Содержание

Введение 1

Глава 1. Уровни и процесс проектирования СБИС 7

Уровни представления и проектирования СБИС 7
Традиционный подход к поуровневому проектированию СБИС 10
Блочно-ориентированное проектирование 11
Проектирование Систем-на-кристалле 12
Синтез реализации проектируемых схем на кристалле СБИС 15
Элементная база технической реализации СБИС 15
Логический синтез реализации устройства 21
Модели в проектировании систем на СБИС 27

Глава 2. Операторы и данные языка VHDL 31

Базовые элементы языка VHDL 31
Лексические элементы 31
Описания констант, переменных, сигналов 32
Типы данных 33
Скалярные типы данных 34
Числовые типы 34
Тип *Integer* 35
Тип *Real* 36
Физические типы данных 36
Операции над физическими типами 37
Описание времени 38
Перечислимые типы данных 38
Описание типа с использованием диапазона 38
Описание типа с использованием списка значений 38
Атрибуты скалярных типов 41
Преобразование скалярных типов 44
Подтипы 44
Встроенные подтипы 44
Составные типы данных 45
Массивы 45
Определение начальных значений 46

Атрибуты данных типа "массив" 48
Массивы неограниченной длины 49
Операции над массивами 50
Фрагменты массивов 51
Преобразование типов 51
Записи 52
Указательные типы данных (access) 53
Назначение указательных типов 53
Описание данных указательного типа 54
Указатели на массивы 56
Организация связанных структур данных 57
Частичное предописание 57
Абстрактные типы данных 59
Механизм шаблонов пакетов 59
Ограничения на практическое применение указательных типов 59
Операторы 60
Оператор присваивания 60
Операции над данными в выражениях 60
Управляющие операторы 64
Операторы условного перехода 65
Циклы 66
Пустой оператор 69
Операторы управления сбором информации в ходе моделирования 70
Оператор assert 70
Оператор report 71

ГЛАВА 3. БАЗОВЫЕ КОНСТРУКЦИИ МОДЕЛЕЙ НА ЯЗЫКЕ VHDL 73

Сигналы 73
Структура описания объекта моделирования 77
Декларативная часть 77
Описание архитектуры объекта моделирования 80
Библиотеки 81
Пакеты 83
Описание поведения объекта моделирования 85
Процессы для описания архитектуры 86
Последовательный оператор присваивания значения сигналу 87
Задержки в модели устройства 89
Инерционная задержка 90
Транспортная задержка 91
Разрешение неоднозначности установления значения сигнала 92
Дельта-задержка сигналов 95
Параллельные операторы присваивания значения сигналу 99
Простой параллельный оператор присваивания 99
Оператор условного присваивания 99
Оператор селективного присваивания 102
Примеры присваивания значений сигналам 103
Атрибуты сигналов 107
Операторы ожидания *wait* 108

Параллельный оператор контроля в ходе моделирования *Assert* 109
Структурное описание объекта моделирования **110**
Компоненты 110
Декларация компонента 111
Включение компонента в модель объекта (instantiation) 112
Оператор генерации (Generate) 115
Пример структурного описания в OrCAD Express 9.1 118
Задание конфигурации компонентов 121
Конфигурационная спецификация 121
Конфигурационная декларация 123
Конфигурирование моделей с многоуровневой иерархией 124
Использование полностью сконфигурированного объекта моделирования 126
Связывание по умолчанию 127
Отложенные присоединения компонентов 128
Вычисляемые сигналы. Разделяемые сигналы и функции разрешения коллизий **128**
Описание вычисляемых сигналов 129
Охраняемые сигналы 132
Настраиваемые параметры модели (Generics) **138**
Блоки **141**
Назначение блоков в модели 141
Охраняемые присваивания новых значений сигналам 142
Явно определенные сигналы-сторожа 144
Спецификация отсоединения (disconnect) 146
Вложенность блоков и организация иерархической структуры модели 147
Конфигурирование объектов, содержащих блоки 148
Подпрограммы **150**
Процедуры 150
Описание процедуры 150
Вызов процедуры 152
Сигналы в качестве параметров процедуры 155
Операторы параллельного вызова процедур 157
Функции 157
Перегрузка процедур и функций 160
Перегрузка операторных символов 161
Видимость описаний 161
Описание подпрограмм в пакетах 163
Файлы **163**
Описание файла 164
Работа с файлами 165
Соответствие между физическими и логическими файлами 166
Видимость описаний и автоматическое открытие/закрытие файлов в модели 167
Открытие и закрытие файлов без использования автоматических режимов 167
Файлы как параметры подпрограмм 168

Глава 4. Проектирование на VHDL **169**

Использование конструкций VHDL для моделирования **169**
Особенности использования временных задержек в операторе присваивания значения сигналу при поведенческом моделировании 169

Использование процессов, сигналов и переменных в поведенческом моделировании ... 172
Процессы и присваивание значений сигналам ... 172
Присваивание значений сигналу разными процессами, параллельными операторами ... 177
Использование списка чувствительности процесса и атрибутов сигналов. ... 179
Использование пакетов ... 182
Использование в моделях механизма wait ... 184
Использование типа запись ... 188
Использование типа запись для внутренних сигналов ... 188
Использование типа запись для входных и выходных сигналов ... 193
Поведенческое моделирование элементов памяти ... 195
Пример использования подпрограмм в поведенческом моделировании ... 205
Объединение сигналов на шине ... 210
Выбор типа сигнала для выходов на общую шину ... 210
Особенности использования типов *std_logic* и *std_ulogic* для организации общих шин в OrCAD. Функции разрешения коллизий ... 217
Особенности программирования на VHDL для синтеза ... 224
От программирования на VHDL для моделирования к программированию для синтеза ... 224
Сигналы, переменные, константы ... 227
Разрядность сигналов ... 228
Перечислимые типы и кодировка значений ... 229
Начальные значения ... 231
Переменные ... 231
Операторы присваивания ... 232
Задержки в операторах присваивания игнорируются! ... 232
Оптимизация выражений в операторах присваивания ... 233
Использование переменных и констант в операторах присваивания ... 234
Синтез управляющих конструкций, содержащих операторы присваивания ... 234
Общие подходы к синтезу комбинационных и последовательностных схем по программе на VHDL ... 234
Условные операторы ... 236
Работа с типами сигналов из стандартных пакетов и библиотек (на примере std_logic) ... 241
Операторы цикла ... 243
Процессы и компоненты ... 247
Список чувствительности процесса ... 247
Использование процессов и компонентов для описания объектов ... 249
Описание модели конечного автомата ... 252
Синтез устройств, описания которых включают в себя подпрограммы ... 255
Глава 5. Практика применения VHDL ... 257
Язык VHDL как средство формализованного представления спецификаций интерфейсов и протоколов ... 257
Моделирование на VHDL обмена данными между устройствами по шине PCI ... 269
Структура модели ... 269
Модель ведущего устройства ... 270

