_symbol
allocator ::=
new subtype_indication
| new qualified_expression
architecture_body ::=
architecture identifier of entity_name is
architecture_declarative_part
begin
architecture_statement_part ]
end [ architecture ] [ architecture_simple_name ] ;
architecture_declarative_part ::=
{ block_declarative_item }
architecture_statement_part ::=
{ concurrent_statement }
array_type_definition ::=
unconstrained_array_definition
| constrained_array_definition
assertion ::=
assert condition
[ report expression ]
[ severity expression ]
assertion_statement ::= [ label: ] assertion ;

```



```
association_element ::=
[formal_part =>] actual_part
association_list ::=
association_element {, association_element}
attribute_declaration ::=
attribute identifier : type_mark ;
attribute_designator ::= attribute_simple_name
attribute_name ::=
prefix [signature]'attribute_designator [ (expres
sion) ]
attribute_specification ::=
attribute attribute_designator of entity_specifica
tion is expression;
base ::= integer
base_specifier ::= B | O | X
base_unit_declaration ::= identifier ;
based_integer ::=
extended_digit { [ underline ] extended_digit }
based_literal ::=
base # based_integer [. based_integer ] # [ expo
nent ]
basic_character ::=
basic_graphic_character | format_effector
basic_graphic_character ::=
upper_case_letter | digit | special_character |
space_character
basic_identifier ::=
letter { [ underline ] letter_or_digit }
binding_indication ::=
[ use entity_aspect ]
[ generic_map_aspect ]
[ port_map_aspect ]
bit_string_literal :: base_specifier « [ bit_value ] «
bit_value ::= extended_digit { [ underline ]
extended_digit }
block_configuration ::=
for block_specification
{ use_clause }
```

```

{ configuration_item }
end for ;
block_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| configuration_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
block_declarative_part ::=
{ block_declarative_item }
block_header ::=
[ generic_clause
[ generic_map_clause ;] ]
[ port_clause
[ port_map_clause ;] ]
block_specification ::=
architecture_name
| block_statement_label
| generate_statement_label [ (index_specification) ]
block_statement ::=
block_label:
block [ (guard_expression) ] [ is ]
block_header
block_declarative_part
begin
block_statement_part

```

```
end block [ block_label ] ;
block_statement_part ::=
{ concurrent_statement }
case_statement ::=
[ case_label: ]
case expression is
case_statement_alternative
{ case_statement_alternative }
end case [ case_label ] ;
case_statement_alternative ::=
when choices =>
sequence_of_statements
character_literal ::= ' graphic_character '
choice ::=
simple_expression
| discrete_range
| element_simple_name
| others
choices ::= choice { | choice }
component_configuration ::=
for component_specification
[ binding_indication ; ]
[ block_configuration ]
end for ;
component_declaration ::=
component identifier [is]
[ local_generic_clause]
[ local_port_clause]
end component [ component_simple_name];
component_instantiation_statement ::=
instantiation_label:
instantiated_unit
[ generic_map_aspect ]
[ port_map_aspect ] ;
component_specification ::=
instantiation_list : component_name
composite_type_definition ::=
array_type_definition
```

```
| record_type_definition
concurrent_assertion_statement ::=
[ label: ] [ postponed ] assertion ;
concurrent_procedure_call_statement ::=
[ label: ] [ postponed ] procedure_call ;
concurrent_signal_assignment_statement ::=
[ label: ] [ postponed ] conditional_signal_assignment
| [ label: ] [ postponed ] selected_signal_assignment
concurrent_statement ::=
block_statement
| process_statement
| concurrent_procedure_call_statement
| concurrent_assertion_statement
| concurrent_signal_assignment_statement
| component_instantiation_statement
| generate_statement
condition ::= boolean_expression
condition_clause ::= until condition
conditional_signal_assignment ::=
target <= options conditional_waveforms ;
conditional_waveforms ::=
{ waveform when condition else }
waveform [ when condition ]
configuration_declaration ::=
configuration identifier of entity_name is
configuration_declarative_part
block_configuration
end [configuration] [ configuration_simple_name];
configuration_declarative_item ::=
use_clause
| attribute_specification
| group_declaration
configuration_declarative_part ::=
{ configuration_declarative_item }
configuration_item ::=
block_configuration
| component_configuration
configuration_specification ::=
```

```
for component_specification binding_indication;
constant_declaration ::=
constant identifier_list : subtype_indication
[ := expression ] ;
constrained_array_definition ::=
array index_constraint of element_subtype_indication
constraint ::=
range_constraint
| index_constraint
context_clause ::= { context_item }
context_item ::=
library_clause
| use_clause
decimal_literal ::= integer [ . integer ] [ exponent ]
declaration ::=
type_declaration
| subtype_declaration
| object_declaration
| interface_declaration
| alias_declaration
| attribute_declaration
| component_declaration
| group_template_declaration
| group_declaration
| entity_declaration
| configuration_declaration
| subprogram_declaration
| package_declaration
delay_mechanism ::=
transport
| [ reject time_expression ] inertial
design_file ::= design_unit { design_unit }
design_unit ::= context_clause library_unit
designator ::= identifier | operator_symbol
direction ::= to | downto
disconnection_specification ::=
disconnect guarded_signal_specification after
time_expression ;
```

```

discrete_range ::= discrete_subtype_indication | range
element_association ::=
[ choices => ] expression
element_declaration ::= identifier_list :
element_subtype_definition ;
element_subtype_definition ::= subtype_indication
entity_aspect ::=
entity entity_name [(architecture_identifier)]
| configuration configuration_name
| open
entity_class ::=
entity | architecture | configuration
| procedure | function | package
| type | subtype | constant
| signal | variable | component
| label | literal | units
| group | file
entity_class_entry ::= entity_class [<>]
entity_class_entry_list ::=
entity_class_entry {, entity_class_entry }
entity_declaration ::=
entity identifier is
entity_header
entity_declarative_part
[ begin
entity_statement_part ]
end [ entity ] [ entity_simple_name ] ;
entity_declarative_item ::
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration

```

```
| attribute_specification
| disconnection_specification
| use_clause
| group_template_declaration
| group_declaration
entity_declarative_part ::=
{ entity_declarative_item }
entity_designator ::= entity_tag [signature]
entity_header ::=
[ formal_generic_clause ]
[ formal_port_clause ]
entity_name_list ::=
entity_designator {, entity_designator}
| others
| all
entity_specification ::=
entity_name_list : entity_class
entity_statement ::=
concurrent_assertion_statement
| passive_concurrent_procedure_call
| passive_process_statement
entity_statement_part ::=
{ entity_statement }
entity_tag ::= simple_name | character_literal |
operator_symbol
enumeration_literal ::= identifier | character_literal
enumeration_type_definition ::=
(enumeration_literal {, enumeraton_literal })
exit_statement ::=
[ label: ] exit [ loop_label ] [ when condition ] ;
exponent ::= E [ + ] integer | E - integer
expression ::=
relation { and relation }
| relation { or relation }
| relation { xor relation }
| relation [ nand relation ]
| relation [ nor relation ]
| relation { xnor relation }
```

```

extended_digit ::= digit | letter
extended_identifier ::=
| graphic_character { graphic_character } \
factor ::=
primary [ ** primary ]
| abs primary
| not primary
file_declaration ::=
file identifier_list : subtype_indication
[ file_open_information ] ;
file_logical_name ::= string_expression
file_open_information ::=
[ open file_open_kind_expression ] is file_logical_name
file_type_definition ::= file of type_mark
floating_type_definition ::= range_constraint
formal_designator ::=
generic_name
| port_name
| parameter_name
formal_parameter_list ::= parameter_interface_list
formal_part ::=
formal_designator
| function_name(formal_designator)
| type_mark(formal_designator)
full_type_declaration ::=
type identifier is type_definition ;
function_call ::=
function_name [ (actual_parameter_part) ]
generate_statement ::=
generate_label:
generation_scheme generate
[ { block_declarative_item }
begin ]
{ concurrent_statement }
end generate [ generate_label] ;
generation_scheme ::=
for generate_parameter_specification
| if condition

```



```
generic_clause ::=
generic(generic_list);
generic_list ::= generic_interface_list
generic_map_aspect ::=
generic map (generic_association_list)
graphic_character ::=
basic_graphic_character | lower_case_letter |
  other_special_character
group_constituent ::= name | character_literal
group_constituent_list ::= group_constituent
{, group_constituent }
group_declarataion ::=
group identifier : group_template_name
(group_consituent_list);
group_template_declaration ::=
group identifier is (entity_class_entry_list) ;
guarded_signal_specification ::=
guarded_signal_list : type_mark
identifier ::=
basic_identifier | extended_identifier
identifier_list ::= identifier {, identifier }
if_statement ::=
[ if_label: ]
if condition then
sequence_of_statements
{ elsif condition then
sequence_of_statements }
[ else
sequence_of_statements ]
end if [ if_label ] ;
incomplete_type_declaration ::= type identifier ;
index_constraint ::= (discrete_range
{, discrete_range })
index_specification ::=
discrete_range
| static_expression
index_subtype_definition ::= type_mark range <>
indexed_name ::= prefix (expression {, expression })
```

```

instantiated_unit ::=
[component] component_name
| entity entity_name [(architecture_name)]
| configuration configuration_name
instantiation_list ::=
instantiation_label {, instantiation_label}
| others
| all
integer ::= digit { [ underline ] digit }
integer_type_definition ::= range_constraint
interface_constant_declaration ::=
[constant] identifier_list : [in]
subtype_indication [:= static_expression]
interface_declaration ::=
interface_constant_declaration
| interface_signal_declaration
| interface_variable_declaration
| interface_file_declaration
interface_element ::= interface_declaration
interface_file_declaration ::=
file identifier_list : subtype_indication
interface_list ::=
interface_element {; interface_element}
interface_signal_declaration ::=
[signal] identifier_list : [mode]
subtype_indication [bus]
[:= static_expression]
interface_variable_declaration ::=
[variable] identifier_list : [mode] subtype_indication
[:= static_expression]
iteration_scheme ::=
while condition
| for loop_parameter_specification
label ::= identifier
letter ::= upper_case_letter | lower_case_letter
letter_or_digit ::= letter | digit
library_clause ::= library logical_name_list ;
library_unit ::=

```

```
primary_unit
| secondary_unit
literal ::=
numeric_literal
| enumeration_literal
| string_literal
| bit_string_literal
| null
logical_name ::= identifier
logical_name_list ::= logical_name {, logical_name }
logical_operator ::= and | or | nand | nor | xor | xnor
loop_statement ::=
[ loop_label: ]
[ iteration_scheme ] loop
sequence_of_statements
end loop [ loop_label ] ;
miscellaneous_operator ::= ** | abs | not
mode ::= in | out | inout | buffer | linkage
multiplying_operator ::= * | / | mod | rem
name ::=
simple_name
| operator_symbol
| selected_name
| indexed_name
| slice_name
| attribute_name
next_statement ::=
[ label: ] next [ loop_label ] [ when condition ] ;
null_statement ::=
[ label: ] null ;
numeric_literal ::=
abstract_literal
| physical_literal
object_declaration ::=
constant_declaration
| signal_declaration
| variable_declaration
| file_declaration
```

```
operator_symbol ::= string_literal
options ::= [ guarded ] [delay_mechanism]
package_body ::=
package body package_simple_name is
package_body_declarative_part
end [ package body ] [ package_simple_name ] ;
package_body_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| use_clause
| group_template_declaration
| group_declaration
package_body_declarative_part ::=
{ package_body_declarative_item }
package_declaration ::=
package identifier is
package_declarative_part
end [ package ] [ package_simple_name ] ;
package_declarative_item ::=
subprogram_declaration
| type_declaration
| subtype_declaration
| constant_declaration
| signal_declaration
| shared_variable_declaration
| file_declaration
| alias_declaration
| component_declaration
| attribute_declaration
| attribute_specification
| disconnection_specification
| use_clause
```

```
| group_template_declaration
| group_declaration
package_declarative_part ::=
{ package_declarative_item }
parameter_specification ::=
identifier in discrete_range
physical_literal ::= [ abstract_literal ] unit_name
physical_type_definition ::=
range_constraint
units
primary_unit_declaration
{ secondary_unit_declaration }
end units [ physical_type_simple_name ]
port_clause ::=
port(port_list);
port_list ::= port_interface_list
port_map_aspect ::=
port map (port_association_list)
prefix ::=
name
| function_call
primary ::=
name
| literal
| aggregate
| function_call
| qualified_expression
| type_conversion
| allocator
| (expression)
primary_unit ::=
entity_declaration
| configuration_declaration
| package_declaration
primary_unit_declaration ::= identifier
procedure_call ::= procedure_name
[ (actual_parameter_part) ]
procedure_call_statement ::= [ label: ]
```

```

procedure_call ;
process_declarative_item ::=
subprogram_declaration
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
process_declarative_part ::=
{ process_declarative_item }
process_statement ::=
[ process_label: ]
[ postponed ] process [ (sensitivity_list) ] [ is ]
process_declarative_part
begin
process_statement_part
end process [ process_label ] ;
process_statement_part ::=
{ sequential_statement }
qualified_expression ::=
type_mark'(expression)
| type_mark'aggregate
range ::=
range_attribute_name
| simple_expression direction simple_expression
range_constraint ::= range range
record_type_definition ::=
record
element_declaration
{ element_declaration }
end record [ record_type_simple_name ]

```

```
relation ::=
shift_expression [ relational_operator
shift_expression ]
relational_operator ::= = | /= | < | <= | > | >=
report_statement ::=
[label:] report expression
[severity expression] ;
return_statement ::=
[ label: ] return [ expression ] ;
scalar_type_definition ::=
enumeration_type_definition
| integer_type_definition
| physical_type_definition
| floating_type_definition
secondary_unit ::=
architecture_body
| package_body
secondary_unit_declaration ::= identifier =
physical_literal ;
selected_name ::= prefix.suffix
selected_signal_assignment ::=
with expression select
target <= options selected_waveforms ;
selected_waveforms ::=
{ waveform when choices, }
waveform when choices
sensitivity_clause ::= on sensitivity_list
sensitivity_list ::= signal_name {, signal_name}
sequence_of_statements ::=
{ sequential_statement }
sequential_statement ::=
wait_statement
| assertion_statement
| report_statement
| signal_assignment_statement
| variable_assignment
| procedure_call_statement
| if_statement
```

```

| case_statement
| loop_statement
| next_statement
| exit_statement
| return_statement
| null_statement
shift_expression ::=
simple_expression [ shift_operator simple_expres-
sion ]
shift_operator ::= sll | srl | sla | sra | rol | ror
sign ::= + | -signal_
assignment_statement ::=
[ label: ] target <= [ delay_mechanism ] waveform ;
signal_declaration ::=
signal identifier_list : subtype_indication
[signal_kind] [:= expression] ;
signal_kind ::= register | bus
signal_list ::=
signal_name {, signal_name }
| others
| all
signature ::= [ [ type_mark {, type_mark } ]
[ return type_mark ]
simple_expression ::=
[ sign ] term { adding_operator term }
simple_name ::= identifier
slice_name ::= prefix (discrete_range)
string_literal ::= « { graphic_character } «
subprogram_body ::=
subprogram_specification is
subprogram_declarative_part
begin
subprogram_statement_part ]
end [ subprogram_kind ] [ designator ] ;
subprogram_declaration ::=
subprogram_specification ;
subprogram_declarative_item ::=
subprogram_declaration

```



```
| subprogram_body
| type_declaration
| subtype_declaration
| constant_declaration
| variable_declaration
| file_declaration
| alias_declaration
| attribute_declaration
| attribute_specification
| use_clause
| group_template_declaration
| group_declaration
subprogram_declarative_part ::=
{ subprogram_declarative_item }
subprogram_kind ::= procedure | function
subprogram_specification ::=
procedure designator [ (formal_parameter_list) ]
| [ pure | impure ] function designator
| (formal_parameter_list) ]
return type_mark
subprogram_statement_part ::=
{ sequential_statement }
subtype_declaration ::=
subtype identifier is subtype_indication ;
subtype_indication ::=
[ resolution_function_name ] type_mark [ constraint ]
suffix ::=
simple_name
| character_literal
| operator_symbol
| all
target ::=
name
| aggregate
term ::=
factor { multiplying_operator factor }
timeout_clause ::= for time_expression
type_conversion ::= type_mark(expression)
```

```
type_declaration ::=
full_type_declaration
| incomplete_type_declaration
type_definition ::=
scalar_type_definition
| composite_type_definition
| access_type_definition
| file_type_definition
type_mark ::=
type_name
| subtype_name
unconstrained_array_definition ::=
array (index_subtype_definition {, index_subtype_
definition })
of element_subtype_indication
use_clause ::=
use selected_name {, selected_name} ;
variable_assignment_statement ::=
[ label: ] target := expression ;
variable_declaration ::=
[shared] variable identifier_list :
subtype_indication [:= expression] ;
wait_statement ::=
[label:] wait [sensitivity_clause] [condition_clause]
[timeout_clause] ;
waveform ::=
waveform_element {, waveform_element}
| unaffected
waveform_element ::=
value_expression [after time_expression]
| null [after time_expression]
```

Глава 5. **Язык описания аппаратуры Verilog HDL**

5.1. Общие сведения

Как известно [45], язык описания аппаратуры Verilog был разработан фирмой «Gateway Design Automation» в 1984 г. После поглощения последней таким монстром, как фирма «Caddence», язык стал получать все более широкое распространение среди разработчиков и стал не менее популярен, чем VHDL.

В отличие от VHDL, структура и синтаксис которого напоминают такие «сложные» языки, как АДА или АЛГОЛ, синтаксис Verilog напоминает очень популярный в среде как программистов, так и разработчиков встроенных систем и систем ЦОС старый добрый C++. Verilog позволяет достаточно эффективно выполнить описание и провести моделирование (simulate) и синтез цифровых схем благодаря применению встроенных примитивов (built-in primitives), примитивов пользователя (user-defined primitives), средств временного контроля (timing checks), моделирования задержки распространения от входа до выхода (pin-to-pin delay simulation), возможностью задания внешних тестовых сигналов (external stimulus).

Как и VHDL, Verilog изначально предназначался для моделирования цифровых систем и как средство описания синтезируемых проектов стал использоваться с 1987 года. В настоящее время ведущие пакеты синтеза систем на ПЛИС, такие, как продукты фирм «Synopsys», «Caddence», «Mentor Graphics», многих производителей ПЛИС поддерживают синтез с описания на языке Verilog. В языке Verilog поддерживается набор типов логических вентилей (Gate types).

Для логических вентилях определены ключевые слова (keywords): AND («И»), NAND («И-НЕ»), OR («ИЛИ»), NOR («ИЛИ-НЕ»), XOR («исключающее ИЛИ»), XNOR («исключающее ИЛИ-НЕ»), BUF (буферный элемент), NOT (отрицание «НЕ»).

В Verilog при использовании вентилях необходимо задать входы и выходы элемента, а также (не обязательно) имя вентиля. Например, вентиля AND и OR должны иметь один выход и два и более входов. Так, для вентиля «И» (AND) имеем.

```
and <name><list of arguments>
and myand(out, in1, in2, in3);
and (out, in1, in2
```

Вентиля BUF и NOT могут иметь один вход и один и более выходов. Имя вентиля является необязательным. Примеры использования приведены ниже.

```
buf mybuf(out1, out2, out3, in);
not (out, in);
```

Говоря о синтаксисе языка Verilog, следует помнить, что он является контекстно-зависимым языком, то есть строчные и прописные буквы различаются. Все ключевые слова (keywords) задаются в нижнем регистре (строчные буквы). Для обозначения пустого пространства (white space) используются символы пробела, табуляции и новой строки.

В Verilog поддерживаются два типа комментариев. Они аналогичны принятым в C++, если надо закомментировать одну строку, то используется две косые черты в ее начале:

```
// это комментарий
```

Чтобы закомментировать несколько строк, используется следующая конструкция:

```
/* Это
   комментарии... */
```

Очевидно, что комментарии не могут быть вложенными.

5.2. Операторы

В Verilog существует три типа операторов — с одним, двумя и тремя операндами. Унарные операторы располагаются слева от операнда, бинарные между операндами и тернарный оператор разделяет три операнда двумя операторами.

Примеры операторов:

```
clock = ~clock; // унарный оператор отрицания
           // clock — операнд
c = a || b; // || — бинарный оператор ИЛИ, a и b операнды
r = s ? t : u; // ?: — тернарный оператор, читается как
           // r = [if s is true then t else u]
```

Как правило, при написании описаний на Verilog комментарии одной строкой (//) используют для ввода текстовых комментариев к коду, а конструкцию /* */ используют при отладке для «закомментирования» фрагментов кода.

5.3. Числа в Verilog

5.3.1. Целые числа (Integers)

Целые числа могут быть двоичными (binary), обозначаются b или B, десятичными (decimal, d или D), шестнадцатеричными (hexidecimal, h или H) или восьмеричными (octal, o или O). Для определения чисел используются следующие форматы:

1. <разрядность>'<основание><число> — полное описание числа.
2. <основание><число> — используется разрядность представления, заданная в системе по определению, но не менее 32 бит.
3. <число> — используется, когда по умолчанию десятичное основание. Разрядность определяет число бит под представление числа.

Например:

```
8'b10100010 // 8-разрядное число в двоичной системе
8'hA2       // 8-разрядное число в шестнадцатеричной системе
```

5.3.2. Неопределенное и высокоимпедансное состояния (x and z values)

Символ x используется для задания неопределенного состояния, символ z показывает третье (высокоимпедансное). При использовании в качестве цифры в числах вместо символа z можно использовать "?". Это рекомендуется делать в операторах выбора (CASE expressions) для улучшения читаемости кода. Ниже приведены примеры использования символов x и z в числах.

```
4'b10x0
4'b101z
12'dz
12'd?
8'h4x
```

5.3.3. Отрицательные числа (Negative numbers)

Отрицательное число задается с помощью знака минус перед разрядностью числа. Примеры отрицательных чисел:

```
-8'd5
8'b-5 // неправильно, знак минус перед числом, а не разрядностью!
```

5.3.4. Подчеркивание (Underscore)

Знак подчеркивания (_, Underscores) может быть записан в любом месте числа, что позволяет использовать его как разделитель разрядов, улучшающий читабельность.

```
16'b0001_1010_1000_1111 // использование подчеркивания
8'b_0001_1010 // некорректное использование подчеркивания
```

5.3.5. Действительные числа (Real)

Действительные числа (Real numbers) могут быть представлены либо в десятичном виде, либо в стандартной форме с плавающей точкой (scientific format). Примеры действительных чисел:

```
1.8
```

```
3_2387.3398_3047
3.8e10    // е или Е для обозначения порядка
2.1e-9
3.    // неправильно!
```

5.3.6. Строки (Strings)

Строка заключается в кавычки «...» и не может занимать более одной линии. Примеры использования строк:

```
«hello world»; // правильное использование строк
«good
b
y
e
wo
rld»;    // неправильное использование строк
```

5.4. Цепи в Verilog (Nets)

Для обозначения цепей используются следующие ключевые слова: wire, supply0, supply1. Величина по умолчанию (default value): z. Разрядность по умолчанию (default size): 1 бит.

По своему назначению цепь (Net) в Verilog сходна с сигналом в VHDL. Цепи обеспечивают непрерывное модифицирование сигналов на выходах цифровой схемы относительно изменения сигналов на ее входах.

Если драйвер (источник) сигнала цепи имеет некоторое значение, то и цепь принимает то же значение. Если драйверы цепи принимают различные значения, цепь принимает значение наиболее «сильного» сигнала (strongest), если же «сила» каждого сигнала равнозначна, то цепь принимает неопределенное состояние (x).

Для обозначения цепи наиболее часто применяется ключевое слово wire, ключевые слова supply0 и supply1 используются для моделирования источников питания (power supplies) в схеме.

5.5. Регистры (Registers)

Для обозначения регистров используется следующее ключевое слово: reg. Величина по умолчанию: x. Разрядность по умолчанию: 1 бит.

Основное различие между цепями (Nets) и регистрами (Registers) состоит в том, что значение регистра должно быть назначено явно. Эта величина сохраняется до тех пор, пока не сделано новое назначение. Рассмотрим использование этого свойства на примере триггера с разрешением (защелки, E-type Flipflop):

```
module E_ff(q, data, enable, reset, clock);
    output q;
    input data, enable, reset, clock;
    reg q;

    always @(posedge clock)
        if (reset == 0)
            q = 1'b0;
        else if (enable==1)
            q = data;
endmodule
```

Регистр q хранит записанную в него величину до тех пор, пока не произойдет нового назначения сигнала. Для того чтобы провести моделирование этого примера, напомним модуль верхнего уровня (higher level module), который формирует тестовый сигнал и позволяет провести наблюдение за выходами триггера. В этом случае сигнал q является цепью, драйвером (источником) которой служит модуль E_ff.

```
module stimulus;
    reg data, enable, clock, reset;
    wire q;

    initial begin
        clock = 1'b0;
        forever #5 clock = ~clock;
    end

    E_ff eff0(q, data, enable, reset, clock);

    initial begin
```



```
reset = 1'b0;
#10 reset = 1'b1;
    data = 1'b1;
#20 enable = 1;
#10 data = 1'b0;
#10 data = 1'b1;
#10 enable = 0;
#10 data = 1'b0;
#10 data = 1'b1;
#10 enable = 1;
#10 reset = 1'b0;
#30 $finish;
end

initial
    $monitor($time, "q = %d", q);

endmodule
```

5.6. Векторы (Vectors)

Как цепи, так и регистры могут иметь произвольную разрядность, если их объявлять как векторы. Ниже приведены примеры объявлений векторов.

```
reg [3:0] output; // выход 4-разрядного регистра
wire [31:0] data; // 32-разрядная цепь
reg [7:0] a;
data[3:0] = output; // частичное назначение
output = 4'b0101; // назначение на целый регистр
```

Важным является порядок назначения элементов в векторе. Первый элемент регистра является наиболее значимым (most significant):

```
reg [3:0] a; // a3 — старший разряд
reg [0:3] b; // b0 — старший разряд
```

5.7. Массивы (Arrays)

Регистры (Registers), целые числа (Integers) и временные (Time) типы данных можно объявлять как массивы, как это показано в нижеследующем примере:

Объявление:

```
<data_type_spec> {size} <variable_name> {array_size}
```

Использование:

```
<variable_name> {array_reference} {bit_reference}
```

```
reg data [7:0]; // 8 1-разрядных элементов
integer [3:0] out [31:0]; // 32 4-разрядных элемента
data[5]; // 5 8-разрядных элементов
```

5.8. Регистровые файлы (Memories)

Регистровый файл (Memories) представляет собой массив регистров (Array of registers). Ниже представлен синтаксис объявления регистрового файла.

```
reg [15:0] mem16_1024 [1023:0]; // регистровый файл 1K × 16
mem16_1024[489]; // 489 элемент файла mem16_1024
```

Желательно для обозначения регистровых файлов использовать информативные имена, например mem16_1024, чтобы избежать путаницы.

5.9. Элементы с третьим состоянием (Tri-state)

Как уже упоминалось, в языке Verilog ситуация, когда драйверами одной цепи являются более одного источника с разными значениями, разрешается в пользу источника, имеющего большую «силу». Наименьшую «силу» имеет сигнал z, обозначающий третье или высокоимпедансное (high-impedance) состояние. Правда, эти рассуждения актуальны именно на этапе моделирования, но никак не синтеза ПЛИС, о них следует помнить при разработке тестов. Таким образом, драйвер в третьем состоянии

не влияет на итоговое значение сигнала цепи. Пример драйвера с третьим состоянием приведен ниже.

```
module triDriver(bus, drive, value);
    inout [3:0] bus;
    input      drive;
    input [3:0] value;

    assign #2 bus = (drive == 1) ? value : 4'bz;

endmodule // triDriver
```

В данном примере, когда управляющий сигнал принимает **ВЫСОКИЙ** уровень, шина принимает значение входной величины, в противном случае шина переходит в третье состояние.

В следующем примере используется несколько управляющих комбинаций для трех буферов с тремя состояниями.

```
module myTest;
    wire [3:0] bus;
    reg drive0, drive1, drive2;

    integer      i;

    triDriver mod1 (bus, drive0, i[3:0]);
    triDriver mod2 (bus, drive1, 4'hf);
    triDriver mod3 (bus, drive2, 4'h0);

    initial begin
        for (i = 0; i < 12; i = i + 1) begin
            #5 {drive2, drive1, drive0} = i;
            #5 $display ($time, "%b %b %d", i[2:0], bus, bus);
            end
            $finish;
        end
    end

endmodule
```

Результаты прогона модели имеют вид:

```
10 000 zzzz z
20 001 0001 1
30 010 1111 15
40 011 xx11 X
50 100 0000 0
60 101 0x0x X
70 110 xxxx x
80 111 xxxx x
90 000 zzzz z
100 001 1001 9
110 010 1111 15
120 011 1x11 X
```

5.10. Арифметические операторы (Arithmetic operators)

Символы (keysymbols): *, /, +, -, % .

Бинарные операторы умножения, деления, сложения, вычитания, определения остатка от деления представлены в нижеследующем примере.

```
module arithTest;
    reg [3:0] a, b;

    initial begin
        a = 4'b1100; // 12
        b = 4'b0011; // 3

        $displayb(a * b); // умножение - 4'b1000
                           // 4 МСБ
        $display(a / b); // деление 4
        $display(a + b); // сложение 15
        $display(a - b); // вычитание 9
        $display((a + 1'b1) % b); // остаток 1
    end

endmodule // arithTest
```

Унарные плюс и минус имеют более высокий приоритет (precedance), чем бинарные операторы.

Следует заметить, что если хотя бы один бит в одном из операндов неопределен (равен *x*), то и результат операции также будет неопределен.

5.11. Логические операторы (Logical operators)

Ключевые символы: `&&`, `||`, `!`.

К логическим операциям относятся «И» (AND), «ИЛИ» (OR) и «НЕ» (NOT). Результат логической операции может принимать значения истинно (TRUE, "1") или ложно (FALSE, "0"), а также иметь неопределенное состояние (unknown, *x*). При выполнении логического оператора все неопределенные величины, как и операнды в третьем состоянии, принимаются как имеющие НИЗКИЙ логический уровень (FALSE). В качестве операнда могут выступать как переменная (Variable), так и логическое выражение (Expression). Пример работы логических операторов приведен ниже.

```
module logicalTest;

    reg [3:0] a, b, c;

    initial begin
        a = 2; b = 0; c = 4'hx;

        $display(a && b); // И 0
        $display(a || b); // ИЛИ 1
        $display(!a);     // НЕ 0
        $display(a || c); // 1, unknown || 1 (=1)
        $display(!c);     // unknown
    end

endmodule // logicalTest
```

5.12. Операторы отношения (Relational Operators)

Ключевые символы: >, <, >=, <=.

К операторам отношения относятся операторы «больше», «меньше», «больше или равно», «меньше или равно». Результат операции — истина или ложь. Если хотя бы один операнд неопределен, то и результат операции будет неопределен.

```
module relatTest;
    reg [3:0] a, b ,c, d;

    initial begin
        a=2;
        b=5;
        c=2;
        d=4'hx;

        $display(a < b); // true, 1
        $display(a > b); false, 0
        $display(a >= c); // true, 1
        $display(d <= a); unknown
    end
endmodule // relatTest
```

5.13. Операторы эквивалентности (Equality)

Ключевые символы: ==, !=, ===, !==.

К операторам эквивалентности (Equality operators) относятся оператор логического равенства (logical equality), неравенства (logical inequality), выборочного равенства (CASE equality) и неравенства (inequality). Эти операторы сравнивают операнды побитно. Логические операторы возвращают неопределенный результат, если операнд содержит неопределенные биты, в отличие от выборочных операторов. В случае неравной длины операндов более короткий дополняется нулями.

Ниже приведен пример использования операторов эквивалентности.

```
module equTest;
    reg [3:0] a, b ,c, d, e, f;
    initial begin
        a = 4; b = 7;      // these default to
decimal bases
        c = 4'b010;
        d = 4'bx10;
        e = 4'bx101;
        f = 4'bxx01;

        $displayb(c);      // outputs 0010
        $displayb(d);      // outputs xx10

        $display(a == b); // logical equality, evaluates to 0
        $display(c != d); // logical inequality, evaluates to x
        $display(c != f); // logical inequality, evaluates to 1
        $display(d === e); // case equality, evaluates to 0
        $display(c !== d); // case inequality, evaluates to 1
    end
endmodule // equTest
```

Ну и, наверное, совершенно понятно, что операторы эквивалентности и присваивания — совершенно различные операторы.

5.14. Поразрядные операторы (Bitwise operators)

Ключевые символы: `~, &, |, ^, (~^, ^~)`.

К поразрядным операторам относятся поразрядное отрицание, поразрядные логические «И», «ИЛИ», исключающее «ИЛИ», исключающее «ИЛИ-НЕ». Поразрядные операторы выполняются только над операндами, имеющими одинаковую разрядность. В том случае, если разрядность одного операнда меньше другого, недостающие разряды дополняются нулями. Ниже приведен пример использования поразрядных операторов.

```
module bitTest;
    reg [3:0] a, b ,c;
```

```

initial begin
  a = 4'b1100; b = 4'b0011; c = 4'b0101;
  $displayb(~a); // bitwise negation, evaluates to 4'b0011
  $displayb(a & c); // bitwise and, evaluates to 4'b0100
  $displayb(a | b); // bitwise or, evaluates to 4'b1111
  $displayb(b ^ c); // bitwise xor, evaluates to 4'b0110
  $displayb(a ~^ c); // bitwise xnor, evaluates to 4'b0110
end
endmodule // bitTest

```

5.15. Операторы приведения (Reduction operator)

Ключевые символы: $\&$, $\sim\&$, $|$, $\sim|$, \wedge , $\sim\wedge$, $\wedge\sim$.

Операторы приведения — «И», «ИЛИ», «И-НЕ», «ИЛИ-НЕ», «исключающее ИЛИ», «исключающее ИЛИ-НЕ» (два варианта). Они выполняются над многоразрядным операндом пошагово, бит за битом, начиная с двух крайних левых разрядов, выдавая на выходе одноразрядный результат. Очевидно, что такой подход позволяет реализовать проверку на четность (нечетность). Ниже приведены примеры использования операторов приведения.

```

module reductTest;
  reg [3:0] a, b ,c;

  initial begin
    a = 4'b1111;
    b = 4'b0101;
    c = 4'b0011;
    $displayb(& a); //, (то же 1&1&1&1), равен 1 1
    $displayb(| b); // (same as 0|1|0|1), evaluates to 1
    $displayb(^ b); // искл.ИЛИ (same as 0^1^0^1), eval-
    uates to 0
  end

endmodule // reductTest

```

Безусловно, следует замечать различия между логическими операторами, поразрядными операторами и операторами приведения. Несмотря на схожесть символов этих операторов, число операндов в каждом случае различно.

5.16. Операторы сдвига (SHIFT operator)

Ключевые символы: `>>`, `<<`.

Операторы сдвига позволяют осуществить сдвиг операнда как вправо, так и влево. Пример их использования приведен ниже.

```
module shiftTest;
    reg [3:0] a;

    initial begin
        a = 4'b1010;

        $displayb(a << 1);    // shift left by 1, evaluates to
        4'b0100
        $displayb(a >> 2);    // shift right by 2, evaluates to
        4'b0010

        end

    endmodule // shiftTest
```

Этот оператор часто применяют для реализации регистров сдвига, длинных алгоритмов перемножения и т.п.

5.17. Конкатенация (объединение, Concatenation)

Ключевой символ: `{,}`

Объединение позволяет увеличить разрядность цепей, регистров и т.д.

```
module concatTest;
    reg a;
    reg [1:0] b;
    reg [5:0] c;
    initial begin
        a = 1'b1;
        b = 2'b00;
        c = 6'b101001;
    end
endmodule
```

```
$displayb({a, b}); // produces a 3-bit number 3'b100
$displayb({c[5:3], a}); // produces 4-bit number 4'b1011

end

endmodule // concatTest
```

5.18. Повторение (Replication)

Повторение (Replication) может быть использовано для многократного повторения объединения, как показано в нижеследующем примере.

```
module replicTest;
    reg a;
    reg [1:0] b;
    reg [5:0] c;

    initial begin
        a = 1'b1;
        b = 2'b00;

        $displayb({4{a}}); // результат - 1111
        c = {4{a}};
        $displayb(c);      // результат - 001111
    end

endmodule // replicTest
```

5.19. Системные директивы (System tasks)

Наверное, этот раздел не будет интересен тем, кто пишет только синтезируемые описания на Verilog. Но поскольку язык является не только средством описания проекта, но и довольно мощным инструментом для поведенческого моделирования систем, то следует сказать несколько слов о встроенных директивах компилятора, позволяющих выполнить моделирование и произвести анализ его результатов.

5.19.1. Директивы вывода результатов моделирования (Writing to standard output)

Ключевые слова: \$display, \$displayb, \$displayh, \$displayo, \$write, \$writeb, \$writeh, \$writeo.

Наиболее часто применяется директива \$display. Она может быть использована для вывода на экран строк, выражений или переменных. Ниже приведен пример использования директивы \$display.

```
$display("Hello Dr Blair");
-- output: Hello Dr Blair

$display($time) // current simulation time.
-- output: 460

counter = 4'b10;
$display("The count is %b", counter);
-- output: The count is 0010
```

Синтакс определения формата вывода аналогичен синтаксису printf в языке программирования С. Ниже (в **Табл. 5.1**) приведено его описание для директивы \$display.

Таблица 5.1. Описание синтаксиса для директивы \$display

Формат	Описание
%d или %D	decimal
%b или %B	binary
%h или %H	hexadecimal
%o или %O	octal
%m или %M	hierarchical name
%t или %T	Time format
%e или %E	Real in scientific format
%f или %F	Real in decimal format
%g или %G	Real in shorter of above two

Для специальных символов используются следующие эскейп-последовательности (escape sequence):

<code>\n</code>	Новая строка
<code>\t</code>	табуляция
<code>\\</code>	<code>\</code>
<code>\></code>	<code><</code>
<code>%%</code>	<code>%</code>

Директива `$write` идентична директиве `$display`, за исключением того, что она не осуществляет автоматический переход на новую строку в конце вывода информации. Если спецификация вывода не определена, то по умолчанию используются следующие форматы:

Директива	Формат по умолчанию
<code>\$display</code>	decimal
<code>\$displayb</code>	binary
<code>\$displayh</code>	hexadecimal
<code>\$displayo</code>	octal
<code>\$write</code>	decimal
<code>\$writeb</code>	binary
<code>\$writeh</code>	hexadecimal
<code>\$writeo</code>	octal

Так, например, следующий фрагмент кода:

```
$write(5'b01101);
$writeb(<< «, 5'b01101);
$writeh(<< «, 5'b01101);
$writeo(<< «, 5'b01101,>>\n»);
```

И результат его работы:

```
13  01101 0d 15
14
```

**5.19.2. Контроль процесса моделирования
(Monitoring a simulation)**

Ключевые слова: `$monitor`, `$monitoron`, `$monitoroff`.
Формат директивы `$monitor` практически аналогичен формату `$display`.

Разница заключается в том, что выход формируется при любом изменении переменных, которое произойдет в определенное время. Наблюдение может быть включено или отключено с помощью директив `$monitoron` или `$monitoroff` соответственно. По умолчанию в начале моделирования наблюдение за его ходом включено. Ниже приведен пример наблюдения за моделированием.