Организация устройства памяти 277
Модель контроллера шины 285
Описание модели на верхнем уровне иерархии 286
Проектирование систем-на-кристалле на основе шины AMBA 291
Высокоскоростная шина AMBA AHB для систем-на-кристалле 293
Сигналы шины AMBA AHB 295
Организация обменов по шине AHB 301
Управление доступом к шине. Арбитр шины AHB 304
Выбор ведущего устройства, которому будет предоставлена шина 304
Выбор ведомого устройства, которое будет участвовать в обмене 306
Контроль выполнения обменов с блокировкой 306
Расщепленные транзакции 307
Поведение ведущего устройства в ходе обмена 307
Поведение ведомого устройства в ходе обмена 309
Проектирование на VHDL блоков подключения на шину AMBA AHB 311
Реализация на VHDL компонентов интерфейса шины AMBA AHB 311
Интерфейс ведущего устройства на шину AMBA AHB 313
Интерфейс ведущего устройства при чтении одного слова 313
Интерфейс ведущего устройства при чтении последовательности слов 320
Интерфейс ведомого устройства на шину AMBA AHB 324
Устройства, выполняющие функции ведущего и ведомого на шине AMBA AHB 331
Компонент вычисления специальной функции 331
Компонент ведомого устройства 338
Компонент ведущего устройства 343
Структурное описание устройства в целом 346
Организация шины AMBA AHB для взаимодействия модулей в системах-на-кристалле. Блок связей на основе мультиплексоров 357
Узел Мультиплексор 358
Узел Арбитр 361
Компонент определения конца запроса 364
Компонент выбора ведущего устройства 368
Компонент выбора ведомого устройства 372
Компонент контроля расщепленных транзакций 374
Структурное описание узла Арбитр 376
Структурное описание Блока связей на основе мультиплексоров 378
Пример функционирования шины AHB на основе мультиплексоров 382
Анализ характеристик шины AMBA AHB для организации связей модулей в системах-на-кристалле 389

Глава 6. Проектирование на VHDL в среде OrCAD Express 405

Проектирование в OrCAD Express 407
Создание первоначального описания схемы 408
Функциональное моделирование 409
Логический синтез и оптимизация 410
Размещение и трассировка 413
Временное моделирование 414
Документирование и архивация 415

Результаты проектирования 415
Компоненты фрейма Express Simulate **415**
Окно сессии 415
Менеджер проектов 417
Представление проекта в файловом виде 417
Представление проекта в иерархическом виде 418
Командная строка 418
Область команд 418
Область информации 419
Окно информации о стеке 419
Рабочая область 419
Работа с проектом **420**
Формирование проекта 420
Работа с VHDL-файлами 421
Компиляция проекта 422
Формирование тестовых наборов 423
Интерактивный способ формирования тестовых наборов 423
Файлы тестов на VHDL (Test Bench-файлы) 428
Выполнение моделирования и просмотр результатов 435
Работа с результатами моделирования 435
Выполнение моделирования 437
Отладка модели 437
Пошаговое выполнение 437
Механизм точек останова 438
Механизм событий 439
Инструментарий, позволяющий получить дополнительную информацию в ходе моделирования 439

ГЛАВА 7. ПРОЕКТИРОВАНИЕ СБИС НА ЯЗЫКЕ VHDL В СРЕДЕ FOUNDATION EXPRESS **441**

Менеджер проектов **441**
Окно менеджера проектов 442
Браузер иерархии 442
Диаграмма проекта 445
Окно сообщений 445
Панель инструментов и полоса статуса 446
Понятие проекта **446**
Типы проектов 447
Управление проектом 448
Создание нового проекта 448
Открытие проекта 448
Копирование проекта 449
Удаление проекта 449
Удаление информации о реализации 449
Сохранение проекта в виде архива 449
Восстановление проекта из архива 450
Изменение свойств физической реализации проекта 451
Работа с библиотеками 451
Выбор библиотек, входящих в состав проекта 452

Изменение порядка просмотра библиотек 452
Получение информации о библиотеке....453
Просмотр содержимого библиотеки....453
Запуск менеджера библиотек 453
VHDL-библиотеки....453
Создание исходного описания проектируемой системы 453
Редактор описаний на языке высокого уровня HDL editor....454
Редактор диаграмм состояний State diagram editor....454
Редактор схем Schematic editor....454
Импорт готовых файлов исходного описания проектируемого устройства 454
Подготовка исходного описания проекта к синтезу....455
Синтез проекта....455
Процесс синтеза....455
Редактирование ограничений 458
Вкладка Global 459
Вкладка Ports....460
Вкладка Advanced....461
Организация работы с окном ограничений....461
Моделирование и проверка функционирования проекта 462
Загрузка описания проекта в программу моделирования Logic Simulator 462
Основное окно программы моделирования 463
Выбор сигналов, состояния которых будут отображаться программой моделирования....465
Определение значений сигналов и работа с результатами моделирования 467
Работа с бинарным счетчиком....469
Использование инструмента Formula 470
Использование инструмента Clock 471
Принудительная установка (Forcing) значения сигналов 471
Удаление связи между внешним источником и сигналом....472
Взаимодействие модели и временных диаграмм внешних стимулирующих воздействий....472
Использование временных диаграмм в качестве источников 473
Редактирование временных диаграмм....473
Выполнение моделирования....475
Моделирование в пошаговом режиме 475
Точки останова 475
Моделирование в режиме длительного промежутка времени....476
Сохранение и загрузка файлов моделирования 479
Анализ результатов моделирования. 479
Измерение временного промежутка между событиями....481
Отслеживание signal conditions 481
Комментарии во временных диаграммах....482
Моделирование устройств памяти....482
Загрузка и сохранение содержимого блоков памяти в файлах....483
Просмотр и редактирование содержимого кристаллов и блоков памяти....484
Режимы моделирования....485
Функциональное моделирование (Functional) 485
Моделирование с единичной задержкой (Unit delay)....486

Временное моделирование (Timing) 486
Сравнение результатов временного и функционального моделирования 488
Функциональное моделирование с регулируемой единичной задержкой (Glitch) 488
Выборочное моделирование 488
Создание физической реализации проектируемого устройства 489
Версии и редакции проекта 489
Процесс создания физической реализации 490
Оптимизация схемы 492
FPGA-редактор 492
Проверка временных характеристик модели после этапа реализации 493
Анализатор временных характеристик схемы (timing analyzer) 494
Генерация файла прошивки FPGA 495
Дополнительные возможности 495
Использование LogiBLOX-компонентов 496
Генерация и использование макроячеек (IP-блоков, CORE-модулей) 497

Приложение 1. Некоторые особенности использования конструкций языка VHDL при Синтезе 509

Приложение 2. VHDL, поддерживаемый Orcad Express 9.1, и его отличия от стандарта IEEE 1076-1993 511

Приложение 3. VHDL, поддерживаемый Foundation Express 2.1, и его отличия от стандарта IEEE 1076-1993 529

Приложение 4. Пакет std_logic_1164 539

Приложение 5. Пакет std_logic_arith 541

Приложение 6. Пакет Numeric_std 545

Приложение 7. Пакет STD_LOGIC_UNSIGNED 549

Приложение 8. Пакет TEXTIO 551

Чтение из файла 552
Определение конца файла и конца строки 553
Запись в файл 554

Приложение 9. Директивы Foundation Express 555

Директивы включения и выключения трансляции 555
Директивы решающих функций 556
Директивы component implication 556

Приложение 10. Зарезервированные ключевые слова 557

Литература 559

Введение

Согласно известному эмпирическому правилу, так называемому закону Мура (Moore's Law), число транзисторов на кристалле СБИС (сверхбольших интегральных схем) удваивается каждые 18 месяцев (для процессорных схем). Это правило, сформулированное в 1965 году [14], когда на кристалле интегральной схемы размещалось всего 30 транзисторов, ко всеобщему удивлению, продолжает работать до сих пор [6,13] (когда на кристалле процессорной СБИС размещается уже свыше 50 миллионов транзисторов!). Для логических СБИС к 2010 году прогнозируется миллиард транзисторов на кристалле [5]. В некоторых СБИС — программируемых логических интегральных схемах (ПЛИС — FPGA), вполне доступных и отечественным разработчикам, уже содержатся миллионы вентилей.

Возрастание сложности СБИС на базе развития интегральной технологии позволяет иметь в аппаратуре все большее число компонентов, схемотехнически реализовывать все более многообразные и сложные функции. Для эффективного использования этих возможностей необходим переход на новые технологии проектирования и применения аппаратно реализованных узлов, блоков, систем.

Типичная логическая схема в графическом представлении содержит на странице фрагмент, эквивалентный порядку 200 вентилей [15]. Соответственно, схема СБИС на 10 тыс. вентилей будет объемом в 50 страниц. Легко представить себе, во что выльется (по времени) составление и ввод в графической нотации схем СБИС сложностью 50 тыс., 100 тыс., 500 тыс. вентилей и далее.

Альтернативой рисования детализированных схем из низкоуровневых элементов являются языки описания аппаратуры высокого уровня. Собирательно языки этого класса называют языками HDL (Hardware Description Language). Они не только обеспечивают компактную запись для проектируемой схемы, дают значительное сокращение трудоемкости и сроков разработки больших схем, но и упрощают миграцию, перенос проекта на

разные варианты интегральных технологий, реализацию их в СБИС с учетом специфики технологий различных производителей. Разработчик получает возможность оценивать варианты реализации проектируемого устройства в СБИС при различных вариантах проектных ограничений, на различных технологиях, у различных производителей.

Однако проблема не ограничивается только количественными характеристиками описания проектируемых схем. Используемые при разработке и применении аппаратуры традиционные схемы разных уровней (структурные, функциональные, принципиальные) являются как бы синтаксическим описанием аппаратно реализованных технических решений. Описание их работы, выполняемых функций традиционно дается словесно, с использованием привычных, но не стандартизированных и не всегда однозначных в понимании дополнительных графических форм (временных диаграмм и др.). Традиционные семантические формы спецификации функционирования цифровой аппаратуры — таблицы истинности, конечные автоматы, сети Петри и другие — оказываются пригодными для спецификации лишь очень небольших, по современным меркам, фрагментов аппаратуры. Используемые формы спецификаций не являются исчерпывающими, позволяют отразить лишь отдельные аспекты функционирования описываемой системы. Например, стандартных формализованных механизмов спецификации временно́го поведения аппаратно реализованных устройств просто нет. Некоторые специальные формализмы, типа темпоральной логики (temporal logic), не получили широкого распространения и в практике проектирования и эксплуатации аппаратуры не применяются.

Используя традиционные средства описания аппаратуры, сложно получить целостное, достаточно строгое, однозначное описание современных многокомпонентных и функционально сложных цифровых систем.

Возрастающая алгоритмическая сложность аппаратно реализованных устройств приводит к тому, что, как проблемы разработки, описания и применения аппаратуры (hardware), так и подходы к их решению, становятся подобны проблемам и методам решения для современных программных систем (software).

Перспективное направление решения этих проблем — применение *алгоритмического подхода*, создание алгоритмического языка для описания аппаратуры, *программирования* и структуры, функционирования аппаратных средств обработки информации.

Наиболее распространенным языком этого класса, специфицированным международными стандартами, является язык VHDL [3,10], который разработан в рамках американского проекта создания нового поколения высокоскоростной элементной базы (Very High Speed Integrated Circuits — VHSIC). Аббревиатура VHDL расшифровывается как **VHSIC Hardware Description Language**. Расширение языка VHDL — язык VHDL-AMS (Very-High-Speed

IC Hardware Description Language — Analog and Mixed Signal) [9] включает также возможности моделирования систем, содержащих и цифровую, и аналоговую части.

Язык VHDL предназначен для решения комплекса задач в ходе проектирования и применения цифровых систем, их аппаратных средств [27], в том числе:

1. Описания структуры системы, декомпозиции системы на подсистемы, спецификации связей и взаимодействия подсистем.
2. Спецификации функционирования системы, узлов, блоков, реализуемых функций. Спецификация дается в алгоритмической форме, с использованием привычных современному специалисту программных конструкций алгоритмического языка, включающих в себя спецификацию временно́го поведения сигналов и блоков.
3. Моделирования системы и ее работы на основе четкой спецификации структуры системы, а также функционирования ее компонентов.
4. Синтеза схемотехнической реализации системы, автоматической генерации детальной структуры на основе строгой спецификации системы на языке VHDL – спецификации на более абстрактном уровне.

Определив язык, мы получаем возможность писать на нем программы, то есть использовать его для описания структуры и функционирования системы (пункты 1 и 2 в приведенном перечне задач). Такое описание, достаточно формализованное и однозначное, будет уже иметь самостоятельную ценность, как средство передачи знаний о спроектированной аппаратно реализованной цифровой системе (устройстве, блоке) от разработчика к специалисту, ее применяющему.

В этом качестве язык VHDL постепенно становится стандартным при документировании аппаратных средств, причем не только на уровне СБИС, но и на уровне плат и блоков. Высокий уровень описания проектируемого устройства на языке VHDL дает так называемый самодокументирующий характер описанию проекта. Текст программы на языке VHDL сам по себе является документацией, которую опытный специалист легко читает. Не случайно есть тенденция — в перечень необходимой документации на цифровые электронные блоки и устройства включать описание на языке VHDL. Так, Европейское космическое агентство (ESA) стандартизировало язык VHDL — и как средство обмена информацией при проектировании и применении заказных СБИС (ASIC), и как инструмент для работы на уровне конструктивно-функциональных модулей, аппаратных узлов на уровне плат [16, 17]. В отечественной промышленности также предполагается ввести документирование на VHDL как обязательную составляющую технической документации на изделия [23].

HDL-языки составляют необходимую основу для развития методологии проектирования систем на СБИС с повторным использованием готовых компонентов, уже апробированных и отработанных на других проектах (такую методику называют устоявшимся словосочетанием *design re-use*).

Основное внимание в настоящей книге уделяется этапам функционального и логического проектирования систем на СБИС. Именно с этими этапами имеет дело разработчик цифровых систем, реализуя заданные техническим заданием требования к проектируемой системе. Для заказных и полузаказных СБИС логический проект, сформированный разработчиком, поступает на схемотехническое и топологическое проектирование кристалла производителями СБИС. Однако, при реализации проектируемого цифрового устройства на FPGA (ПЛИС) — готовых кристаллов с фиксированной топологией, специальные формы схемотехнического проектирования для данного класса кристаллов, размещение реализации схемы на кристалле и трассировка также выполняются разработчиком систем на СБИС самостоятельно. Во всех вариантах реализации цифровых систем, современная технология проектирования СБИС базируется на использовании HDL-языков, прежде всего — языка VHDL.

В *главе 1* рассматриваются общие принципы представления и проектирования систем на СБИС, основные классы СБИС, как элементной базы технической реализации проектируемых цифровых систем, типовые этапы разработки системы, место языка высокого уровня VHDL в этих процессах.

В *главе 2* описываются базовые элементы языка VHDL, типы данных и операторов.