```
module myTest;
    integer a,b;

    initial begin
        a = 2;
        b = 4;
        forever begin
            #5 a = a + b;
            #5 b = a - 1;
        end // forever begin
    end // initial begin

    initial #40 $finish;

    initial begin
        $monitor($time, «a = %d, b = %d», a, b);
    end // initial begin

endmodule // myTest
```

Результат прогона:

```
0 a = 2, b = 4
5 a = 6, b = 4
10 a = 6, b = 5
15 a = 11, b = 5
20 a = 11, b = 10
25 a = 21, b = 10
30 a = 21, b = 20
35 a = 41, b = 20
```

5.19.3. Окончание моделирования (Ending a simulation)

Ключевые слова: \$stop, \$finish.

Директива \$finish завершает симуляцию и передает управление операционной системе. Директива \$stop приостанавливает моделирование и переводит систему с Verilog в интерактивный режим (см. пример ниже).

```
initial begin
    clock = 1'b0;
    #200 $stop
    #500 $finish
End
```

5.20. Проектирование комбинационных схем, пример проектирования мультиплексора 4 в 1

Рассмотрим проектирование комбинационных логических устройств на примере мультиплексора 4 в 1. При проектировании мультиплексора рассмотрим различные методы его построения.

5.20.1. Реализация на уровне логических вентилей (Gate level implementation)

Рассмотрим пример реализации нашего мультиплексора 4 в 1 на уровне логических вентилей.

```
module multiplexor4_1(out, in1, in2, in3, in4,
cntrl1, cntrl2);

    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;
    wire notcntrl1, notcntrl2, w, x, y, z;

    not (notcntrl1, cntrl1);
    not (notcntrl2, cntrl2);

    and (w, in1, notcntrl1, notcntrl2);
    and (x, in2, notcntrl1, cntrl2);
```

```
and (y, in3, cntrl1, notcntrl2);
and (z, in4, cntrl1, cntrl2);

or (out, w, x, y, z);
```

```
endmodule
```

Начнем подробное построчное рассмотрение этого примера. Итак, «в первых строках» у нас:

```
module multiplexor4_1(out, in1, in2, in3, in4,
cntrl1, cntrl2);
```

Первая строка — описание модуля, ключевое слово — `module` — используется вместе с именем модуля, по которому осуществляется ссылка на модуль. В скобках приведен список портов модуля, причем вначале перечисляются выходы, затем входы. Каждая строка завершается точкой с запятой — это, как известно, правило для многих языков высокого уровня.

```
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;
```

Все порты в списке должны быть объявлены как входы, выходы или двунаправленные выводы (`inout`), в этом случае они по умолчанию назначаются типом `wire`, если нет других указаний. Когда назначено имя цепи, система моделирования на базе Verilog ожидает неявное назначение выходного сигнала, оценивая его, чтобы осуществлять передачу этого сигнала к внешним модулям.

```
wire notcntrl1, notcntrl2, w, x, y, z;
```

В данной строке определяются внутренние цепи, осуществляющие объединение узлов мультиплексора. Как видим, понятия цепи в Verilog и сигнала в VHDL очень схожи.

```
not (notcntrl1, cntrl1);
not (notcntrl2, cntrl2);
```

В этих строчках описываются вентили «НЕ» с входами `cntrl1` и `cntrl2` и выходами `notcntrl1` и `notcntrl2` соответственно. Следует помнить, что в описании портов вентиля всегда вначале идут выходы, затем входы.

```
and (w, in1, notcntrl1, notcntrl2);
and (x, in2, notcntrl1, cntrl2);
and (y, in3, cntrl1, notcntrl2);
and (z, in4, cntrl1, cntrl2);

or (out, w, x, y, z);
```

Аналогично описываются и вентили «И» и «ИЛИ».

```
endmodule
```

Конец модуля завершается ключевым словом `endmodule`.

5.20.2. Реализация мультиплексора с помощью логических операторов (Logic statement implementation)

Реализация мультиплексора с использованием логических операторов имеет вид:

```
module multiplexor4_1 (out, in1, in2, in3,
in4, cntrl1, cntrl2);
output out;
input in1, in2, in3, in4, cntrl1, cntrl2;

assign out = (in1 & ~cntrl1 & ~cntrl2) |
              (in2 & ~cntrl1 & cntrl2) |
              (in3 & cntrl1 & ~cntrl2) |
              (in4 & cntrl1 & cntrl2);

endmodule
```

Ну, в начале файла все понятно — имя проекта, порты ...

```
module multiplexor4_1 (out, in1, in2, in3,
in4, cntrl1, cntrl2);
```



```
output out;  
input in1, in2, in3, in4, cntrl1, cntrl2;
```

Обратим внимание, что ни порты, ни их количество не изменилось — стыковка с внешними модулями также останется неизменной. В какой-то мере это напоминает проект с использованием VHDL, когда для одного интерфейса можно описать несколько архитектурных тел.

```
assign out = (in1 & ~cntrl1 & ~cntrl2) |  
             (in2 & ~cntrl1 &  cntrl2) |  
             (in3 &  cntrl1 & ~cntrl2) |  
             (in4 &  cntrl1 &  cntrl2);  
  
endmodule
```

С помощью конструкции `assign` осуществляется непрерывное назначение (`continuous assignment`) цепи `out`. В этом случае ее значение заново вычисляется каждый раз, когда меняется хоть один из операндов.

5.20.3. Реализация с помощью оператора выбора (CASE statement implementation)

Рассмотрим использование оператора выбора (`CASE statement`) для реализации мультиплексора. Следует заметить, что этот способ, наверное, наиболее прост и наименее трудоемок.

```
module multiplexor4_1 (out, in1, in2, in3,  
                      in4, cntrl1, cntrl2);  
output out;  
input  in1, in2, in3, in4, cntrl1, cntrl2;  
reg    out;  
  
always @(in1 or in2 or in3 or in4 or  
cntrl1 or cntrl2)  
case ({cntrl1, cntrl2})  
2'b00 : out = in1;  
2'b01 : out = in2;  
2'b10 : out = in3;  
2'b11 : out = in4;
```

```
default : $display(«Please check control bits»);  
        endcase  
endmodule
```

Ну, первые три строки знакомы — добавить к сказанному ранее практически нечего.

```
module multiplexor4_1 (out, in1, in2, in3,  
    in4, cntrl1, cntrl2);  
    output out;  
    input in1, in2, in3, in4, cntrl1, cntrl2;  
    reg out;
```

Единственное отличие — выход out определен как регистр, это сделано для того, чтобы назначать его значения явно и не управлять ими, такое назначение сигнала называется процедурным назначением (procedural assignment). Данные типа wire не могут быть назначены явно, они нуждаются в сигнале драйвера, такое назначение называется непрерывным назначением (continuous assignment).

```
always @(in1 or in2 or in3 or in4 or cntrl1 or cntrl2)
```

Эта конструкция читается так же, как и пишется, т.е. значение вычисляется всегда при изменении хотя бы одного операнда — система постоянно их отслеживает. Несколько забегаю вперед, следует отметить, что данная конструкция является синтезируемой во многих системах проектирования, в частности и в MAX+PLUS II фирмы «Altera». Список переменных называется списком чувствительности (sensitivity list), поскольку данная конструкция чувствительна к их изменениям. Данный термин нам уже известен из описания языка VHDL.

Ключевое слово always @ (expr or expr ...).

```
case ({cntrl2, cntrl1})  
    2'b00 : out = in1;  
    2'b01 : out = in2;  
    2'b10 : out = in3;  
    2'b11 : out = in4;
```

```
default : $display(«Please check control bits»);  
endcase  
endmodule
```

Синтаксис оператора выбора case в Verilog сходен с синтаксисом оператора выбора в языке C++. Условием является конкатенация или объединение переменных cntl2 и cntl1 в двухразрядное число. Завершает оператор выбора ключевое слово endcase.

Здесь нелишне напомнить, что следует различать процедурное и непрерывное назначение сигналов, что и иллюстрируют вышеприведенные примеры.

5.20.4. Реализация с использованием условного оператора (Conditional operator implementation)

В принципе ничего нового мы уже не видим — ясно видно, что в нашем случае условный оператор проигрывает оператору CASE в наглядности представления (хотя для большинства систем проектирования синтезируемые реализации окажутся идентичными).

```
module multiplexor4_1 (out, in1, in2, in3,  
    in4, cntl1, cntl2);  
    output out;  
    input in1, in2, in3, in4, cntl1, cntl2;  
  
    assign out = cntl1 ? (cntl2 ? in4 : in3) :  
        (cntl2 ? in2 : in1);  
  
endmodule
```

5.20.5. Тестовый модуль (The stimulus module)

Вообще говоря, первоначально язык Verilog задумывался как язык верификации цифровых устройств (Verify Logic), поэтому одной из мощных возможностей языка является наличие средств для задания тестовых сигналов. Ниже приводится пример такого модуля для нашего мультиплексора 4 в 1.

```
module muxstimulus;
```

```
    reg IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;
    wire OUT;

    multiplexor4_1 mux1_4(OUT, IN1, IN2, IN3, IN4,
        CNTRL1, CNTRL2);

    initial begin
        IN1 = 1; IN2 = 0; IN3 = 1; IN4 = 0;
        $display(«Initial arbitrary values»);
        #0 $display(«input1 = %b, input2 = %b, input3 = %b,
            input4 = %b\n», IN1, IN2, IN3, IN4);

        {CNTRL1, CNTRL2} = 2'b00;
        #1 $display(«cntrl1=%b, cntrl2=%b, output is %b»,
            CNTRL1, CNTRL2, OUT);
        {CNTRL1, CNTRL2} = 2'b01;
        #1 $display(«cntrl1=%b, cntrl2=%b output is %b»,
            CNTRL1, CNTRL2, OUT);

        {CNTRL1, CNTRL2} = 2'b10;
        #1 $display(«cntrl1=%b, cntrl2=%b output is %b»,
            CNTRL1, CNTRL2, OUT);

        {CNTRL1, CNTRL2} = 2'b11;
        #1 $display(«cntrl1=%b, cntrl2=%b output is %b»,
            CNTRL1, CNTRL2, OUT);

        end

    endmodule
```

Рассмотрим данный пример подробнее.

```
module muxstimulus;
```

Поскольку наш генератор тестов является модулем верхнего уровня иерархии (top level module), то в его описании отсутствует список портов. Для задния модуля используется ключевое слово module.

```
reg IN1, IN2, IN3, IN4, CNTRL1, CNTRL2;  
wire OUT;
```

В данном случае мы обеспечиваем подачу тестовых сигналов на входы нашего мультиплексора и снятие сигнала с его выхода, следовательно, мы должны назначить сигналы для входов мультиплексора и сигнал, который управляется (driven) его выходом. Поэтому и использованы данные типа `reg` для входов и `wire` для выхода.

```
multiplexor4_1 mux1_4(OUT, IN1, IN2, IN3,  
IN4, CNTRL1, CNTRL2);
```

В этой строке происходит обращение к тестируемому модулю `multiplexor4_1`, в таких случаях принят следующий синтаксис:

```
<module_name> <instance_name> (port list);
```

Имя экземпляра (instance name) очень похоже на понятие языка VHDL и является необходимым при обращении к определяемым пользователем модулям (user defined modules), дабы не нарушать иерархию проекта.

Список портов (port list) устанавливает соответствие между сигналами тестового и тестируемого модулей, при этом порядок переменных в списке портов должен соответствовать порядку их объявления в тестовом модуле.

```
initial begin  
    IN1 = 1; IN2 = 0; IN3 = 1; IN4 = 0;  
    $display("Initial arbitrary values");  
    #0 $display("input1 = %b, input2 = %b, input3 = %b,  
input4 = %b\n", IN1, IN2, IN3, IN4);
```

Собственно моделирование осуществляется конструкцией, определяемой ключевыми словами `initial begin ... end`. Она предназначена для объединения инструкций, которые могут выполняться одновременно.

Перед инициализацией процесса моделирования сообщение об этом выводится с помощью `$display`. Конструкция `#0` перед директивой вывода означает, что вывод на экран осуществляется после назначения сигналов на входы. Синтакс директивы `$display` подобен синтаксису функции `printf` в языке C++.

```
$display( expr1, expr2, ..., exprN);
```

Expr N может быть переменной, выражением или строкой.

```
{CNTRL1, CNTRL2} = 2'b00;
#1 $display("cntrl1=%b, cntrl2=%b, output is %b",
CNTRL1, CNTRL2, OUT);
```

Здесь осуществляется назначение сигналов управления.

```
CNTRL1 = 0;
CNTRL2 = 0;
```

Оператор конкатенации (concatenation operator) { } может быть использован для группового назначения. Следует помнить, что число разрядов в числе и переменной должно совпадать.

```
{CNTRL1, CNTRL2} = 2'b01;
#1 $display("cntrl1=%b, cntrl2=%b output is %b",
CNTRL1, CNTRL2, OUT);
```

```
{CNTRL1, CNTRL2} = 2'b10;
#1 $display("cntrl1=%b, cntrl2=%b output is %b",
CNTRL1, CNTRL2, OUT);
```

```
{CNTRL1, CNTRL2} = 2'b11;
#1 $display("cntrl1=%b, cntrl2=%b output is %b",
CNTRL1, CNTRL2, OUT);
```

```
end
endmodule
```

В этом фрагменте сигналы управления изменяются, и изменение выхода отслеживается директивой display. В этом примере директива \$display выполняется с некоторой задержкой относительно сигналов управления, что позволяет отобразить корректные значения.

5.21. Модули проекта (Design blocks modules)

Рассмотрим правила построения иерархического проекта. В качестве примера мы будем рассматривать триггер с разрешением. В Verilog возможны два подхода для его реализации:

— сверху вниз (top down) — начать с описания Е-триггера и далее добавлять детали проекта;

— снизу вверх (bottom up) — начать с базовых блоков, а затем объединить их в единый проект.

Мы воспользуемся технологией снизу вверх, хотя, наверное, наиболее удобно использование обоих подходов. Начнем с описания D-триггера как модуля.

Ключевые слова для задания модуля: `module <module_name> ... endmodule`, as below. Ниже приведено его описание.

```
module dff(q, data, clock);
    output q;
    input  data, clock;
    reg q;

    always @(posedge clock)
        q = data;
endmodule // dff
```

Аналогичное поведенческое описание мультиплексора с инверсией (inverting multiplexor) имеет вид:

```
module mux2_1(out, in1, in2, cntrl);
    output out;
    input  in1, in2, cntrl;

    assign out = cntrl ? ~in1 : ~in2;
endmodule // mux2_1
```

Тогда описание модуля верхнего уровня для Е-триггера, состоящего из D-триггера и мультиплексора, принимает вид:

```
module e_ff(q, data, enable, reset, clock);
    output q;
    input data, enable, reset, clock;
    wire norout, muxout;

    mux2_1 mod1 (muxout, data, q, enable);
```

```
        nor (norout, reset, muxout);
        dff dff0 (q, norout, clock);
    endmodule
```

Обратим еще раз внимание на обращение к модулю, определяемому пользователем, через имя экземпляра модуля:

```
name_of_module instance_name (port_list);
```

5.21.1. Тестирование

Ниже приведен текст модуля для тестирования нашего триггера. Как и в примере с мультиплексором, он не имеет списка портов, поскольку является самым верхним в иерархии.

```
module e_ffStimulus;
    reg data, enable, reset, clock;
    wire q;

    e_ff mod1 (q, data, enable, reset, clock);

    initial begin
        clock = 1'b0;
        forever clock = #5 ~clock;
    end

    initial begin
        enable = 1'b0;      // переменная разрешения
        reset = 1'b1;       // активный уровень сброса - 1
        #20 reset = 1'b0;   // сброс
        data = 1'b1;        установка в HIGH
        #10 enable = 1'b1;  // and then enable data latching
        #10 data = 1'b1;    // изменение данных
        #20 data = 1'b0;    // изменение данных
        #30 data = 1'b1;    // изменение данных
        #10 data = 1'b0;    // изменение данных
        #10 data = 1'b1;    // изменение данных
        #20 enable = 1'b0;  // запрет защелкивания
    end
endmodule
```



```
#10 data = 1'b0;    // изменение данных
#10 reset = 1'b1;   // сброс снова
#20 $finish;        // все
    end             // такт остановлен
    initial begin
        $display($time, "reset, enable, data, q");
        $display($time, "%d %d %d %d",
            reset, enable, data, q);
        forever #10 $display($time, "%d %d %d %d",
            reset, enable, data, q);
    end

    endmodule // e_ffStimulus
```

5.22. Порты (Ports)

Порты обеспечивают связь модуля с другими модулями проекта. Ниже приведены порты для нашего примера.

```
module d_ff(q, d, reset, clock);
module e_ff(q, d, enable, reset, clock);
module Stimulus;
```

Каждый порт может быть объявлен как вход, выход или двунаправленный. Все объявленные порты по умолчанию назначаются в цепи, для изменения типа сигнала необходимо выполнить назначение отдельно.

Например:

```
module d_ff(q, d, reset, clock);
output q;    // обязательное объявление портов
input d, reset, clock; // как входы и выходы
reg q;       // снова объявляем как регистр
```

Не забываем, что выходы модуля идут первыми в списке портов.

5.23. Правила соединения (Connection rules)

Выше мы рассматривали два типа модулей: внешние и внутренние (outer and inner), в нашем примере внешним модулем был Е-триггер, внутренним — триггер типа D. Ниже рассмотрим основные правила соединения модулей.

5.23.1. Входы (inputs)

Входы внутреннего модуля всегда должны иметь тип `wire`, поскольку они управляются внешними сигналами. Входы внешнего модуля могут иметь тип как `wire`, так и регистр.

5.23.2. Выходы (outputs)

Во внутреннем модуле выходы могут иметь тип как цепь, так и регистр, в то время как выходы внешнего модуля должны быть типа цепь, поскольку управляются внутренними модулями.

5.23.3. Двухнаправленные выводы (inouts)

Двухнаправленный порт всегда имеет тип цепь.

5.23.4. Соответствие портов (Port matching)

Очевидно, обращаясь к модулю, необходимо, чтобы совпадала разрядность портов.

5.23.5. Присоединение портов (Connecting ports)

Присоединение портов осуществляется либо по порядку (ordered list), либо по имени порта (or by name). Например:

```
module d_ff( q, d, reset, clock);  
    ...  
endmodule  
  
module e_ff(q, d , enable, reset, clock);  
    output q;  
    input d, enable, reset, clock;  
    wire inD;  
    ...  
endmodule
```

```
        d_ff dff0(q, inD, reset, clock);  
        ...  
endmodule
```

5.24. Базовые блоки (Basic blocks)

В Verilog принято два типа назначений: непрерывное и процедурное.

Непрерывное назначение может быть выполнено для цепей или для их объединений (concatenation of nets). Операнды могут иметь произвольный тип данных.

Процедурное назначение может быть выполнено для данных типов reg, INTEGER, REAL или Time.

5.24.1. Инициализация (Initial block)

Ключевое слово: initial.

Блок инициализации состоит из группы операторов, заключенных в операторные скобки begin... end, которые будут выполняться с момента старта моделирования. Ниже приведен пример инициализации моделирования.

```
initial  
clock = 1'b0;  
  
initial  
begin  
    alpha = 0;  
    #10 alpha = 1;    // генерация сигнала  
    #20 alpha = 0;  
    #5  alpha = 1;  
    #7  alpha = 0;  
    #10 alpha = 1;  
    #20 alpha = 0;  
end;
```

5.24.2. Конструкция Always (Always block)

Ключевое слово: always.

Блок Always означает, что действие выполняется непрерывно, пока

процесс моделирования не будет остановлен директивой \$finish или \$stop.

Примечание: Директива \$finish останавливает работу модулятора, тогда как директива \$stop вносит паузу.

Ниже приведен пример формирования тактового импульса с использованием конструкции always. Заметим, что, вообще говоря, для этих целей удобнее использовать циклы.

```
module pulse;

    reg clock;
    initial clock = 1'b0; // запуск такта
    always #10 clock = ~clock; // такт
    initial #5000 $finish // конец моделирования

endmodule
```

5.25. Пример проектирования последовательностного устройства: двоичный счетчик

Рассмотрим пример проектирования двоичного счетчика, считающего от 0 до 12 и по достижении 12 сбрасывающегося в 0 и начинающего счет заново. Рассмотрим описание на вентиляном уровне.

```
// 4-bit binary counter

    module counter4_bit(q, d, increment, load_data,
global_reset, clock);
    output [3:0] q;
    input [3:0] d;
    input load_data, global_reset, clock, increment;
    wire t1, t2, t3; // internal wires
    wire seel2, reset; // internal wires

    et_ff etff0(q[0], d[0], increment, load_data, reset,
clock);
    et_ff etff1(q[1], d[1], t1, load_data, reset, clock);
```

```
et_ff etff2(q[2], d[2], t2, load_data, reset, clock);
et_ff etff3(q[3], d[3], t3, load_data, reset, clock);

and a1(t1, increment, q[0]);
and a2(t2, t1, q[1]);
and a3(t3, t2, q[2]);
    recog12 r1(see12, q); // счет
    or or1(reset, see12, global_reset); // сброс

endmodule // counter4_bit

module et_ff(q, data, toggle, load_data, reset, clock);
    output q;
    input  data, toggle, load_data, reset, clock;
    wire m1, m2;

    mux mux0(m1, ~q, q, toggle);
    mux mux1(m2, data, m1, load_data);
    dff dff0(q, m2, reset, clock);

endmodule // et_ff

module mux(out, in1, in2, cntrl);
    output out;
    input  in1, in2, cntrl;
    assign out = cntrl ? in1 : in2;

endmodule // mux

module dff(q, data, reset, clock);
    output q;
    input data, reset, clock;
    reg q;

    always @(posedge clock) // at every clock
edge, if reset is 1, q is
    if (reset == 1)
```

```

        q = 0;
    else q = data;

    endmodule

    module recog12(flag, in);
        input [3:0] in;
        output flag;

        assign flag = (in == 4'b1100) ? 1 : 0;

    endmodule // recog12

    module stumulus;
        wire [3:0] q;
        reg [3:0] d;
        reg load_data, global_reset, clk, increment;

    counter4_bit mod1 (q, d, increment, load_data, global_reset, clk);

    initial begin
        global_reset = 0;
        clk = 0;
        increment = 0;
        load_data = 0;
        d = 4'b0100;

        #10 global_reset = 1;
        #20 global_reset = 0;
        #20 load_data = 1;
        #20 load_data = 0;
        #20 increment = 1;
        #200 global_reset = 1;
        #20 global_reset = 0;
        #50 load_data = 1;
        #20 load_data = 0;
        #10 increment = 0;
    end

```

```
        #20 $finish;
    end // initial begin
        always #5 clk = ~clk;

    always #10 $display ($time,"%b %b %b %d -> %b %d",
        increment, load_data, global_reset, d, q, q);
    endmodule // stumulus
```

Начнем подробное рассмотрение этого примера.

```
module counter4_bit(q, d, increment, load_data, glob-
al_reset, clock);
output [3:0] q;
input [3:0] d;
input load_data, global_reset, clock, increment;
wire t1, t2, t3; // internal wires
wire see12, reset; // internal wires
```

Как нам уже известно, первые строки модуля содержат его имя и объявление портов. Выход счетчика *q*, все остальные входные сигналы, *t1*, *t2*, *t3*, *see12* и *RESET* объявлены как внутренние цепи, подключенные к входам, соответствующим субмодулям проекта.

```
et_ff etff0(q[0], d[0], increment, load_data, reset,
clock);
et_ff etff1(q[1], d[1], t1, load_data, reset, clock);
et_ff etff2(q[2], d[2], t2, load_data, reset, clock);
et_ff etff3(q[3], d[3], t3, load_data, reset, clock);
```

В этом фрагменте кода мы определяем 4 счетных триггера с разрешением.

```
and a1(t1, increment, q[0]);
and a2(t2, t1, q[1]);
and a3(t3, t2, q[2]);
```

Сигнал увеличения счетчика (increment signal) объединяется по «И» с выходом последнего триггера для переноса сигнала счета на следующий триггер.

```
recog12 r1(see12, q);  
or or1(reset, see12, global_reset);
```

Каждый раз, когда выход q изменяется, переменная see12 модифицируется. Когда она достигнет 12, внутренний сигнал сброса обеспечит сброс счетчика на очередном такте. Далее рассмотрим поведенческую модель нашего счетчика.

5.25.1. Поведенческая модель счетчика (Behavioural model)

Предшествующее описание счетчика было осуществлено на уровне вентилей и триггеров, т.е. на структурном уровне. С другой стороны, при наличии достаточно мощных средств синтеза поведенческое описание позволяет получить более наглядное описание проекта. Ниже приводится пример поведенческого описания (Behavioural description) счетчика.

```
// 4-bit binary up-counter - of period 13  
  
module counter4_bit(q, d, increment, load_data, global_reset, clock);  
    output [3:0] q;  
    input [3:0] d;  
    input load_data, global_reset, clock, increment;  
  
    reg [3:0] q;  
    always @(posedge clock)  
        if (global_reset)  
            q = 4'b0000;  
        else if (load_data)  
            q = d;  
        else if (increment) begin  
            if (q == 12)  
                q = 0;  
            else
```



```
    q = q + 1;
end
endmodule // counter4_bit
```

В отличие от описания на регистровом уровне различие в первых строках только одно — выход *q* объявлен как регистр. Данная модель позволяет путем проведения небольших модификаций получить описание другого типа счетчика — счетчика в обратном направлении (down counter). Ниже приведен пример такого описания.

```
// 4-bit binary down-counter - of period 13
module counter4_bit(q, d, decrement,
load_data, global_reset, clock);

    always @(posedge clock)
        if (global_reset)
            q = 12;
        else if (load_data)
            q = d;
        else if (decrement)
            begin
                if (q == 0)
                    q = 12;
                else
                    q = q - 1;
            end

endmodule
```

Далее рассмотрим реверсивный счетчик (счетчик с переменным направлением счета, up-down counter) с периодом 16.

```
// 4-bit binary up-down-counter - of period 16
module counter4_bit(q, d, updown, count, load_data,
global_reset, clock);

    always @(posedge clock)
        if (reset)
```

```
        q = 0;
    else if (load_data)
        q = d;
    else if (count)
        begin
            if (updown)
                q = q + 1;
            else
                q = q - 1;
        end
endmodule
```

5.26. Временной контроль (Timing control)

В языке Verilog существует три варианта осуществления временного контроля (timing control): использование задержек (delay), событий (event) и уровни чувствительности (level sensitive). Рассмотрим их подробнее.

5.26.1. Задержки (delay)

Синтакс: `timing_control_statement ::= delay_based statement*`

`delay_based ::= # delay_value`

В данном методе вводится задержка между поступлением переменной на вход блока и результатом.

```
initial begin
    a = 0;          // выполняется в момент t = 0
    #10 b = 2;      // выполняется в момент t = 10
    #15 c = a;      // выполняется в момент t = 25
    #b c = 4;       // выполняется в момент t = 27
    b = 5;          // выполняется в момент t = 27
end
```

Величина задержки может быть определена как постоянная или переменная. Следует помнить, что системное время измеряется в тех единицах, которые установлены в системе. Ниже приводится пример формирования тактового сигнала (creation of a clock signal).

```
initial begin
    clock = 1'b0;
    forever #5 clock = ~clock;
end
```

В данном примере каждые 5 единиц времени сигнал переходит из ВЫСОКОГО уровня в НИЗКИЙ и наоборот.

5.26.2. Событийный контроль (event-based control)

При событийном контроле справедлив следующий синтаксис оператора события (в форме Бэкуса—Науэра):

```
event_control_statement ::=
@ event_identifier
| @ (event_expression)
event_expression ::=
| exp
| event_id
| posedge exp
| negedge exp
| event_exp or event_exp.
```

При событийном контроле происходит вычисление результата при каждом изменении входных сигналов.

```
@ (clock) a = b; // при каждом такте выполняется a = b
@ (negedge clock) a = b; // когда clock -> 0, то a = b
a = @(posedge clock) b;
```

5.27. Защелкивание (triggers)

Функционирование защелки рассмотрим на примере.

```
event data_in;
always @(negedge clock)
    if (data[8]==1) -> data_in;
```

```
always @(data_in) mem[0:1] = buf;
```

В этом примере в каждом отрицательном фронте синхроимпульса проверяется, равно ли data[8] единице, если так, то data_in = 1.

5.28. Список сигналов возбуждения (sensitivity list)

Синтакс: event_list_statement::=@ (event_exp or event_exp)

event_exp::= (event_exp or event_exp).

Если мы желаем, чтобы блок выполнялся тогда, когда изменилась любая из входных переменных, то список сигналов возбуждения разделяется операторами OR.

```
always @ (i_change or he_changes or she_changes)
somebody_changed = 1;
```

Здесь изменение хотя бы одной переменной приводит к изменению остальных.

5.29. Задержка распространения в вентиле (Gate delays)

Синтакс: delay_statement::= gate_name delay_exp gate_details

delay_exp::= # delay_time

Данный тип задержки применим только к простейшим вентилям, определяемым в Verilog.

```
and #(5) a1(out, in1, in2);
```

5.30. Операторы ветвления (Branch statements)

5.30.1. Оператор IF (IF statement)

Синтакс: if (conditional_expression) statement {else statement}.

Как известно, оператор IF является оператором условного перехода и выполняется стандартным для языков высокого уровня способом. Условие вычисляется, и в зависимости от результата выполняется либо первый опе-

ратор, либо второй. Группы операторов в ветви выделяются операторными скобками (как в языке Паскаль) `begin... end`. Рассмотрим реализацию нашего мультиплексора с помощью оператора `IF`.

```
module multiplexor4_1 (out, in1, in2, in3 ,in4,
cntrl1, cntrl2);
    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;

    assign out = (~cntrl2 & ~cntrl1 & in1) |
                 ( cntrl2 & ~cntrl1 & in2) |
                 (~cntrl2 &  cntrl1 & in3) |
                 ( cntrl2 &  cntrl1 & in4) ;

endmodule
```

Рассмотрим пример реализации мультиплексора 4 в 1 с помощью логических уравнений.

```
module multiplexor4_1 (out, in1, in2, in3,in4,
cntrl1, cntrl2);
    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;
    reg out;

    always @(in1 or in2 or in3 or in4 or cntrl1
or cntrl2)
        if (cntrl1==1)
            if (cntrl2==1)
                out = in4;
            else out = in3;
        else
            if (cntrl2==1)
                out = in2;
            else out = in1;

endmodule
```

5.30.2. Оператор выбора (CASE statement)

Рассмотренный выше метод реализации мультиплексора удобен при малом числе входов, при их увеличении гораздо разумнее использовать оператор выбора, причем его синтаксис аналогичен оператору выбора в C++.

Синтакс: `conditional::= case (condition) case_item+ endcase`
`case_item::= expression+ (seperated by commas) : statement* | default : statement*`

Рассмотрим реализацию мультиплексора 4 в 1 с помощью оператора выбора.

```
module multiplexor4_1 (out, in1, in2, in3, in4,
cntrl1, cntrl2);
    output out;
    input in1, in2, in3, in4, cntrl1, cntrl2;
    reg out; // note must be a register

    always @(in1 | in2 | in3 | in4 | cntrl1 |
cntrl2)
        case ({cntrl2, cntrl1}) // concatenation
            2'b00 : out = in1;
            2'b01 : out = in2;
            2'b10 : out = in3;
            2'b11 : out = in4;
            default : $display("Please check control
bits");
        endcase
    endmodule
```

Вариантами оператора выбора являются операторы `casez` и `casex`.

5.30.3. Оператор ветвления (Conditional operator)

Оператор по синтаксису анлогичен такому же оператору в C++, так же, как и в C++ этот оператор может быть заменен конструкцией IF-ELSE.

Синтакс: `conditional_expression ? true_expression : false_expression`

Пример реализации мультиплексора 2 в 1:

```
assign out = control ? in2 : in1;
```

5.31. Циклы (Looping constructs)

В Verilog существует четыре типа циклов — конструкции WHILE, FOR, REPEAT и FOREVER. Циклы возможно использовать только внутри блоков Initial и Always. Основное назначение циклов:

- WHILE — выполнение операторов, пока условие истинное;
- FOR — в этом цикле возможно инициализировать, проверять и увеличивать индексную переменную явным способом;
- REPEAT — цикл с заданным числом повторений;
- FOREVER — вечный цикл: повторение до конца процесса моделирования.

5.31.1. Цикл WHILE (WHILE LOOP)

Синтакс: `looping_statement::= while (conditional) statement`

Цикл WHILE выполняется до тех пор, пока условие истинно, в качестве условия может быть использовано любое логическое выражение. Операторы в цикле группируются с использованием операторных скобок `begin ... end`. Ниже приведен пример WHILE.

```
/* How many cups of volume 33 ml does it
   take to fill a jug of volume 1 litre */

'define JUGVOL 1000
'define CUPVOL 33

module cup_and_jugs;
    integer cup, jug;

    initial begin
        cup = 0; jug = 0;
        while (jug < `JUGVOL) begin
            jug = jug + `CUPVOL;
            cup = cup + 1;
        end // while (jug < JUG_VOL)
        $display("It takes %d cups", cup);
    end
endmodule
```

```
end // initial begin

endmodule // cup_and_jugs
Notice the use of the "define" statement.
```

5.31.2. Цикл FOR (FOR LOOP)

Синтакс: FOR (reg_initialisation ; conditional ; reg_update) statement

Цикл FOR в Verilog аналогичен циклу FOR в языке C++. Ниже приведен пример его использования.

```
integer list [31:0];
integer index;

initial begin
    for(index = 0; index < 32; index = index + 1)
        list[index] = index + index;
end
```

5.31.3. Цикл REPEAT (REPEAT LOOP)

Синтакс: looping_statement::= repeat (conditional) statement

Цикл REPEAT выполняется конечное число раз, условие (Conditional) может быть константой, переменной или сигналом, но обязательно быть целым числом. Ниже приведен пример цикла REPEAT.

```
module comply;
    int count;
    initial begin
        count = 128;
        repeat (count) begin
            $display("%d seconds to comply", count);
            count = count - 1;
        end
    end
endmodule
```

Примечание: Цикл будет выполняться 128 тактов независимо от величины счета, которая может изменяться после запуска самого цикла.

5.31.4. Вечный цикл (FOREVER LOOP)

Синтаксис: forever statement

Цикл FOREVER выполняется непрерывно, пока моделирование не будет остановлено командой \$finish. Ниже приведен пример вечного цикла.

```
reg clock;

initial begin
    clock = 1'b0;
    forever #5 clock = ~clock;
end

initial #2000 $finish;
```

5.32. Файлы в Verilog

5.32.1. Открытие файла (Opening a file)

Ключевые слова: \$open

Синтакс: <file_handle> = \$fopen(«<file_name>»);

В процессе моделирования в языке Verilog имеется возможность записывать результаты прогона в файл, что в дальнейшем позволяет провести его более подробный анализ. Ниже приведен пример открытия файлов.

```
integer handleA, handleB;
// an integer has 32 bits so is perfect
// for this usage

initial begin
    handleA = $fopen("myfile.out");
// handleA = 0000_0000_0000_0000_0000_0000_0000_0010
    handleB = $fopen("anotherfile.out");
// handleB = 0000_0000_0000_0000_0000_0000_0000_0100
end
```

Когда файл открыт, его можно использовать для записи. Всего допустимо открыть не более 31 файла одновременно.

5.32.2. Запись в файл (Writing to a file)

Ключевые слова: \$fdisplay, \$fmonitor, \$fwrite

Синтакс: \$fdisplay(<file_handles>, --same as display--);

\$fmonitor(<file_handles>, --same as monitor--);

\$fwrite(<file_handles>, --same as write--);

Для записи может быть открыто несколько файлов, ниже приведен пример записи числа 0000_0000_0000_0000_0000_0000_0011 в файлы.

```
integer channelsA;

initial begin
    channelsA = handleA | 1;
    $fdisplay(channelsA, "Hello");
end
```

5.32.3. Закрытие файла (Closing a file)

Ключевое слово: \$fclose

Синтакс: \$fclose(handle);

Когда используется несколько файлов, то обычно файлы, которые уже не требуются, закрывают для облегчения работы операционной системы. Для этого и служит директива \$fclose, после закрытия работа с данным файлом, естественно, невозможна.

5.32.4. Инициализация регистровых файлов (памяти) (Initialising memories)

Ключевые слова: \$readmemb, \$readmemh

Синтакс: \$readmemb(<«file_name»>, <memory_name>»);

\$readmemb(<«file_name»>, <memory_name>, memory_start»);

\$readmemb(<«file_name»>, <memory_name>, memory_start, memory_finish»);

Для инициализации регистровых файлов используется следующая конструкция. Содержимое регистрового файла хранится в файле с форматом, приведенном ниже, все адреса шестнадцатеричные.

```
@003
00000011
00000100
```

```
00000101
00000110
00000111
00001000
00001001
```

Для улучшения читаемости возможна и такая запись:

```
@003
00000011
00000100
00000101
@006
00000110
00000111
@008
00001000
00001001
```

Пример инициализации памяти приведен ниже.

```
module testmemory;
    reg [7:0] memory [9:0];
    integer index;

    initial begin
        $readmemb("mem.dat", memory);

        for(index = 0; index < 10; index = index + 1)
            $display("memory[%d] = %b", index[4:0],
                memory[index]);
    end
endmodule // testmemory
```

Содержимое файла mem.data имеет вид:

```
1000_0001
1000_0010
0000_0000
0000_0001
0000_0010
0000_0011
```

```
0000_0100
0000_0101
0000_0110
0000_0000
```

5.33. Задание векторов входных сигналов для моделирования (Verilog input vectors)

Для задания входных векторов для моделирования системы требуется весьма ограниченное подмножество языка Verilog. Рассмотрим простой пример.

```
initial
begin
    clk = 1'b0;
    globalReset = 1'b1;
    in = 1'b1;
end
```

Выражение 1'b0 представляет собой двоичное представление 0. Существуют две основные части:

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

Блок Initial начинает выполняться с началом моделирования, поэтому в нем инициализируются сигналы и определяется последовательность их изменений.

Тактовый сигнал существует в течение всего времени моделирования. Задержка между моментами смены сигналов (событиями) определяется с помощью знака #.

Например:

```
#160 globalReset = 0;
#160 in = 0;
#160 in = 1;
#320 in = 0;
```

В приведенном примере после 160 единиц времени ожидания сигнал `globalReset` устанавливается в 0, затем 160 единиц ожидания и установка сигнала `in` в 0 и т.д.

Наиболее часто конструкция `always` используется для задания тактового сигнала.

Например:

```
always
begin
    #10 clk = ~clk;
end
```

Здесь сигнал `clk` должен быть проинвертирован каждые 10 единиц измерения времени, что, по сути, означает, что период тактового сигнала — 20 единиц измерения времени (time units). Кроме того, в процессе моделирования неплохо использовать цикл `REPEAT` для задания заданного числа повторений.