В *главе 3* даются базовые конструкции языка VHDL для поведенческого и структурного описания проектируемого цифрового устройства. Акцентируется внимание на специальном классе объектов языка VHDL — сигналах, не имеющих аналогов в традиционных языках программирования.

В *главе 4* излагаются методы использования конструкций языка VHDL для моделирования поведения, а также особенности программирования на VHDL для синтеза проектируемой системы на СБИС.

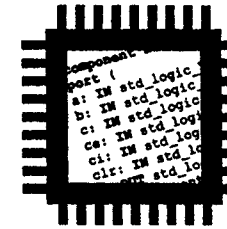
В *главе 5* приводятся примеры использования языка VHDL при выполнении различных практических задач, решаемых при проектировании и применении систем на СБИС — от формализованной спецификации интерфейсов и протоколов, моделирования устройств, взаимодействующих по шине, до проектирования систем-на-кристалле. Рассмотрение ведется на примерах с использованием шины PCI на уровне блоков цифрового устройства и внутрикристалльной шины AMBA (для систем-на-кристалле).

Для того чтобы работать с программами на языке VHDL, выполнять их, требуется система программирования, которая бы проверяла корректность программы на данном языке, организовывала трансляцию, компоновку и

выполнение оттранслированной программы при моделировании. Примером такой системы программирования для языка VHDL является программный комплекс семейства OrCAD — OrCAD Express, рассматриваемый в *главе 6* (реализует задачи 1, 2 и 3). С другой стороны, при синтезе устройства, описанного программой на языке VHDL, пакеты проектирования должны выполнять трансляцию модели на VHDL, компиляцию аппаратной реализации устройства в кристалле СБИС. Для синтеза реализации проектируемых цифровых систем на FPGA используются специализированные пакеты сквозного автоматизированного проектирования с использованием языка VHDL (задачи 1, 2, 3 и 4). В качестве примера такого пакета в *главе 7* описывается организация работы в пакете Foundation Express по проектированию цифровых систем на FPGA фирмы Xilinx.

В приложениях приводится справочная информация, полезная при практическом проектировании систем на СБИС на языке VHDL.

При подготовке книги использованы материалы курсов лекций, читаемых авторами в Санкт-Петербургском государственном университете аэрокосмического приборостроения.



Глава 1

Уровни и процесс проектирования СБИС

Уровни представления и проектирования СБИС

Понятие СБИС как сложного многокомпонентного объекта объединяет в себе и логические функции (преобразования информации, обработки данных), и структурную организацию многокомпонентной системы, и специфические вопросы их реализации в интегральной технологии как физического объекта — сложного полупроводникового прибора со специальной многослойной топологией. И представление такого объекта, и организация его проектирования требуют учета всего комплекса факторов, составляющих понятие СБИС, причем с учетом их взаимодействия и взаимного влияния.

СБИС, в зависимости от взглядов на ее природу и организацию, принято рассматривать в трех областях (измерениях) [11]:

- в *Функциональной* (Поведенческой) области;
- в *Структурной* области;
- в *Геометрической* (Топологической) области.

В каждой области формируется своя иерархия представления СБИС, отражающая их многоуровневую организацию в терминах этой области (рис. 1.1). Например, в Структурной области это будет:

- *уровень крупных структурных блоков*; иллюстрируя, говорят "уровень блоков типа Процессор, Память, Коммутатор" (Processor, Memory, Switch — PMS-level;
- *уровень регистров*, функциональных блоков, межрегистровых связей (Register Transfers Level — RTL-level);
- *уровень логических вентилей* (Gates, Gate level);
- *уровень транзисторов* (Transistor level).

Процесс проектирования СБИС носит последовательный характер спуска по этим уровням (по каждой из трех областей представления). Однако спуск по уровням в каждой из областей связан и перемежается с движением по оси иерархии других областей. Поуровневый характер проектирования внутри каждой области сочетается с согласованным движением по осям других измерений (областей).

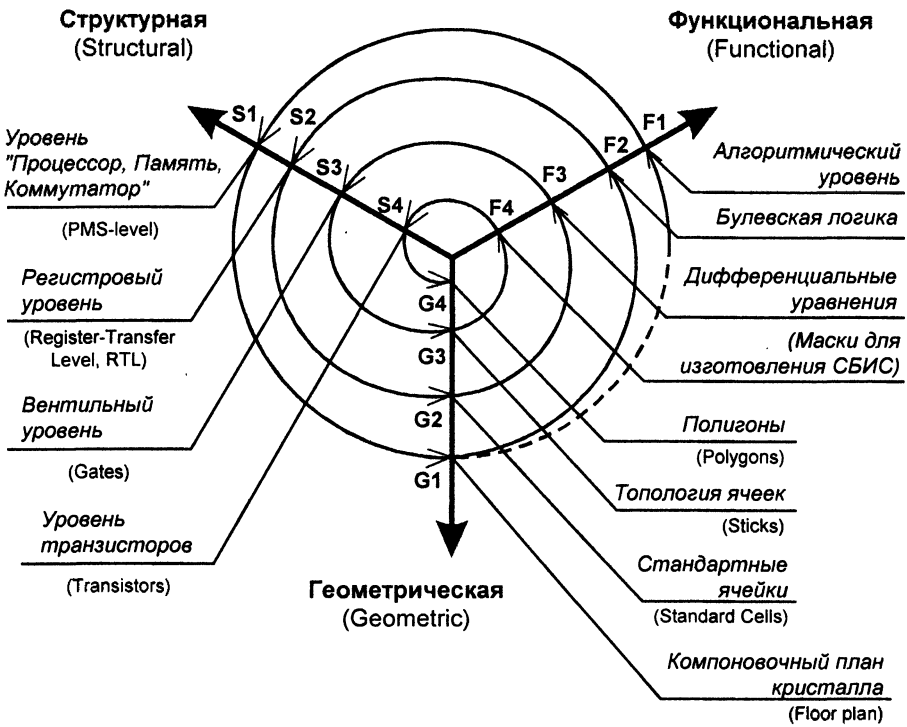


Рис. 1.1. Области и уровни моделей в проектировании СБИС. Диаграмма Гайского-Кана (Gajski and Kuhn)

Иллюстрация этого взаимодействия представлена на так называемой диаграмме Гайского-Кана в виде спиралевидной кривой, начинающейся с Алгоритмического уровня в функциональной области, а заканчивающейся на уровне Полигонов, представляющих собой послойные маски для изготовления кристалла (рис. 1.1).

Подобная многомерная иерархия, отражающая объективные взаимосвязи разных аспектов построения СБИС, учитывается в методиках и в организации процессов их проектирования, реализуется в поддерживающих системах автоматизированного проектирования.

Сама организация проектирования СБИС также носит отчетливо иерархический характер. В комплексе процессов проектирования СБИС принято выделять ряд основных уровней (рис. 1.2). Их реализация в организационных процессах создания СБИС — от постановки задачи до изготовления кристаллов СБИС в целом следует этому общему представлению. Однако по мере эволюции СБИС, технологии их производства и методик проектирования модифицируются задачи, решаемые на каждом из уровней, меняются требования к методикам и инструментарию их поддержки.

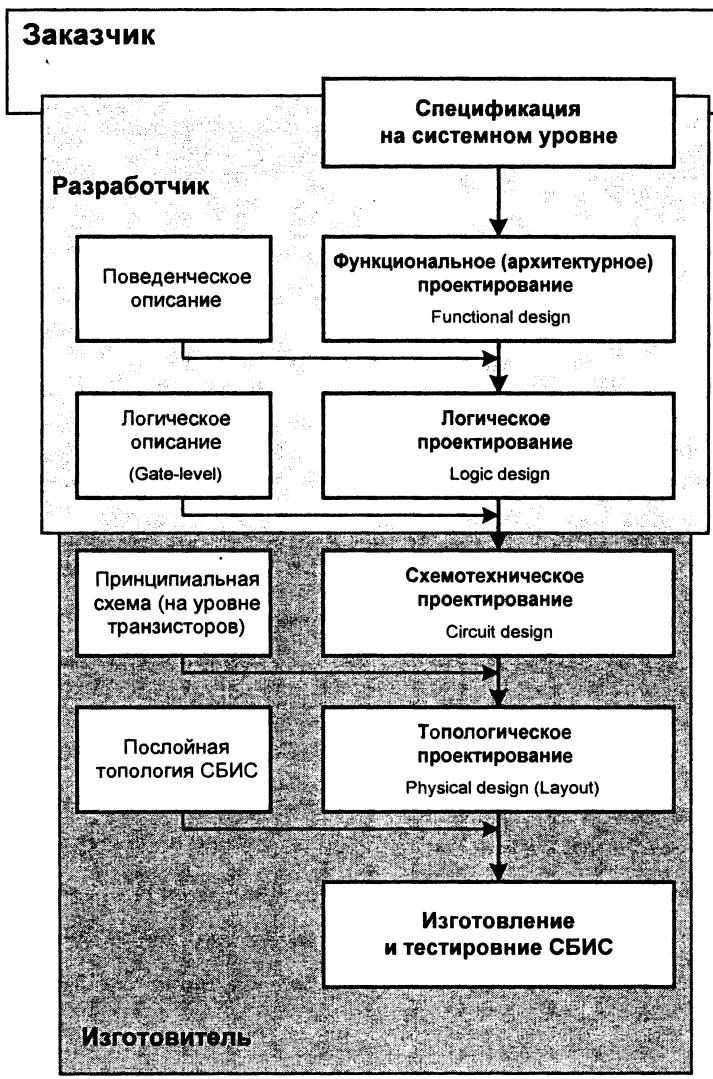


Рис. 1.2. Уровни проектирования СБИС

Традиционный подход к поуровневому проектированию СБИС

В поуровневом, многоэтапном процессе проектирования стараются использовать автоматизированные системы синтеза реализации проектируемого изделия на более низком (более детальном) уровне по спецификации проектируемой СБИС на более высоком уровне. Традиционные САПР поддерживают синтез представления изделия на вентильном уровне (логический проект) по спецификации проектируемого устройства уровня регистровых передач, RTL-уровня, а также синтез топологии СБИС — по спецификации устройства на вентильном уровне (кремниевая компиляция). При этом ответственность за выполнение требований по занимаемой площади кристалла, производительности и потребляемой мощности проектируемого узла, за сбалансированность и оптимизацию принимаемых решений возлагается на САПР.

Сложность задачи синтеза (с оптимизацией) определяет ограничения существующих пакетов синтеза в САПР. Ограничения САПР по сложности автоматически синтезируемых узлов на вентильном уровне определяют требования к разбиению проекта СБИС на раздельно синтезируемые узлы. Проект СБИС, разбитый на узлы, специфицируется на RTL-уровне, и в таком виде вводится в САПР. Далее, с этого уровня идет поэтапный переход с проекта одного уровня на проект нижележащего уровня, с использованием автоматического синтеза и верификации (от функционального проекта — к логическому, от логического — к физическому проекту, с получением послойной топологии СБИС).

Традиционные САПР проектирования заказных и полузаказных СБИС используют автоматические средства синтеза и верификации, начиная с перехода от RTL-уровня к логическому уровню. Проект функционального уровня, RTL-уровня спецификации проектируемой СБИС разработчик выполняет сам, вручную, на основе изложенных на бумаге требований к проекту, спецификации проекта СБИС на системном уровне. Получающийся в результате проект на уровне RTL (и тесты для него) являются результатом неформализованного творчества разработчика, а не автоматизированных процессов синтеза и верификации.

Важно также представлять себе взаимоотношения и разделение труда между основными участниками процесса создания новой СБИС: заказчиком, разработчиком (проектировщиком — designer) и изготовителем (рис. 1.2). При создании заказных и полузаказных СБИС (ASIC — Application Specific Integrated Circuit) эти три участника обычно разделены как по персоналиям, так и по организациям, по фирмам, участвующим в этом процессе. Разработчик СБИС класса ASIC доводит проектирование до логического проекта, отрабатывает проектируемое устройство на моделях в САПР Логического проектирования, включая и комплекс тестов для разработанного устройства. Далее

проект передается фирме-производителю СБИС (ASIC vendor), которая уже самостоятельно проводит физическое проектирование (с учетом специфики своей технологии изготовления СБИС на конкретном оборудовании) и изготавливает кристаллы.

Блочно-ориентированное проектирование

По мере роста сложности СБИС, стоимости и временных затрат проектирования новой СБИС "с нуля", все большее значение придается повторному использованию ранее спроектированных и использованных в СБИС компонентов для создания новых изделий.

В рамках этой тенденции получило развитие так называемое блочно-ориентированное проектирование (Block-based design). Оно позволяет использовать в новом проекте, в качестве готовых компонентов, функциональные узлы и топологические фрагменты из предыдущих проектов СБИС. Реализация этой, вполне естественной по своей постановке, задачи требует частичного изменения в традиционной схеме проектирования СБИС, изменения соотношений и взаимодействия между проектированием на системном уровне, RTL-уровне и физическим проектированием.

Повторное использование относится к реализации функций системного уровня. Проектирование начинается с построения *поведенческой модели* на системном уровне, где отрабатывается системный проект, принимаются решения о разделении реализации функций аппаратными или программными средствами и т. д. Затем компоненты системного уровня нового проекта разбиваются и отображаются на набор специфицированных ранее функциональных блоков RTL-уровня. Для выбранных блоков RTL-уровня проводится оценочное проектирование, устанавливаются условия их соответствия ограничениям по времени, потребляемой мощности, занимаемой площади кристалла. По результатам проведенной работы определяется, какие из готовых блоков можно оставить в новом проекте, а какие лучше спроектировать заново.

Типичным для этой методики является использование в новых проектах готовых блоков программируемых процессоров (процессорных ядер — processor cores), т. е. центральной части процессоров сигналов, популярных микропроцессоров, микроконтроллеров. Блоки могут браться либо как готовые, отработанные и надежные топологические фрагменты (hard blocks), либо как спецификация на уровне схемотехнического проекта (firm blocks), либо как блоки на RTL-уровне, которые далее надо проектировать и верифицировать на нижних уровнях.

При верификации как повторно используемых, так и новых блоков, в блочно-ориентированном проектировании применяют реалистические тестовые наборы, полученные на модели системы на поведенческом уровне. Этот под-

ход знаменует переход к *внутрисистемной верификации* проектируемых блоков ("System-in" verification), которая одна только и может дать достоверные результаты верификации для сложных СБИС комплексных систем.

Таким образом, блочно-ориентированный подход требует интегрированных сред разработки, сквозных языков и методов спецификации систем по уровням проектирования СБИС.