```
repeat(10)
begin
    #40 in = 0;
    #20 in = 1;
end
```

Для генерации случайных чисел в диапазоне `[0, n-1]` используется следующая команда

```
{ $random } % n
```

Комбинируя эти команды, создадим генератор пачек из 14 случайных чисел каждые 20 единиц времени:

```
repeat (14)
begin
    #20 in = {$random} / 16 % 2;
end
```

5.34. Список операторов Verilog

Дабы обобщить рассмотренные ранее ключевые моменты работы с

Таблица 5.2. Список операторов языка Verilog

Тип оператора	Символ	Операция	Число операндов
Арифметические (Arithmetic)	*	Умножение	2
	/	деление	2
	+	сумма	2
	-	разность	2
	%	остаток	2
Логические (Logical)	!	«НЕ»	1
	&&	«И»	2
		«ИЛИ»	2
Отношения (Relational)	>	больше чем	2
	<	меньше чем	2
	>=	больше либо равно	2
	<=	меньше либо равно	2
Эквивалентности (Equality)	=	равенство	2
	!=	неравенство	2
	=	выборочное равенство	2
	!=	выборочное неравенство	2
Поразрядные (Bitwise)	~	поразрядное отрицание	1
	&	поразрядное «И»	2
		поразрядное «ИЛИ»	2
	^	поразрядное «исключающее ИЛИ»	2
	^~or^	поразрядное «исключающее ИЛИ-НЕ»	2
Приведение (Reduction)	&	приведенное «И»	1
	&~	приведенное «И-НЕ»	1
	~&	приведенное «ИЛИ»	1
		приведенное «ИЛИ-НЕ»	1
	~	приведенное «исключающее ИЛИ»	1
	^~or^	приведенное «исключающее ИЛИ-НЕ»	1
Сдвиг (SHIFT)	>>	правый сдвиг	2
	<<	левый сдвиг	2

Таблица 5.2 (окончание)

Тип оператора	Символ	Операция	Число операндов
Конкатенация (Concatenation)		объединение	любое
Репликация (Replication)		повторение	любое
Условный (Conditional)	?:	условный	3

языком, приведем список его операторов (Табл. 5.2).

5.35. Приоритет операторов

[illegible]

Ниже рассматривается приоритет операторов Verilog.

5.36. Ключевые слова (keywords)

Ниже приведены ключевые слова языка Verilog в алфавитном порядке. Естественно, их нельзя использовать для имен вентилей, модулей, портов и т.п.

always	and	assign	attribute	begin	buf	bufif0
bufif1	case	casex	casez	cmos	deassign	default

defram disable edge else end endattribute endcase
 endfunction endmodule endprimitive endspecify endtable
 endtask event for force forever fork function highz0
 highz1 if initial inout input integer join large
 macromodule meduim module nand negedge nmos nor
 not notif0 notif1 or output arameter pmos posedge
 primitive pull0 pull1 pulldown pullup rcmos real
 realtime reg release repeat rtranif1 scaled signed
 small specify specpram strength strong0 strong1 sup-
 ply0 supply1 table task time tran tranif0 tranif1 tri
 tri0 tri1 triand trior tireg unsigned vectored wait
 wand weak0 weak1 while wire wor xnor xor

5.37. Директивы компилятора

\$bitstoreal \$countdrivers \$display \$fclose \$fdisplay
 \$fmonitor \$fopen \$fstrobe \$fwrite \$finish \$getpattern
 \$history \$incsave \$input \$itor \$key \$list \$log
 \$monitor \$monitoroff monitoron \$nokey

5.38. Типы цепей (Net types)

supply0 supply1 tri triand trior tireg tri0 tri1
 wand wire wor

Глава 6. **Примеры проектирования цифровых устройств с использованием языков описания аппаратуры VHDL и Verilog**

6.1. Общие сведения

Языки описания аппаратуры VHDL и Verilog HDL представляют собой языки высокого уровня (в отличие от AHDL и ABEL HDL) и предназначены для описания цифровых схем и их элементов в первую очередь на поведенческом уровне.

Эти языки поддерживают различные уровни абстракции проекта, включая поддержку особенностей архитектуры ПЛИС, под которую выполняется проект (Architecture-Specific Design). Использование языков описания аппаратуры высокого уровня позволяет проектировать цифровые системы, обладающие высокой мобильностью, то есть возможностью переносимости на другую элементную базу. Для создания законченного проекта необходимо только лишь произвести компиляцию в соответствующей системе в элементную базу выбранного производителя. Но в этом и основной недостаток языков высокого уровня — недостаточный учет специфических архитектурных особенностей используемой элементной базы (specific target technology). В этом разделе мы попытаемся рассмотреть некоторые примеры использования языков описания аппаратуры для создания проектов различного уровня сложности.

При проведении синтеза логической структуры ПЛИС с использованием языков описания аппаратуры (HDL Synthesis-Based Design Flow) различают четыре основные стадии проектирования:

- создание и функциональная верификация проекта;
- реализация проекта в САПР ПЛИС;
- программирование ПЛИС;
- верификация всей системы.

При функциональной верификации проекта ввод описания проекта осуществляется на регистровом уровне (RTL-level) в поведенческой области (behavioral). После ввода описания проекта поведенческое (функциональное) моделирование (верификация) позволяет оценить степень правильности функционирования алгоритма. После проведения функционального моделирования описание синтезируется в список цепей на вентиляном уровне (GATE-level) в структурной области (structural). После осуществления синтеза можно выполнить структурное (временное и функциональное) моделирование устройства. В результате мы получаем список цепей (как правило, в формате EDIF) для временной верификации проекта.

Рассмотрим применение языков описания аппаратуры высокого уровня (VHDL и Verilog HDL) для описания различных цифровых устройств. Данные описания, если не особо указывается, не ориентированы на какую-либо конкретную систему проектирования или семейство ПЛИС и могут быть воплощены в различных базисах. Такой тип описаний получил название независимого от технологии стиля описания устройств (Technology Independent Coding Styles). Начнем рассмотрение способов описания независимых от технологии устройств с последовательностных устройств (Sequential Devices). В англоязычной литературе приняты термины защелка для устройств, тактируемых уровнем, и триггер — для устройств, тактируемых фронтом тактового импульса.

6.2. Триггеры и регистры

Для описания триггерных схем в VHDL используются операторы WAIT и IF вместе с процессом, использующим атрибуты переднего или заднего фронтов синхроимпульса (см. главу 4). Ниже приведены примеры создания описаний срабатывания по фронту:

(clk'event and clk='1') — атрибут срабатывания по переднему фронту
(clk'event and clk='0') — атрибут срабатывания по заднему фронту
rising_edge(clock) — вызов функции по переднему фронту
falling_edge(clock) — вызов функции по заднему фронту

В этих примерах иллюстрируется применение атрибута переднего фронта (rising edge 'event attribute). Использование атрибутов следует рекомендовать в тех случаях, когда система проектирования не поддерживает

вызов функции по событию. Однако использование функций позволяет избежать коллизий, связанных с переходом из неопределенного состояния, поскольку функция определяет только переходы уровней (из «0» в «1» или из «1» в «0»), а не переход из неопределенного состояния в «1» или «0». Это становится достаточно важным в случае использования многозначных типов данных, например `std_logic`, который имеет 9 возможных значений ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'). Далее рассмотрим основные типы триггеров.

6.2.1. Триггеры, тактируемые передним фронтом (Rising Edge Flipflop)

Ниже приводится пример описания D-триггера без цепей асинхронного сброса (RESET) или предустановки (PRESET). На **Рис. 6.1** приведено схемное обозначение рассматриваемого триггера.

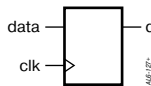


Рис. 6.1. Схемное обозначение D-триггера

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
port (data, clk : in std_logic;
      q : out std_logic);
end dff;
architecture behav of dff is
begin
  process (clk) begin
    if (clk'event and clk = '1') then
      q <= data;
    end if;
  end process;
end behav;
```

Пример описания на Verilog:

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

6.2.2. Триггеры, тактируемые передним фронтом, с асинхронным сбросом (Rising Edge Flipflop with Asynchronous Reset)

Обозначение устройства приведено на **Рис. 6.2**.

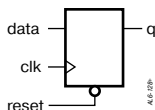


Рис. 6.2. Схемное обозначение триггера с асинхронным сбросом

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_async_rst is
port (data, clk, reset : in std_logic;
q : out std_logic);
end dff_async_rst;
architecture behav of dff_async_rst is
begin
process (clk, reset) begin
if (reset = '0') then
q <= '0';
elsif (clk'event and clk = '1') then
q <= data;
end if;
```

```
end process;
end behav;
```

Пример описания на Verilog:

```
module dff_async_rst (data, clk, reset, q);
input data, clk, reset;
output q;
reg q;
always @(posedge clk or negedge reset)
if (~reset)
q = 1'b0;
else
q = data;
endmodule
```

6.2.3. Триггеры, тактируемые передним фронтом, с асинхронной предустановкой (Rising Edge Filpflip with Asynchronous Preset)

Обозначение устройства на схеме приведено на **Рис. 6.3**.

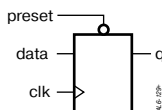


Рис. 6.3. Схемное обозначение триггера с асинхронной предустановкой

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_async_pre is
port (data, clk, preset : in std_logic;
q : out std_logic);
end dff_async_pre;
```

```
architecture behav of dff_async_pre is
begin
process (clk, preset) begin
if (preset = '0') then
q <= '1';
elsif (clk'event and clk = '1') then
q <= data;
end if;
end process;
end behav;
```

Пример описания на Verilog:

```
module dff_async_pre (data, clk, preset, q);
input data, clk, preset;
output q;
reg q;
always @(posedge clk or negedge preset)
if (~preset)
q = 1'b1;
else
q = data;
endmodule
```

6.2.4. Триггеры, тактируемые передним фронтом, с асинхронным сбросом и предустановкой (Rising Edge Filpflop with Asynchronous Reset and Preset)

Обозначение на схеме приведено на Рис. 6.4.

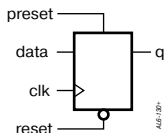


Рис. 6.4. Схемное обозначение триггера с асинхронным сбросом и предустановкой

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_async is
port (data, clk, reset, preset : in std_logic;
q : out std_logic);
end dff_async;
architecture behav of dff_async is
begin
process (clk, reset, preset) begin
if (reset = '0') then
q <= '0';
elsif (preset = '1') then
q <= '1';
elsif (clk'event and clk = '1') then
q <= data;
end if;
end process;
end behav;
```

Пример описания на Verilog:

```
module dff_async (reset, preset, data, q, clk);
input clk;
input reset, preset, data;
output q;
reg q;
always @(posedge clk or negedge reset or posedge
preset)
if (~reset)
q = 1'b0;
else if (preset)
q = 1'b1;
else q = data;
endmodule
```

6.2.5. Триггеры, тактируемые передним фронтом, с синхронным сбросом (Rising Edge Filpflop with Synchronous Reset)

Обозначение на схеме приведено на **Рис. 6.5**.

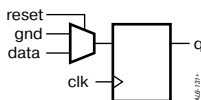


Рис. 6.5. Схема обозначения триггера с синхронным сбросом

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_sync_rst is
port (data, clk, reset : in std_logic;
q : out std_logic);
end dff_sync_rst;
architecture behav of dff_sync_rst is
begin
process (clk) begin
if (clk'event and clk = '1') then
if (reset = '0') then
q <= '0';
else q <= data;
end if;
end if;
end process;
end behav;
```

Пример описания на Verilog:

```
module dff_sync_rst (data, clk, reset, q);
input data, clk, reset;
output q;
reg q;
always @(posedge clk)
```



```

if (~reset)
q = 1'b0;
else q = data;
endmodule

```

6.2.6. Триггеры, тактируемые передним фронтом, с синхронной предустановкой (Rising Edge Filpflip with Synchronous Preset)

Обозначение на схеме приведено на **Рис. 6.6.**

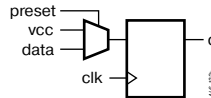


Рис. 6.6. Схемное описание триггера с синхронной предустановкой

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity dff_sync_pre is
port (data, clk, preset : in std_logic;
q : out std_logic);
end dff_sync_pre;
architecture behav of dff_sync_pre is
begin
process (clk) begin
if (clk'event and clk = '1') then
if (preset = '0') then
q <= '1';
else q <= data;
end if;
end if;
end process;
end behav;

```

Пример описания на Verilog:

```
module dff_sync_pre (data, clk, preset, q);
input data, clk, preset;
output q;
reg q;
always @(posedge clk)
if (~preset)
q = 1'b1;
else q = data;
endmodule
```

6.2.7. Триггеры, тактируемые передним фронтом, с асинхронным сбросом и разрешением тактового сигнала (Rising Edge Filpflip with Asynchronous Reset and Clock Enable)

Пример схемного обозначения приведен на Рис. 6.7.

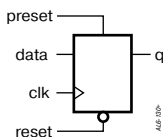


Рис. 6.7. Схемное описание триггера с асинхронным сбросом и разрешением тактового сигнала

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity dff_ck_en is
port (data, clk, reset, en : in std_logic;
q : out std_logic);
end dff_ck_en;
architecture behav of dff_ck_en is
begin
process (clk, reset) begin
if (reset = '0') then
q <= '0';
```

```

elseif (clk'event and clk = '1') then
  if (en = '1') then
    q <= data;
  end if;
end if;
end process;
end behav;

```

Пример описания на Verilog:

```

module dff_ck_en (data, clk, reset, en, q);
  input data, clk, reset, en;
  output q;
  reg q;
  always @(posedge clk or negedge reset)
    if (~reset)
      q = 1'b0;
    else if (en)
      q = data;
endmodule

```

Далее рассмотрим защелки на основе D-триггеров (D-Latches).

6.2.8. Защелка с разрешением выхода (D-Latch with Data and Enable)

Обозначение на схеме приведено на **Рис. 6.8**.

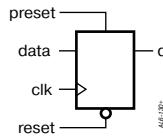


Рис. 6.8. Схемное обозначение защелки с разрешением выхода

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;

```

```
entity d_latch is
port(enable, data: in std_logic;
y : out std_logic);
end d_latch;
architecture behave of d_latch is
begin
process (enable, data)
begin
if (enable = '1') then
y <= data;
end if;
end process;
end behave;
```

Пример описания на Verilog:

```
module d_latch (enable, data, y);
input enable, data;
output y;
reg y;
always @(enable or data)
if (enable)
y = data;
endmodule
```

6.2.9. Защелка с входом данных с разрешением (D-Latch with Gated Asynchronous Data)

Пример обозначения приведен на **Рис. 6.9**.

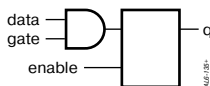


Рис. 6.9. Схемное описание защелки с выходом данных с разрешением

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```

entity d_latch_e is
port (enable, gate, data : in std_logic;
      q : out std_logic);
end d_latch_e;
architecture behave of d_latch_e is
begin
process (enable, gate, data) begin
if (enable = '1') then
q <= data and gate;
end if;
end process;
end behave;

```

Пример описания на Verilog:

```

module d_latch_e(enable, gate, data, q);
input enable, gate, data;
output q;
reg q;
always @ (enable or data or gate)
if (enable)
q = (data & gate);
endmodule

```

6.2.10. Защелка с входом разрешения (D-Latch with Gated Enable)

Пример обозначения приведен на Рис. 6.10.



Рис. 6.10. Схемное описание защелки с входом разрешения

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity d_latch_en is

```

```

port (enable, gate, d: in std_logic;
      q : out std_logic);
end d_latch_en;
architecture behave of d_latch_en is
begin
  process (enable, gate, d) begin
    if ((enable and gate) = '1') then
      q <=d;
    end if;
  end process;
end behave;

```

Пример описания на Verilog:

```

module d_latch_en(enable, gate, d, q);
input enable, gate, d;
output q;
reg q;
always @ (enable or d or gate)
if (enable & gate)
q =d;
endmodule

```

6.2.11. Защелка с асинхронным сбросом (D-Latch with Asynchronous Reset)

Пример обозначения приведен на Рис. 6.11.

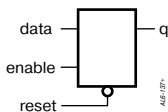


Рис. 6.11. Схемное описание защелки с асинхронным сбросом

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;

```

```

entity d_latch_rst is
port (enable, data, reset: in std_logic;
      q : out std_logic);
end d_latch_rst;
architecture behav of d_latch_rst is
begin
process (enable, data, reset) begin
if (reset = '0') then
q <= '0';
elsif (enable = '1') then
q <= data;
end if;
end process;
end behav;

```

Пример описания на Verilog:

```

module d_latch_rst (reset, enable, data, q);
input reset, enable, data;
output q;
reg q;
always @ (reset or enable or data)
if (~reset)
q = 1'b0;
else if (enable)
q = data;
endmodule

```

6.3. Построение устройств потоковой обработки данных (Datapath logic)

Как правило, устройства потоковой обработки данных представляют собой структурированные повторяющиеся функции. Рассмотрение таких устройств начнем с приоритетного шифратора. Приоритетный шифратор (Рис. 6.12) строится с использованием оператора IF-THEN-ELSE.

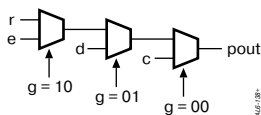


Рис. 6.12. Схемное описание приоритетного шифратора

Пример описания шифратора на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity my_if is
port (c, d, e, f: in std_logic;
s : in std_logic_vector(1 downto 0);
pout : out std_logic);
end my_if;
architecture my_arc of my_if is
begin
myif_pro: process (s, c, d, e, f) begin
if s = "00" then
pout <= c;
elsif s = "01" then
pout <= d;
elsif s = "10" then
pout <= e;
else pout <= f;
end if;
end process myif_pro;
end my_arc;
```

Пример описания на Verilog:

```
module IF_MUX (c, d, e, f, s, pout);
input c, d, e, f;
input [1:0]s;
output pout;
reg pout;
always @(c or d or e or f or s) begin
```



```

if (s == 2'b00)
pout = c;
else if (s == 2'b01)
pout = d;
else if (s == 2'b10)
pout = e;
else pout = f;
end
endmodule

```

Другим часто используемым устройством является мультиплексор (Multiplexor). Как правило, для построения мультиплексора (**Рис. 6.13**) удобно использовать оператор CASE. Оператор CASE обеспечивает параллельную обработку. Оператор выбора CASE используется для выбора одного варианта из нескольких в зависимости от условий.

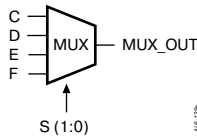


Рис. 6.13. Мультиплексор 4 в 1

Средства синтеза с VHDL позволяют автоматически выполнить параллельную обработку без приоритета, в то время как средства синтеза с Verilog поддерживают приоритет в выполнении оператора CASE. В ряде случаев необходимо ввести специфические инструкции в код для того, чтобы оператор выбора не имел приоритета.

Ниже приводятся примеры описания мультиплексора 4 в 1.

Пример описания на VHDL:

```

-4:1 Multiplexor
library IEEE;
use IEEE.std_logic_1164.all;
entity mux is
port (c, d, e, f : in std_logic;
      s : in std_logic_vector(1 downto 0);

```

```

muxout : out std_logic);
end mux;
architecture my_mux of mux is
begin
mux1: process (s, c, d, e, f) begin
case s is
when "00" => muxout <= c;
when "01" => muxout <= d;
when "10" => muxout <= e;
when others => muxout <= f;
end case;
end process mux1;
end my_mux;

```

Пример описания на Verilog:

```

//4:1 Multiplexor
module MUX (C, D, E, F, S, MUX_OUT);
input C, D, E, F;
input [1:0] S;
output MUX_OUT;
reg MUX_OUT;
always @(C or D or E or F or S)
begin
case (S)
2'b00 : MUX_OUT = C;
2'b01 : MUX_OUT = D;
2'b10 : MUX_OUT = E;
default : MUX_OUT = F;
endcase
end
endmodule

```

Для более сложного случая рассмотрим пример мультиплексора 12 в 1.
Пример описания на VHDL:

```

-- 12:1 mux
library ieee;

```

```
use ieee.std_logic_1164.all;
-- Entity declaration:
entity mux12_1 is
port
(
mux_sel: in std_logic_vector (3 downto 0);-- mux
select
A: in std_logic;
B: in std_logic;
C: in std_logic;
D: in std_logic;
E: in std_logic;
F: in std_logic;
G: in std_logic;
H: in std_logic;
I: in std_logic;
J: in std_logic;
K: in std_logic;
M: in std_logic;
mux_out: out std_logic -- mux output
);
end mux12_1;
-- Architectural body:
architecture synth of mux12_1 is
begin
proc1: process (mux_sel, A, B, C, D, E, F, G, H,
I, J, K, M)
begin
case mux_sel is
when "0000" => mux_out<= A;
when "0001" => mux_out <= B;
when "0010" => mux_out <= C;
when "0011" => mux_out <= D;
when "0100" => mux_out <= E;
when "0101" => mux_out <= F;
when "0110" => mux_out <= G;
when "0111" => mux_out <= H;
```

```

when "1000" => mux_out <= I;
when "1001" => mux_out <= J;
when "1010" => mux_out <= K;
when "1011" => mux_out <= M;
when others => mux_out <= '0';
end case;
end process proc1;
end synth;

```

Пример описания мультиплексора 12 в 1 на Verilog:

```

// 12:1 mux
module mux12_1(mux_out,
mux_sel, M, L, K, J, H, G, F, E, D, C, B, A);
output mux_out;
input [3:0] mux_sel;
input M;
input L;
input K;
input J;
input H;
input G;
input F;
input E;
input D;
input C;
input B;
input A;
reg mux_out;
// create a 12:1 mux using a case statement
always @ ({mux_sel[3:0]} or M or L or K or J or H
or G or F or
E or D or C or B or A)
begin: mux_blk
case ({mux_sel[3:0]}) // synthesis full_case
parallel_case
4'b0000 : mux_out = A;
4'b0001 : mux_out = B;

```

```

4'b0010 : mux_out = C;
4'b0011 : mux_out = D;
4'b0100 : mux_out = E;
4'b0101 : mux_out = F;
4'b0110 : mux_out = G;
4'b0111 : mux_out = H;
4'b1000 : mux_out = J;
4'b1001 : mux_out = K;
4'b1010 : mux_out = L;
4'b1011 : mux_out = M;
4'b1100 : mux_out = 1'b0;
4'b1101 : mux_out = 1'b0;
4'b1110 : mux_out = 1'b0;
4'b1111 : mux_out = 1'b0;
endcase
end
endmodule

```

Кроме обычного оператора выбора в языке описания аппаратуры Verilog используется оператор выбора CASEX. Ниже приводится описание на Verilog мультиплексора 4 в 1.

```

//8 bit 4:1 multiplexor with don't care X, 3:1
equivalent mux
module mux4 (a, b, c, sel, q);
input [7:0] a, b, c;
input [1:0] sel;
output [7:0] q;
reg [7:0] q;
always @ (sel or a or b or c)
case (sel)
2'b00:q =a;
2'b01:q =b;
2'b1x:q =c;
default:q =c;
endcase
endmodule

```

Дешифратор, пожалуй, является самым распространенным комбинационным устройством. Ниже приводится пример дешифратора 3 в 8.

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity decode is
port ( Ain : in std_logic_vector (2 downto 0);
      En : in std_logic;
      Yout : out std_logic_vector (7 downto 0));
end decode;
architecture decode_arch of decode is
begin
process (Ain)
begin
if (En='0') then
Yout <= (others => '0');
else
case Ain is
when "000" => Yout <= "00000001";
when "001" => Yout <= "00000010";
when "010" => Yout <= "00000100";
when "011" => Yout <= "00001000";
when "100" => Yout <= "00010000";
when "101" => Yout <= "00100000";
when "110" => Yout <= "01000000";
when "111" => Yout <= "10000000";
when others => Yout <= "00000000";
end case;
end if;
end process;
end decode_arch;
```

Пример описания дешифратора на Verilog имеет вид:

```
module decode (Ain, En, Yout);
input En;
input [2:0] Ain;
```

```

output [7:0] Yout;
reg [7:0] Yout;
always @ (En or Ain)
begin
  if (!En)
    Yout = 8'b0;
  else
    case (Ain)
      3'b000 : Yout = 8'b000000001;
      3'b001 : Yout = 8'b000000010;
      3'b010 : Yout = 8'b000000100;
      3'b011 : Yout = 8'b000001000;
      3'b100 : Yout = 8'b000010000;
      3'b101 : Yout = 8'b000100000;
      3'b110 : Yout = 8'b001000000;
      3'b111 : Yout = 8'b010000000;
      default : Yout = 8'b000000000;
    endcase
  end
endmodule

```

6.4. Счетчики

Счетчики являются достаточно широко распространенными устройствами, их классификация и принципы построения изложены в литературе [34—38]. Следует помнить, что большинство программ синтеза не позволяет получить приемлемых результатов по быстродействию при разрядности счетчика более 8 разрядов, в этом случае часто применяются специфические приемы синтеза, зависящие от технологии, по которой выполнена ПЛИС.

Рассмотрим пример построения 8-разрядного счетчика, считающего в прямом направлении и имеющего цепи разрешения счета и асинхронного сброса.

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;

```

```

use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity counter8 is
port (clk, en, rst : in std_logic;
count : out std_logic_vector (7 downto 0));
end counter8;
architecture behav of counter8 is
signal cnt: std_logic_vector (7 downto 0);
begin
process (clk, en, cnt, rst)
begin
if (rst = '0') then
cnt <= (others => '0');
elsif (clk'event and clk = '1') then
if (en = '1') then
cnt <= cnt + '1';
end if;
end process;
count <= cnt;
end behav;

```

Пример описания на Verilog:

```

module count_en (en, clock, reset, out);
parameter Width = 8;
input clock, reset, en;
output [Width-1:0] out;
reg [Width-1:0] out;
always @(posedge clock or negedge reset)
if(!reset)
out = 8'b0;
else if(en)
out = out +1;
endmodule

```

Другой пример иллюстрирует построение 8-разрядного счетчика с загрузкой и асинхронным сбросом (8-bit Up Counter with Load and Asynchronous Reset):

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity counter is
port (clk, reset, load: in std_logic;
data: in std_logic_vector (7 downto 0);
count: out std_logic_vector (7 downto 0));
end counter;
architecture behave of counter is
signal count_i : std_logic_vector (7 downto 0);
begin
process (clk, reset)
begin
if (reset = '0') then
count_i <= (others => '0');
elsif (clk'event and clk = '1') then
if load = '1' then
count_i <= data;
else
count_i <= count_i + '1';
end if;
end if;
end process;
count <= count_i;
end behave;
```

Пример описания на Verilog:

```
module count_load (out, data, load, clk, reset);
parameter Width = 8;
input load, clk, reset;
input [Width-1:0] data;
output [Width-1:0] out;
reg [Width-1:0] out;
```

```
always @(posedge clk or negedge reset)
if(!reset)
out = 8'b0;
else if(load)
out = data;
else
out = out +1;
endmodule
```

Пример построения счетчика с предварительной загрузкой, входами разрешения и остановки счета (8-bit Up Counter with Load, Count Enable, Terminal Count and Asynchronous Reset) приведен ниже.

Пример описания на Verilog:

```
module count_load (out, cout, data, load, clk, en,
reset);
parameter Width = 8;
input load, clk, en, reset;
input [Width-1:0] data;
output cout; // carry out
output [Width-1:0] out;
reg [Width-1:0] out;
always @(posedge clk or negedge reset)
if(!reset)
out = 8'b0;
else if(load)
out = data;
else if(en)
out = out +1;
// cout=1 when all out bits equal 1
assign cout = &out;
endmodule
```

Следующий пример — счетчик с произвольным модулем счета и всеми остальными функциями: сброс, загрузка и т.п. (N-bit Up Counter with Load, Count Enable, and Asynchronous Reset)

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;
entity counter is
generic (width : integer := n);
port (data : in std_logic_vector (width-1 downto 0);
load, en, clk, rst : in std_logic;
q :out std_logic_vector (width-1 downto 0));
end counter;
architecture behave of counter is
signal count : std_logic_vector (width-1 downto 0);
begin
process(clk, rst)
begin
if rst = '1' then
count <= (others => '0');
elsif (clk'event and clk = '1') then
if load = '1' then
count <= data;
elsif en = '1' then
count <= count + 1;
end if;
end if;
end process;
q <= count;
end behave;

```

6.5. Арифметические устройства

Проектирование устройств обработки информации немислимо без реализации арифметических операций — сложения, умножения, вычитания и деления. Ниже приводятся примеры использования арифметических операторов при проектировании на языках описания аппаратуры высокого уровня.

Пример выполнения арифметических операций приведен ниже.

Пример описания на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity arithmetic is
port (A, B: in std_logic_vector(3 downto 0);
Q1: out std_logic_vector(4 downto 0);
Q2, Q3: out std_logic_vector(3 downto 0);
Q4: out std_logic_vector(7 downto 0));
end arithmetic;
architecture behav of arithmetic is
begin
process (A, B)
begin
Q1 <= ('0' & A) + ('0' & B); --addition
Q2<=A -B;--subtraction
Q3<=A /B;--division
Q4<=A *B;--multiplication
end process;
end behav;

```

Конечно, такое описание не всегда приводит к хорошим результатам. Ниже мы еще вернемся к теме построения быстродействующих арифметических устройств. Здесь же заметим, что если требуется выполнить умножение или деление на число, являющееся степенью двойки, то арифметическая операция легко выполняется путем сдвига на необходимое число разрядов вправо или влево. Например, выражение:

$$Q \leq C/16 + C*4;$$

может быть представлено как

$$Q \leq \text{shr}(C, "100") + \text{shl}(C, "10");$$

или на VHDL:

$$Q \leq "0000"&C(8\text{downto }4)+C(6\text{downto }0) \& "00";$$

Функции "shr" и "shl" находятся в пакете IEEE.std_logic_arith.all.

Пример арифметических операций в Verilog:

```

module arithmetic (A, B, Q1, Q2, Q3, Q4);
input [3:0] A, B;

```

```

output [4:0] Q1;
output [3:0] Q2, Q3;
output [7:0] Q4;
reg [4:0] Q1;
reg [3:0] Q2, Q3;
reg [7:0] Q4;
always @(A or B)
begin
Q1 =A +B;//addition
Q2 =A -B;//subtraction
Q3 =A /2;//division
Q4 =A *B;//multiplication
end
endmodule

```

Реализация регистров сдвига имеет вид:

```
Q = {4b'0000 C[8:4]} + {C[6:0] 2b'00};
```

Операторы отношения сравнивают два операнда и выдают значение TRUE или FALSE. Приведенные ниже примеры показывают применение операторов отношения. Пример использования операторов отношения в VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity relational is
port (A,B :in std_logic_vector(3 downto 0);
Q1, Q2, Q3, Q4 : out std_logic);
end relational;
architecture behav of relational is
begin
process (A, B)
begin
-Q1<=A >B;--greater than
-Q2<=A <B;--less than
-Q3 <= A >= B; -- greater than equal to
if (A <= B) then -- less than equal to

```

```
Q4 <= '1';
else
Q4 <= '0';
end if;
end process;
end behav;
```

Пример использования операторов отношения в Verilog:

```
module relational (A, B, Q1, Q2, Q3, Q4);
input [3:0] A, B;
output Q1, Q2, Q3, Q4;
reg Q1, Q2, Q3, Q4;
always @(A or B)
begin
//Q1 =A >B;//greater than
//Q2 =A <B;//less than
//Q3 =A >=B;//greater than equal to
if (A <= B) //less than equal to
Q4 =1;
else
Q4 =0;
end
endmodule
```

Оператор эквивалентности (Equality operator) используется для сравнения операндов. Пример реализации оператора эквивалентности на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity equality is
port (
A: in STD_LOGIC_VECTOR (3 downto 0);
B: in STD_LOGIC_VECTOR (3 downto 0);
Q1: out STD_LOGIC;
Q2: out STD_LOGIC
);
```

```

end equality;
architecture equality_arch of equality is
begin
process (A, B)
begin
Q1<=A =B; -- equality
if (A /= B) then -- inequality
Q2 <= '1';
else
Q2 <= '0';
end if;
end process;
end equality_arch;

```

Другой вариант описания на VHDL имеет вид:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity equality is
port (
A: in STD_LOGIC_VECTOR (3 downto 0);
B: in STD_LOGIC_VECTOR (3 downto 0);
Q1: out STD_LOGIC;
Q2: out STD_LOGIC
);
end equality;
architecture equality_arch of equality is
begin
Q1 <= '1' when A =B else '0'; -- equality
Q2 <= '1' when A /=B else '0'; -- inequality
end equality_arch;

```

Пример описания на Verilog:

```

module equality (A, B, Q1, Q2);
input [3:0] A;
input [3:0] B;
output Q1;

```

```

output Q2;
reg Q1, Q2;
always @(A or B)
begin
  Q1 =A ==B;//equality
  if (A != B) //inequality
  Q2 =1;
  else
  Q2 =0;
end
endmodule

```

Реализация оператора сдвига может быть полезна в различных схемах умножения-деления, а также нормализации данных. Пример оператора сдвига на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity shift is
port (data : in std_logic_vector(3 downto 0);
      q1, q2 : out std_logic_vector(3 downto 0));
end shift;
architecture rtl of shift is
begin
  process (data)
  begin
    q1 <= shl (data, "10"); -- logical shift left
    q2 <= shr (data, "10"); -- logical shift right
  end process;
end rtl;

```

Другой вариант оператора сдвига:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity shift is

```



```

port (data : in std_logic_vector(3 downto 0);
      q1, q2 : out std_logic_vector(3 downto 0));
end shift;
architecture rtl of shift is
begin
  process (data)
  begin
    q1 <= data(1 downto 0) & "10"; -- logical shift left
    q2 <= "10" & data(3 downto 2); -- logical shift right
  end process;
end rtl;

```

Пример описания на Verilog:

```

module shift (data, q1, q2);
input [3:0] data;
output [3:0] q1, q2;
parameter B = 2;
reg [3:0] q1, q2;
always @ (data)
begin
  q1 = data << B; // logical shift left
  q2 = data >> B; // logical shift right
end
endmodule

```

6.6. Конечные автоматы (Finite state machine)

Конечные автоматы (Finite state machine) и средства их проектирования были рассмотрены выше (см. главу 2). В ряде случаев автоматная модель (описание) устройства позволяет получить быструю и эффективную реализацию последовательностного устройства. Как известно, обычно рассматривают два типа автоматов — автомат Мили (Mealy) и Мура (Moore). Выход автомата Мура является функцией только текущего состояния, в то время как выход автомата Мили функция как текущего состояния, так и начального внешнего воздействия. Обычно конечный автомат состоит из трех основных частей:

1. Регистр текущего состояния (Sequential Current State Register). Этот регистр представляет собой набор тактируемых D-триггеров, синхронизи-

руемых одним синхросигналом, и используется для хранения кода текущего состояния автомата. Очевидно, что для автомата с n -состояниями требуется $\log_2(n)$ триггеров.

2. Логика переходов (Combinational Next State Logic). Как известно, конечный автомат может находиться в каждый конкретный момент времени только в одном состоянии. Каждый тактовый импульс вызывает переход автомата из одного состояния в другое. Правила перехода определяются комбинационной схемой, называемой логикой переходов. Следующее состояние определяется как функция текущего состояния и входного воздействия.

3. Логика формирования выхода (Combinational Output Logic). Выход цифрового автомата обычно определяется как функция текущего состояния и исходной установки (в случае автомата Мили). Формирование выходного сигнала автомата определяется с помощью логики формирования выхода.

На Рис. 6.14 и 6.15 приведены структуры автоматов Мура и Мили соответственно.

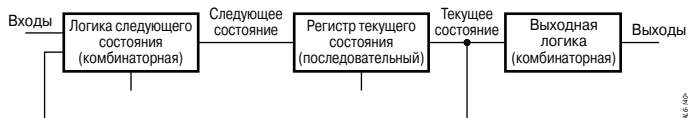


Рис. 6.14. Автомат Мура

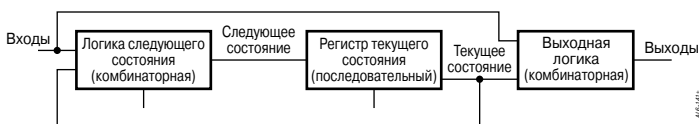


Рис. 6.15. Автомат Мили

Для обеспечения стабильной и безотказной работы используется сброс автомата в начальное состояние. Таким образом всегда обеспечивается инициализация автомата в заранее predetermined состоянии при первом тактовом импульсе. В случае, если сброс не предусмотрен, невозможно предсказать, с какого начального состояния начнется функционирование, что может привести к сбоям в работе всей системы. Особенно эта ситуация актуальна при включении питания системы. Поэтому мы настоятельно рекомендуем использовать схемы сброса и начальной установки при проекти-

ровании устройств на ПЛИС. Обычно применяют асинхронные схемы сброса из-за того, что не требуется использовать дешифратор неиспользуемых (избыточных) состояний, что упрощает логику переходов.

С другой стороны, из-за того, что ПЛИС, выполненные по архитектуре FPGA, имеют достаточное число регистров (триггеров), использование автоматных моделей позволяет получить достаточно быстродействующую и в то же время наглядную реализацию при приемлемых затратах ресурсов.

Ниже рассмотрим пример проектирования автомата Мили. Диаграмма переходов автомата приведена на **Рис. 6.16**.

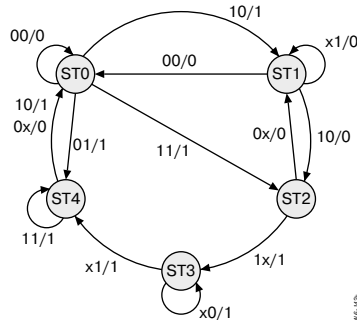


Рис. 6.16. Диаграмма переходов автомата Мили

Пример автомата Мили на VHDL:

```

— Автомат Мили с 5 состояниями
library ieee;
use ieee.std_logic_1164.all;
entity mealy is
port (clock, reset: in std_logic;
data_out: out std_logic;
data_in: in std_logic_vector (1 downto 0));
end mealy;
architecture behave of mealy is
type state_values is (st0, st1, st2, st3, st4);
signal pres_state, next_state: state_values;
begin

```

```
- FSM register
statereg: process (clock, reset)
begin
  if (reset = '0') then
    pres_state <= st0;
  elsif (clock'event and clock = '1') then
    pres_state <= next_state;
  end if;
end process statereg;
- FSM combinational block
fsm: process (pres_state, data_in)
begin
  case pres_state is
    when st0 =>
      case data_in is
        when "00" => next_state <= st0;
        when "01" => next_state <= st4;
        when "10" => next_state <= st1;
        when "11" => next_state <= st2;
        when others => next_state <= (others <= 'x');
      end case;
    when st1 =>
      case data_in is
        when "00" => next_state <= st0;
        when "10" => next_state <= st2;
        when others => next_state <= st1;
      end case;
    when st2 =>
      case data_in is
        when "00" => next_state <= st1;
        when "01" => next_state <= st1;
        when "10" => next_state <= st3;
        when "11" => next_state <= st3;
        when others => next_state <= (others <= 'x');
      end case;
    when st3 =>
      case data_in is
        when "01" => next_state <= st4;
        when "11" => next_state <= st4;
```