Проектирование Систем-на-кристалле

Дальнейшим развитием технологии проектирования систем на СБИС является проектирование *Систем-на-кристалле* — СнК (System-on-a-chip Design, SoC). В отличие от блочно-ориентированного проектирования, которое базировалось, в основном, на повторном использовании собственных разработок, проектирование СнК заключается в возможности использования *готовых компонентов из разных источников*, от разных разработчиков. На это сориентирован как процесс создания комплексной СБИС, так и разработка самих компонентов для многократного использования в разных проектах (design reuse).

Проектирование СнК является следующим шагом в эволюции технологии построения систем на СБИС, включает в себя комбинацию инструментария и методологии проектирования сложных систем на одном кристалле в пределах короткого временного цикла.

Как и блочно-ориентированное проектирование, СнК является иерархической технологией проектирования, начинающейся на системном уровне. Иерархическое проектирование (design hierarchy), а также использование в проектах преимущественно готовых функциональных компонентов, позволяют успешно справляться с большой сложностью проектируемых систем, размещаемых на кристалле, радикально сократить временной цикл создания СБИС, повысить производительность разработчиков.

В качестве основных компонентов проекта СнК используются готовые, предварительно верифицированные и апробированные блоки со стандартизированными интерфейсами. Проектирование СнК включает две основные составляющие:

- авторизацию блоков для применения в проекте (block authoring);
- интеграцию блоков в систему-на-кристалле (system-chip integration).

Авторизация блоков использует методологию, аналогичную блочно-ориентированному проектированию, дополняя ее двумя ключевыми методиками: стандартизацией интерфейсов блоков, а также проектированием для виртуальной системы (virtual system design). Проектирование компонентов для виртуальной системы означает, что разработчик блоков (компонентов многократного применения в разных СнК) должен использовать средства для

установления системных ограничений на применение его компонента в конкретных СнК (для верификации соответствия компонента этим ограничениям и для адаптации компонента к ним при включении в проект СнК).

Интеграция СнК заключается в проектировании и верификации архитектуры на системном уровне, а также организации интерфейсов между блоками, компонуемыми в СнК. Здесь важную роль играет ориентация на архитектуру со стандартизированными интерфейсами блоков, а также на стандартизацию форм и языков спецификации интерфейсов компонуемых блоков. Разработчики блоков снабжают их стандартизированным комплектом спецификаций, включая спецификации интерфейсов, причем комплектом многоуровневым — от системного уровня до того уровня, который реализован в компоненте (для топологических компонентов — до физического уровня).

Концепция СнК, принципы иерархического проектирования, заложенные в ее основу, дают возможность строить и использовать готовые блоки (компоненты СнК разного уровня). Такие готовые блоки часто называют *IP-блоками*, *IP-ядрами* (IP-cores; IP — от Intellectual Property, интеллектуальный продукт[1]) или *виртуальными компонентами* (*virtual components — VC*), виртуальной элементной базой для систем на кристалле.

Различают три основных класса блоков:

- *программные IP-блоки* (soft blocks) — блоки, специфицированные в языке описания аппаратуры (например, на языке VHDL, на программируемых уровнях проектирования СБИС — функциональном, логическом; RTL-спецификации). Программные IP-блоки наиболее гибки в применении, технологически независимы, могут быть использованы в проектах, ориентированных на различные технологии реализации. Например, soft IP-блоки могут легко переноситься из проектов устройств на FPGA в проекты ASIC, проекты полузаказных и заказных СБИС. Недостатком программных блоков является сложность прогнозирования характеристик их реализации на конкретной технологии (скоростных характеристик, занимаемой площади кристалла, потребляемой мощности). Еще одна проблема — сложность защиты авторских прав разработчика программного IP-блока (приходится передавать исходный текст на VHDL пользователю [4]);
- *схемотехнические блоки* (firm blocks) — блоки, специфицированные на схемотехническом уровне, без привязки к конкретной топологической реализации. Они более гибкие, чем блоки физического уровня, однако при применении в СнК потребуется проектирование (генерация) и верификация перевода блока на физический уровень. Схемотехнические IP-блоки имеют достаточно предсказуемые характеристики по занимаемой площади и быстродействию, представляются на вентильном уровне, в

[1] Здесь аббревиатура IP не имеет никакого отношения к распространенному термину "IP-протокол" в сети Интернет.

форме списков связей (netlists). В них уже отражены определенные технологические ограничения реализации, эти блоки структурно и топологически оптимизированы по производительности и занимаемой площади, используют справочные данные о планируемой технологии реализации (reference technology library), Firm IP-блоки могут содержать данные о взаимном расположении элементов на площади кристалла. Однако полную трассировку связей внутри блока они все же не включают. Схемотехнические IP-блоки занимают промежуточное положение между программными IP-блоками и аппаратными IP-блоками, они более оптимизированы и предсказуемы по характеристикам, чем soft IP, но, в то же время, более жестко завязаны на технологию реализации; более гибки, чем hard IP, но, по сравнению с последними, менее оптимизированы и требуют дополнительных этапов схемотехнического и топологического синтеза для конечной реализации на кристалле

- *физические (топологические) блоки* (hard blocks) — блоки, специфицированные на физическом уровне реализации СБИС. Они представляют собой реализацию блока в топологии на кристалле (Polygon level data), дают наиболее оптимизированную форму реализации функций блока, имеют вполне определенные характеристики по быстродействию, площади, потребляемой мощности. Однако физические блоки имеют жесткую привязку к технологии изготовления кристалла и топологическим ограничениям, поэтому перенос их в изделия, проектируемые под другие технологические требования, может повлечь за собой их полное перепроектирование.

Разработка виртуальных компонентов для проектирования СнК (компонентов многократного применения в различных проектах СБИС) превращается в самостоятельную форму деятельности. Сами виртуальные компоненты всех перечисленных выше классов получили собирательное название *интеллектуальных продуктов* для построения СнК (Intellectual Property, сокращенно — IP). В отечественной практике такие виртуальные компоненты называют также *сложными функциональными блоками* (СФ-блоками). Компоненты-IP разрабатываются и применяются фирмами-разработчиками СБИС, как традиционными фирмами-изготовителями СБИС, так и новыми независимыми фирмами, не имеющими своего производства интегральных схем (для них даже появилось специальное название — fabless companies), для которых собственно разработка и продажа IP-компонентов, проектирование СнК становятся основным бизнесом.

Технологии создания Систем-на-кристалле, поддерживающие их иерархические системы автоматизированного проектирования, унифицированные процедуры взаимодействия, сложившаяся система разделения труда между разработчиками и изготовителями СБИС, делает создание систем на СБИС доступным небывало широкому кругу пользователей, создает качественно новые возможности для создания высокоэффективных и экономичных систем на СБИС.

В этих процессах особую роль играют методики и языки спецификации проекта на разных уровнях, реализующие их системы автоматизации программирования, моделирования и проектирования. Язык VHDL решает значительную часть этих задач, от системного до логического уровня. VHDL стал фактическим стандартом в проектировании СБИС (включая СнК), он является важной составляющей рынка IP, поддерживается практически всеми основными современными системами автоматизации проектирования СБИС.

Синтез реализации проектируемых схем на кристалле СБИС

Синтез схемы по ее модели-спецификации на языке высокого уровня (в нашей книге — на языке VHDL) выполняется воплощением проектируемого цифрового устройства в виде так называемого логического проекта — схемы на вентильном уровне (Gate-level), то есть схемы, детализированной до уровня отдельных вентилей — примитивов представления цифрового устройства на логическом уровне.

Элементная база технической реализации СБИС

Наиболее часто говорят о представлении схемы на вентильном уровне. Сложность реализации схемы оценивают в числе вентилей, однако далеко не всегда в современной интегральной схемотехнике фактическая реализация проектируемого устройства идет в логических вентилях.

Специалистам в области цифровой техники хорошо известно, что цифровая схема может быть представлена в различных базисах [28]. Выбор логических элементов при синтезе схемы определяется элементной базой технической реализации, которая обычно связана с классом СБИС, используемым для воплощения проектируемого устройства.

Полностью заказные СБИС. Вентильное представление схем логического проекта, полученное в ходе проектирования, реализуется последующим прямым переводом в принципиальную схему на уровне транзисторов, с дальнейшей разработкой послойного рисунка топологии проектируемого кристалла. Разработанная СБИС требует постановки серийного производства в полном объеме, для всех этапов изготовления СБИС. Разработка и освоение серийного производства СБИС такого рода требует огромных затрат, которые оказываются оправданными только при массовом производстве подобных изделий, выпуске многомиллионными сериями.

Это определяет необходимость либо высокой универсальности применения заказных СБИС, многоцелевого использования их в изделиях (например,

СБИС памяти), либо ориентации на рынки, сферы применения, которые сами могут обеспечить многомиллионные объемы рынка для них (например, периферийные СБИС, чипсеты для ПЭВМ, СБИС для мобильных телефонов, и др.). Обычный проектировщик устройств и систем на СБИС, систем в интегральном исполнении, редко имеет дело с разработкой СБИС такого рода.

При заданном уровне интегральной технологии заказные СБИС обеспечивают наилучшие характеристики по плотности размещения схемы на кристалле (а значит — по сложности схем, которые можно разместить на кристалле СБИС), по быстродействию реализуемых в СБИС устройств. Проблема этого класса СБИС — большая трудоемкость, а следовательно, высокая стоимость разработки и постановки производства.

Полузаказные СБИС (ASIC) на Базовых матричных кристаллах (БМК, GA). При проектировании цифровых устройств в интегральном исполнении, не ориентированных на многомиллионный выпуск устройств для конкретных изделий, во многих случаях наиболее приемлемым вариантом является их реализация на основе базовых матричных кристаллов (БМК). Иные названия этого класса СБИС — вентильная матрица (Gate Array, GA), вентильная матрица с масочным программированием.

В этом классе интегральных схем проектирование и производство СБИС разбивается на две стадии. Первая стадия, общая для всех устройств, реализуемых на данном БМК, — общая и по проектированию, и по серийному производству. Вторая стадия — индивидуальная для каждого проектируемого устройства.

На первой стадии проектируется кристалл, как универсальное техническое средство для реализации самых разных устройств. В силу универсальности, кристалл может производиться в массовом количестве, что обеспечивает его приемлемую стоимость. Однако такой кристалл является еще только заготовкой для создания СБИС проектируемых устройств. На второй стадии, для того чтобы реализовать на кристалле конкретную функцию, конкретное устройство, набор схемных компонентов заготовки преобразуется в нужную схему посредством выполнения дополнительных соединений между ними. Эти соединения реализуются на кристалле путем напыления дополнительных слоев на уже готовый кристалл-заготовку. Для задания индивидуального рисунка межсоединений, соответствующих конкретному устройству, проектируется несколько специальных фотошаблонов (масок), число которых в несколько раз меньше, чем общее число шаблонов для изготовления законченной СБИС. Таким образом, с помощью нескольких дополнительных шаблонов универсальный кристалл массового выпуска как бы программируется на выполнение конкретных функций, на реализацию конкретного устройства.

Такие СБИС называются полузаказными (в англоязычной литературе им соответствует термин ASIC — Application Specific Integrated Circuit).

Схемные компоненты кристалла-заготовки БМК — некоторый набор элементов схем, называют базовыми ячейками. Базовые ячейки регулярно, в большом количестве повторяются на площади кристалла БМК. Они занимают внутреннюю область кристалла БМК. Периферийную область кристалла занимают другие схемные компоненты — базовые ячейки ввода/вывода, ориентированные на реализацию внешних связей БМК. Для формирования связей поверх базового кристалла-заготовки, в БМК могут быть предусмотрены специальные свободные зоны между базовыми ячейками (такие БМК называют *канальными* БМК). Если таких зон на площади кристалла не выделяется, то вся площадь заполняется базовыми ячейками, и любая область может быть использована как для создания логической схемы, так и для создания соединений (*бесканальные* БМК).

БМК могут строиться как массив базовых ячеек — законченных логических элементов, или как массив не скоммутированных в логическую схему транзисторов. В классе бесканальных БМК их называют, соответственно, *море вентилей* и *море транзисторов*. Для повышения эффективности реализации, на БМК функционально-сложных устройств, включающих не только комбинационные схемы, но и значительное число элементов памяти, в современные БМК высокой степени интеграции, кроме базовых ячеек, включаются специализированные блоки памяти, реализованные на уровне топологии кристалла-заготовки. Такие БМК называют *блочными* БМК. В однородную структуру БМК как моря вентилей или море транзисторов как бы погружаются специализированные блоки различного назначения — блоки памяти, блоки умножителей. Реализованные на уровне топологии кристалла, они обладают существенно большей компактностью и лучшими скоростными характеристиками.

При всем разнообразии видов БМК, сохраняется традиция оценивать их сложность в *эквивалентных вентилях*, под которыми понимается группа элементов, соответствующая возможности реализации функции логического вентиля — двухвходового элемента И-НЕ или ИЛИ-НЕ.

Заданием дополнительных соединений внутри и между базовыми ячейками реализуются функционально-законченные компоненты схемы устройства, которые принято называть функциональными ячейками. Функциональная ячейка может строиться внутри одной базовой ячейки, или на нескольких базовых ячейках. Создание функциональных ячеек идет уже на второй из указанных стадий — на стадии создания СБИС конкретного вида путем наложения дополнительных слоев соединений на кристалл-заготовку БМК.

Функциональные ячейки могут формироваться на этапе синтеза схемы проектируемого устройства по его описанию, на языке VHDL, или проектиро-

ваться заранее, могут объединяться в библиотеки и уже оттуда включаться в качестве готовых компонентов схемной реализации устройства. Обычно выпускаемые БМК сопровождаются библиотеками функциональных ячеек, которые поставляются разработчиками и производителями БМК. Сделанные вручную, высококвалифицированными проектировщиками, досконально знающими возможности и специфику своего БМК, функциональные ячейки из таких библиотек, как правило, обладают лучшими характеристиками, чем аналогичные фрагменты схем, синтезируемые автоматически. Поэтому библиотеки функциональных элементов активно используются как программными пакетами автоматического синтеза, так и проектировщиками устройств на БМК. Функциональные ячейки в значительной степени формируют базис реализации устройства, проектируемого на языке высокого уровня. Учет их состава и характеристик может оказывать существенное влияние на структурное проектирование синтезируемых моделей-спецификаций разрабатываемых устройств на СБИС.