```
when others => next_state <= st3;
end case;
when st4 =>
case data_in is
when "11" => next_state <= st4;
when others => next_state <= st0;
end case;
when others => next_state <= st0;
end case;
end process fsm;
-- Mealy output definition using pres_state
w/ data_in
outputs: process (pres_state, data_in)
begin
case pres_state is
when st0 =>
case data_in is
when "00" => data_out <= '0';
when others => data_out <= '1';
end case;
when st1 => data_out <= '0';
when st2 =>
case data_in is
when "00" => data_out <= '0';
when "01" => data_out <= '0';
when others => data_out <= '1';
end case;
when st3 => data_out <= '1';
when st4 =>
case data_in is
when "10" => data_out <= '1';
when "11" => data_out <= '1';
when others => data_out <= '0';
end case;
when others => data_out <= '0';
end case;
end process outputs;
end behave;
```

Описание автомата Мили на Verilog имеет вид:

```
// Example of a 5-state Mealy FSM
module mealy (data_in, data_out, reset, clock);
output data_out;
input [1:0] data_in;
input reset, clock;
reg data_out;
reg [2:0] pres_state, next_state;
parameter st0=3'd0, st1=3'd1, st2=3'd2, st3=3'd3,
st4=3'd4;
// FSM register
always @(posedge clock or negedge reset)
begin: statereg
if(!reset)// asynchronous reset
pres_state = st0;
else
pres_state = next_state;
end // statereg
// FSM combinational block
always @(pres_state or data_in)
begin: fsm
case (pres_state)
st0: case(data_in)
2'b00: next_state=st0;
2'b01: next_state=st4;
2'b10: next_state=st1;
2'b11: next_state=st2;
endcase
st1: case(data_in)
2'b00: next_state=st0;
2'b10: next_state=st2;
default: next_state=st1;
endcase
st2: case(data_in)
2'b0x: next_state=st1;
2'b1x: next_state=st3;
endcase
```

```
st3: case(data_in)
2'bx1: next_state=st4;
default: next_state=st3;
endcase
st4: case(data_in)
2'b11: next_state=st4;
default: next_state=st0;
endcase
default: next_state=st0;
endcase
end // fsm
// Mealy output definition using pres_state
w/ data_in
always @(data_in or pres_state)
begin: outputs
case(pres_state)
st0: case(data_in)
2'b00: data_out=1'b0;
default: data_out=1'b1;
endcase
st1: data_out=1'b0;
st2: case(data_in)
2'b0x: data_out=1'b0;
default: data_out=1'b1;
endcase
st3: data_out=1'b1;
st4: case(data_in)
2'b1x: data_out=1'b1;
default: data_out=1'b0;
endcase
default: data_out=1'b0;
endcase
end // outputs
endmodule
```

Ниже рассмотрен пример автомата Мура (Moore machine) с тем же графом переходов. Описание на VHDL:

```

- Example of a 5-state Moore FSM
library ieee;
use ieee.std_logic_1164.all;
entity moore is
port (clock, reset: in std_logic;
data_out: out std_logic;
data_in: in std_logic_vector (1 downto 0));
end moore;
architecture behave of moore is
type state_values is (st0, st1, st2, st3, st4);
signal pres_state, next_state: state_values;
begin
- FSM register
statereg: process (clock, reset)
begin
if (reset = '0') then
pres_state <= st0;
elsif (clock = '1' and clock'event) then
pres_state <= next_state;
end if;
end process statereg;
- FSM combinational block
fsm: process (pres_state, data_in)
begin
case pres_state is
when st0 =>
case data_in is
when "00" => next_state <= st0;
when "01" => next_state <= st4;
when "10" => next_state <= st1;
when "11" => next_state <= st2;
when others => next_state <= (others <= 'x');
end case;
when st1 =>
case data_in is
when "00" => next_state <= st0;
when "10" => next_state <= st2;

```



```
when others => next_state <= st1;
end case;
when st2 =>
case data_in is
when "00" => next_state <= st1;
when "01" => next_state <= st1;
when "10" => next_state <= st3;
when "11" => next_state <= st3;
when others => next_state <= (others <= 'x');
end case;
when st3 =>
case data_in is
when "01" => next_state <= st4;
when "11" => next_state <= st4;
when others => next_state <= st3;
end case;
when st4 =>
case data_in is
when "11" => next_state <= st4;
when others => next_state <= st0;
end case;
when others => next_state <= st0;
end case;
end process fsm;
-- Moore output definition using pres_state only
outputs: process (pres_state)
begin
case pres_state is
when st0 => data_out <= '1';
when st1 => data_out <= '0';
when st2 => data_out <= '1';
when st3 => data_out <= '0';
when st4 => data_out <= '1';
when others => data_out <= '0';
end case;
end process outputs;
end behave;
```

Описание автомата Мура на Verilog:

```
// Example of a 5-state Moore FSM
module moore (data_in, data_out, reset, clock);
output data_out;
input [1:0] data_in;
input reset, clock;
reg data_out;
reg [2:0] pres_state, next_state;
parameter st0=3'd0, st1=3'd1, st2=3'd2, st3=3'd3,
st4=3'd4;
//FSM register
always @(posedge clock or negedge reset)
begin: statereg
if(!reset)
pres_state = st0;
else
pres_state = next_state;
end // statereg
// FSM combinational block
always @(pres_state or data_in)
begin: fsm
case (pres_state)
st0: case(data_in)
2'b00: next_state=st0;
2'b01: next_state=st4;
2'b10: next_state=st1;
2'b11: next_state=st2;
endcase
st1: case(data_in)
2'b00: next_state=st0;
2'b10: next_state=st2;
default: next_state=st1;
endcase
st2: case(data_in)
2'b0x: next_state=st1;
2'b1x: next_state=st3;
```

```

endcase
st3: case(data_in)
2'b1: next_state=st4;
default: next_state=st3;
endcase
st4: case(data_in)
2'b1: next_state=st4;
default: next_state=st0;
endcase
default: next_state=st0;
endcase
end // fsm
// Moore output definition using pres_state only
always @(pres_state)
begin: outputs
case(pres_state)
st0: data_out=1'b1;
st1: data_out=1'b0;
st2: data_out=1'b1;
st3: data_out=1'b0;
st4: data_out=1'b1;
default: data_out=1'b0;
endcase
end // outputs
endmodule // Moore

```

6.7. Элементы ввода/вывода

Без элементов ввода/вывода невозможна организация полноценного обмена данных, организация шин и т.п. Рассмотрим особенности описания на языках описания аппаратуры элементов ввода/вывода.

Элемент с тремя состояниями (Tri-state buffer) позволяет организовать подключение нескольких устройств к общей шине. Обозначение такого элемента на схеме приведено на **Рис. 6.17**.

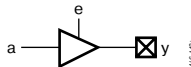


Рис. 6.17. Элемент с тремя состояниями

Пример описания на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tristate is
port (e,a :in std_logic;
y : out std_logic);
end tristate;
architecture tri of tristate is
begin
process (e, a)
begin
if e = '1' then
y <=a;
else
y <= 'Z';
end if;
end process;
end tri;
```

Другой вариант описания элемента с тремя состояниями на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tristate is
port (e,a :in std_logic;
y :out std_logic);
end tristate;
architecture tri of tristate is
begin
Y <=awhen (e = '1') else 'Z';
end tri;
```

На Verilog элемент с тремя состояниями описывается следующим образом:

```
module TRISTATE (e, a, y);
input a, e;
output y;
```

```

reg y;
always @(eor a) begin
  if (e)
    y = a;
  else
    y = 1'bz;
  end
endmodule

```

или

```

module TRISTATE (e, a, y);
  input a, e;
  output y;
  assign y = e ?a :1'bZ;
endmodule

```

Приведенные выше примеры показывают логику функционирования элемента с тремя состояниями. Примеры создания экземпляра компоненты приведены ниже.

Экземпляр компонента на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity tristate is
  port (e,a :in std_logic;
        y :out std_logic);
end tristate;
architecture tri of tristate is
  component TRIBUFF
    port (D, E: in std_logic;
          PAD: out std_logic);
  end component;
begin
  U1: TRIBUFF port map (D => a,
    E =>e,
    PAD => y);
end tri;

```

Экземпляр компонента на Verilog:

```
module TRISTATE (e, a, y);
input a, e;
output y;
TRIBUFF U1 (.D(a), .E(e), .PAD(y));
endmodule
```

Двунаправленные элементы ввода/вывода (Bi-directional buffer). Двунаправленный элемент ввода/вывода используется либо как элемент ввода, либо как выходной буфер с возможностью перехода в третье состояние. Ниже приводятся примеры создания описания двунаправленного буфера (Рис. 6.18).

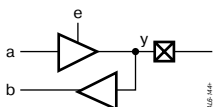


Рис. 6.18. Схемное описание двунаправленного буфера

Пример логики двунаправленного буфера на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity bidir is
port (y : inout std_logic;
e, a: in std_logic;
b :out std_logic);
end bidir;
architecture bi of bidir is
begin
process (e, a)
begin
case e is
when '1'=>y <=a;
when '0' => y <= 'Z';
when others => y <= 'X';
end case;
end process;
end bi;
```

```

end process;
b <=y;
end bi;

```

Пример описания двунаправленного буфера на Verilog:

```

module bidir (e, y, a, b);
input a, e;
inout y;
output b;
reg y_int;
wire y, b;
always @(a or e)
begin
if (e == 1'b1)
y_int <= a;
else
y_int <= 1'bz;
end
assign y = y_int;
assign b = y;
endmodule

```

Примеры создания экземпляров компоненты на VHDL:

```

library IEEE;
use IEEE.std_logic_1164.all;
entity bidir is
port (y : inout std_logic;
e, a: in std_logic;
b :out std_logic);
end bidir;
architecture bi of bidir is
component BIBUF
port (D, E: in std_logic;
Y :out std_logic;

```

```
PAD: inout std_logic);
end component;
begin
U1: BIBUF port map (D => a,
E => e,
Y => b,
PAD => y);
end bi;
```

Пример описания экземпляра буфера на Verilog:

```
module bidir (e, y, a, b);
input a, e;
inout y;
output b;
BIBUF U1 (.PAD(y), .D(a), .E(e), .Y(b) );
Endmodule
```

6.8. Параметризация

Как уже отмечалось выше, в последние годы наметилась тенденция к разработке и использованию параметризованных модулей (ядер, макросов, мегафункций) при проектировании устройств на ПЛИС с использованием языков описания аппаратуры. В языках описания аппаратуры существуют специальные конструкции, позволяющие обеспечить полную параметризацию проекта. Это так называемые родовые (параметризованные, настраиваемые) типы данных (Generics) и параметры (Parameters). Мы будем использовать термин настраиваемый тип. Настраиваемые типы данных и параметры используются обычно для определения размерности компонента (например, разрядность шин, коэффициентов и т.п.). Конкретное значение параметров определяется при создании конкретного экземпляра компонента. Рассмотрим пример использования параметризации на примере сумматора.

Пример описания параметризуемого сумматора на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```



```

use IEEE.std_logic_unsigned.all;
entity adder is
generic (WIDTH : integer := 8);
port (A, B: in UNSIGNED(WIDTH-1 downto 0);
CIN: in std_logic;
COUT: out std_logic;
Y: out UNSIGNED(WIDTH-1 downto 0));
end adder;
architecture rtl of adder is
begin
process (A,B,CIN)
variable TEMP_A,TEMP_B,TEMP_Y:UNSIGNED(A'length
downto 0);
begin
TEMP_A := '0' & A;
TEMP_B := '0' & B;
TEMP_Y := TEMP_A + TEMP_B + CIN;
Y <= TEMP_Y (A'length-1 downto 0);
COUT <= TEMP_Y (A'length);
end process;
end rtl;

```

Параметр Width определяет разрядность данных. Пример конкретного экземпляра компонента с 16-разрядными данными приведен ниже:

```

U1: adder generic map(16) port map (A_A, B_A,
CIN_A, COUT_A, Y_A);

```

Пример описания сумматора на Verilog:

```

module adder (cout, sum, a, b, cin);
parameter Size = 8;
output cout;
output [Size-1:0] sum;
input cin;
input [Size-1:0] a, b;
assign {cout, sum}= a + b + cin;
endmodule

```

Параметр Size определяет ширину сумматора. Реализация компонента 16-разрядного сумматора приведена ниже:

```
adder #(16) adder16(cout_A, sun_A, a_A, b_A, cin_A)
```

6.9. Специфика проектирования устройств с учетом архитектурных особенностей ПЛИС

В отличие от специализированных БИС (ASIC) ПЛИС, выполненные по FPGA-архитектуре, имеют модульную матричную структуру. Каждая комбинационная ступень реализации сложной функции вносит свой вклад в суммарную задержку распространения сигнала. В результате приходится учитывать ограничения, связанные со слишком длинными цепями распространения сигналов. Использование соответствующих приемов описания устройств (coding style) позволяет улучшить временные характеристики спроектированного устройства.

Рассмотрим способы снижения числа ступеней вычисления сложной функции (Reducing logic levels) для уменьшения длины критических путей распространения сигнала (critical paths). Ниже приводятся примеры оптимального проектирования таких устройств.

Пример 1. В данном примере сигнал critical (Рис. 6.19) проходит через три ступени.

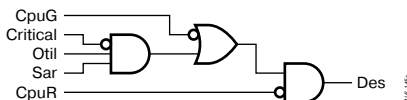


Рис. 6.19. Прохождение сигнала critical через три ступени

Пример написан на языке VHDL:

```
if ((( Crtical='0' and Obi='1' and Sar='1')
or CpuG='0') and CpuR='0') then
Des <= Adr;
elsif (((Crtical='0' and Obi='1' and Sar='1')
or CpuG='0') and CpuR='1') then
Des <= Bdr;
elsif (Sar='0' and.....
```

В результате сигнал `critical` является запаздывающим. Чтобы уменьшить число ступеней, используем конструкцию IF-THEN-ELSE statement. В результате сигнал `critical` проходит только через одну ступень, как показано на **Рис. 6.20**.

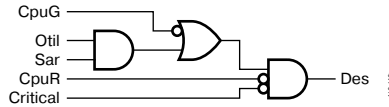


Рис. 6.20. Прохождение сигнала `critical` через одну ступень

```

if (Critical='0') then
  if (((Obi='1' and Sar='1')
    or CpuG='0') and CpuR='0') then
    Des <= Adr;
  elsif (((Obi='1' and Sar='1')
    or CpuG='0' and CpuR='1') then
    Des <= Bdr;
  end if;
end if;

```

Пример 2. В данном случае сигнал, обозначенный `critical` (**Рис. 6.21**), проходит через две ступени.

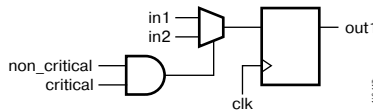


Рис. 6.21. Прохождение сигнала `critical` через две ступени

```

if (clk'event and clk ='1') then
  if (non_critical and critical) then
    out1 <= in1
  else
    out1 <= in2
  end if;
end if;

```

Для уменьшения числа ступеней для сигнала *critical* используем мультиплексирование входных сигналов *in1* и *in2*, используя для управления мультиплексором сигнал *non_critical*. Выход мультиплексора затем мультиплексируется с входом *in2* при управлении сигналом *critical*. В итоге сигнал *critical* проходит только через одну ступень задержки (**Рис. 6.22**).

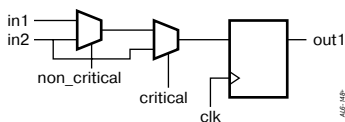


Рис. 6.22. Прохождение сигнала *critical* через одну ступень задержки

Реализация на VHDL приведена ниже.

```

signal out_temp : std_logic
if (non_critical)
    out_temp <= in1;
else out_temp <= in2;
    if (clk'event and clk = '1') then
        if (critical) then
            out1 <= out_temp;
        else out1 <= in2;
        end if;
    end if;
end if;
end if;

```

6.10. Совместное использование ресурсов

Другим путем повышения эффективности описаний цифровых устройств является совместное использование ресурсов (Resource sharing). Совместное использование ресурсов позволяет снизить число логических элементов для реализации операторов на языках описания аппаратуры. Ниже приводятся два примера совместного использования ресурсов. Примеры написаны на языке Verilog.

Пример 1. Реализация четырех сумматоров (счетчиков).

```

if (...(siz == 1)...)
    count = count + 1;
else if (...((siz == 2)...)
    count = count + 2;
else if (...(siz == 3)...)
    count = count + 3;
else if (...(siz == 0)...)
    count = count + 4;

```

Покажем, как можно избавиться от двух «лишних» счетчиков:

```

if (...(siz == 0)...)
    count = count + 4;
else if (...)
    count = count + siz

```

Пример 2. В данном примере используется неполное совместное использование ресурсов для реализации сумматоров (**Рис. 6.23**).

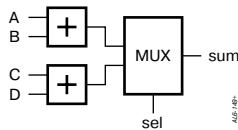


Рис. 6.23. Неполное совместное использование ресурсов для реализации сумматоров

```

if (select)
    sum<=A +B;
else
    sum<=C +D;

```

Сумматоры занимают значительные ресурсы, для их уменьшения переписываем код с целью ввести два мультиплексора и один сумматор, как показано ниже (**Рис. 6.24**).

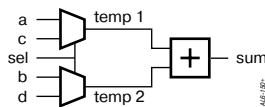


Рис. 6.24. Схемное описание уменьшения ресурсов, занимаемых сумматорами

```

if (sel)
    temp1 <= A;
    temp2 <= B;
else
    temp1 <= C;
    temp2 <= D;
sum <= temp1 + temp2;

```

Следует помнить, что в данном примере сигнал выбора `sel` не является запаздывающим сигналом.

Еще одним способом организации совместного использования ресурсов является использование операторов цикла. Арифметические операторы и мультиплексоры занимают значительные ресурсы. Если имеется оператор внутри цикла, программа синтеза должна оценить все состояния. В следующем примере на языке VHDL описываются четыре сумматора и один мультиплексор. Такая реализация может быть рекомендована только если сигнал выбора `req` — запаздывающий сигнал (**Рис. 6.25**).

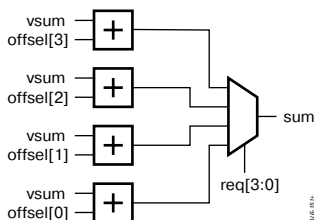


Рис. 6.25. Схемное описание четырех сумматоров и одного мультиплексора, в котором сигнал `req` — запаздывающий сигнал

```

vsum := sum;
for i in 0 to 3 loop
    if (req(i)='1') then
        vsum <= vsum + offset(i);
    end if;
end loop;

```

Если сигнал выбора `req` не является критическим, оператор может быть вынесен за пределы цикла, что приведет к тому, что вместо четырех сумматоров будет использован один (**Рис. 6.26**).

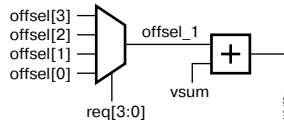


Рис. 6.26. Схемное описание использования только одного сумматора при вынесении за пределы цикла оператора выбора

Пример описания приведен ниже.

```
vsum := sum;
for i in 0 to 3 loop
    if (req(i)='1') then
        offset_1 <= offset(i);
    end if;
end loop;
vsum <= vsum + offset_1;
```

Использование кодирования для сочетаемости (Coding for Combinability) также позволяет выкроить ресурсы ПЛИС.

Для ПЛИС некоторых архитектур возможно выполнение всей логики работы как комбинационной, так и последовательностной в пределах одного логического элемента (ячейки), что значительно снижает задержки по критическим путям и экономит ресурсы.

Однако следует помнить, что такое объединение возможно только в том случае, если комбинационная схема управляет только одним триггером. В приведенном примере описания на VHDL элемент AND формирует управляющий сигнал для двух триггеров (**Рис. 6.27**), что препятствует введению элемента AND в последовательностный модуль.

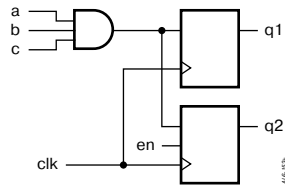


Рис. 6.27. Схемное описание элемента AND, формирующего управляющий сигнал для двух триггеров

```
one :process (clk, a, b, c, en) begin
  if (clk'event and clk = '1') then
    if (en = '1') then
      q2 <= a and b and c;
    end if;
    q1 <= a and b and c;
  end if;
end process one;
```

Для того чтобы объединить схему внутри одного элемента и убрать внешние цепи, удобно использовать дублирование логического элемента AND, уменьшая тем самым критические пути распространения сигнала (**Рис. 6.28**).

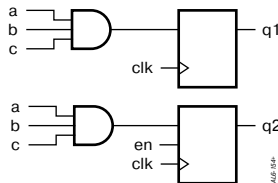


Рис. 6.28. Схемное описание использования двух логических элементов AND

```
part_one: process (clk, a, b, c, en) begin
  if (clk'event and clk = '1') then
    if (en = '1') then
      q2 <= a and b and c;
    end if;
  end if;
end process part_one;
```



```

end process part_one;
part_two: process (clk, a, b, c) begin
if (clk'event and clk ='1') then
    q1 <= a and b and c;
end if;
end process part_two;

```

6.11. Дублирование регистра

Задержка в цепи распространения сигнала возрастает с увеличением числа входов, нагруженных на эту цепь. Такая ситуация еще может быть приемлема для цепей сброса, но для цепей разрешения это недопустимо. Пример такой ситуации приведен ниже (**Рис. 6.29**).

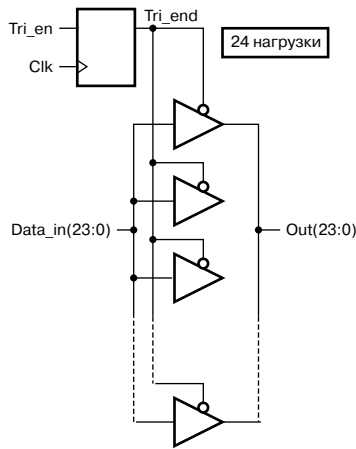


Рис. 6.29. Схемное описание цепи, в которой коэффициент разветвления сигнала разрешения равен 24

В данном VHDL примере коэффициент разветвления сигнала разрешения Tri_en равен 24.

```

architecture load of four_load is
signal Tri_en std_logic;
begin
loadpro: process (Clk)

```

```

begin
  if (clk'event and clk = '1') then
    Tri_end <= Tri_en;
  end if;
end process loadpro;
endpro : process (Tri_end, Data_in)
begin
  if (Tri_end = '1') then
    out <= Data_in;
  else
    out <= (others => 'Z');
  end if;
end process endpro;
end load;

```

Число разветвлений (fanout) можно уменьшить вдвое, используя два регистра (Рис. 6.30).

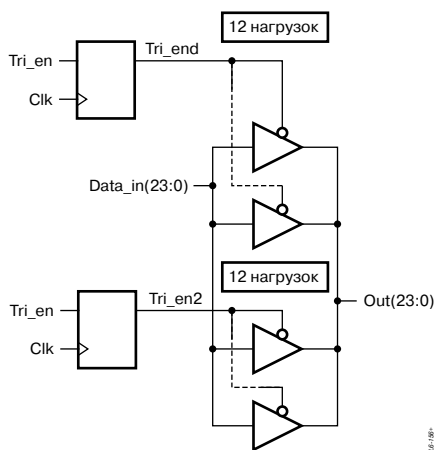


Рис. 6.30. Схемное описание цепи, в которой используются два регистра для уменьшения вдвое числа разветвлений

Ниже приведен пример реализации дублирования регистров на VHDL:

```
architecture loada of two_load is
    signal Tri_en1, Tri_en2 : std_logic;
begin
    loadpro: process (Clk)
    begin
        if (clk'event and clk ='1') then
            Tri_en1 <= Tri_en;
            Tri_en2 <= Tri_en;
        end if;
    end process loadpro;
    process (Tri_en1, Data_in)
    begin
        if (Tri_en1 = '1') then
            out(23:12) <= Data_in(23:12);
        else
            out(23:12) <= (others => 'Z');
        end if;
    end process;
    process (Tri_en2, Data_in)
    begin
        if (Tri_en2 = '1') then
            out(11:0) <= Data_in(11:0);
        else
            out(11:0) <= (others => 'Z');
        end if;
    end process;
```

Разделение проекта на части оптимальных размеров позволяет обеспечить оптимальное использование ресурсов за счет их оптимизации при синтезе. Известно, что подавляющее большинство программ синтеза достигает наиболее приемлемого результата при размере блока порядка 2...5 тысяч эквивалентных вентиляей.

Приведенный ниже пример показывает, как можно сэкономить ресурсы путем замены встроенного в модуль регистра (**Рис. 6.31**) на выделенный регистр (**Рис. 6.32**).

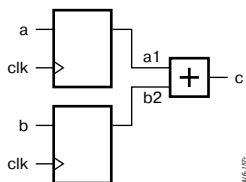


Рис. 6.31. Схемное описание цепи со встроенным в модуль регистром

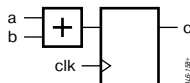


Рис. 6.32. Схемное описание цепи с выделенным регистром

```

process (clk, a, b) begin
  if (clk'event and clk = '1') then
    a1 <= a;
    b1 <= b;
  end if;
end process;
process (a1, b1)
begin c <= a1 + b1;
end process;
process (clk, a, b) begin
  if (clk'event and clk = '1') then
    c <= a + b;
  end if;
end process;

```

6.12. Создание описаний с учетом особенностей архитектуры ПЛИС (Technology Specific Coding Techniques)

Помимо способов создания описаний, независимых от используемой элементной базы и ее архитектурных особенностей, в практике проектирования устройств на ПЛИС применяется так называемое создание описа-

ний с учетом особенностей архитектуры ПЛИС (Technology Specific Coding Techniques). Рассмотрим некоторые примеры проектирования цифровых устройств для использования с учетом особенностей архитектуры FPGA ПЛИС.

Для реализации мультиплексоров, как правило, применяется конструкция CASE, что значительно выгоднее с точки зрения быстродействия и затрат ресурсов, чем конструкция IF-THEN-ELSE. Большинство производителей ПЛИС рекомендуют использовать оператор CASE.

Как правило, при реализации на ПЛИС внутренняя шина с тремя состояниями эмулируется как мультиплексор. Пример эмуляции шины в виде мультиплексора приведен на **Рис. 6.33**.

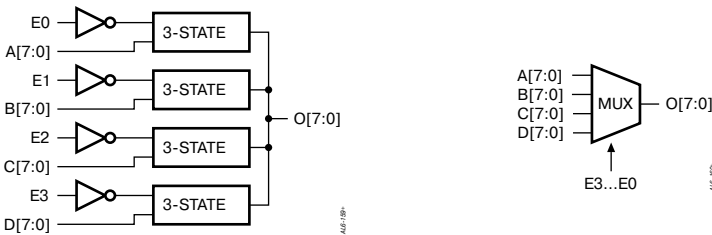


Рис. 6.33. Эмуляция шины в виде мультиплексора

Пример эмуляции шины на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity tribus is
port (A, B, C, D : in std_logic_vector
      (7 downto 0);
      E0, E1, E2, E3 : in std_logic;
      Q :out std_logic_vector(7 downto 0));
end tribus;
architecture rtl of tribus is
begin
  Q <=A when(E0 = '1') else "ZZZZZZZZ";
  Q <=B when(E1 = '1') else "ZZZZZZZZ";
```

```
Q <=C when(E2 = '1') else "ZZZZZZZZ";
Q <=D when(E3 = '1') else "ZZZZZZZZ";
end rtl;
```

Пример мультиплексора на VHDL:

```
library IEEE;
use IEEE.std_logic_1164.all;
entity muxbus is
port (A, B, C, D : in std_logic_vector
      (7 downto 0));
E0, E1, E2, E3 : in std_logic;
Q :out std_logic_vector(7 downto 0));
end muxbus;
architecture rtl of muxbus is
signal E_int : std_logic_vector(1 downto 0);
begin
process (E0, E1, E2, E3)
variable E : std_logic_vector(3 downto 0);
begin
E :=E0 &E1 &E2 &E3;
case E is
when "0001" => E_int <= "00";
when "0010" => E_int <= "01";
when "0100" => E_int <= "10";
when "1000" => E_int <= "11";
when others => E_int <= "-";
end case;
end process;
process (E_int, A, B, C, D)
begin
case E_int is
when "00" => Q <= D;
when "01" => Q <= C;
when "10" => Q <= B;
when "11" => Q <= A;
when others =>Q <=(others => '-');
end case;
```

```
end process;
end rtl;
```

Эмуляция шины на Verilog:

```
module tribus (A, B, C, D, E0, E1, E2, E3, Q);
    input [7:0]A, B, C, D;
    output [7:0]Q;
    input E0, E1, E2, E3;
    assign Q[7:0] = E0 ? A[7:0] : 8'bzzzzzzzz;
    assign Q[7:0] = E1 ? B[7:0] : 8'bzzzzzzzz;
    assign Q[7:0] = E2 ? C[7:0] : 8'bzzzzzzzz;
    assign Q[7:0] = E3 ? D[7:0] : 8'bzzzzzzzz;
endmodule
```

Реализация мультиплексора на Verilog:

```
module muxbus (A, B, C, D, E0, E1, E2, E3, Q);
    input [7:0]A, B, C, D;
    output [7:0]Q;
    input E0, E1, E2, E3;
    wire [3:0] select4;
    reg [1:0] select2;
    reg [7:0]Q;
    assign select4 = {E0, E1, E2, E3};
    always @ (select4)
    begin
        case(select4)
        4'b0001 : select2 = 2'b00;
        4'b0010 : select2 = 2'b01;
        4'b0100 : select2 = 2'b10;
        4'b1000 : select2 = 2'b11;
        default : select2 = 2'bxx;
        endcase
    end
    always @ (select2 or A or B or C or D)
    begin
        case(select2)
```

```

2'b00 :Q =D;
2'b01 :Q =C;
2'b10 :Q =B;
2'b11 :Q =A;
endcase
end
endmodule

```

Хотелось бы привести несколько примеров по реализации устройств памяти. Приведенный ниже пример иллюстрирует описание статической памяти (SRAM) на VHDL для реализации на ПЛИС, не имеющих встроенных блоков памяти. В ряде случаев этот прием позволяет, не выходя за рамки одного устройства, реализовать небольшой буфер. Пример реализован на VHDL:

```

//#####
//#- Behavioral description of a single-port SRAM with:#
//#- Active High write enable (WE) #
//#- Rising clock edge (Clock) #
//#####
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity reg_sram is
generic (width : integer:=8;
depth : integer:=8;
addr : integer:=3);
port (Data : in std_logic_vector (width-1 downto 0);
Q : out std_logic_vector (width-1 downto 0);
Clock : in std_logic;
WE : in std_logic;
Address : in std_logic_vector (addr-1 downto 0));
end reg_sram;
architecture behav of reg_sram is
type MEM is array (0 to depth-1) of std_logic_
vector(width-1

```



```

downto 0);
signal ramTmp : MEM;
begin
process (Clock)
begin
if (clock'event and clock='1') then
if (WE = '1') then
ramTmp (conv_integer (Address)) <= Data;
end if;
end if;
end process;
Q <= ramTmp(conv_integer(Address));
end behav;

```

Модель на Verilog имеет вид:

```

`timescale 1 ns/100 ps
//#####
//# Behavioral single-port SRAM description :      #
//# Active High write enable (WE)                  #
//# Rising clock edge (Clock)                      #
//#####
module reg_sram (Data, Q, Clock, WE, Address);
parameter width = 8;
parameter depth = 8;
parameter addr = 3;
input Clock, WE;
input [addr-1:0] Address;
input [width-1:0] Data;
output [width-1:0] Q;
wire [width-1:0] Q;
reg [width-1:0] mem_data [depth-1:0];
always @(posedge Clock)
if(WE)
mem_data[Address] = #1 Data;
assign Q = mem_data[Address];
endmodule

```

Следующий пример иллюстрирует создание модели двухпортовой статической памяти (Dual-Port SRAM). Описывается модуль 8×8 ячеек. Описание на VHDL имеет вид:

```

#####
//# Behavioral description of dual-port SRAM with :#
//# Active High write enable (WE) #
//# Active High read enable (RE) #
//# Rising clock edge (Clock) #
#####
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity reg_dpam is
generic (width : integer:=8;
depth : integer:=8;
addr : integer:=3);
port (Data : in std_logic_vector (width-1 downto 0);
Q : out std_logic_vector (width-1 downto 0);
Clock : in std_logic;
WE : in std_logic;
RE : in std_logic;
WAddress: in std_logic_vector (addr-1 downto 0);
RAddress: in std_logic_vector (addr-1 downto 0));
end reg_dpam;
architecture behav of reg_dpam is
type MEM is array (0 to depth-1) of std_logic_
vector(width-1
downto 0);
signal ramTmp : MEM;
begin
-- Write Functional Section
process (Clock)
begin
if (clock'event and clock='1') then
if (WE = '1') then

```

```

    ramTmp (conv_integer (WAddress)) <= Data;
  end if;
end if;
end process;
-- Read Functional Section
process (Clock)
begin
  if (clock'event and clock='1') then
    if (RE = '1') then
      Q <= ramTmp(conv_integer (RAddress));
    end if;
  end if;
end process;
end behav;

```

Модель двухпортовой памяти на Verilog:

```

`timescale 1 ns/100 ps
#####
/** Behavioral dual-port SRAM description :      #
/** Active High write enable (WE)                #
/** Active High read enable (RE)                 #
/** Rising clock edge (Clock)                   #
#####
module reg_dpam (Data, Q, Clock, WE, RE,
  WAddress, RAddress);
  parameter width = 8;
  parameter depth = 8;
  parameter addr = 3;
  input Clock, WE, RE;
  input [addr-1:0] WAddress, RAddress;
  input [width-1:0] Data;
  output [width-1:0] Q;
  reg [width-1:0] Q;
  reg [width-1:0] mem_data [depth-1:0];
#####
/** Write Functional Section                      #

```

```
//#####
    always @(posedge Clock)
    begin
        if(WE)
            mem_data[WAddress] = #1 Data;
        end
//#####
//# Read Functional Section                                     #
//#####
    always @(posedge Clock)
    begin
        if(RE)
            Q = #1 mem_data[RAddress];
        end
    endmodule
```

Глава 7. Примеры реализации алгоритмов ЦОС на ПЛИС

7.1. Реализация цифровых фильтров на ПЛИС семейства FLEX фирмы «Altera»

Многие цифровые фильтры (ЦФ) достаточно просто реализовать в виде КИХ-фильтра. С появлением БИС семейства FLEX8000 и FLEX10K фирмы «Altera» появилась возможность создания высокопроизводительных и гибких КИХ-фильтров высокого порядка. При этом ПЛИС благодаря особенностям своей архитектуры позволяют достигнуть наилучших показателей производительности по сравнению с другими способами реализации ЦФ. Так, при реализации ЦФ на базе ЦОС среднего класса можно достичь производительности обработки данных порядка 5 миллионов отсчетов в секунду (MSPS, million samples per second). Использование готовых специализированных БИС позволяет обеспечить производительность 30...35 MSPS. При использовании ПЛИС семейств FLEX8000 и FLEX10K достигаются величины более 100 MSPS.

Рассмотрим особенности реализации КИХ-фильтров на базе ПЛИС семейств FLEX8000 и FLEX10K с учетом специфики их архитектуры на примере КИХ-фильтра с 8 отводами (**Рис. 7.1**). На **Рис. 7.1** введены следующие обозначения: x_n — сигнал на выходе n -го регистра, h_n — коэффициент СФ, y_n — выходной сигнал. Сигнал на выходе фильтра будет иметь вид:

$$y_n = \sum_{n=1}^8 x_n h_n \quad (1)$$

Для обеспечения линейности фазовой характеристики КИХ-фильтра коэффициенты h_n СФ-фильтра выбираются симметричными относительно

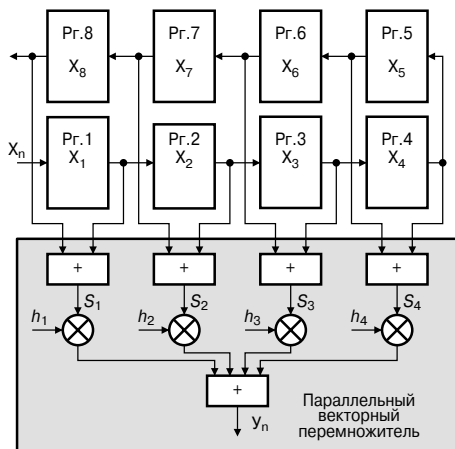


Рис. 7.1. КИХ-фильтр с 8 отводами

центральной величины. Такое построение позволяет сократить число перемножителей. Поскольку компоненты вектора h постоянны для любого фильтра с фиксированными характеристиками, то в качестве параллельного векторного перемножителя (ПВП) удобно использовать таблицу перекодировок, входящую в состав логического элемента [10-13] ПЛИС. Работа параллельного векторного перемножителя описывается следующим уравнением:

$$y = s_1h_1 + s_2h_2 + s_3h_3 + s_4h_4 \quad (2)$$

При использовании ТП операция перемножения выполняется параллельно. В качестве примера рассмотрим реализацию 2-разрядного векторного перемножителя. Пусть вектор коэффициентов h имеет вид:

h_1	h_2	h_3	h_4
01	11	10	11

Вектор сигналов s имеет вид:

s_1	s_2	s_3	s_4
11	00	10	01

Произведение на выходе ПВП принимает вид:

h_n	01	11	10	11	
s_n	11	00	10	01	
Частичное произведение P_1	01	00	00	11	= 100
Частичное произведение P_2	01	00	10	00	= 011
Выходной сигнал y	011	000	100	011	= 1010

Аналогично формируется результат P_1 в случае 4-разрядных сигналов s_n . Частичное произведение P_2 вычисляется аналогично, только результат необходимо сдвинуть на 1 разряд влево. Структура 4-разрядного ПВП представлена на **Рис. 7.2**.

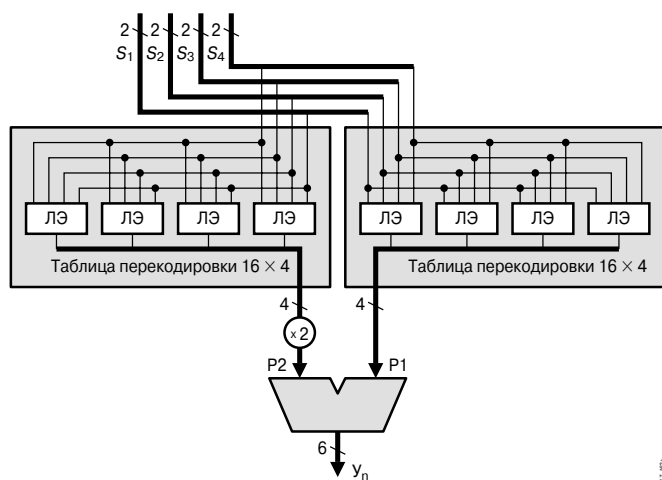


Рис. 7.2. Параллельный векторный перемножитель

Очевидно, что с ростом разрядности представления данных возрастает размерность ТП, а следовательно, требуется большее число ЛЭ для реализации алгоритма фильтрации. Так, для реализации 8-разрядного ПВП требуется 8 ТП размерностью 16×8 . Операция умножения на 2 легко обеспечивается сдвиговыми регистрами.

Включение сдвиговых регистров на выходах промежуточных сумматоров позволяет обеспечить конвейеризацию обработки данных и, следовательно, повысить производительность. При этом регистры входят в состав

соответствующих ЛЭ, поэтому сохраняется компактность структуры, не требуется задействовать дополнительные ЛЭ.

Основным достоинством параллельной архитектуры построения фильтров является высокая производительность. Однако, если число задействованных ЛЭ является критичным, предпочтительнее строить фильтр по последовательной или комбинированной архитектуре. В **Табл. 7.1** приведены сравнительные характеристики КИХ-фильтров одинаковой разрядности и порядка, но выполненных по различной архитектуре.

Таблица 7.1. Сравнительные характеристики КИХ-фильтров

Архитектура	Разрядность данных	Порядок	Размер, число ЛЭ	Тактовая частота [МГц]	Число тактов до получения результата	Быстродействие, [MSPS]
Параллельная	8	16	468	101	1	101
Последовательная	8	16	272	63	9	7

С целью уменьшить число используемых ЛЭ применяется последовательная архитектура построения фильтра. Так же, как и при параллельном построении фильтра, для вычисления частичных P_1, P_2, \dots, P_N , $N = W+1$ произведений применяется ТП, W — разрядность данных. Такой ЦФ обрабатывает только один разряд входного сигнала в течение такта. Последовательно вычисляемые частичные произведения накапливаются в масштабирующем аккумуляторе (МА). МА обеспечивает сдвиг содержимого вправо на один разряд каждый такт. После $W+1$ тактов на выходе появляется результат. Блок управления обеспечивает формирование управляющих сигналов, обеспечивающих корректное выполнение операций.

Комбинированная архитектура является компромиссом между экономичностью и производительностью. В этом случае применяются как последовательные, так и параллельные регистры. В результате распараллеливания вычислений обеспечивается несколько большая производительность, чем у последовательного фильтра.

Одним из эффективных способов повышения производительности фильтра является конвейеризация. В случае, если длина вектора s не является степенью двойки, используются дополнительные регистры для обеспечения синхронизации. С целью построения фильтров более высокого порядка используют последовательное соединение фильтров (каскадирование) меньшего порядка.

Увеличение разрядности данных требует обеспечения большей точности вычислений и разрядности коэффициентов. Увеличение разрядности данных на один разряд требует использования дополнительной ТП при параллельной архитектуре и увеличения на один такт времени фильтрации при последовательной архитектуре. При этом для обеспечения достаточной точности представления данных для фильтра 32-го порядка требуется 19 разрядов длины выходного слова при 8-разрядных входных данных.

Построение фильтра нечетного порядка достигается удалением одного из регистров сдвига. Если применять не сумматоры, а вычитатели, то легко реализовать фильтр с антисимметричной характеристикой.

Довольно легко реализуются фильтры с прореживанием или, наоборот, с интерполяцией отсчетов. При проектировании прореживающего фильтра обеспечивается покаскадное уменьшение тактовой частоты, что позволяет существенно снизить энергопотребление. Интерполирующий фильтр выполняет противоположную операцию — увеличение частоты выборки в определенное число раз. Одним из способов реализации такого алгоритма является дополнение отсчетов нулями.

Возможности архитектуры ПЛИС семейств FLEX позволяют реализовать двумерные фильтры для обработки изображений, а также решетчатые структуры. Для проектирования фильтров необходим набор специализированных программных средств, позволяющих синтезировать требуемую системную функцию фильтра, провести моделирование его работы, а также библиотеки параметризуемых мегафункций, содержащих реализации типовых структур ЦФ.

Фирма «Altera» представляет в составе Altera DSP Design KIT программный продукт FIRGEN. В функции программы FIRGEN входит моделирование КИХ-фильтра с заданными коэффициентами, а также генерация файлов, предназначенных для реализации фильтра штатными средствами пакета MAX+PLUS II. Для построения отклика фильтра, полученного в результате моделирования средствами FIRGEN, можно использовать либо пакет Microsoft Excel, либо входящий в состав Altera DSP Design KIT продукт GNUplot. Схематически процесс разработки фильтра представлен на **Рис. 7.3**. В **Табл. 7.2** сведены данные о мегафункциях фильтров, входящих в состав Altera DSP Design KIT.

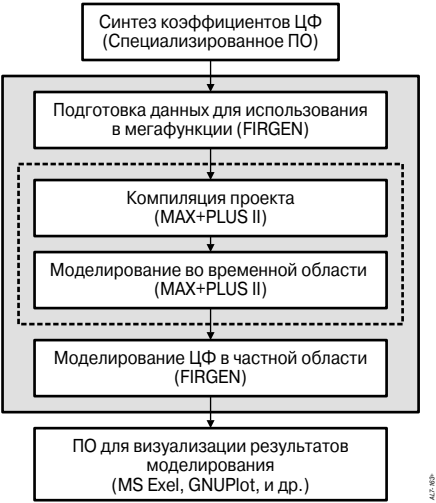


Рис. 7.3. Процесс разработки фильтра в пакете MAX+PLUS II

Таблица 7.2. Данные о мегафункциях фильтров

Модель	Разрядность входных данных	Число отводов	Разрядность представления			Размер, число ЛЭ	Производительность [MSPS]	
			Коэффициентов	Внутренняя	Выходных данных		С конвейером	Без конвейера
Fir_08tp	8	8	8	17	17	296	101	28
Fir_16tp	8	16	8	10	10	468	101	20
Fir_24tp	8	24	8	10	10	653	100	18
Fir_32tp	8	32	8	10	10	862	101	18
Fir_16ts	8	16	8	18	18	272	7	3.4
Fir_64ts	8	64	8	24	24	920	6.5	2.4
Fir_3x3	8	9	8	18	18	327	102	24

7.2. Реализация цифровых полиномиальных фильтров

В общем случае цифровой полиномиальный фильтр размерности r и порядка M определяется конечным дискретным рядом Вольтерра (функциональным полиномом) вида:

$$y(n) = h_0 + \sum_{m=1}^M y_m(n) = h_0 + \sum_{m=1}^M \sum_{n_1} \dots \sum_{n_m} h_m(n_1, \dots, n_m) \prod_{i=1}^m x(n-n_i), \quad (3)$$

где $h_m(n_1, \dots, n_m)$ — многомерные импульсные характеристики (ядра) фильтра, зависящие от векторных аргументов $n_i = [n_{i1} \dots n_{ir}]$. Фильтры данного вида часто называются также фильтрами Вольтерра.

Непосредственная реализация цифровых полиномиальных фильтров связана с вычислением произведения векторов. Вычислительные затраты могут быть сокращены за счет использования свойства симметрии изотропных фильтров.

Таким образом, путем последовательного применения процедуры декомпозиции полиномиальный фильтр произвольного порядка может быть представлен в виде параллельной структуры, состоящей из линейных фильтров. Операция линейной фильтрации связана с вычислением двухмерной свертки и допускает высокоэффективную реализацию в виде структур систолического типа, матричных и волновых процессоров. Процесс двухмерной свертки изображения с маской $N \times N$, в свою очередь, можно свести к вычислению N одномерных сверток, что в конечном итоге позволяет выполнять полиномиальную фильтрацию изображений путем параллельного вычисления обычных одномерных сверток. На **Рис. 7.4** представлен один из наиболее простых вариантов реализации двухмерной свертки изображения с маской 3×3 в виде систолической структуры, состоящей из 9 идентичных процессорных элементов.

Задача двухмерной свертки формулируется следующим образом: даны веса $w_{i,j}$ для $i, j = 1, 2, \dots, k$, так что $k \times k$ — размер ядра и входное изображение $x_{i,j}$ для $i, j = 1, 2, \dots, n$. Требуется вычислить элементы изображения $y_{i,j}$ для $i, j = 1, 2, \dots, n$, определяемые в виде:

$$y_{i,j} = \sum_{l=1}^k \sum_{h=1}^k w_{hl} x_{i+h-1, j+l-1}. \quad (4)$$

При $k = 3$ двухмерная свертка выполняется в виде трех последо-

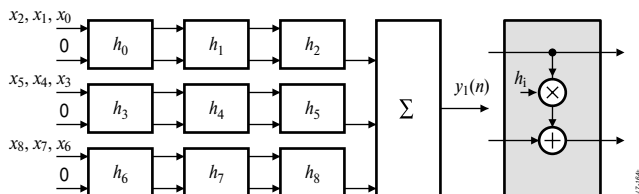


Рис. 7.4. Систолическая реализация двухмерной линейной свертки

вательных одномерных сверток (использующих в качестве весов одну и ту же последовательность $w_{11}, w_{21}, \dots, w_{33}$):

1. Вычисление ($y_{11}, y_{12}, y_{13}, y_{14}, \dots$) при использовании ($x_{11}, x_{21}, x_{31}, x_{12}, x_{22}, x_{32}, x_{13}, x_{23}, x_{33}, \dots$) в качестве входной последовательности.
2. Вычисление ($y_{21}, y_{22}, y_{23}, y_{24}, \dots$) при использовании ($x_{21}, x_{31}, x_{41}, x_{22}, x_{32}, x_{42}, x_{23}, x_{33}, x_{43}, \dots$) в качестве входной последовательности.
3. Вычисление ($y_{31}, y_{32}, y_{33}, y_{34}, \dots$) при использовании ($x_{31}, x_{41}, x_{51}, x_{32}, x_{42}, x_{52}, x_{33}, x_{43}, x_{53}, \dots$) в качестве входной последовательности.

Каждую из этих сверток можно выполнить на одномерном систолическом массиве из девяти ячеек. Отметим, что в любом из этих одномерных массивов ячейка занята вычислениями y_{ij} только 1/3 времени.

Для восстановления ячейка с двумя потоками должна быть 8-разрядной для входных данных и иметь 12 бит для представления весов. Для сетки 32×32 пикселей в каждом блоке памяти требуется хранить 1024 различных весовых коэффициента и иметь 10-разрядную адресную шину для указания положения каждой точки. 12-разрядный умножитель и 24-разрядный сумматор для получения промежуточных результатов вполне обеспечивают сохранение требуемой точности в процессе вычислений.

Для выполнения многомерной свертки требуется единственная модификация базовой ячейки с двумя потоками — добавление требуемого числа систолических входных потоков и соответствующее увеличение размера мультиплексора для выбора данных.

При решении задач фильтрации изображения с целью удаления шумов, восстановления изображения или улучшения его качества приходится иметь дело с данными, имеющими широкий динамический диапазон. Поэтому в качестве формата данных необходимо использовать числа с плавающей запятой.

Для реализации систолических структур полиномиальных фильтров наиболее пригодны ПЛИС семейства FLEX10K, содержащие встроенные

блоки памяти, предназначенные для эффективной реализации функций памяти и сложных арифметических и логических устройств (умножителей, конечных автоматов, цифровых фильтров и т.д.). Один такой блок имеет емкость 2 Кбит и позволяет сформировать память с организацией 2048×1 , 1024×2 , 512×4 или 256×8 , работающую с циклом 12...14 нс. Использование ВБП значительно повышает эффективность и быстродействие создания сложных логических устройств, например умножителей. Так, каждый ВБП может выполнять функции умножителя 4×4 , 5×3 или 6×2 .

На **Рис. 7.5** приведена реализация структуры на ПЛИС, а на **Рис. 7.6** — результаты моделирования систолической структуры в среде MAX PLUS II.

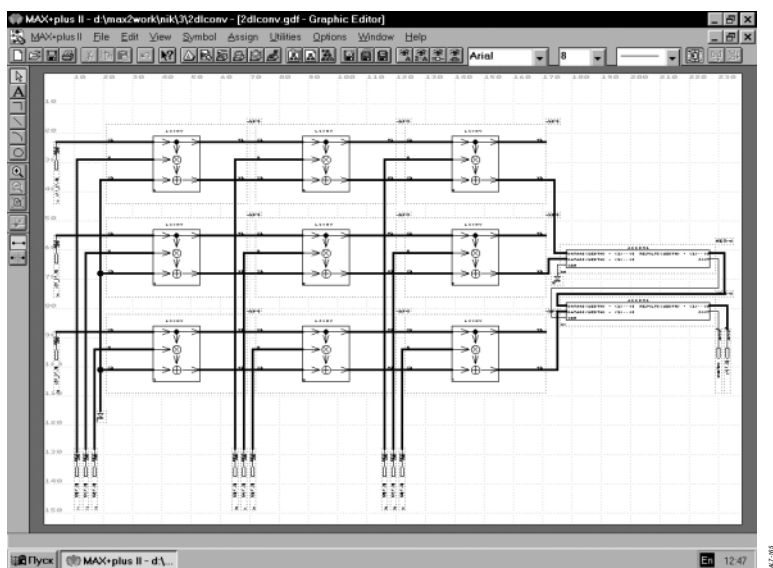


Рис. 7.5. Реализация систолической структуры на ПЛИС

Включение сдвиговых регистров на выходах промежуточных сумматоров позволяет обеспечить конвейеризацию обработки данных и, следовательно, повысить производительность. При этом регистры входят в состав соответствующих ЛЭ, поэтому сохраняется компактность структуры и не требуется задействовать дополнительные ЛЭ.

При реализации операции умножения на константу и возведения в степень целесообразно использовать конструкцию TABLE языка AHDL, как

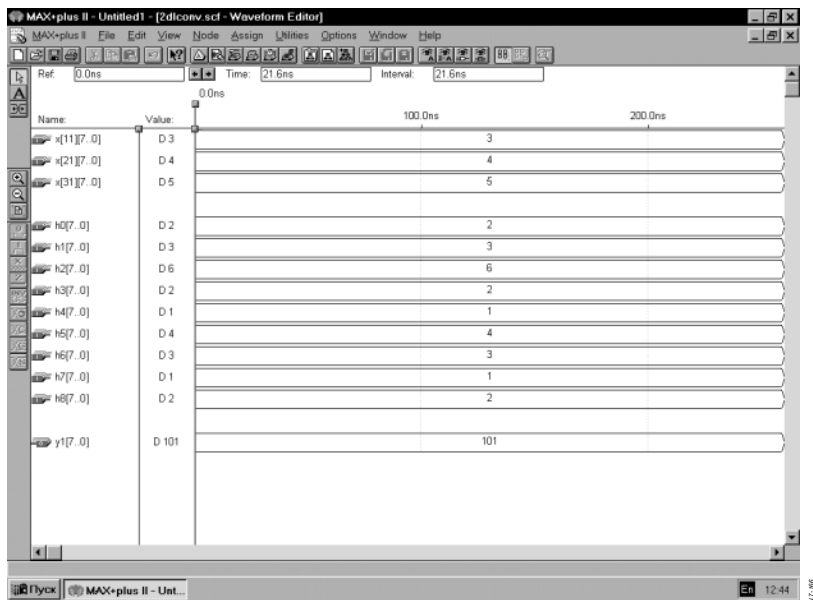


Рис. 7.6. Результаты моделирования систолической структуры на ПЛИС

показывает сравнение, такое построение позволяет обеспечить более компактную и быстродействующую реализацию. Ниже приводится пример такой конструкции:

```
TABLE
net[11..1] => out[15..1];
B"00000000XXXX" => B"0000000000000000";
B"000000010000" => B"0000000101011101";
B"000000100000" => B"0000001010111010";
B"111111110000" => B"0100111000011101";
B"000000010001" => B"0100111000011110";
End TABLE;
```

При реализации перемножителей, как правило, используются матричные структуры, построенные каскадированием битовых сумматоров.

7.3. Алгоритмы функционирования и структурные схемы демодуляторов

Обобщенная структурная схема, по которой реализован демодулятор сигналов с частотной манипуляцией (ЧМн), приведена на **Рис. 7.7**. Как видим, используется аналоговая квадратурная обработка (перемножители, фазовращатель, фильтры нижних частот) и цифровое восстановление символов и символьной частоты, реализованное на ПЛИС. В качестве формирователя квадратур использована ИС RF2711 фирмы «RF Microdevices». Данная микросхема содержит в своем составе два перемножителя и фазовращатель и позволяет работать в диапазоне частот от 0.1 до 200 МГц при ширине спектра до 25 МГц. Опорная частота f_0 формируется с помощью синтезатора прямого синтеза частот AD9830 [1] фирмы «Analog Devices». Сигнал с выхода синтезатора фильтруется с помощью активного ФНЧ, реализованного на операционном усилителе (ОУ) AD8052 по схеме Рауха. В настоящее время наблюдается тенденция к «оцифровыванию» обрабатываемого сигнала на промежуточных частотах (ПЧ) порядка десятков МГц, формируя квадратуры «в цифре». Для этих целей возможно использовать ИС AD6620. Однако это не всегда оправдано, в основном из-за сложностей с управлением такой микросхемой в системах, где отсутствует собственный контроллер.

В качестве АЦП удобно использовать специализированный квадратурный аналого-цифровой преобразователь (АЦП) AD9201. Пожалуй, единственным его недостатком является необходимость демультимплексирования отсчетов синфазной и квадратурной составляющих.

Ниже приводится описание конструкций каждого из блоков в составе ПЛИС: детектора ЧМн-сигнала и синхронизатора. Определены преимущества каждой из предлагаемых схем.

Детектор ЧМн-сигнала предназначен для преобразования исходного модулированного радиосигнала в последовательность прямоугольных им-

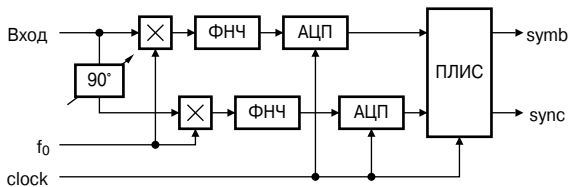


Рис. 7.7. Структурная схема демодулятора сигналов с частотной манипуляцией

пульсов, появляющихся с частотой следования символов и обладающих той же поляриностью. Частота исходного радиосигнала равна $f = f_0 - \Delta f/2$, если передается символ «0», и $f = f_0 + \Delta f/2$, если передается символ «1»; при этом по техническому заданию $\Delta f T_c = 1$, где T_c — длительность символа (индекс модуляции единица, сигнал без разрыва фазы). Таким образом, измерение разности $(f - f_0)$ — это и есть та операция, которую должен осуществлять детектор. В Приложении 1 показано, что при наличии отсчетов квадратур исходного радиосигнала: S_k и C_k , $k = 0, 1, 2$ и т.д., величина $(f - f_0)_k$ может быть вычислена следующим образом:

$$(f - f_0)_k = \frac{1}{2\pi} \left[\frac{S_k C_{k-1} - C_k S_{k-1}}{S_k^2 + C_k^2} \right]. \quad (5)$$

Отсюда вытекает структурная схема детектора, которая приведена на **Рис. 7.8**.

Следует отметить преимущества предлагаемого алгоритма демодуляции ЧМн-сигнала:

- детектор не требует точной настройки квадратурного генератора (**Рис. 7.8**) на частоту f_0 , что позволяет ему устойчиво функционировать при значительных (до 30%) уходах частоты входного сигнала вследствие эффекта Доплера;
- операция деления на двучлен не является обязательной, если динамика входного сигнала невелика либо стабилизация амплитуды осуществляется при помощи Автоматической регулировки усиления (АРУ) в ВЧ-тракте;
- инвариантность алгоритма к фазе опорного и входного сигналов, а также к амплитуде входного сигнала (при наличии нормирующего множителя) увеличивает помехоустойчивость.

Синхронизатор. При достаточно больших отношениях сигнал/шум (ОСШ) на входе демодулятора (20...30 дБ) восстановленную последовательность

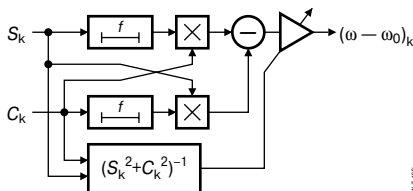


Рис. 7.8. Структурная схема детектора

символов можно снимать непосредственно с выхода детектора. Однако при снижении ОСШ (до 10...15 дБ) форма сигнала на выходе детектора начинает искажаться (появляются ложные перепады, смещение фронтов по времени и т.п.). Поэтому на выход детектора (внутри ПЛИС) подключается еще один блок-синхронизатор, предназначение которого — восстановить истинную форму демодулированного радиосигнала за счет его накопления и анализа в течение N подряд идущих символов (в описанной далее версии демодуляторов $N = 10$). Синхронизатор реализует оптимальный (по критерию максимума правдоподобия) алгоритм оценки сигнала прямоугольной формы на фоне белого гауссовского шума. Восстановлению подлежат истинные моменты смены символов в исходном радиосигнале (тактовая синхронизация), а также истинная полярность символов.

Главными элементами синхронизатора (**Рис.7.9**) являются линия задержки на $N \times M$ отсчетов (M — число отсчетов на символ) и $(N + 1)$ сумматоров, реализующих операцию накопления. Синхронизатор функционирует следующим образом: каждый отсчет входного сигнала порождает сдвиг в линии задержки, после чего вычисляются суммы каждой M подряд идущих отсчетов, определяются их модули и производится усреднение результатов по N суммам (символам). Если в какой-то момент времени каждое суммирование (по M отсчетам) будет производиться внутри одного символа, значение усредненного сигнала будет максимальным, на выходе порогового устройства

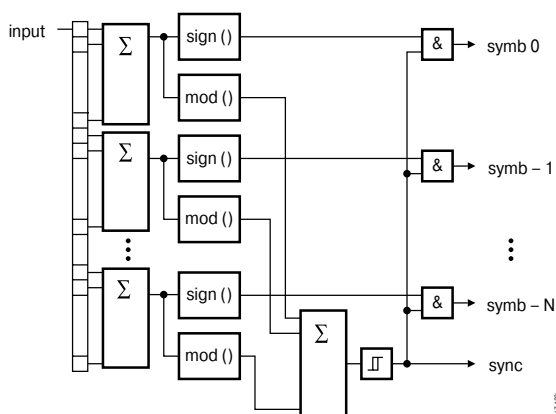


Рис. 7.9. Структурная схема синхронизатора

появится синхроимпульс, и в этот же момент будут считаны знаки накопленных сумм, с высокой вероятностью совпадающие с полярностями символов.

К преимуществам предлагаемого алгоритма следует отнести:

- высокую эффективность (устойчивость к помехам, к уходу частоты следования символов от номинальной, к снижению частоты дискретизации и др.);

- способность точно восстанавливать моменты смены символов во входном сигнале при длинных (до $N - 1$) включительно) сериях «нулей» и «единиц», причем в конце серии отсутствует переходный процесс (направленный на устранение накопленной ошибки), что характерно для аналоговых устройств;

- наличие на выходе демодулятора одновременно N подряд идущих символов, что может быть важно при корреляционной обработке потока данных (например, при поиске синхропосылок);

- простоту операций (суммирование, сдвиг) и хорошую адаптацию к реализации на базе ПЛИС, что не характерно для большинства традиционных алгоритмов, содержащих петли обратной связи (синхронизатор с запаздывающим и опережающим стробированием и др.).

Реализация цифровой части алгоритма демодуляции и выделения синхроимпульса была выполнена на кристалле фирмы «Altera» FLEX10K50. Реализованное устройство состоит из входных цифровых КИХ-фильтров с восемью отводами, непосредственно блока демодуляции сигнала и блока синхронизатора.

Для реализации входных КИХ-фильтров был использован пакет Altera DSP Design Kit. Данный пакет был выбран ввиду того, что он позволяет по рассчитанным коэффициентам цифрового фильтра быстро получить AHDL-описание устройства с данными характеристиками и максимально доступной точностью при заданных значениях точности входных данных и внутреннего представления коэффициентов фильтрации. Для этого исходные коэффициенты фильтра масштабируются, что приводит к тому, что выходной сигнал фильтра также является промасштабированным на ту же величину, которую в большинстве случаев необходимо учитывать при дальнейших вычислениях. Однако для данной задачи это не является существенным, и масштабирующий множитель не учитывался в последующих операциях. Кроме того, Altera DSP Design Kit позволяет сгенерировать векторный файл для моделирования работы фильтра, а также позволяет преобразовать выходные данные отклика фильтра к масштабу входных данных и построить график отклика фильтра на входное воздействие.

Реализованный восьмиотводный фильтр имеет симметричную характеристику. В отличие от классической реализации КИХ-фильтров в виде набора умножителей для взвешивания задержанных отсчетов входного сигнала и выходного сумматора данная реализация вообще не содержит умножителей. Все операции умножения заменены операциями распределенной арифметики, что возможно благодаря постоянству коэффициентов фильтрации и наличию в логических элементах FLEX10K таблиц перекодирования. Сигналы с выходов фильтров (восемь бит в дополнительном коде) подаются на квадратурные входы блока частотной демодуляции.

Для реализации демодулятора ЧМн-сигналов понадобились два регистра для хранения значений квадратур в предыдущий (k -й) момент времени — S_k и C_k , два умножителя и один сумматор. Все вычисления в схеме производятся в дополнительном коде, за исключением умножителей, операнды и выходные данные которых представлены в прямом коде со знаком, что требует предварительно преобразовывать сигналы в дополнительный код до умножения и конвертировать в дополнительный код результат умножения. Выходной сигнал демодулятора имеет разрядность, равную пятнадцати битам, однако для выделения символов нужно рассматривать только старший (знаковый) разряд результата. Входным сигналом синхронизатора является выход блока демодулятора.

Заметим, что для реализации суммирования вида:

$$sum_i = y_k + y_{k-1} + y_{k-2} + \dots + y_{k-n+1} \quad (6)$$

нецелесообразно использовать каскад из n двухвходовых сумматоров, так как на каждом такте результат этого суммирования может быть получен из значения суммы на предыдущем такте путем вычитания y_{k-n-1} и прибавления. А именно: $sum_{i+1} = sum_i + y_{k+1} - y_{k-n-1}$.

Таким образом, для реализации этой части алгоритма синхронизации понадобится один регистр для хранения значения суммы на предыдущем такте и три сумматора, один из которых используется для изменения знака значения y_{k-n-1} (так как все числа представлены в дополнительном коде). Кроме того, необходимы регистры для хранения значений y_k , y_{k-1} , y_{k-n+1} . Если же не учитывать эти n регистров, то количество элементов для выполнения такой операции суммирования не будет зависеть от числа операндов и позволит сэкономить ячейки ПЛИС при количестве слагаемых в сумме $n > 4$. В данном же случае количество слагаемых в каждой сумме равно количеству отсчетов сигнала, приходящихся на символ, т.е., $n = 8$, и эффект в увеличении скорости, а главное, в уменьшении занимаемого места ощутим.

Был реализован блок суммирования, выходными сигналами которого являлись как значение суммы, так и значение, для удобства последовательного соединения таких блоков по входам, что необходимо ввиду того, что выходящее из одной суммы слагаемое становится слагаемым следующей суммы:

$$X_k = \lnch(sum_i) + \lnch(sum_{i+1}) + \lnch(sum_{i+m-1}). \quad (7)$$

Так как синхронизатор работает по принципу максимума правдоподобия, то схема должна выставлять синхроимпульс в момент достижения выходным сигналом максимума. Для определения момента наступления локальных максимумов этот сигнал дифференцируется, и определяются моменты смены знака продифференцированного сигнала. Результаты моделирования в системе MAX+PLUS II приведены на **Рис. 7.10**.



Рис. 7.10. Результаты моделирования в системе MAX+PLUS II

В заключение отметим, что все узлы системы были реализованы в виде параметризованных мегафункций с использованием языка описания аппаратуры AHDL, что позволяет с легкостью использовать их для приложений, требующих другой точности вычислений.

7.4. Реализация генератора ПСП на ПЛИС

Генератор формирования М-последовательностей был написан в виде параметризованной макрофункции, описывающей устройство, упрощенная структура которого показана на **Рис. 7.11**. Параметрами макрофункции являются длина характеристического многочлена и число, описывающее начальные состояния триггеров. Ниже приводится листинг описания этой функции на AHDL.

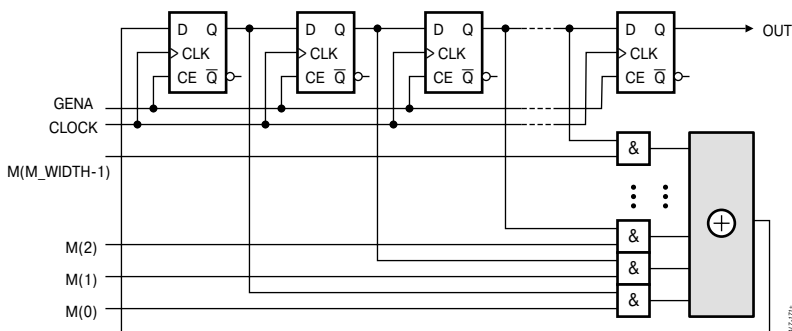


Рис. 7.11. Структурная схема генератора формирования М-последовательностей

Функция M_GENERATE:

Функция M_GENERATE

Генерирует М-последовательность, заданную
характеристическим многочленом М

Входы:

- М - вход многочлена;
- GENA - разрешение генерации (default=GND);
- CLK - тактовый вход;
- LOAD - вход предустановки (default=GND);

Выходы:

- OUT - выход М-последовательности

Параметры:

- M_WIDTH - длина полинома;
- M_BEGIN - начальное состояние
(default="100000..00");

Версия 1.0

```

INCLUDE "lpm_xor.inc";
INCLUDE "lpm_constant.inc";

PARAMETERS
(
    M_WIDTH,
    M_BEGIN = 0
);

SUBDESIGN m_generate
(
    M[M_WIDTH-1..0]      : INPUT = GND;
    CLK                  : INPUT;
    GENA                 : INPUT = GND;
    LOAD                 : INPUT = GND;
    OUT                  : OUTPUT;
)
VARIABLE
    dffs[M_WIDTH-2..0]   : DFFE;
-- триггеры регистра сдвига
    shift_node[M_WIDTH-2..0] : NODE;
    xor_node[M_WIDTH-2..0]   : NODE;
    shiftin, shiftout        : NODE;
-- вход и выход регистра сдвига

    IF (USED(M_BEGIN)) GENERATE
        ac : lpm_constant
            WITH (LPM_WIDTH = M_WIDTH,
LPM_CVALUE = M_BEGIN);
    END GENERATE;

BEGIN

    ASSERT (M_WIDTH>0)
        REPORT "Значение параметра M_WIDTH должно
быть больше нуля"

```

```
SEVERITY ERROR;
%----- общие выводы триггеров-----%
dffs[].ena = GENA;
dffs[].clk = CLK;

%----- асинхронные операции -----%
IF (USED(M_BEGIN)) GENERATE
dffs[].clrn = !load # ac.result[M_WIDTH-2..0];
установка начального состояния
dffs[].prn = !load # !ac.result[M_WIDTH-2..0];
триггеров, заданное M_BEGIN
ELSE GENERATE
dffs[M_WIDTH-3..0].clrn = !load;
установка начального состояния
dffs[M_WIDTH-2].prn = !load;
триггеров "10000..00
END GENERATE;

%----- обратные связи -----%
xor_node[] = dffs[] & M[M_WIDTH-2..0];
shiftin = lpm_xor(xor_node[])
WITH (LPM_SIZE = M_WIDTH-1, LPM_WIDTH=1);

%----- операции сдвига -----%
shift_node[] = (shiftin, dffs[M_WIDTH-2..1]);
shiftout = dffs[0];

%-----синхронные операции -----%
dffs[].d = !load & shift_node[];

%-----подключим выход -----%
OUT = shiftout;
END;
```

Описание функции:

Входы:

- M — шина (группа выводов) с размером M_WIDTH. На этот вход подаются коэффициенты характеристического многочлена;
- CLK — вход тактового сигнала;
- GENA — вход разрешения генерации. При GENA = «0» генерация M-последовательности запрещена;
- LOAD — вход предустановки. При LOAD = «1» триггеры регистра сдвига устанавливаются в состояние, определяемое параметром M_BEGIN.

Выходы:

- OUT — выход M-последовательности.

Для включения данной функции в другие схемы были также созданы включаемый файл, содержащий описание, и графический символ-элемент (см. **Рис. 7.12**).

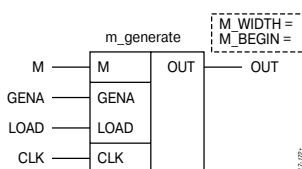


Рис. 7.12. Графический символ-элемент

Файл «M_GENERATE.INC»:

```
FUNCTION m_generate (m[(m_width) - (1)..0], clk,
gena, load)
  WITH (M_WIDTH, M_BEGIN)
    RETURNS (out);
```

Для моделирования работы генератора в графическом редакторе пакета MAX+PLUS II была создана тестовая схема, показанная на **Рис. 7.13**. Результаты моделирования показаны на **Рис. 7.14** (на вход одного генератора подан характеристический многочлен $M_1 = 11001$, на вход другого — $M_2 = 10010000001$).

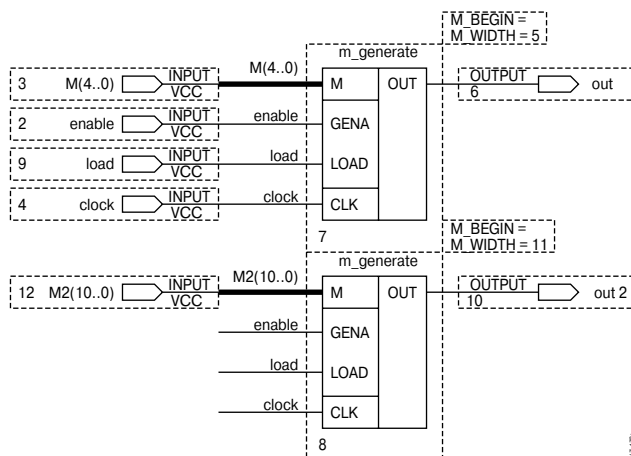


Рис. 7.13. Тестовая схема в графическом редакторе пакета MAX+PLUS II

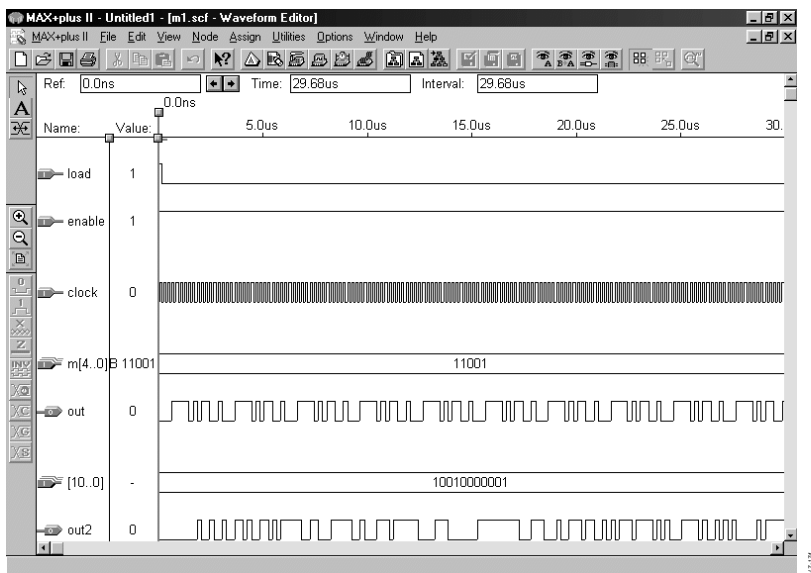


Рис. 7.14. Результаты моделирования

7.5. Примеры описания цифровых схем на VHDL

Рассмотрим некоторые примеры описания цифровых схем на VHDL.

Примером описания цифрового автомата является преобразователь параллельного кода в последовательный. Преобразователь кода представляет собой устройство, на вход которого подаются n -разрядное число в параллельном коде d , сигнал загрузки $load$ и синхроимпульсы clk . По сигналу загрузки происходит запись входного слова во внутренний регистр и последовательная выдача в течение n тактов этого входного слова в последовательном коде на выходе o синхроимпульсами $oclk$. После окончания преобразования на выходе e появляется ВЫСОКИЙ уровень сигнала в течение одного такта. Такого рода преобразователи кода часто используются для управления синтезаторами частот 1104ПЛ1 и им подобными. Описание этого устройства на языке VHDL приведено ниже.

```
library ieee;
use ieee.std_logic_1164.all;

entity Serial is
  port (
    clk    : in STD_LOGIC;
    load   : in STD_LOGIC;
    reset  : in STD_LOGIC;
    d      : in STD_LOGIC_vector (3 downto 0);
    oclk   : out STD_LOGIC;
    o      : out STD_LOGIC;
    e      : out STD_LOGIC
  );
end;

architecture behavioral of Serial is
  type t1 is range 0 to 4;
  signal s : STD_LOGIC_vector (2 downto 0);
  signal i : t1;

begin

  process (clk)
```

```
begin
if reset = '1' then
    i <= 0;
else
    if (clk'event and clk='1') then
        if (i = 0 and load = '1') then
            s(2 downto 0) <= d(3 downto 1);
            o <= d(0);
            i <= 4;
        end if;
        if (i > 1) then
            o <= s(0);
            s(1 downto 0) <= s(2 downto 1);
            i <= i - 1;
        end if;
        if (i = 1) then
            e <= '1';
            i <= 0;
        else
            e <= '0';
        end if;
    end if;
end if;
if i>0 then
    oclk <= not clk;
else
    oclk <= '0';
end if;

end process;

end behavioral;
```

По переднему фронту синхроимпульса *clk* при ВЫСОКОМ уровне на входе загрузки происходит загрузка трех старших разрядов входного слова *d*[3..1] во временный регистр *s*[2..0]. Младший разряд входного слова *d*[0] подается на выход *o*. На выходе *oclk* появляются синхроимпульсы. На сигнале *i* собран внутренний счетчик, выдающий сигнал окончания преобразования *e*. При по-

ступлении последующих синхроимпульсов происходит выдача на выход остальных бит входного слова, хранящихся в регистре $s[2..0]$.

Моделирование этого устройства было проведено в системе проектирования OrCAD 9.0. Для тестирования схемы использовался тест:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity test_serial is end test_serial;
architecture testbench of test_serial is
component serial
    port (
        clk : in std_logic;
        load : in std_logic;
        reset : in std_logic;
        d : in std_logic_vector(3 downto 0);
        oclk : out std_logic;
        o : out std_logic;
        e : out std_logic
    );
end component;
signal clk : std_logic;
signal load : std_logic := '0';
signal reset : std_logic;
signal d : std_logic_vector(3 downto 0);
signal oclk : std_logic;
signal o : std_logic;
signal e : std_logic;
begin
    process begin
        for i in 0 to 50 loop
            clk <= '0'; wait for 5 ns;
            clk <= '1'; wait for 5 ns;
        end loop;
    end process;
    process begin
        reset <= '1'; wait for 10 ns;
        reset <= '0' ;
```

```

        load <= '1';
        d <= "1010"; wait for 10 ns;
        load <= '0';
        d <= "0000"; wait for 500 ns;
    end process;
dut : serial_port map (
    clk => clk,
    load => load,
    reset => reset,
    d => d,
    oclk => oclk,
    o => o,
    e => e
);
end testbench;
```

Результаты моделирования представлены на **Рис. 7.15**.

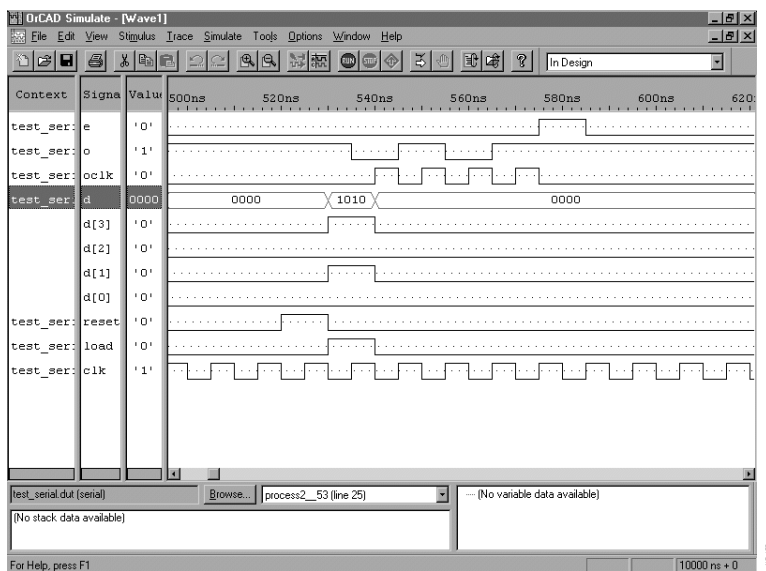


Рис. 7.15. Результаты моделирования

В качестве примера описания устройства ЦОС рассмотрим цифровой КИХ-фильтр. Работа цифрового КИХ-фильтра описывается разностным уравнением:

$$y_n = A_0x_n + A_1x_{n-1} + A_2x_{n-2} + \dots, \quad (8)$$

где y_n — реакция системы в момент времени n ;

x_n — входное воздействие;

A_i — весовой коэффициент i -й входной переменной.

На VHDL описание фильтра имеет вид:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity f is
port (
    din: in std_logic_vector(7 downto 0);
    sout: out std_logic_vector(15 downto 0);
    r: in std_logic;
    c: in std_logic
);
end f;
architecture behavior of f is
    constant h00 : std_logic_vector(7 downto 0) :=
"00000000";
    constant h01 : std_logic_vector(7 downto 0) :=
"00000001";
    constant h02 : std_logic_vector(7 downto 0) :=
"00000100";
    constant h03 : std_logic_vector(7 downto 0) :=
"00001111";
    constant h04 : std_logic_vector(7 downto 0) :=
"00100100";
    constant h05 : std_logic_vector(7 downto 0) :=
"01000010";
    constant h06 : std_logic_vector(7 downto 0) :=
"01100100";
    constant h07 : std_logic_vector(7 downto 0) :=
```

```
"01111100";
  constant h08 : std_logic_vector(7 downto 0) :=
"01111111";
  constant h09 : std_logic_vector(7 downto 0) :=
"01101010";
  constant h10 : std_logic_vector(7 downto 0) :=
"01000010";
  constant h11 : std_logic_vector(7 downto 0) :=
"00010011";
  constant h12 : std_logic_vector(7 downto 0) :=
"11101100";
  constant h13 : std_logic_vector(7 downto 0) :=
"11010110";
  constant h14 : std_logic_vector(7 downto 0) :=
"11010101";
  constant h15 : std_logic_vector(7 downto 0) :=
"11100011";
  constant h16 : std_logic_vector(7 downto 0) :=
"11110111";
  constant h17 : std_logic_vector(7 downto 0) :=
"00001010";
  constant h18 : std_logic_vector(7 downto 0) :=
"00010100";
  constant h19 : std_logic_vector(7 downto 0) :=
"00010011";
  constant h20 : std_logic_vector(7 downto 0) :=
"00001100";
  constant h21 : std_logic_vector(7 downto 0) :=
"00000010";
  constant h22 : std_logic_vector(7 downto 0) :=
"11111000";
  constant h23 : std_logic_vector(7 downto 0) :=
"11110101";

  signal x00, x01, x02, x03, x04, x05, x06, x07,
         x08, x09, x10, x11, x12, x13, x14, x15,
         x16, x17, x18, x19, x20, x21,
         x22, x23 : std_logic_vector(7 downto 0);
```

```
signal m00, m01, m02, m03, m04, m05, m06, m07,  
       m08, m09, m10, m11, m12, m13, m14, m15,  
       m16, m17, m18, m19, m20, m21,  
       m22, m23 : std_logic_vector(15 downto 0);
```

```
begin
```

```
m00 <= (signed(x00)*signed(h00));  
m01 <= (signed(x01)*signed(h01));  
m02 <= (signed(x02)*signed(h02));  
m03 <= (signed(x03)*signed(h03));  
m04 <= (signed(x04)*signed(h04));  
m05 <= (signed(x05)*signed(h05));  
m06 <= (signed(x06)*signed(h06));  
m07 <= (signed(x07)*signed(h07));  
m08 <= (signed(x08)*signed(h08));  
m09 <= (signed(x09)*signed(h09));  
m10 <= (signed(x10)*signed(h10));  
m11 <= (signed(x11)*signed(h11));  
m12 <= (signed(x12)*signed(h12));  
m13 <= (signed(x13)*signed(h13));  
m14 <= (signed(x14)*signed(h14));  
m15 <= (signed(x15)*signed(h15));  
m16 <= (signed(x16)*signed(h16));  
m17 <= (signed(x17)*signed(h17));  
m18 <= (signed(x18)*signed(h18));  
m19 <= (signed(x19)*signed(h19));  
m20 <= (signed(x20)*signed(h20));  
m21 <= (signed(x21)*signed(h21));  
m22 <= (signed(x22)*signed(h22));  
m23 <= (signed(x23)*signed(h23));
```

```
sout <=(signed(m00)+signed(m01)+signed(m02)  
        +signed(m03)+signed(m04)+signed(m05)  
        +signed(m06)+signed(m07)+signed(m08)  
        +signed(m09)+signed(m10)+signed(m11)  
        +signed(m12)+signed(m13)+signed(m14)  
        +signed(m15)+signed(m16)+signed(m17)  
        +signed(m18)+signed(m19)+signed(m20)
```



```
+signed(m21)+signed(m22)+signed(m23));

process (c,r)
begin
  if r='1' then
    x00 <= (others => '0');
    x01 <= (others => '0');
    x02 <= (others => '0');
    x03 <= (others => '0');
    x04 <= (others => '0');
    x05 <= (others => '0');
    x06 <= (others => '0');
    x07 <= (others => '0');
    x08 <= (others => '0');
    x09 <= (others => '0');
    x10 <= (others => '0');
    x11 <= (others => '0');
    x12 <= (others => '0');
    x13 <= (others => '0');
    x14 <= (others => '0');
    x15 <= (others => '0');
    x16 <= (others => '0');
    x17 <= (others => '0');
    x18 <= (others => '0');
    x19 <= (others => '0');
    x20 <= (others => '0');
    x21 <= (others => '0');
    x22 <= (others => '0');
    x23 <= (others => '0');
  elsif (c'event and c='1') then
    x00(7 downto 0) <= din(7 downto 0);
    x01(7 downto 0) <= x00(7 downto 0);
    x02(7 downto 0) <= x01(7 downto 0);
    x03(7 downto 0) <= x02(7 downto 0);
    x04(7 downto 0) <= x03(7 downto 0);
    x05(7 downto 0) <= x04(7 downto 0);
    x06(7 downto 0) <= x05(7 downto 0);
    x07(7 downto 0) <= x06(7 downto 0);
```

```

x08(7 downto 0) <= x07(7 downto 0);
x09(7 downto 0) <= x08(7 downto 0);
x10(7 downto 0) <= x09(7 downto 0);
x11(7 downto 0) <= x10(7 downto 0);
x12(7 downto 0) <= x11(7 downto 0);
x13(7 downto 0) <= x12(7 downto 0);
x14(7 downto 0) <= x13(7 downto 0);
x15(7 downto 0) <= x14(7 downto 0);
x16(7 downto 0) <= x15(7 downto 0);
x17(7 downto 0) <= x16(7 downto 0);
x18(7 downto 0) <= x17(7 downto 0);
x19(7 downto 0) <= x18(7 downto 0);
x20(7 downto 0) <= x19(7 downto 0);
x21(7 downto 0) <= x20(7 downto 0);
x22(7 downto 0) <= x21(7 downto 0);
x23(7 downto 0) <= x22(7 downto 0);
end if;
end process;
end behavior;

```

Входные данные считываются с входа `din[7..0]` в дополнительном коде по переднему фронту синхросигнала `s`. На сигналах `x0 ÷ x23` построен сдвиговый регистр, обеспечивающий задержку данных на 24 такта. Сигналы с регистров умножаются на весовые коэффициенты `h0 ÷ h23` и суммируются. Для тестирования схемы использован тест:

```

-- Test bench shell

library ieee;
use ieee.std_logic_1164.all;

entity test_f is end test_f;

architecture testbench of test_f is

component f
port (

```

```
        din : in std_logic_vector(7 downto 0);
        sout : out std_logic_vector(15 downto 0);
        r : in std_logic;
        c : in std_logic
    );
end component;
signal    din : std_logic_vector(7 downto 0);
signal    sout : std_logic_vector(15 downto 0);
signal    r : std_logic;
signal    c : std_logic;

begin

    process begin
        for i in 0 to 50 loop
            c <= '0'; wait for 5 ns;
            c <= '1'; wait for 5 ns;
        end loop;
    end process;
    process begin
        r <= '1'; wait for 10 ns;
        r <= '0' ;
        din <= "00000001"; wait for 10 ns;
        din <= "00000000"; wait for 500 ns;
    end process;

    dut : f port map (
        din => din,
        sout => sout,
        r => r,
        c => c
    );

end testbench;
```

Тест моделирует подачу на цифровой фильтр аналога δ -функции. На выходе фильтра — его импульсная характеристика. Результаты моделирования представлены на **Рис. 7.16**.



Рис. 7.16. Результаты моделирования

7.6. Реализация нейрона на AHDL

Рассмотрим реализацию на языке описания аппаратуры структуры трех-входового (количество входов легко варьируется) нейрона с логистической функцией активации (также может варьироваться). Весовые коэффициенты являются переменными, задаваемыми пользователем. Входные величины (как входы, так и веса) представлены шестнадцатиразрядным кодом:

$\underbrace{X}_{\text{Знак}}.\underbrace{XXXXXXXXXXXXXXXXXX}_{\text{Мантисса}}$

При обработке данных шестнадцатиразрядный код преобразуется к двенадцатиразрядному коду вида:

$\underbrace{X}_{\text{Знак}} \cdot \underbrace{XXXXXXXX}_{\text{Мантисса}} \cdot \underbrace{XXX}_{\text{Показательная часть}}$

Эта операция снижает точность, но обеспечивает обработку данных, имеющих более широкий динамический диапазон:

от 1.1111111.1111 = -4161536_д до 0.1111111.1111 = 4161536_д

Действия производятся только с целыми числами. Значения весов и входов преобразуем к целому числу и округляем, например число -7.2341 может быть представлено числом -72341. В этом случае во всех остальных значениях входов и весов запятая должна быть передвинута на 4 знака вправо (для десятичного числа).

Модель нейрона описывается в виде основного тела программы, к которому подключается набор процедур, выполняющих основные действия. Соответствующие распечатки текстов находятся в пункте приложения.

Рассмотрим назначение подключаемых процедур:

```
FUNCTION mult_nrn (xx[7..0], yy[7..0], a[6..0], b[6..0], clk)
RETURNS (p[14..0]);
FUNCTION sum8 (a[7..0], b[7..0], clk, cin) RETURNS (sum[7..0], crr);
FUNCTION norm8 (a[16..1]) RETURNS (s[12..1]);
FUNCTION sum4 (a[3..0], b[3..0], clk, cin) RETURNS (sum[3..0]);
FUNCTION norm_sum (a[12..1], b[12..1], clk)
RETURNS (as[12..1], bs[12..1]);
FUNCTION dop_kod (xx[7..0], clk) RETURNS (yy[7..0]);
FUNCTION perep (clk, xxa[11..6], xxb[4..1]) RETURNS (yy[11..1]);
FUNCTION sigm (net[12..1]) RETURNS (out[16..1]); .
```

mult_nrn — представляет собой 8-разрядный умножитель: 7 разрядов данных, 1 разряд — знаковый.

sum8 — 8-разрядный сумматор.

norm8 — выполняет нормализацию 16-разрядного числа к 12-разрядному. При этом в 7 значащих разрядах старший разряд обязательно знаковый.

sum4 — 4-разрядный сумматор.

norm_sum — используется при необходимости сложения чисел с отличающейся показательной частью. Данная функция приводит младшее по модулю из этих чисел к показательной части старшего числа.

dop_kod — преобразует 8-разрядное число из прямого кода в обратный и из обратного в прямой. При этом, если входные данные находятся в дополнительном коде (что соответствует числу 0 в восьмом разряде), то перевод осуществляется в прямой код, и наоборот.

perep — функция, необходимая при переполнении разрядной сетки при выполнении действия функцией sum8. При этом получаем единичный раз-

ряд в переменной `srr`. Рассматриваемая функция осуществляет сдвиг на разряд вправо значащей части и прибавление единицы к показательной.

`sigm` — логистическая функция. Представляет функцию активации нейрона в виде сигмоиды $out = 2/(1+\exp(-0.07net)) - 1$. Сигмоида сжата в пределах $net = (-142; 142)$, $out = \text{int}([0,1]*10000)$. На вход подается нормализованное число к виду: 12 разряд — знаковый (+1, -0); 11—5 разряд — мантисса (если экспоненциальная часть не нулевая, то 11 разряд = V_{CC}); 4—1 разряд — экспоненциальная часть. На выходе имеем число вида: 16 разряд — знаковый (+1, -0); 15—1 разряд — мантисса.

Приведем функции, используемые в подпрограммах:

`FUNCTION mult16b (xx, yy, a, b) RETURNS (s, c);`

`FUNCTION dop_kod1 (xx[3..0], clk) RETURNS (yy[3..0]);` .

`mult16b` — задействует процедуру `MULT1B`. Является базовым одно-разрядным умножителем, на котором происходит построение умножителей большей размерности.

`dop_kod1` — аналог функции `dop_kod`, но для 4-разрядных чисел (используется для показательной части).

Ниже приводится текст основной программы `NEIRON`.

```
FUNCTION mult_nrn (xx[7..0], yy[7..0], a[6..0],
b[6..0], clk) RETURNS (p[14..0]);
```

```
FUNCTION sum8 (a[7..0], b[7..0], clk, cin) RETURNS
(sum[7..0], crr);
```

```
FUNCTION norm8 (a[16..1]) RETURNS (s[12..1]);
```

```
FUNCTION sum4 (a[3..0], b[3..0], clk, cin) RETURNS
(sum[3..0]);
```

```
FUNCTION norm_sum (a[12..1], b[12..1], clk)
RETURNS (as[12..1], bs[12..1]);
```

```
FUNCTION dop_kod (xx[7..0], clk) RETURNS
(yy[7..0]);
```

```
FUNCTION perep (clk, xxa[11..6], xxb[4..1])
RETURNS (yy[11..1]);
```

```
FUNCTION sigm (net[12..1]) RETURNS (out[16..1]);
```

```
SUBDESIGN neuron_
```

```
%Последние разряды в процедурах умножения во вход-
ных переменных и в ответе являются знаковыми%
```

```
%Реализуется трехвходовый однослойный нейрон%
(
clk:INPUT;

n[3..0]:INPUT;%Число входов нейрона 2...10%

w[16..1]:INPUT;%Весы%
ww[16..1]:INPUT;
www[16..1]:INPUT;

x_[16..1]:INPUT;%Входы%
xx[16..1]:INPUT;
xxx[16..1]:INPUT;

z[12..1]:OUTPUT;%Выход%
zz[16..1]:OUTPUT;

%Тестовые выходы%
o[12..1]:OUTPUT;%Умножение%
oo[12..1]:OUTPUT;
ooo[12..1]:OUTPUT;

jas[12..1]:OUTPUT;%Выравнивание перед сложением%
jbs[12..1]:OUTPUT;
jjas[12..1]:OUTPUT;
jjbs[12..1]:OUTPUT;

f[12..1]:OUTPUT;%Проверка суммы%
f_[11..5]:OUTPUT; %Проверка суммы (без переноса)%

ka[12..5]:OUTPUT;%Переменные для проверки работы
с доп. кодом и вычитания с ним%
kb[11..5]:OUTPUT;
kabs[12..5]:OUTPUT;
)

VARIABLE
rx[3..1]:norm8;%Переменные нормализации входов%
```

```

rw[3..1]:norm8;%Переменные нормализации весов%
rs[3..1]:norm8;%Переменные нормализации промежу-
точных результатов%

s:mult_nrn;%Переменные умножения%
ss:mult_nrn;
sss:mult_nrn;

e[3..1]:sum4;%Переменные сложения показателей при
умножении%
ee[3..1]:sum4;

q[2..1]:sum8;%Переменные сложения%
qq:sum8;

m[12..1]:NODE;%Вспомогательные переменные сумми-
рования%
mm[12..1]:NODE;

t[2..1]:norm_sum;%Переменные приведения к одному
показателю перед сложением%

d[4..1]:dor_kod;%Переменные перевода слагаемо-
го в дополнительный код%

p[2..1]:perer;%Переменная, переносящая переполне-
ние разрядной сетки в показатель%

sg:sgm;%Переменная сигмоиды%

BEGIN
%Нормализация входов%
rx[1].a[16..1]=x_[16..1];
rx[2].a[16..1]=xx[16..1];
rx[3].a[16..1]=xxx[16..1];
%Нормализация весов%
rw[1].a[16..1]=w[16..1];
rw[2].a[16..1]=ww[16..1];

```



```

    rw[3].a[16..1]=www[16..1];
%—————%

    %Умножение входов на веса%
    %Примечание: числа были приведены к виду
12.11.10.9.8.7.6.5.4.3.2.1%
    %знак числа|значение числа|показатель степени
двойки - сдвиг%
    %Числа перемножаются, показатели складываются%
    s.xx[7..0]=rx[1].s[12..5];
    s.yy[7..0]=rw[1].s[12..5];
    s.a[6..0]=GND;
    s.b[6..0]=GND;
    s.clk=clk;

    e[1].a[3..0]=rx[1].s[4..1];
    e[1].b[3..0]=rw[1].s[4..1];
    e[1].clk=clk;
    e[1].cin=GND;

    %Проводим нормализацию промежуточных результатов%
    rs[1].a[16]=s.p[14]; %Знак%
    rs[1].a[15]=GND;
    rs[1].a[14..1]=s.p[13..0];
    %Добавление показателя после нормализации%
    ee[1].a[3..0]=e[1].sum[3..0];
    ee[1].b[3..0]=rs[1].s[4..1];
    ee[1].clk=clk;
    ee[1].cin=GND;

%—————%
    ss.xx[7..0]=rx[2].s[12..5];
    ss.yy[7..0]=rw[2].s[12..5];
    ss.a[6..0]=GND;
    ss.b[6..0]=GND;
    ss.clk=clk;

    e[2].a[3..0]=rx[2].s[4..1];
    e[2].b[3..0]=rw[2].s[4..1];
    e[2].clk=clk;
    e[2].cin=GND;

```

```
%Проводим нормализацию промежуточных результатов%
rs[2].a[16]=ss.p[14]; %Знак%
rs[2].a[15]=GND;
rs[2].a[14..1]=ss.p[13..0];
    %Добавление показателя после нормализации%
    ee[2].a[3..0]=e[2].sum[3..0];
    ee[2].b[3..0]=rs[2].s[4..1];
    ee[2].clk=clk;
    ee[2].cin=GND;

%—————%

sss.xx[7..0]=rx[3].s[12..5];
sss.yy[7..0]=rw[3].s[12..5];
sss.a[6..0]=GND;
sss.b[6..0]=GND;
sss.clk=clk;
    e[3].a[3..0]=rx[3].s[4..1];
    e[3].b[3..0]=rw[3].s[4..1];
    e[3].clk=clk;
    e[3].cin=GND;

%Проводим нормализацию промежуточных результатов%
rs[3].a[16]=sss.p[14]; %Знак%
rs[3].a[15]=GND;
rs[3].a[14..1]=sss.p[13..0];
    %Добавление показателя после нормализации%
    ee[3].a[3..0]=e[3].sum[3..0];
    ee[3].b[3..0]=rs[3].s[4..1];
    ee[3].clk=clk;
    ee[3].cin=GND;

%—————%
%тест%
%демонстрируют правильность выполнения умножения
входов на веса и нормализации%
o[12..5]=rs[1].s[12..5]; o[4..1]=ee[1].sum[3..0];
oo[12..5]=rs[2].s[12..5];
oo[4..1]=ee[2].sum[3..0];
ooo[12..5]=rs[3].s[12..5];
ooo[4..1]=ee[3].sum[3..0];
%—————%
```

```
%Общий сумматор%
%во всех 3 случаях 14 разряд — знаковый%
%Dля корректного сложения необходимо равенство по-
казательных частей. Приводим показательные части двух
слагаемых к показательной части старшего из них %

%Производим выравнивание по показательным частям%
t[1].a[12..5]=rs[1].s[12..5];
t[1].a[4..1]=ee[1].sum[3..0];
t[1].b[12..5]=rs[2].s[12..5];
t[1].b[4..1]=ee[2].sum[3..0];
t[1].clk=clk;
%—————%
%тест%
%демонстрируют правильность выполнения выравнива-
ния значений перед сложением%
jas[12..1]=t[1].as[12..1];
jbs[12..1]=t[1].bs[12..1];
%—————%

%Произведем проверку знака (0 соответствует
(-) и доп. коду) и запрос к дополнительному коду%
%Перевод необходим лишь при различных знаках%
If ((t[1].as[12]==GND)&(t[1].bs[12]==VCC))==VCC
then
d[1].clk=clk;
d[1].xx[7]=GND;
d[1].xx[6..0]=t[1].as[11..5];
%—————%

%тест%
ka[12..5]=t[1].bs[12..5];
kb[11..5]=d[1].yy[6..0];
%—————%

%Суммируем%
```

```

q[1].clk=clk;
q[1].cin=GND;
q[1].a[7..0]=t[1].bs[12..5];
q[1].b[7]=GND; q[1].b[6..0]=d[1].yy[6..0];
%—————%

%тест%
kabs[12..5]=q[1].sum[7..0];
%—————%

%Проверка на получение отрицательного ре-
зультата в доп. коде%
%If (q[1].sum[7]==GND) then???% %То перевод
из доп. кода и постановка знака - (0)%
If (q[1].sum[7]==VCC) then
    d[2].clk=clk;
    d[2].xx[7]=GND;
d[2].xx[6..0]=q[1].sum[6..0];

    m[12]=GND; m[11..5]=d[2].yy[6..0];
Else
    m[12]=VCC;m[11..5]=q[1].sum[6..0];
End If;
m[4..1]=t[1].as[4..1];%Показательная часть при
вычитании не изменяется. Может быть t[1].bs[4..1]%

%Перевод необходим лишь при различных знаках%
ElsIf((t[1].as[12]==VCC)&(t[1].bs[12]==GND))==VCC
then
    d[1].clk=clk;
    d[1].xx[7]=GND;
d[1].xx[6..0]=t[1].bs[11..5];
%—————%

%тест%
ka[12..5]=t[1].as[12..5];
kb[11..5]=d[1].yy[6..0];
%—————%

```

```
%Суммируем%
q[1].clk=clk;
q[1].cin=GND;
q[1].a[7..0]=t[1].as[12..5];
q[1].b[7]=GND; q[1].b[6..0]=d[1].yy[6..0];
%—————%

%тест%
kabs[12..5]=q[1].sum[7..0];
%—————%

%Проверка на получение отрицательного ре-
зультата в доп. коде%
%If (q[1].sum[7]==GND) then??? %То перевод
из доп. кода и постановка знака - (0)%
If (q[1].sum[7]==VCC) then
    d[2].clk=clk;
    d[2].xx[7]=GND;
d[2].xx[6..0]=q[1].sum[6..0];

    m[12]=GND; m[11..5]=d[2].yy[6..0];
Else
    m[12]=VCC;m[11..5]=q[1].sum[6..0];
End If;
m[4..1]=t[1].as[4..1];%Показательная часть при
вычитании не изменяется. М.б. t[1].bs[4..1]%

Else %В случае двух положительных или двух
отрицательных слагаемых%
%Суммируем%
q[1].clk=clk;
q[1].cin=GND;
q[1].a[6..0]=t[1].bs[11..5];%Восьмой бит -
знаковый%
q[1].b[6..0]=t[1].as[11..5];
%—————%
```

```

%тест%
%демонстрируют правильность выполнения сложения
(без переносов)%
f_[11..5]=q[1].sum[6..0];
%—————%

%Проверка на переполнение разрядной сетки
уместна лишь при сложении чисел одного знака%
If (q[1].sum[7]==VCC) then
    p[1].clk=clk;
    p[1].xxa[11..6]=q[1].sum[6..1];
    p[1].xxb[4..1]=t[1].bs[4..1];%можно и
t[1].as[4..1]%
    m[12]=t[1].bs[12];%можно и t[1].as[12] -
их знак одинаков%
    m[11..1]=p[1].yy[11..1];
Else
    m[12]=t[1].bs[12];%можно и t[1].as[12] -
их знак одинаков%
    m[11..5]=q[1].sum[6..0];
    m[4..1]=t[1].bs[4..1];%можно и
t[1].as[4..1]%

    End If;
End If;
%—————%

%тест%
%демонстрируют правильность выполнения сложения%
f[12..1]=m[12..1];
%—————%

%Производим выравнивание по показательным ча-
стям%
t[2].a[12..5]=m[12..5]; t[2].a[4..1]=m[4..1];
t[2].b[12..5]=rs[3].s[12..5];
t[2].b[4..1]=ee[3].sum[3..0];
t[2].clk=clk;

```

```

%—————%
%тест%
%демонстрируют правильность выполнения выравни-
вания значений перед сложением%
jjas[12..1]=t[2].as[12..1];
jjbs[12..1]=t[2].bs[12..1];
%—————%

%Произведем проверку знака (0 соответствует
(-) и доп. коду) и запрос к дополнительному коду%
%Перевод необходим лишь при различных знаках%
If ((t[2].as[12]==GND)&(t[2].bs[12]==VCC))==VCC
then
    d[3].clk=clk;
    d[3].xx[7]=GND;
d[3].xx[6..0]=t[2].as[11..5];

    %Суммируем%
    q[2].clk=clk;
    q[2].cin=GND;
    q[2].a[7..0]=t[2].bs[12..5];
    q[2].b[7]=GND; q[2].b[6..0]=d[3].yy[6..0];

    %Проверка на получение отрицательного ре-
зультата в доп. коде%
    %If (q[2].sum[7]==GND) then??? %То перевод
из доп. кода и постановка знака - (0)%
    If (q[2].sum[7]==VCC) then
        d[4].clk=clk;
        d[4].xx[7]=GND;
d[4].xx[6..0]=q[2].sum[6..0];
        mm[12]=GND; mm[11..5]=d[4].yy[6..0];
    Else
        mm[12]=VCC; mm[11..5]=q[2].sum[6..0];
    End If;
    mm[4..1]=t[2].as[4..1];%Показательная часть
при вычитании не изменяется. Может быть
t[2].bs[4..1]%

```

```

        %Перевод необходим лишь при различных знаках%
        ElseIf
        ((t[2].as[12]==VCC) & (t[2].bs[12]==GND)) == VCC then
            d[3].clk=clk;
            d[3].xx[7]=GND;
        d[3].xx[6..0]=t[2].bs[11..5];

        %Суммируем%
        q[2].clk=clk;
        q[2].cin=GND;
        q[2].a[7..0]=t[2].as[12..5];
        q[2].b[7]=GND; q[2].b[6..0]=d[3].yy[6..0];
        %Проверка на получение отрицательного ре-
        зультата в доп. коде%
        %If (q[2].sum[7]==GND) then???% %То перевод
        из доп. кода и постановка знака - (0)%
        If (q[2].sum[7]==VCC) then
            d[4].clk=clk;
            d[4].xx[7]=GND;
        d[4].xx[6..0]=q[2].sum[6..0];

        mm[12]=GND; mm[11..5]=d[4].yy[6..0];
        Else
            mm[12]=VCC; mm[11..5]=q[2].sum[6..0];
        End If;
        mm[4..1]=t[2].as[4..1]; %Показательная часть
        при вычитании не изменяется. Может быть
        t[2].bs[4..1]%
        Else %В случае двух положительных или двух
        отрицательных слагаемых%
        %Суммируем%
        q[2].clk=clk;
        q[2].cin=GND;
        q[2].a[6..0]=t[2].bs[11..5]; %Восьмой бит -
        знаковый%
        q[2].b[6..0]=t[2].as[11..5];
    
```



```

        %Проверка на переполнение разрядной сетки
уместна лишь при сложении чисел одного знака%
        If (q[2].sum[7]==VCC) then
            p[2].clk=clk;
            p[2].xxa[11..6]=q[2].sum[6..1];
            p[2].xxb[4..1]=t[2].bs[4..1];%можно и
t[1].as[4..1]%

            mm[12]=t[2].bs[12];%можно и t[1].as[12] —
их знак одинаков%
            mm[11..1]=p[2].yy[11..1];
        Else
            mm[12]=t[2].bs[12];%можно и t[1].as[12] —
их знак одинаков%
            mm[11..5]=q[2].sum[6..0];
            mm[4..1]=t[2].bs[4..1];%можно и
t[1].as[4..1]%

        End If;
    End If;

    sg.net[12..1]=mm[12..1];
    zz[16..1]=sg.out[16..1];
    z[12..1]=mm[12..1];
END

```

7.7. Построение быстродействующих перемножителей

Реализация операции умножения аппаратными методами всегда являлась сложной задачей при разработке высокопроизводительных вычислителей. Аппаратная реализация алгоритма умножения в первую очередь предназначена для получения максимального быстродействия выполнения этой операции в устройстве. На ПЛИС можно разрабатывать умножители с быстродействием более 100 МГц и располагать более 100 модулей умножителей в одном кристалле.

Для операндов небольшой разрядности (4 и менее) наиболее результативна структура простого матричного суммирования (**Рис. 7.17**). Она реа-

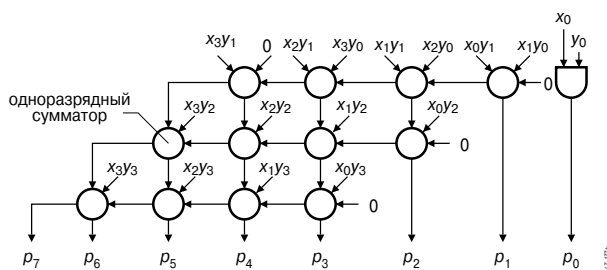


Рис. 7.17. Структура матричного умножителя с последовательным распространением переноса

лизует параллельный умножитель как массив одноразрядных сумматоров, соединенных локальными межсоединениями, при этом общее число сумматоров напрямую определяется разрядностью множимого и множителя.

Так, полный параллельный умножитель 4×4 требует для своей реализации 12 сумматоров. При дальнейшем наращивании разрядности матрица одноразрядных сумматоров значительно разрастается, одновременно увеличивается критический путь и реализация умножителя становится неэкономичной.

Одним из способов уменьшения аппаратных затрат служит использование алгоритма Бута. В соответствии с данным алгоритмом определенным образом анализируются парные разряды множителя и в зависимости от их комбинации над множимым выполняются некоторые преобразования (пример для умножителя 12×12 бит приведен в Табл. 7.3).

Таблица 7.3. Преобразование множимого в зависимости от состояния множителя

Пары разрядов множителя 0+1, 2+3, 4+5, 6+7, 8+9	Выполняемое действие
00	Прибавить 0
01	Прибавить множимое
10	Прибавить (2×множимое)
11	Прибавить (3×множимое)

Пара разрядов множителя 10+11 (знак)	Выполняемое действие
00	Прибавить 0
01	Прибавить множимое
10	Прибавить (2×множимое)
11	Прибавить множимое

Как видно из таблицы, выполняемые над множимым действия отличаются для двух старших разрядов, один из которых знаковый. Подобная коррекция позволяет осуществлять операцию умножения над числами в дополнительном коде.

Сформированные частичные произведения, представляющие собой результат преобразования множимого, необходимо суммировать с соответствующим весовым фактором. Результатом их суммирования по алгоритму простого матричного множителя и будет конечный продукт произведения двух операндов. Анализ различных методов свертки частичных произведений показал, что наиболее приемлемым для реализации на ПЛИС является алгоритм на основе иерархического дерева многоразрядных масштабирующих сумматоров. При этом достигается сокращение аппаратных затрат по сравнению с параллельным матричным множителем до двух раз.

Пример реализации данного решения для случая 12-разрядных операндов приведен на **Рис. 7.18**. В данной схеме выбор соответствующего преобразования множимого Y осуществляется мультиплексором на основании анализа парных разрядов множителя X в соответствии с **Табл. 7.3**. Входные значения мультиплексоров 0 и Y подаются непосредственно, значение $2Y$ формируется сдвигом множимого на один разряд, а $3Y$ формируется путем сложения Y и $2Y$.

Сумматоры ADD осуществляют суммирование частичных произведений, причем их разрядность для разных ступеней иерархии неодинакова. На каждом шаге суммирования, как видно, имеется несколько сквозных разрядов, непосредственно дополняющих выходные значения сумматоров. За счет них и за счет разрядов расширения знака осуществляется необходимое масштабирование результата. Размерность произведения P в конечном итоге равна сумме размерностей операндов.

За счет регулярности приведенной структуры быстроедействие можно резко повысить путем введения конвейеризации каждой ступени. При этом разделяются конвейерными регистрами как сумматоры, так и мультиплексоры. Выигрыш в быстроедействии благодаря данному решению оказывается особенно ощутимым для большой разрядности операндов.

Для иллюстрации метода умножения на константу рассмотрим, как выполняется умножение двух десятичных чисел «в столбик» на примере 85×37 :

$$\begin{array}{r}
 85 \\
 \times 7 \\
 \hline
 595
 \end{array}
 \rightarrow 7 \times 85 \rightarrow \begin{array}{r}
 85 \\
 \times 37 \\
 \hline
 595 \\
 + 2550 \\
 \hline
 3145
 \end{array}$$

Как видно из примера, используется каждая цифра множителя для поразрядного перемножения со всеми цифрами множимого и совсем не обязательно знать всю таблицу произведений операндов полной разрядности. Для перемножения двух переменных достаточно иметь таблицу произведений мно-

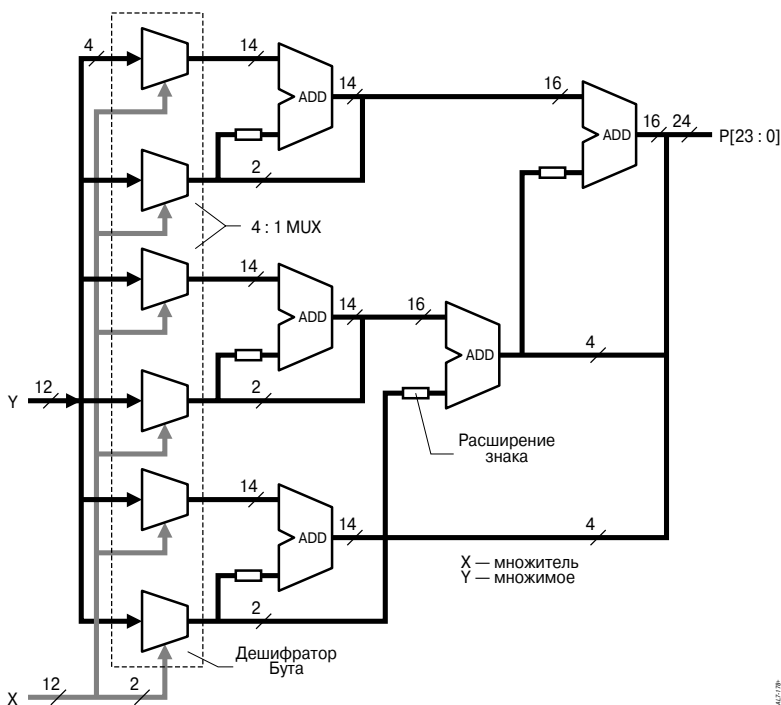


Рис. 7.18. Структурная схема 12-разрядного умножителя на основе алгоритма Бута

жимого, в данном случае константы, на весь ранг возможных цифр множителя и осуществить корректное суммирование полученных частичных произведений. Для случая 85×476 это иллюстрируется следующим образом:

$$\begin{array}{rcl}
 0 \times 85 & = & 000 \\
 1 \times 85 & = & 085 \\
 2 \times 85 & = & 170 \\
 3 \times 85 & = & 255 \\
 4 \times 85 & = & 340 \\
 5 \times 85 & = & 425 \\
 6 \times 85 & = & 510 \\
 7 \times 85 & = & 595 \\
 8 \times 85 & = & 680 \\
 9 \times 85 & = & 765
 \end{array}
 \begin{array}{r}
 \swarrow \\
 \longrightarrow \\
 \longrightarrow \\
 + \\
 \hline
 \end{array}
 \begin{array}{r}
 85 \\
 476 \\
 510 \\
 5950 \\
 34000 \\
 \hline
 40460
 \end{array}$$

Таблица произведений множимого записывается во фрагмент памяти на ПЛИС и называется таблицей перекодировок. ТП адресуются четырьмя разрядами, следовательно, необходимо представить операнды в шестнадцатеричном коде и массив ТП будет содержать таким образом массив произведений константы на ряд цифр 0, 1, ..., E, F:

$0 \times 55 = 000$	$8 \times 55 = 2A8$
$1 \times 55 = 055$	$9 \times 55 = 2FD$
$2 \times 55 = 0AA$	$A \times 55 = 352$
$3 \times 55 = 0FF$	$B \times 55 = 3A7$
$4 \times 55 = 154$	$C \times 55 = 3FC$
$5 \times 55 = 1A9$	$D \times 55 = 451$
$6 \times 55 = 1FE$	$E \times 55 = 4A6$
$7 \times 55 = 253$	$F \times 55 = 4FB$

Как видно из приведенного примера, для того чтобы осуществить операцию умножения двух двенадцатиразрядных двоичных чисел, одно из которых — константа, необходимо иметь ТП на 16 значений с разрядностью

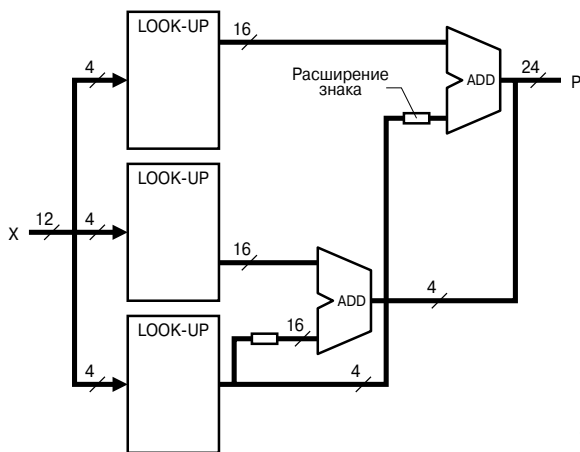


Рис. 7.19. 12-разрядный умножитель на константу

выходного значения 16 (для положительных чисел) и два 16-разрядных сумматора (Рис. 7.19).

Четыре младших разряда выходного произведения нижней (Рис. 7.19) ТП непосредственно в соответствии с алгоритмом функционирования формируют четыре младших разряда окончательного произведения. При подаче на сумматор указанного произведения необходимо произвести расширение знака в соответствии со значением старшего разряда. Например, для 12-разрядных операндов:

Младшее произведение ($\times 1$)	1111	1110	0110	1101	Сквозное суммирование
Старшее произведение ($\times 16$) +	0010	1101	0011		
Результат	0010	1011	1001	1101	

Реализация по данному принципу операции умножения отрицательных чисел, представленных в дополнительном коде, не требует каких-либо изменений в технической реализации и не задействует дополнительную специальную логику коррекции, изменится лишь прошивка просмотрной таблицы, адресуемой знаковым битом входного операнда.

Приложение 1. Система проектирования Quartus

Увеличение логической емкости ПЛИС и появление новой идеологии проектирования систем-на-кристалле (System on Chip) привели к тому, что ведущие производители ПЛИС вместе с выпуском на рынок собственно кристаллов с эквивалентной емкостью более 500 000 вентилях существенно обновили программное обеспечение, предоставив разработчику возможность использовать все преимущества новых БИС. В середине 1999 года на рынок вышел САПР 4-го поколения фирмы «Altera» — система Quartus.

Отличительные свойства данного пакета:

- Интеграция с программным обеспечением третьих фирм (Advanced Tools Integration). В рамках программы NATIVE LINK обеспечена совместимость с САПР ведущих производителей ПО. Поддерживаются стандарты EDIF, SDF, Vital 95, VHDL 1987 и 1993, Verilog HDL.

- Возможность коллективной работы над проектом (Workgroup Computing).

- Возможность анализа сигналов «внутри» ПЛИС с использованием функции Signal Tap.

- Итерационная компиляция проекта, позволяющая не изменять уже отлаженные участки проекта (nSTEP Compiler).

- Улучшенные средства синтеза в архитектуре APEX (CoreSyn).

- Многоплатформенность (Win NT, Sun, HP).

- Полная интеграция системы.

- Разнообразие средств описания проекта.

- Поддержка языков описания аппаратуры.

- Internet-поддержка.

- Поддержка мегафункций MegaCore.

В дополнение к уже привычным редакторам, используемым в пакете MAX+PLUS II, введен редактор блоков (Block Editor), позволяющий упростить графическое описание проекта, используя механизм параметризуемых блоков. Поуровневый планировщик (FloorPlan Editor) имеет возможность распределять ресурсы как внутри ЛБ, так и по мегаблокам. Новым средством, облегчающим работу над иерархическим проектом, является навигатор проекта (Project Navigator), позволяющий легко ориентироваться во всех файлах проекта. Улучшены возможности синтеза с заданными временными параметрами (Time Driven Compilation).

Возрастающее внимание уделяется функциональному и поведенческому моделированию с использованием языков описания аппаратуры, в том числе тестирование проектов из нескольких ПЛИС. Наличие встроенного логического анализатора Signal TAP позволяет проводить контроль сигналов внутри ПЛИС. Механизм подсказок сориентирован на использование Internet-технологий.

Для нормальной работы Quartus на базе PC совместимого компьютера требуется следующая аппаратная поддержка:

- процессор с тактовой частотой не менее 400 МГц;
- от 256 Мбайт до 1 Гбайт оперативной памяти;
- 4 Гбайт свободного пространства на жестком диске;
- Windows NT 4.0 и более поздние версии;
- привод CD-ROM;
- желательно монитор с диагональю не менее 17 дюймов;
- установленный Microsoft Internet Explorer 4.0 и более поздние версии.

К сожалению, объем книги не позволяет полностью и подробно изложить особенности работы с пакетом, но в планах автора написание отдельной книги по новым САПР разработки ПЛИС.

Приложение 2. Интерфейсы передачи данных и сопряжение устройств

В этом приложении рассмотрим вопросы выбора и использования стандартных протоколов и интерфейсов передачи данных, используемых в современной аппаратуре обработки сигналов.

С задачей разработки устройств обмена данными в той или иной мере сталкивался практически каждый разработчик. В случае выбора протокола для нового изделия всегда встает вопрос о компромиссе между сложностью аппаратных средств интерфейса («амуниции») и протоколом передачи данных («конституции»). Кроме того, присматриваясь к новомодному интерфейсу, не следует забывать, что очень часто в наших скромных задачах достаточно возможностей старых добрых RS-232 или RS-485, реализация которых к тому же исключительно дешева и многократно отработана.

Последние несколько лет, помимо прочих прелестей, принесли разработчику и целый ряд новых интерфейсов, позволяющих без помех передавать большие объемы информации на значительное расстояние. Современные ПЛИС имеют встроенную аппаратную реализацию таких интерфейсов, как GTL, LVDS, практически вся современная элементная база устройств обработки сигналов рассчитана на напряжение питания не выше 3.3 В, что вызывает необходимость разработки способов сопряжения указанных интерфейсов с традиционными. В то же время на русском языке практически отсутствует литература по этому вопросу. Ряд компаний выпустил руководства по применению ИС для реализации технических средств интерфейса, но, к сожалению, они не всегда доступны российскому читателю.

На **Рис. П2.1** представлены области использования различных интерфейсов передачи данных в координатах расстояние — скорость передачи. Как можно заметить, если требуется передача информации на расстояние больше нескольких десятков сантиметров, стандартные логические уров-

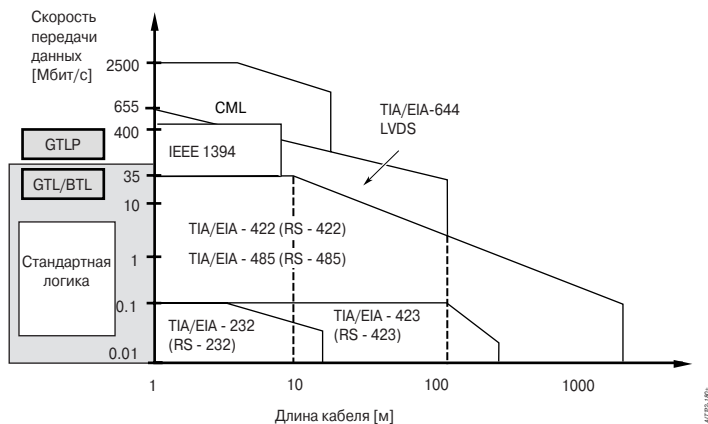


Рис. П2.1. Области применения интерфейсов передачи данных

ни перестают удовлетворять. На помощь приходят специализированные протоколы. Какой же из них выбрать для разрабатываемой системы? Какая элементная база позволит реализовать его аппаратно? Какие особенности применения данного интерфейса? Ответ на эти вопросы предстоит дать в этом разделе книги.

При выборе протокола передачи данных следует обращать внимание на несколько основных параметров. Это скорость передачи данных, расстояние между источником и приемником данных, заранее заданные уровни сигналов, совместимость, вид интерфейса (параллельный или последовательный). В Табл. П2.1 приведены краткая характеристика интерфейсов и данные об основных производителях ИС, поддерживающих данный интерфейс. Разумеется, последний столбец отражает лишь малую долю существующих решений; в тех случаях, когда производителей слишком много, в таблице скромно указано семейство ИС.

По способу организации передачи данных различают однопроводные (single-ended) и дифференциальные (differential) интерфейсы. На Рис. П2.2 приведена обобщенная схема однопроводного интерфейса.

Таблица П2.1. Интерфейсы передачи данных

Тип интерфейса	Скорость передачи данных по одной линии	Расстояние между источником и приемником данных	Стандарт	Производители элементной базы, поддерживающие интерфейс или семейство ИС
Последовательный	25/50 Мбит/с	1.5 м	IEEE1394 – 1995	Texas Instruments, Intel, и др.
	100-400 Мбит/с	4.5 м	IEEE1394 – 1995/p1394.a	Texas Instruments, Intel, и др.
	12 Мбит/с	5 м	USB 2.0	Texas Instruments, Intel, и др.
	35 Мбит/с	10 м (1200 м)	TIA/EIA 485 (RS-485) (ISO8482)	Texas Instruments, Analog Devices, Maxim, Sipex, и др.
	200 Мбит/с	0.5 м	LVDM (в разработке)	LVDM
	10 Мбит/с	10 м (1200 м)	TIA/EIA 422 (RS-422) (ITU-T V.11)	Texas Instruments, Analog Devices, Maxim, Sipex, и др.
	200/100 Мбит/с	0.5 м/10 м	TIA/EIA 644 (LVDS) (в разработке)	LVDS
	512 кбит / с	20 м	TIA/EIA 232 (RS-232) (ITU-T V.28)	Texas Instruments, Analog Devices, Maxim, Sipex, и др.
Параллельно-последовательный, последовательно-параллельный	455 Мбит/с	До 10 м	TIA/EIA 644 (LVDS)	Texas Instruments, и др.
	1,25 Гбит/с	До 10 м	IEEE P802.3z	Texas Instruments, и др.
	2,5 Гбит/с	До 10 м	IEEE P802.3z	Texas Instruments, и др.
Параллельный	35 Мбит/с	10 м (1200 м)	TIA/EIA 485 (RS-485) (ISO8482)	Texas Instruments, Analog Devices, Maxim, Sipex, и др.
	40/20 Мбит/с	12/25 м	SCSI	Многие производители
	40 Мбит/с	12 м	LVD- SCSI	Многие производители
	200/100 Мбит/с	0.5/10 м	LVDM (в разработке)	LVDM
	33/66 Мбит/с	0.2 м	Compact PCI	TI, PLX, разработчики прошивок для ПЛИС

Таблица П2.1 (окончание)

Тип интерфейса	Скорость передачи данных по одной линии	Расстояние между источником и приемником данных	Стандарт	Производители элементной базы, поддерживающие интерфейс или семейство ИС
Параллельный	33/66 Мбит/с	0.2 м	PCI	TI, PLX, разработчики прошивок для ПЛИС
	Тактовая частота до 4 МГц	10 м	IEEE Std 1284 – 1994	AC1284, LVC161284 LV161284
	Тактовая частота до 20 МГц	0.5 м	CMOS, JESD20, TTL, IEEE 1014 – 1987	AC, AHC, ABT, HC, HCT и др.
	Тактовая частота до 33 МГц	0.5 м	LVTTL (JED8-A), IEEE 1014 – 1987	LVTH, ALVT
	Тактовая частота до 40 МГц	0.5 м	VME64 Standart ANSI/VITA1 – 1991	ABTE
	Тактовая частота до 60 МГц	0.5 м	IEEE Std 1194.1 – 1991	BTL/FB+
	Тактовая частота до 60 МГц	0.5 м	JESD8-3	GTL/GTL+
	Тактовая частота до 100 МГц	0.5 м	JESD8-3	GTLP
	Тактовая частота до 200 МГц	0.1 м	EIA/JESD8-8, EIA/JESD8-9	SSTL

При однопроводной передаче данных используется одна сигнальная линия, и ее логический уровень определяется относительно земли. Для простых медленных интерфейсов допускается использование общей земли. В более совершенных интерфейсах каждый сигнальный провод имеет свою землю, и, как правило, они объединяются в витую пару. Преимуществом однопроводных систем является простота и дешевизна реализации. Поскольку каждая линия передачи данных требует только одного сигнального провода, они удобны для передачи параллельных данных на небольшое расстояние. Примером может служить привычный параллельный интерфейс принтера. Другой пример — последовательный интерфейс RS-232. Как видим, однопроводные интерфейсы часто применяются в тех случаях, когда решающим фактором является стоимость реализации. Основным не-

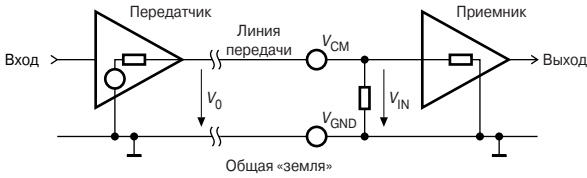


Рис. П2.2. Однопроводный интерфейс

достатком однопроводных систем является их низкая помехоустойчивость. Из-за наводок на общий провод возможен сдвиг уровней сигналов, приводящий к ошибкам. При передаче на расстояние порядка нескольких метров начинают оказывать влияние индуктивность и емкость проводов.

Преодолеть указанные недостатки смогли дифференциальные системы. На **Рис. П2.3** приведена принципиальная схема реализации дифференциальной передачи данных.

Для балансной дифференциальной передачи данных используется пара проводов. На приемном конце линии вычисляется разность между сигналами. Заметим, что такой способ передачи данных пригоден не только для цифровых, но и для аналоговых линий передачи. Ясно, что при дифференциальной передаче удастся в значительной мере подавить синфазную помеху. Отсюда следует основное достоинство дифференциальных протоколов передачи — высокая помехоустойчивость. Недаром один из самых распространенных протоколов в промышленных компьютерах RS-485 построен по дифференциальной схеме. Недостатком дифференциальных схем является их относительно высокая стоимость, а также сложности при выполнении парных согласованных каскадов передатчиков и приемников.

Рассмотрим физические параметры интерфейсов. В литературе принято следующее обозначение уровней:

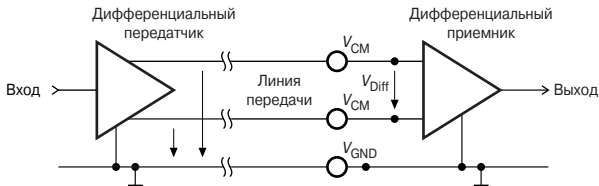


Рис. П2.3. Дифференциальный интерфейс.

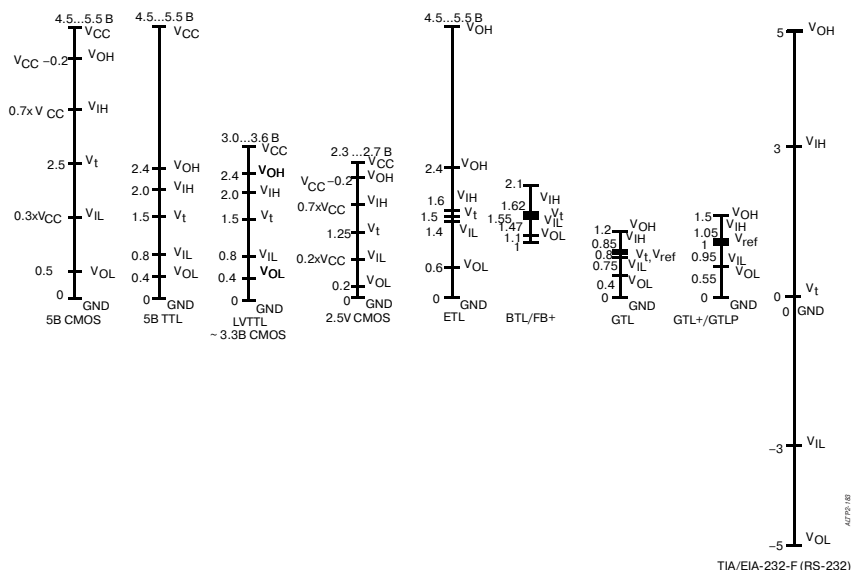


Рис. П2.4. Уровни сигналов в однопроводных интерфейсах

V_{IH} — входное напряжение ВЫСОКОГО уровня (логической единицы);

V_{IL} — входное напряжение НИЗКОГО уровня (логического нуля);

V_{OH} — выходное напряжение ВЫСОКОГО уровня (логической единицы);

V_{OL} — выходное напряжение НИЗКОГО уровня (логического нуля).

На **Рис. П2.4** приведены логические уровни для однопроводных интерфейсов, а на **Рис. П2.5** — для дифференциальных.

Далее мы рассмотрим несколько современных интерфейсов.

Интерфейс TIA/EIA-644 (LVDS — Low Voltage Differential Signaling) используется в скоростных системах передачи данных. Интерфейс LVDS использует дифференциальную передачу данных с довольно низкими уровнями сигналов. Разность сигналов составляет 300 мВ, линии нагружаются сопротивлением 100 Ом. Выходной ток передатчика составляет от 2.47 до 4.54 мА. Интерфейс TIA/EIA-644 обладает лучшими характеристиками потребления по сравнению с TIA/EIA-422 и может служить его заменой в новых разработках. Максимальная скорость передачи данных составляет 655 Мбит/с. Достоинство данного интерфейса — преемственность ИС приемопередатчиков по разводке с драйверами хорошо извест-



Интерфейсы LVDS поддерживают многие современные ПЛИС, такие как APEX фирмы «Altera», Virtex фирмы «Xilinx» и ряд других. Типичными представителями драйверов этого интерфейса являются ИС SN65LVDS31/32, SN65LVDS179 фирмы «Texas Instruments». По электрическим свойствам к интерфейсу LVDS примыкает интерфейс LVDM. Этот протокол поддерживают ИС SN65LVDM176, SN65LVDM050.

— 543 —

Как известно, классические ТТЛ- и КМОП-семейства ИС обеспечивают ток нагрузки до 24 мА при минимальном импедансе линии 50 Ом. С появлением БиКМОП-технологии стало возможным достигнуть выходного тока — 32/64 мА и работы на линию с импедансом 25 Ом. Для этих целей приспособлено семейство ИС SN74ABT25xxx. Данные микросхемы могут быть также использованы в системах так называемой «горячей замены» модулей, съемные модули могут подключаться или отключаться по ходу работы прибора.

При проектировании подключаемых модулей необходимо выполнить несколько требований, которые, во-первых, предупредят поломку модуля при подключении к работающей системе и, во-вторых, не приведут к сбоям в работе системы. Интерфейс между подключаемым и основным модулями состоит из шин питания, земли и сигнальных шин. Модель микросхемы, подключаемой к системе, показана на **Рис. П2.6**.

Защита входов и выходов микросхем осуществляется с использованием диодных ключей. Диод D_1 защищает микросхему от электростатических разрядов, ограничивая напряжение на входе микросхемы. Этого диода

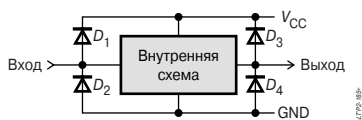


Рис. П2.6. Диоды на входе и выходе ИС

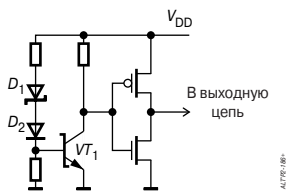


Рис. П2.7. Цепь, отключающая выход при пониженном напряжении питания в микросхемах БиКМОП

нет в микросхемах семейств ТТЛ, ЭСЛ и БиКМОП. Диод D_2 — паразитный диод, появление которого предопределено структурой входа микросхем. В цифровых микросхемах дополнительно к этому диоду ставят еще один с меньшим сопротивлением для того, чтобы ограничивать пики напряжения, меньшие уровня логического нуля. Этот диод также защищает вход микросхемы от электростатического разряда. Очень часто для защиты входов цифровых схем применяют диодную сборку BAV99. Недавно в сети обнаружился сайт, целиком посвященный этому объекту (www.bav99.al.ru).

Для защиты выходов используют диоды D_3 и D_4 . Диод D_3 используется в микросхемах КМОП для за-

щиты от электростатических разрядов. Диод D_4 защищает от напряжения на выходе, которое меньше уровня логического нуля.

При разработке подключаемых модулей лучше использовать микросхемы БиКМОП, поскольку они выгодно отличаются от прочих тем, что имеют схему (Рис. П2.7), которая держит выход микросхемы в состоянии высокого импеданса в момент включения микросхемы. Эта цепь

следит за напряжением питания и состоит из двух диодов D_1 и D_2 и транзистора VT_1 , на базу которого подается напряжение. При напряжении питания, которое меньше установленного (например для серии АВТ/ВСТ $V_{COFF} \sim 2.5$ В, для LVT $V_{COFF} \sim 1.8$ В), выход этой цепи переходит в состояние логической единицы. При этом он отключает сигнал на выходе микросхемы, независимо от сигнала, который присутствует на входе. Это свойство микросхем БиКМОП гарантирует, что поведение схемы предсказуемо даже при очень низком напряжении питания.

При горячем подключении модуля поведение системы будет предсказуемо, если соблюдаются по крайней мере два условия:

- на разъеме есть один или несколько контактов земли, удлиненных относительно других контактов;
- интерфейс состоит только из биполярных или БиКМОП-микросхем с тристабильными выходами или с выходами с открытым коллектором.

Проблема конфликтов на шине стоит особенно остро, когда встречаются выходные сигналы разных уровней — НИЗКОГО и ВЫСОКОГО. На Рис. П2.8 показан этот процесс. Ток, который возникает в результате конфликта, достигает 120 мА, и в этой борьбе выживает та микросхема, которая имеет на выходе НИЗКИЙ уровень. Микросхема с ВЫСОКИМ уровнем на выходе работает в режиме короткого замыкания и сгорает.

Для того, чтобы избежать такого конфликта, нужна дополнительная схема, которая во время включения питания держала бы выходы в состоянии высокого импеданса.

Основным элементом этой схемы может быть ИС TLC7705. Такие микросхемы используются для генерации сигнала RESET при включении

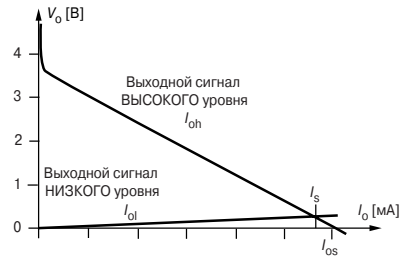


Рис. П2.8. Ток короткого замыкания при конфликтах на шине



1

... ..

[illegible]

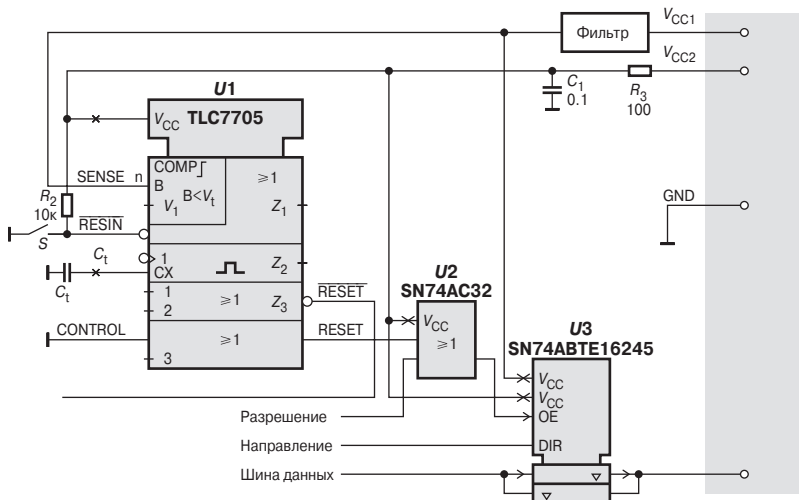


Рис. П2.10. Схема интерфейса с использованием микросхем серии ETL

Броски напряжения в цепях питания системы при подключении модуля появляются точно так же, как броски в сигнальных цепях. При этом величина заряжаемой емкости колеблется от десятков до сотен мкФ и зависит от емкости блокирующих конденсаторов на подключаемой плате. Один из путей к ограничению скачка напряжения — включение в цепь питания коммутатора, который медленно включается. На **Рис. П2.11** предложена схема, в которой роль коммутатора играет p -МОП-транзистор VT . RC -цепочка обеспечивает медленное изменение сигнала на базе транзистора. Диод D быстро разряжает конденсатор после того, как модуль был выключен.

Предполагается, что транзистор имеет малое сопротивление во включенном состоянии. При работе рассеиваемая мощность на транзисторе невелика из-за небольшого падения напряжения. При необходимости можно параллельно включать несколько транзисторов.

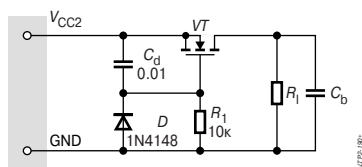


Рис. П2.11. Схема медленного включения модуля с использованием транзистора

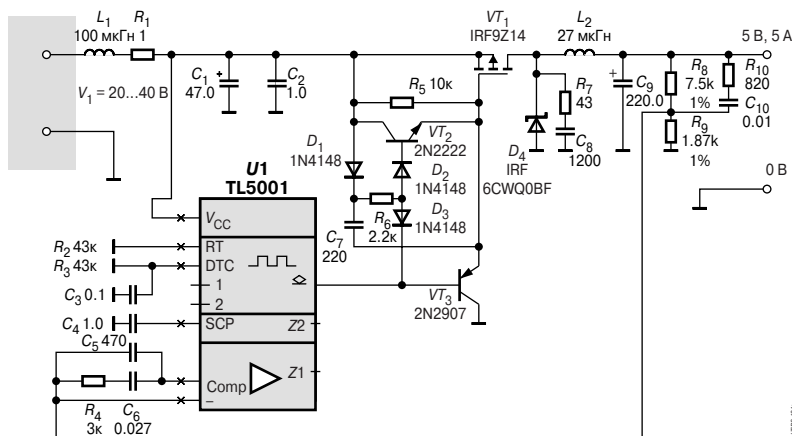


Рис. П2.12. Децентрализованный источник питания

В подключаемых модулях удобно использовать собственные источники питания. На **Рис. П2.12** приведена схема источника питания, который получает из системы от 10 до 40 В и преобразует их импульсным способом в 5 В. Схема не дает броска напряжения при включении.

Как известно, значительная масса ИС выпускается по КМОП-технологии, обеспечивающей все более высокое быстродействие, меньшее энергопотребление и большую интеграцию. Время задержки распространения сигнала на клапан у КМОП ИС стало сопоставимо с лучшими ИС ЭСЛ. С уменьшением проектных норм, естественно, снижается и напряжение питания ИС, рост быстродействия приводит к необходимости разработки «лазеек», недаром за последние несколько лет появилось несколько десятков новых протоколов передачи данных.

Одним из новых интерфейсов является интерфейс BTL (Backplane Transfer Logic). Фактически BTL является физической реализацией протокола Futurebus. Рассматривая в прошлом разделе проблему сопряжения с общей шиной, мы упомянули о том, что наилучшим решением для драйверов являются биполярные или БиКМОП-схемы. Однако в случае, когда в системе используются напряжения питания 5 и 3.3 В, бывает сложно обеспечить корректное сопряжение логических уровней из-за значительного размаха напряжений, особенно при достаточно больших токах нагрузки. Шина с понижен-

ным размахом напряжения (voltage swing) позволяет решить такие проблемы. В частности, интерфейс BTL реализует шину с открытым коллектором. Поддержку этого интерфейса выполняет семейство ИС SN74FBxxx. Интерфейс BTL позволяет работать с размахом напряжения 1.1 В. Уровень логического нуля равен 1 В, логическая единица — 2.1 В. Величина нагрузочного резистора равна характеристическому сопротивлению линии, что обеспечивает корректное сопряжение. Для автоматического определения логических уровней используется дифференциальный входной каскад, опорное напряжение которого (1.55 В) равно середине размаха напряжений между ВЫСОКИМ и НИЗКИМ уровнями. Основное назначение интерфейса — телекоммуникационное оборудование, где важным является возможность «горячей замены» модулей. К другим особенностям интерфейса следует отнести низкий уровень помех при переключении, максимальная длительность фронта — 2 нс.

К интерфейсу BTL по идеологии примыкает интерфейс GTL (Gunning Transceiver Logic). Его поддерживают ИС семейства SN74GTLxxx. Принципиальная схема интерфейса GTL приведена на **Рис. П2.13**.

Вследствие отсутствия диодов на открытом выходе уровень логического нуля равен 0.4 В, уровень логической единицы равен 1.2 В, таким образом, размах напряжения составляет всего лишь 0.8 В. Нагрузочная способность выхода составляет 40 мА, поэтому сопротивление нагрузочного резистора может быть $0.8 \text{ В} / 40 \text{ мА} = 20 \text{ Ом}$. Поскольку это сопротивление складывается из двух соединенных в параллель резисторов на плате-источнике и плате-приемнике, максимальная рассеиваемая мощность составляет 16 мВт на выход, что позволяет интегрировать драйвер GTL в БИС. В частности, ПЛИС современных семейств [2, 3] поддерживают интерфейс GTL без внешних дополнительных ИС.

Дальнейшим развитием GTL является интерфейс GTL+ (Gunning Transceiver Logic Plus), поддерживаемый семейством SN74GTLPxxx. Интерфейс GTL+ имеет два принципиальных отличия: он оптимизирован под распределенную нагрузку и поддерживает «горячее» подключение модулей. Кроме того, для данного интерфейса характерен уровень логи-

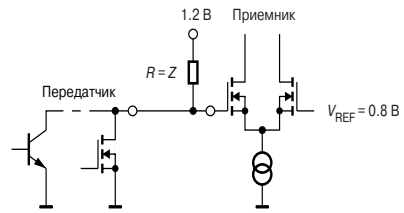


Рис. П2.13. Принципиальная схема интерфейса GTL

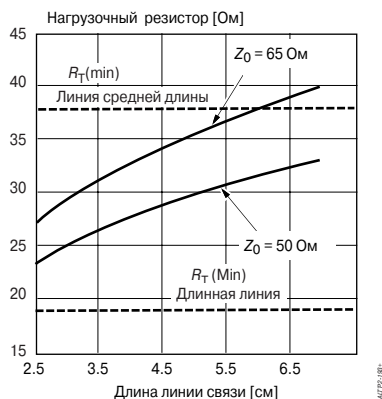


Рис. П2.14. Зависимость нагрузочного резистора от длины проводника для интерфейса GTL+

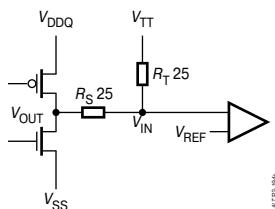


Рис. П2.15. Буферный каскад SSTL

ческого нуля 0.55 В, логической единицы — 1.5 В. Зависимость нагрузочного резистора от длины проводника для интерфейса GTL+ приведена на Рис. П2.14.

В настоящее время популярность интерфейсов GTL и GTL+ растет. Их изначальное предназначение — интерфейс для небольших плат, в частности, между процессором и модулями памяти. Интерфейс GTL+ используется в процессоре Intel Pentium Pro в качестве буфера адресной шины. Быстродействие этих интерфейсов порядка 80 МГц.

Во многих случаях такого быстродействия недостаточно. В частности, тактовая частота подсистем обмена между памятью и процессором в современных вычислителях достигает 200 МГц. Для работы на таких частотах используется интерфейс SSTL (Stub Series-Terminated Logic). Существуют два стандарта, определяющие разновидности интерфейса SSTL. Это стандарты SSTL_3, EIA/JESD8-8, SSTL_2, EIA/JESD8-8. Стандартами определяются схемотехника и номиналы нагрузочных резисторов (терминаторов). Возможны варианты с согласующими резисторами 50 или 25 Ом.

На Рис. П2.15 приведен типичный пример буферного каскада SSTL. Как можно заметить, выходной каскад питается от отдельного источника V_{DDQ} , напряжение которого не должно превышать напряжения питания V_{DD} . Это позволяет поднять напряжение V_{DD} до 3.6 В с целью увеличения быстродействия. Нагрузочная способность — 20 мА.

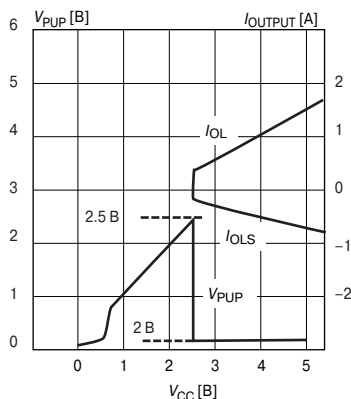
Одним из существенных новшеств, появившихся в цифровой технике, являются ИС, выполненные по комбинированной технологии, сочетающие до-

стоинства как биполярных, так и КМОП ИС. Логические ИС, выполненные по технологии БиКМОП, обладают следующими важными особенностями:

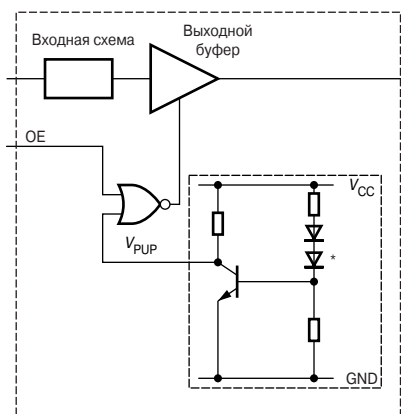
- работа в системах со смешанным напряжением питания;
- установка по включению питания в нужное состояние (сброс или третье состояние, power-up 3-state, power-up reset);
- отключение входа (input disable);
- удержание шины (bus hold).

Рассмотрим реализацию этих функций.

Короткие импульсные помехи, или «глитчи» (glitches), могут появляться в цифровых схемах из-за «горячей» замены платы в цифровых системах, а также в процессе включения или выключения питания. Эти помехи могут привести к сбоям в работе системы. Для уменьшения этого эффекта используют перевод выходов ИС в третье состояние по включению или выключению питания. На **Рис. П2.16** представлена схема, реализующая эту функцию.



а)



* Один диод в сериях LVT(16) и ALVT

б)

АЛТ93-98Р

Рис. 2.16. Перевод выводов ИС в третье состояние: а — ток при включении в третье состояние для АВТ в зависимости от напряжения питания (V_{CC}) и напряжения включения (V_{PUP}); б — схема перевода в Z-состояние по включению питания

Данная схема переводит выходы в третье состояние при достижении уровня напряжения питания 1.2 В для семейств LVT, LVT16 и ALVT и 2.1 В для семейств ABT и MULTIBYTE.

Удержание шины. Как известно, входы КМОП ИС не должны оставаться «висячими». Висячие, или плавающие, входы (floating inputs) могут привести к возрастанию тока, протекающего через входные каскады, что приводит к возрастанию потребляемой мощности, появлению высокочастотных колебаний и выходу микросхемы из строя. Как правило, неиспользуемые входы соединяют либо с землей, либо с напряжением питания через подстраивающий резистор (pull-up resistor или pull-down resistor). Этот способ хорош почти всегда, за исключением случаев, когда из-за экономии места невозможно установить лишний компонент. Кроме того, обилие подстраивающих резисторов приводит к повышению общего потребления системы.

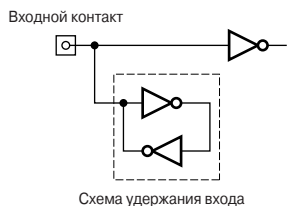


Рис. П2.17. Встроенная схема удержания шины

Семейство ABT16 и низковольтные БиКМОП ИС используют встроенную схему удержания шины (bus hold circuits) (Рис. П2.17 и П2.18), которая не требует внешних резисторов и позволяет сэкономить место на плате. Схема удержания шины хранит предыдущее состояние входа, если в момент включения он оказывается в плавающем состоянии.

Схема запрещения входа (input disable circuit) используется в семействах ABT и MULTIBYTE для обеспечения решения проблемы плавающих входов в отличие от схемы удержания шины. В этом случае входные цепи ведут себя как выход, находящийся в третьем состоянии. Подстраивающий резистор не требуется. Данная схема реализована в буферах и приемопередатчиках с линии и отсутствует у регистров и триггеров.

Свойство сброса по включению питания (power-up reset) используется в регистрах и триггерах. Как известно, для предварительного сброса классических цифровых ИС, как правило, применяются интегрирующие или дифференцирующие RC-цепочки, которые, конечно же, занимают место на плате, но не всегда обеспечивают стабильность функционирования. Гарантировано напряжение НИЗКОГО уровня после сброса $V_{RST} = 0.55 \text{ В}$.

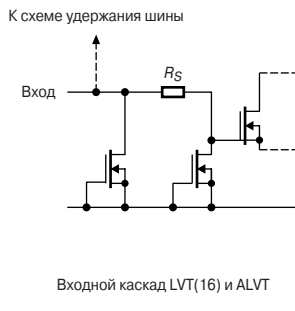
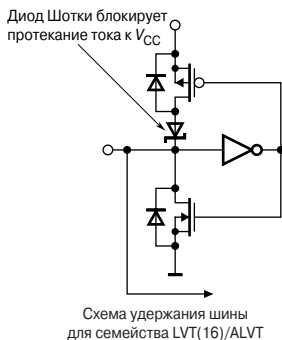


Рис. П2.18. Встроенные схемы удержания шины

Вытекающий ток I_{OFF} при снятом напряжении питания ограничен 100 мА. Данная особенность позволяет реализовать схемы, имеющие возможность перехода в спящий режим.

Для совместимости с традиционными сериями ИС новые семейства имеют входы и выходы, поддерживающие 5-вольтовые логические уровни. Далее мы рассмотрим эту особенность несколько подробнее. Заметим, что чисто биполярные схемы не боятся повышения уровней до 5 В, для КМОП-схем используются диоды для защиты.

На Рис. П2.19 представлены выходные каскады микросхем семейств LVT и ALVT. Диоды Шоттки защищают верхний по схеме PMOS-транзистор, если выходное напряжение превышает напряжение питания больше чем на 0.5 В. Выходной ток I_{EX} ограничен до 125 мА. В третьем состоянии обратносмещенный диод Шоттки предотвращает протекание тока на 3.3-вольтовое питание с выхода, на котором находится 5-вольтовый сигнал.

Однако, несмотря на развитие БиКМОП-технологии, основная линия развития проходит по пути со-

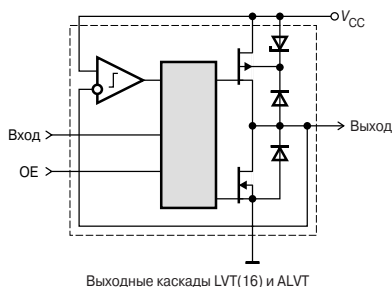


Рис. П2.19. Выходные каскады микросхем семейств LVT и ALVT

вершенствования КМОП-систем. Пожалуй, самым популярным семейством логических схем можно назвать семейства 74НС/НСТ/НСU (отечественные аналоги — серии КР1564). Название семейства происходит от High-speed CMOS (HCMOS) — быстродействующие КМОП-схемы. Отличительные особенности этих ИС — достаточно широкий диапазон напряжений питания, малая потребляемая мощность, высокая помехоустойчивость к входным шумам, достаточные быстродействие и нагрузочная способность, работа в широком диапазоне температур — по этим параметрам микросхемы семейств 74НС/НСТ/НСU сопоставимы с современными ТТЛ-схемами семейств Low-power Schottky TTL (LS TTL), при гораздо меньшем потреблении.

В отличие от последних логические ИС семейств 74НС/НСТ/НСU содержат в своем составе как основные функциональные узлы ТТЛ-серий (комбинационные и последовательностные схемы, арифметические устройства, буферы и т.п.), так и наиболее интересные устройства, характерные для семейства HE4000В: аналоговые ключи, мультивибраторы с большой постоянной времени, схемы фазовой автоподстройки частоты. В **Табл. П2.2** приведено сравнение семейства 74НС/НСТ/НСU с другими семействами логических ИС. Время задержки распространения на вентиль сопоставимо с задержкой ТТЛШ-семейств.

В **Табл. П2.3** приведено сравнение характеристик семейств 74НС и 74НСТ с микросхемами LSTTL. Как можно видеть, в современных разработках микросхемы семейств 74НС и 74НСТ являются реальной альтернативой для семейств, выполненных по LSTTL-технологии.

В отличие от первых КМОП ИС серий CD4000, 4000А, 4000Е, выполненных по 6-микронной технологии, микросхемы серий 74НС имеют технологические нормы 3 мкм. Микросхемы 74НС имеют поликремниевый затвор, расположенный над тонкой пленкой изолятора на основе оксида. Исток и сток выполняются в процессе диффузии с использованием ионной имплантации, причем поликремниевый затвор используется как маска в процессе имплантации. Значительное уменьшение паразитных емкостей затвор—исток и затвор—сток достигнуто благодаря уменьшению размеров элементов топологии кристалла.

Быстродействие ИС серии 74НС зависит от многих факторов. Не рекомендуется режим работы на емкостную нагрузку более 50 пФ. В этом случае возможно снижение быстродействия.

Как известно, динамика и быстродействие КМОП ИС определяются стоковой характеристикой транзистора. Поэтому во время работы при по-

Таблица П2.2. Сравнение семейства 74НС/НСТ/НСU/ с другими семействами логических ИС

Технология		КМОП		ТТЛ	ТТЛШ		Улучшенная ТТЛШ		
Семейство	74НС	4000		74	74LS	74S	74ALS	74AS	74F
		CD	HE						
Потребляемая мощность [мВт] Вентиль	Статика	0.0000025	0.001	10	2	19	1.2	8.5	5.5
	Динамика (100 кГц)	0.075	0.1	10	2	19	1.2	8.5	5.5
Потребляемая мощность [мВт] Счетчик	Статика	0.000005	0.001	300	100	500	60	—	190
	Динамика (100 кГц)	0.125	0.120	300	100	500	60	—	190
Задержка распространения [нс]	typ	8	94 40	10	9.5	3	4	1.5	3
	max	14	190 80	20	15	5	7	2.5	4
Энергия переключения [пДж]	(на частоте 100 кГц)	0.52	94 94	100	19	57	4.8	13	16.5
Максимальная тактовая частота [МГц]		55	4 12	25	33	100	60	160	125
Максимальный выходной ток [мА]		4	0.51 0.8	16	8	20	8	20	20
Коэффициент разветвления по выходу (вентиль LS)		10	1 2	40	20	50	20	50	50

ниженных напряжениях питания наблюдается увеличение времени задержки распространения сигнала на вентиль. На **Рис. П2.20** приведена зависимость средней задержки распространения сигнала от напряжения питания.

Температурная задержка распространения определяется только подвижностью носителей, в отличие от ТТЛ-схем, для которых от температуры зависят коэффициенты передачи тока транзисторов, прямое падение напряжения и внутренние сопротивления.

В общем случае задержка КМОП ИС увеличивается на 0.3% на каждый °С при температуре 25°С. При температуре от 25°С до 125°С справедлива зависимость:

Таблица П2.3 Сравнение характеристик семейств 74НС и 74НСТ с микросхемами LSTTL

Параметр	74НСxxx, 74НСТxxx			74LSxxx	
Максимальная рассеиваемая мощность в статическом режиме во всем диапазоне рабочих температур при максимальном напряжении питания [мВт]					
Для вентиля	0.027			6	
Для триггера	0.11			22	
Для четырехразрядного счетчика	0.44			175	
Для буфера линии	0.055			60	
Максимальная потребляемая мощность в динамическом режиме ($C_L = 50$ пФ) [мВт]					
На частоте 1 МГц	0.1	1	10	0.1	10
Для вентиля	0.25	2.25	22	6	22
Для триггера	0.35	2.5	24	22	27
Для четырехразрядного счетчика	0.70	3	27	175	200
Для буфера линии	0.30	2.5	24	60	90
Напряжение питания [В]	2...6 (НС) 4.5...5.5 (НСТ)			4.75...5.25	
Диапазон рабочих температур [°C]	-40...+85 -40...+125			0...+70	
Помехоустойчивость (V_{NMH}/V_{NML} ; при токе нагрузки для КМОП ИС $I_{OHCMOS} = 2$ мА; для ТТЛ ИС $I_{OLSTTL} = 4$ мА)	1.4/1.4 (НС) 2.9/0.7 (НСТ)			0.7/0.4	
Стабильность входного напряжения переключения	±60 мВ			±200 мВ	
Максимальный выходной ток					
Вентиль	-8			-0.4	
Шинный формирователь	-12			-2.6	
Задержка распространения $C_L = 15$ пФ [нс]					
Для вентиля t_{PHL}/t_{PLH}	8/8			8/11	
Для триггера t_{PHL}/t_{PLH}	14/14			15/22	
Максимальная тактовая частота триггера (типичное значение)	50			33	
Максимальный входной ток [мкА]					
I_{IL}	-1			-400...-800	
I_{IH}	1			40	
Ток утечки в третьем состоянии [мкА]	5			20	
Надежность (процент отказов на 1000 часов наработки)	0.0005			0.008	

$$t_p = t_p'(1.003)^{T_{amb} - 25}, \quad (8)$$

где:

t_p' — задержка распространения при 25°C,

T_{amb} — температура перехода °C.

При низких температурах (от -40 до +25°C):

$$t_p = t_p'(0.997)^{25 - T_{amb}}, \quad (9)$$

ИС семейства 74НС работают при напряжении питания от 2 до 6 В. Такой диапазон напряжений питания позволяет применять серию 74НС в приложениях, сопрягающих 5-вольтовые и 3-вольтовые системы. Следует, однако, помнить, что когда ИС 74НС применяются в линейном режиме (например при построении генераторов, одновибраторов и т.п.), следует использовать не менее чем 3-вольтовое питание для предотвращения самопроизвольного перехода в режим насыщения.

Микросхемы 74НСТ совместимы по выводам с LSTTL-схемами, однако они, помимо пониженного потребления, имеют и более широкий диапазон напряжений питания ($\pm 10\%$). Следует помнить, что для этих ИС максимальный ток между выходом и землей или питанием не более 50 мА, для микросхем шинных формирователей и некоторых регистров до 70 мА. Максимально допустимое напряжение питания не выше 7 В.

Благодаря возможности работы при пониженном напряжении питания достаточно просто реализуется резервное питание таких систем от литиевых батарей. На **Рис. П2.21** приведена схема включения батареи в систему с ИС 74НС.

Минимальное напряжение батареи — 2 В плюс одно напряжение на прямосмещенном диоде. Микросхемы сдвига уровня 74НС4049, 4050 предназначены для предотвращения сквозных токов через входные каскады микросхем в случае, если напряжения входных уровней сигналов превышают напряжение питания от батареи. Защита входных и выходных каскадов реализована с использованием нескольких схемных решений.

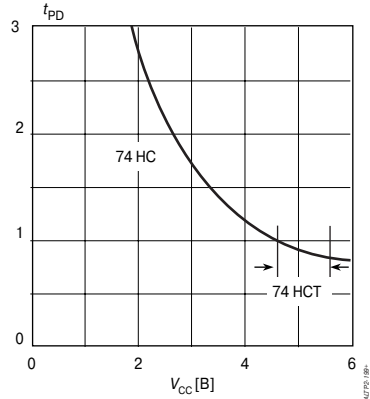


Рис. П2.20. Зависимость средней задержки распространения сигнала от напряжения питания

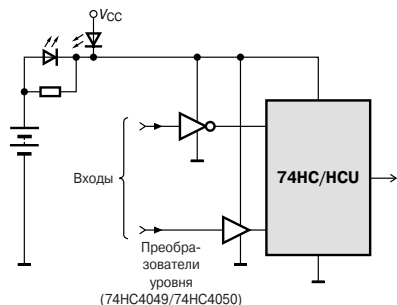


Рис. П2.21. Схема включения батареи в систему с ИС 74НС

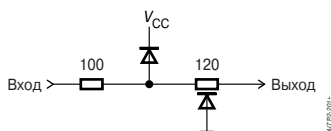


Рис. П2.22. Ограничительные резисторы, защищающие ИС от электростатического напряжения

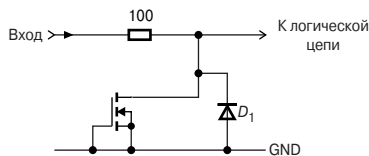


Рис. П2.23. Схема входных каскадов ИС 74НС4049 и 74НС4050

входных каскадов этих ИС приведена на Рис. П2.23. Такое построение схемы позволяет работать с источниками логических сигналов с уровнями выше напряжения питания.

Схемы входных каскадов серий 74НС и 74НСТ приведены на Рис. П2.24 и Рис. П2.25. Серия 74НСТ имеет дополнительные элементы сдвига уровня для совместимости с ТТЛ-схемами.

Как известно, КМОП-схемы из-за высокого входного сопротивления чувствительны к электростатическому напряжению. Для предотвращения статического пробоя в состав элемента входят ограничительные резисторы, как показано на Рис. П2.22.

В качестве элементов защиты использованы полисиликоновый резистор 100 Ом и диодные шунты, обеспечивающие защиту от входных токов. В некоторых схемах, таких как автогенераторы, диоды проводят ток в нормальном режиме работы, и в этом случае необходимо использовать внешние токоограничивающие резисторы. Максимальный положительный входной ток составляет 20 мА на вход. Для устройств со стандартной нагрузочной способностью общий входной ток на устройство не должен превышать 50 мА, для шинных формирователей — 70 мА.

В семействе имеются два достаточно интересных прибора — сдвигатели логических уровней (high-to-low level shifters) 74НС4049 и 74НС4050, которые имеют только защиту от электростатического разряда. Схема

Следует обратить внимание на привязку уровней входов. Как известно, для предотвращения перехода работы входных каскадов в линейный режим неиспользуемые входы ТТЛШ ИС подтягивают к источнику питания через резистор сопротивлением 1...2 кОм. Их входы нельзя подключать к земле или питанию напрямую.

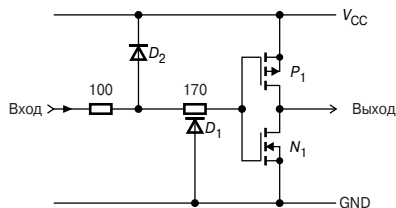


Рис. П2.24. Входные каскады 74НС

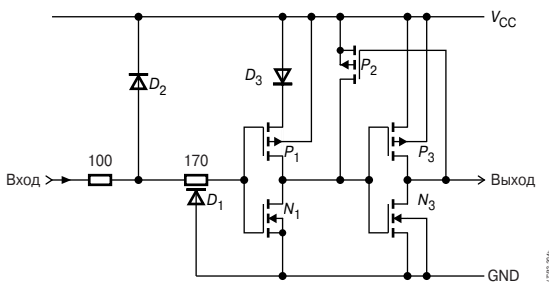


Рис. П2.25. Входные каскады 74НСТ

Для микросхем серий 74НС и 74НСТ возможно прямое подключение к источнику питания или к земле, либо через резистор сопротивлением от 1 кОм до 1 МОм.

Следует помнить, что двунаправленные выводы нельзя подключать к земле или питанию напрямую, только если они используются как входы, их можно подключать к земле через резистор сопротивлением 10 кОм.

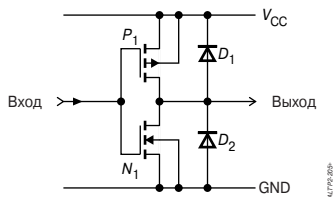


Рис. П2.26. Выходные каскады ИС 74НС

На Рис. П2.26 приведена типовая схема выходного каскада ИС серии 74НС. Защитные диоды ограничивают выходное напряжение в пределах 0.5 В – V_O до $V_{CC} + 0.5$ В.

На Рис. П2.27 приведена схема выходного каскада, реализующего третье состояние. Транзисторы P_3 и

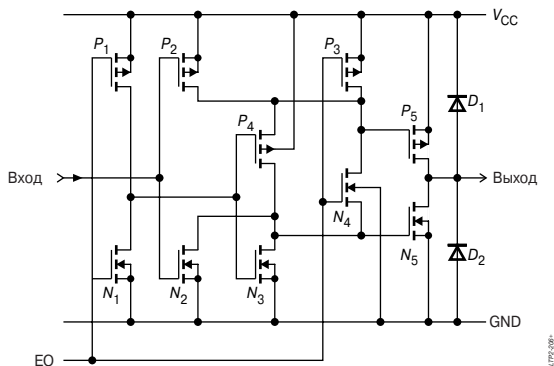


Рис. П2.27. Схема выходного каскада, регулирующего третье состояние

N_3 работают как сдвигатели уровня, управляемые сигналом ЕО, переводя выходные транзисторы P_5 и N_5 в закрытое состояние.

Третий тип выходных каскадов — выход с открытым стоком (open-drain). Данный тип выходного каскада является эквивалентом открытого коллектора для ТТЛ-схем.

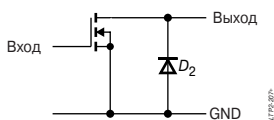


Рис. П2.28. Открытый выход

На Рис. П2.28 приведена схема каскада с открытым стоком. Безусловно, следует помнить о необходимости включения блокирующих емкостей по питанию. В руководящих документах фирм-производителей рекомендуется использовать керамический конденсатор емкостью 22 нФ на каждые два—пять корпусов ИС 74НС плюс танталовый электролитический конденсатор емкостью 1 мкФ на 10 корпусов.

Рассмотрим некоторые новые семейства цифровых интегральных схем, основываясь на стандартах JEDEC, и обратим внимание на некоторые особенности их применения.

Стандарт Interface Standard for Nominal 3 V/3.3 V Supply Digital Integrated Circuits (JESD8-B) определяет основные требования к интерфейсу с ИС с напряжением питания 3.3 В. В соответствии с этим стандартом установлены следующие предельные значения электрических условий эксплуатации (absolute maximum ratings), приведенные в Табл. П2.4.

Под предельно допустимыми значениями понимаются такие значения электрических величин (absolute maximum ratings), превышение которых может привести к выходу устройства из строя. Следует однозначно понимать, что приведенные значения параметров являются именно предельно допустимыми в течение короткого промежутка времени, работа устройства при таких значениях параметров не является нормой и недопустима в режиме нормальной эксплуатации. Следует помнить, что значения входных и выходных токов даны для единственного входа или выхода.

Иное дело рекомендуемые условия эксплуатации (recommended operating conditions). Эти значения параметров являются предпочтительными при штатной эксплуатации изделия, и не стоит их нарушать без крайней необходимости. В **Табл. П2.5** приводятся рекомендуемые условия эксплуатации для 3.3-вольтовых логических ИС.

К 3.3-вольтовым схемам относится ряд семейств ИС, выполненных как по ТТЛ-, так и по КМОП-технологии. В этой связи возникают вопросы, связанные с совместимостью устройств по уровням, потребляемой мощности и т.п. Постараемся привести основные понятия по совместимости различных семейств ИС.

Таблица П2.4. Предельные значения электрических условий эксплуатации

Параметр	Обозначение	Значение
Напряжение питания (supply voltage)	V_{DD}	–0.5 В...4.6 В
Входное постоянное напряжение (DC input voltage)	V_I	–0.5 В... $V_{DD} + 0.5$ В (4.6 В max)
Выходное постоянное напряжение (DC output voltage)	V_O	–0.5 В... $V_{DD} + 0.5$ В (4.6 В max)
Входной ток (DC input current)	I_I при $V_I < 0$ В или $V_I > V_{DD}$	± 20 мА
Выходной ток (DC output current)	I_O при $V_O < 0$ В или $V_O > V_{DD}$	± 20 мА

Таблица П2.5. Рекомендуемые условия эксплуатации для 3.3-вольтовых логических ИС

Диапазон напряжений питания	Нормальный диапазон	Расширенный диапазон
Номинал	3.3 В	3.0 В
Допустимые пределы изменения	3.0...3.6 В	2.7...3.6 В

Входные параметры микросхем семейств LVTTTL и LVCMOS приведены в **Табл. П2.6**. В **Табл. П2.7** приведены основные выходные параметры микросхем LVTTTL. Выходные параметры КМОП семейств LVCMOS приведены в **Табл. П2.8**.

Как известно, для увеличения быстродействия и степени интеграции постоянно снижаются проектные нормы, применяемые при изготовлении БИС. Если еще пару лет назад наиболее популярными и массовыми были 0.5- и 0.35-микронные технологии, то сейчас актуальны 0.25 и менее. Ясно, что уменьшение геометрических размеров логических элементов приводит к уменьшению допустимого напряжения питания. Однако большинство существующих стандартных интерфейсов и протоколов обмена до сих пор оперируют старыми добрыми ТТЛ-уровнями. В этой связи ак-

Таблица П2.6. Входные параметры микросхем семейств LVTTTL и LVCMOS

Обозначение	Параметр	Условия измерений	Значение	
			min	max
V_{IH}	Входное напряжение ВЫСОКОГО уровня (high-level input voltage)	$V_{OUT} \geq V_{OH}$ или $V_{OUT} \geq V_{OL}$	2 В	$V_{DD} + 0.3$ В
V_{IL}	Входное напряжение НИЗКОГО уровня (low-level input voltage)		-0.3 В	0.8 В
I_{IN}	Входной ток (input current)	$V_{IN} = 0$ В или $V_{IN} = V_{DD}$	—	± 5 мкА

Таблица П2.7. Основные выходные параметры микросхем LVTTTL

Обозначение	Параметр	Условия измерений	Значение	
			min	max
V_{OH}	Выходное напряжение ВЫСОКОГО уровня (high-level output voltage)	$V_{DD} = \min$, $I_{OH} = -2$ мА	2.4 В	—
V_{OL}	Выходное напряжение НИЗКОГО уровня (low-level output voltage)	$V_{DD} = \min$, $I_{OL} = 2$ мА	—	0.4 В

Таблица П2.8. Выходные параметры КМОП семейств LVCMOS

Обозначение	Параметр	Условия измерений	Значение	
			min	max
V_{OH}	Выходное напряжение ВЫСОКОГО уровня (high-level output voltage)	$V_{DD} = \min$, $I_{OH} = -100$ мкА	$V_{DD} - 0.2$ В	—
V_{OL}	Выходное напряжение НИЗКОГО уровня (low-level output voltage)	$V_{DD} = \min$, $I_{OL} = 100$ мкА	—	0.2 В

туальным становится вопрос о сопряжении низковольтных и классических логических ИС.

Определимся, что же означает совместимость ИС различных семейств. Приведенные в **Табл. П2.8** параметры определяют совместимость ИС как совместимые с LVTTTL (LVTTTL-compatible). Как можно заметить, выходные напряжения НИЗКОГО и ВЫСОКОГО уровня совершенно однозначно определяются классическими 5-вольтовыми схемами и могут быть использованы для подачи на вход 5-вольтовых ТТЛ совместимых ИС. Однако при обратном переходе от 5-вольтовых систем к 3-вольтовым следует помнить об опасности повреждения входных каскадов, не рассчитанных на напряжение питания 5 В. В этом случае полезны серии 74НС и 74АНС. В этой связи следует четко помнить, что при совместной работе 5-вольтовых и 3-вольтовых компонентов выходное напряжение V_{OH} 5-вольтовых ИС не должно превышать максимального входного напряжения 3-вольтовых ИС V_{IH} .

Говоря о совместимости с низковольтными КМОП ИС (LVCMOS compatibility), необходимо помнить, что в этом случае выходные параметры определены в **Табл. П2.8**. При этом при переключении из одного логического уровня в другой выходной сигнал практически проходит весь диапазон питания, занимая одно из крайних значений (ноль или напряжение питания). Это свойство в англоязычной литературе получило наименование rail-to-rail. Такое построение позволяет прийти к практически нулевому потреблению в статике (при отсутствии переключений).

Для того, чтобы обеспечить совместимость с низковольтными КМОП-устройствами по входу, необходимо соблюдать следующие условия:

1. Минимальное значение входного напряжения ВЫСОКОГО уровня V_{IH} должно быть не менее 2 В, но не выше напряжения питания V_{DD} . Для КМОП-устройств вообще принято, что уровень логической единицы не менее 0.7 напряжения питания.

2. Максимальное значение входного напряжения НИЗКОГО уровня V_{IL} равно 0.8 В.

Таким образом, когда разработчик в технической документации читает «Components are «LVTTTL-compatible» или «LVCMOS-compatible», это означает, что выходные характеристики такого прибора соответствуют приведенным в **Табл. П2.9** и **П2.10** соответственно. Но, как известно, нельзя объять необъятное, поэтому производители часто указывают на отклонения от стандарта, как правило, в сторону превышения стандартных значений (exceed the requirements). К таким превышениям относятся повышен-

Таблица П2.9. Основные параметры ССТ

Обозначение	Параметр	Условия измерения	Значение			Единица измерения
			min	typ	max	
V_{TT}/V_{REF}	Напряжение нагрузки/опорное напряжение	—	1.35	1.5	1.65	В
V_{IH}	Входное напряжение ВЫСОКОГО уровня (high-level input voltage)	—	$V_{REF} + 0.2$	—	—	В
V_{IL}	Входное напряжение НИЗКОГО уровня (low-level input voltage)	—	—	—	$V_{REF} - 0.2$	В
V_{OH}	Выходное напряжение ВЫСОКОГО уровня (high-level output voltage)	50-омная нагрузка	$V_{REF} + 0.4$	—	$V_{REF} + 0.6$	В
V_{OL}	Выходное напряжение НИЗКОГО уровня (low-level output voltage)	50-омная нагрузка	$V_{REF} - 0.6$	—	$V_{REF} - 0.4$	В
I_O	Выходной ток утечки	$V_{SS} < V_{IN} < V_{DD}$, Z-состояние	—	—	± 10	мкА
I_L	Входной ток утечки	$V_{SS} < V_{IN} < V_{DD}$	—	—	± 10	мкА

ная нагрузочная способность, способность функционировать при высоком напряжении на входе и т.п.

Говоря о новых семействах логических ИС, нельзя не остановиться на вопросе новых интерфейсных серий и стандартов. Так, в 1993 году был одобрен стандарт JESD8-4, определяющий параметры интерфейса СТТ (Center Tap Terminated). Этот стандарт определяет спецификацию входных и выходных характеристик для интерфейсных ИС и представляет собой надмножество LVTTTL и LVCMOS. Таким образом, приемники СТТ совместимы со стандартными 3-вольтовыми ИС. В **Табл. П2.9** приводятся основные параметры ССТ. На **Рис. П2.29** приведена типичная схема использования СТТ-интерфейса. На **Рис. П2.30** приведена схема реализации СТТ-интерфейса.

Совершенствование технологии приводит к уменьшению проектных норм, а следовательно, и рабочих напряжений ИС. Уже сейчас новые семейства ПЛИС, такие как APEX фирмы «Altera» и VIRTEX фирмы «Xilinx» [2, 3], работают от источников питания 2.5 В и 1.8 В. В этой связи становятся актуальными вопросы сопряжения с такими устройствами.

Таблица П2.10. Предельно допустимые параметры

Параметр	Обозначение	Значение
Напряжение питания (supply voltage)	V_{DD}	-0.5 В...3.6 В
Входное постоянное напряжение (DC input voltage)	V_{IN}	-0.5 В...3.6 В
Выходное постоянное напряжение (DC output voltage)	V_{OUT}	-0.5 В... V_{DD} + 0.5 В
Входной ток (DC input current)	I_I при $V_I < 0$ В или $V_I > V_{DD}$	± 20 мА
Выходной ток (DC output current)	I_O при $V_O < 0$ В или $V_O > V_{DD}$	± 20 мА

Таблица П2.11. Параметры 2.5-вольтовых ИС

Обозначение	Параметр	Условия измерения		Значение		Единица измерения
				min	max	
V_{DD}	Напряжение питания	—		2.3	2.7	В
V_{IH}	Входное напряжение ВЫСОКОГО уровня (high-level input voltage)	$V_{OUT} > V_{OH}$		1.7	$V_{DD} + 0.3$	В
V_{IL}	Входное напряжение НИЗКОГО уровня (low-level input voltage)	$V_{OUT} < V_{OL}$		-0.3	0.4	В
V_{OH}	Выходное напряжение ВЫСОКОГО уровня (high-level output voltage)	$V_{DD} = \text{MIN},$ $V_I = V_{IH}$ или V_{IL}	$I_{OH} =$ -100 мкА	1.7	$V_{DD} + 0.3$	В
V_{OL}	Выходное напряжение НИЗКОГО уровня (low-level output voltage)	$V_{DD} = \text{MIN},$ $V_I = V_{IH}$ или V_{IL}	$I_{OH} =$ -100 мкА	-0.3	0.4	В
I_I	Входной ток	$V_{DD} = \text{MIN}, V_I = V_{IH}$ или V_{IL}		—	±5	мкА

Примечание: MIN — минимальное значение.

Стандарт JEDEC JESD8-5 определяет логические уровни и параметры ИС, работающих в диапазоне питания 2.5 В. Предельно допустимые параметры, определяемые этим стандартом, приводятся в Табл. П2.10. Параметры 2.5-вольтовых ИС приводятся в Табл. П2.11.

Последние разработки ИС выполняются по 0.18-микронной технологии, при использовании которой напряжение питания составляет всего лишь 1.8 В. Стандарт JESD8-7 определяет как предельно допустимые, так и рабочие параметры 1.8-вольтовых ИС. В Табл. П2.12 приведены предельно допустимые параметры для 1.8-вольтовых ИС. Рабочие характеристики 1.8-вольтовых ИС приведены в Табл. П2.13.



Таблица П2.12. Предельно допустимые параметры для 1.8-вольтовых ИС

Параметр	Обозначение	Значение
Напряжение питания (supply voltage)	V_{DD}	-0.5 В...2.5 В
Входное постоянное напряжение (DC input voltage)	V_{IN}	-0.5 В... $V_{DD} + 0.5$ В
Выходное постоянное напряжение (DC output voltage)	V_{OUT}	-0.5 В... $V_{DD} + 0.5$ В
Входной ток (DC input current)	I_I при $V_I < 0$ В или $V_I > V_{DD}$	± 20 мА
Выходной ток (DC output current)	I_O при $V_O < 0$ В или $V_O > V_{DD}$	± 20 мА

Таблица П2.13. Рабочие характеристики 1.8-вольтовых ИС

Обозначение	Параметр	Условия измерения	Значение		Единица измерения
			min	max	
V_{DD}	Напряжение питания		1.2	1.95	В
V_{IH}	Входное напряжение ВЫСОКОГО уровня (high-level input voltage)	$V_{OUT} > V_{OH}$	$0.7 V_{DD}$	$V_{DD} + 0.3$	В
V_{IL}	Входное напряжение НИЗКОГО уровня (low-level input voltage)	$V_{OUT} < V_{OL}$	-0.3	$0.3 V_{DD}$	В
V_{OH}	Выходное напряжение ВЫСОКОГО уровня (high-level output voltage)	$I_{OH} = -100$ мкА	$V_{DD} - 0.2$	—	В
V_{OL}	Выходное напряжение НИЗКОГО уровня (low-level output voltage)	$I_{OL} = -100$ мкА	—	0.2	В

Приложение 3. **Практические рекомендации по разработке печатных плат**

Современные электронные узлы значительно отличаются от устройств разработки конца 80-х — начала 90-х годов. Во-первых, новые технологии поверхностного монтажа привели к уменьшению габаритов компонентов в 3...6 раз. Во-вторых, появились новые корпуса интегральных схем с малым шагом между выводами (0.5...0.65 мм), корпуса с шариковыми выводами (BGA), новые малогабаритные дискретные компоненты и соединители. В-третьих, повысилась точность изготовления печатных плат, повысились возможности для разводки сложных устройств в малых габаритах. Появление новой элементной базы позволяет говорить о возможности воплощения сложных систем на одной плате и даже на одном кристалле. Это означает, что на одной и той же типичной плате устройства обработки сигналов в малых габаритах размещаются высокочувствительный аналоговый тракт, аналого-цифровой преобразователь, высокоскоростная схема цифровой обработки на процессоре и (или) программируемых логических интегральных схемах, буферные элементы и драйверы линий связи, элементы стабилизаторов напряжения питания и преобразователей уровня и другие узлы. Естественно, что это накладывает свой отпечаток на методологию разработки платы.

При проектировании плат с использованием средств САПР необходимо всегда помнить, что сколь совершенными ни были алгоритмы автоматической трассировки, они никогда не заменят работу конструктора. В лучшем случае в автоматическом режиме возможна трассировка малочувствительных, медленных (до 3...5 МГц) цифровых цепей.

Особое внимание следует уделять проблеме заземления. Земляная шина определяется как эквипотенциальная поверхность, потенциал которой служит для схемы уровнем отсчета напряжений.

При проектировании земляных цепей преследуются две цели. Во-первых, следует помнить, что заземление минимизирует напряжение шумов, возникающее при прохождении токов от нескольких схем через общее сопротивление земли. Во-вторых, необходимо исключить образование контуров заземления, чувствительных к электромагнитным полям и разностям потенциалов.

Таким образом, заземление представляет собой обладающую низким импедансом цепь возврата тока. Отсюда ясно, что протекание любого тока в системе заземления приводит к появлению разности потенциалов. Ясно, что эта разность потенциалов должна быть минимальной. Отсюда следует, что при проектировании топологии земли следует обеспечить импеданс заземления на как можно более низком уровне и контролировать токи, протекающие между источниками и нагрузками.

Поэтому необходимо использовать несколько цепей заземления, соединенных в одной точке. Причем традиционного деления на аналоговое и цифровое заземление может оказаться недостаточно. На **Рис. ПЗ.1** представлена схема заземления двух аналоговых земель, причем одна служит землей для слабого входного сигнала, а другая — для мощного выходного. Следует избегать заземления так называемой гирляндой, надо использовать одноточечные схемы заземления, когда различные земли соединяются в точке ввода у разъема питания.

В первую очередь следует помнить, что если разрабатывается плата, содержащая как аналоговые, так и цифровые узлы и работающая на достаточно высокой тактовой частоте (>1 МГц), то не следует экономить и разрабатывать двухслойную плату. В этом случае необходимо использовать многослойную плату, в которой внутренние слои представляют собой сплошные плоскости земли и питания.

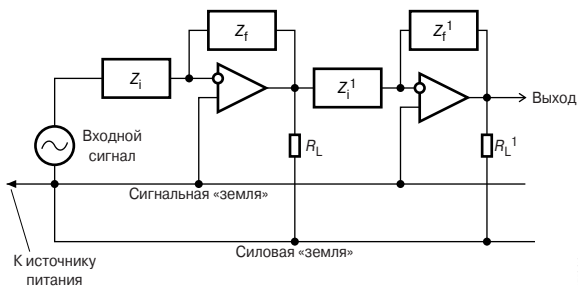


Рис. ПЗ.1. Схема заземления для двух аналоговых земель

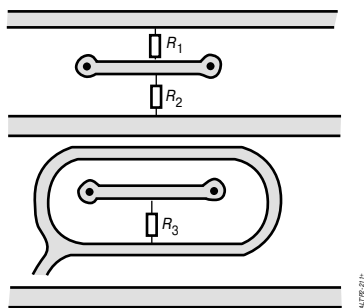


Рис. ПЗ.2. Использование защитного кольца

решающую способность фотоплоттера и другого технологического оборудования, но и помнить о возможных токах утечки через плату. Особенно это критично к входным высокоомным цепям усилителей, в этом случае неплохой мерой является защитное кольцо (Рис. ПЗ.2).

Если передается скоростной сигнал на достаточно большое расстояние, то сделать это можно только по согласованной линии на низкоомную нагрузку.

Несколько слов следует сказать о фильтрующих емкостях. Очень часто начинающий конструктор бездумно устанавливает их где придется, руководствуясь только нормой установки блокирующих емкостей на число тех или иных микросхем. Следует помнить, что для того, чтобы фильтрующие емкости эффективно работали, длина цепи от вывода микросхемы до емкости была минимальной. Использование планарных компонентов практически полностью позволяет решить эту проблему (Рис. ПЗ.3).

Не следует разносить в пространстве сигнальную и возвратные цепи одного сигнала (Рис. ПЗ.4). Это замечание касается также дифференциальных входных цепей, которые должны иметь одинаковую длину.

Следует стремиться выполнить разводку чувствительных аналоговых цепей в одном слое (со стороны установки компонентов) и избегать пересечений проводников, так как нарушение целостности заземляющего слоя вызывает увеличение его индуктивности и, следовательно, возрастает степень взаимного влияния возвратных токов (Рис. ПЗ.5).

Нелишне напомнить о всем известном скин-эффекте и зависимости импеданса от частоты, поэтому область металлизации одного и того же размера будет иметь различный импеданс на низких и высоких частотах. Рекомендуется все свободное пространство платы заполнить сплошной областью металлизации, соединенной с общей шиной, чтобы избежать наводок.

Выбирая шаг сетки трассировки, надо учитывать не только раз-

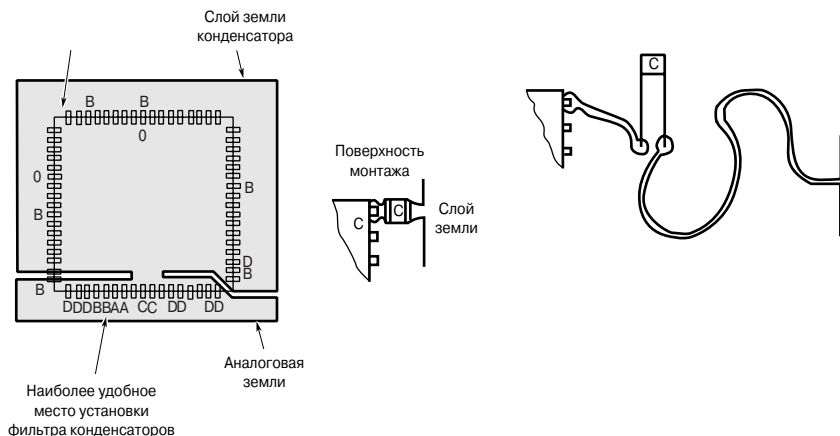


Рис. ПЗ.3. Установка фильтрующих емкостей

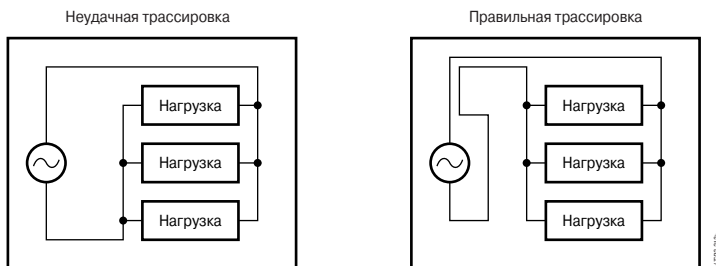


Рис. ПЗ.4. Разводка сигнальной цепи и цепи возврата

При проектировании платы, содержащей скоростные цифровые микросхемы (ПЛИС, сигнальные процессоры и т.п.), следует подумать о теплоотводе. Полезны бывают дополнительные слои для отвода излишнего тепла от микросхем.

Полезно после завершения разработки топологии платы максимально расширить все силовые цепи, земли, ответственные сигнальные цепи. При разработке ответственных устройств не следует забывать о возможностях современных средств по анализу целостности сигналов, тепловых режимов, прочности и т.д.

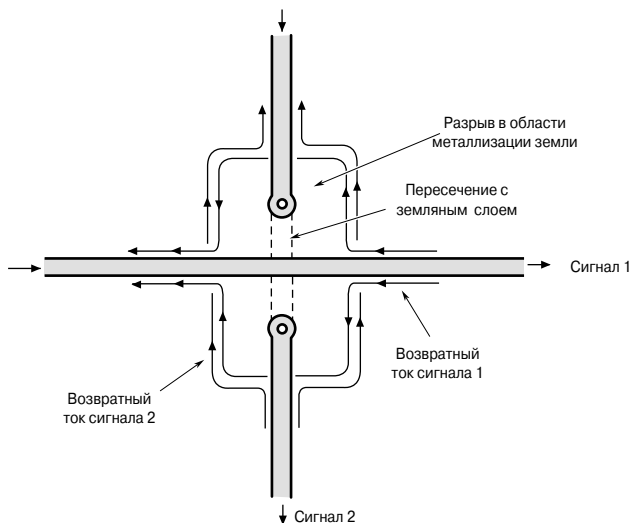


Рис. ПЗ.5. Взаимодействие возвратных токов на вырезе в заземляющей плоскости

Литература

1. **Стещенко В.Б.** Школа разработки аппаратуры цифровой обработки сигналов на ПЛИС. // Chip News, 1999, № 8—10, 2000, № 1—3.
2. **Армстронг Дж.Р.** Моделирование цифровых систем на языке VHDL. Пер. с англ. // М.: Мир, 1992, с.175.
3. **Вицин Н.** Современные тенденции развития систем автоматизированного проектирования в области электроники. // Chip News, № 1, 1997.
4. **Губанов Д.А., Стещенко В.Б., Храпов В.Ю., Шипулин С.Н.** Перспективы реализации алгоритмов цифровой фильтрации на основе ПЛИС фирмы «Altera». // Chip News, № 9—10, 1997.
5. **Губанов Д.А., Стещенко В.Б.** Методология реализации алгоритмов цифровой фильтрации на основе программируемых логических интегральных схем. // Сборник докладов 1-й Международной конференции «Цифровая обработка сигналов и ее применения», 30.06—3.07.1998, Москва, МЦНТИ, том 4, с. 9—19.

6. **Стещенко В.Б.** Особенности проектирования аппаратуры цифровой обработки сигналов на ПЛИС с использованием языков описания аппаратуры // Сборник докладов 2-й Международной конференции «Цифровая обработка сигналов и ее применения», 21.09—24.09.1999, Москва, МЦНТИ, том 2, с. 307—314.
7. **Щербаков М.А., Стещенко В.Б., Губанов Д.А.** Цифровая полиномиальная фильтрация: алгоритмы и реализация на ПЛИС // Инженерная микроэлектроника, № 1 (3), март 1999, с. 12—17.
8. **Губанов Д.А., Стещенко В.Б., Шипулин С.Н.** Современные алгоритмы ЦОС: перспективы реализации. // Электроника: наука, технология, бизнес, № 1, 1999, с. 54—57.
9. **Шипулин С.Н., Губанов Д.А., Стещенко В.Б., Храпов В.Ю.** Тенденции развития ПЛИС и их применение для цифровой обработки сигналов // Электронные компоненты, № 5, 1999, с. 42—45.
10. IEEE Standart 1149.1a –1993.
11. ByteBlasterMV Parallel Port Download Cable, Data Sheet, «Altera corporation», ver.1, April 1998.
12. AN74. High Speed Boards Design, «Altera corporation».
13. **Стещенко В.Б.** ACCEL EDA: технология проектирования печатных плат. // М.: Нолидж, 2000.
14. **Стещенко В.Б.** ПЛИС фирмы «Altera»: проектирование устройств обработки сигналов // М.: «Додэка», 2000.