Полузаказные СБИС имеют существенно меньшую трудоемкость и стоимость производства, чем заказные СБИС. При этом они обеспечивают достаточно хорошие характеристики — как по сложности реализуемых схем, так и по быстродействию. По сравнению с рассматриваемыми ниже FPGA, полузаказные СБИС эффективнее по техническим характеристикам (сложность, быстродействие), а при достаточном серийном выпуске изделий — и по стоимости. Однако сроки создания полузаказных СБИС существенно больше, чем реализация устройства на FPGA.

Программируемые пользователем вентильные матрицы (FPGA). Дальнейшее развитие идеи БМК находят в создании другого класса СБИС — вентильных матриц, программируемых пользователем, по-английски — Field Programmable Gate Array (FPGA), буквально — вентильных матриц, программируемых на месте применения. В русском наименовании СБИС этого класса часто называют *программируемыми логическими интегральными схемами*, ПЛИС; однако не все авторы принимают такое наименование как русскоязычный аналог FPGA.

Общность идеи БМК и FPGA состоит в том, что СБИС выпускается большими сериями, но является не законченным устройством, а кристаллом-заготовкой для реализации на ее базе конкретных устройств в интегральном исполнении. Так же, как и в БМК, в FPGA наполнением кристаллов-заготовок являются массивы идентичных схемных компонентов, которые будут потом конфигурироваться и объединяться связями, чтобы реализовывать те или иные функции. Принципиальная разница между БМК и FPGA состоит в том, что конфигурирование и прокладка связей между базовыми ячейками осуществляется без напыления дополнительных слоев на кристалл. Все это выполняется программными настройками, задаваемыми через внешние контакты готовой СБИС. Такой кристалл-заготовку не надо "до-

делывать" в производстве, как БМК, чтобы получить реализацию проектируемого устройства. Готовый кристалл СБИС FPGA, часто уже корпусированный, заключенный в стандартный корпус, надо лишь запрограммировать, настроить на реализацию нужных функций.

Внутренняя организация FPGA, на общем уровне, похожа на организацию канальных БМК. В ней также есть массив базовых ячеек на внутренней части площади кристалла и массив периферийных ячеек по его периметру. Между базовыми ячейками проходят трассировочные каналы для задания связей. Специфика и новое качество FPGA состоит в виде этих знакомых компонентов кристалла.

Базовая ячейка принимает в FPGA вид *конфигурируемого логического блока (КЛБ)*, Configurable Logic Block (CLB). В современных FPGA реализация КЛБ часто имеет вид, весьма далекий от упоминаемых в названии FPGA вентилей (gates) [15]. Так, широко распространенные семейства FPGA фирмы Xilinx XC4000E, Vertex, Vertex-II имеют КЛБ, построенные на логических блоках табличного типа, так называемых Look-Up Tables — LUT. Табличный блок представляет собой блок памяти (в конфигурируемых КЛБ — перепрограммируемой памяти), которая хранит значения n-местной логической функции для всех возможных значений аргументов. Фактически в памяти прямо записывается таблица истинности задаваемой булевской функции от n аргументов. Набор значений n аргументов подается как адрес на блок памяти, и из соответствующей ячейки выдается на выход значение функции. Для задания n-местной логической функции потребуется однобитный блок памяти, имеющий 2^n ячеек. Например, в используемых в FPGA Xilinx КЛБ содержатся LUT с четырьмя входами, т. е. 2^4=16 бит памяти. Память — оперативная, так что в нее при конфигурировании FPGA может быть записана таблица истинности любой булевской функции от четырех аргументов. КЛБ обычно не исчерпываются одной LUT, они могут включать несколько LUT, а также логические схемы, триггеры. Например, КЛБ в Xilinx XC4000 включает в себя 2 LUT на 4 аргумента, 1 LUT на 3 аргумента, 2 D-триггера, связанных через несколько мультиплексоров, а также логические схемы вентильного уровня.

Понятно, что универсальность и настраиваемость КЛБ требуют накладных расходов и не лучшим образом сказываются на быстродействии устройств. Для повышения характеристик FPGA в них все активнее развивается линия на погружение в кристалл готовых функциональных блоков, спроектированных и оптимизированных на уровне схемотехники и топологии СБИС — блоков памяти, блоков умножителей и других, вплоть до целых процессорных блоков (например, "процессорные ядра" PPC405, "погруженные" в FPGA Xilinx Virtex-II Pro, [19]).

Оборотной стороной гибкости, технологической простоты для пользователя реализации устройств на готовых СБИС FPGA являются ограничения по сложности и быстродействию создаваемых на их основе устройств. По

сравнению с полузаказными СБИС, не говоря уже о заказных СБИС, сложность и быстродействие устройств на FPGA существенно меньше. Нельзя не упомянуть и о достаточно высокой стоимости СБИС FPGA. При достаточной серийности проектируемых устройств, единица СБИС при реализации в полузаказной СБИС будет ощутимо дешевле, чем в FPGA.

Несмотря на все эти ограничения, FPGA являются важным и перспективным средством реализации проектируемых цифровых устройств.

Необходимо учитывать значение FPGA в современных условиях для отечественных разработчиков цифровых систем. FPGA являются изделиями массового производства, выпускаемыми несколькими фирмами, широко представленными на мировом рынке. Они доступны отечественным разработчикам изделий на СБИС, причем высокий технологический уровень их производства в мире может давать отечественному разработчику доступ к созданию СБИС большей сложности, чем это можно сделать в виде, скажем, полузаказных СБИС, на текущих отечественных интегральных технологиях.

Наконец, FPGA являются идеальной платформой для отработки и верификации проектных решений. Разработав проект устройства на языке высокого уровня VHDL, синтезировав его реализацию в FPGA, отладив и верифицировав результат синтеза, отработав тесты для спроектированного устройства, можно получить отлаженный проект, который затем может переноситься в реализацию на полузаказных или заказных СБИС. Отработка IP-блоков на FPGA является типовым этапом его проверки перед включением в проекты систем-на-кристалле.

Мы упомянули только несколько видов кристаллов СБИС. На самом деле различных вариантов значительно больше. Детальный разбор их видов и методов организации можно найти, например, в [28].

Библиотеки функциональных компонентов. В технологической цепочке проектирования СБИС (рис. 1.2) поуровневого синтеза реализации проектируемого устройства — от модели-спецификации на VHDL к логическому проекту, от логического проекта к схемотехническому проекту, от принципиальной схемы на уровне транзисторов к послойной топологии кристалла, автоматический синтез сосуществует с использованием на каждом из этих уровней библиотечных компонентов — несинтезируемых, разрабатываемых вручную. Библиотеки функциональных компонентов разного уровня — функциональных ячеек, макроячеек, IP-ядер — используются при разработке всех классов СБИС, и заказных СБИС, и полузаказных СБИС, и при реализации проектируемых устройств в FPGA.

Библиотека таких компонентов создает новый, высокоуровневый базис реализации. Мы можем, например, описать на VHDL и автоматически синтезировать в FPGA блок умножителя из базовых ячеек общего назначения. Однако такой блок будет иметь существенно худшие характеристики по за-

держкам и занимаемым ресурсам кристалла, чем библиотечный компонент, внутренняя организация и связи которого оптимизированы квалифицированными проектировщиками СБИС, с глубоки учетом специфики конкретного семейства СБИС. По сравнению же с "погруженными" в современные FPGA блоками умножителей — специализированными топологическими компонентами кристалла — разница будет составлять порядки величин.

Логический синтез реализации устройства

Типовые этапы системы разработки на СБИС при проектировании на языке VHDL представлены на рис. 1.3.

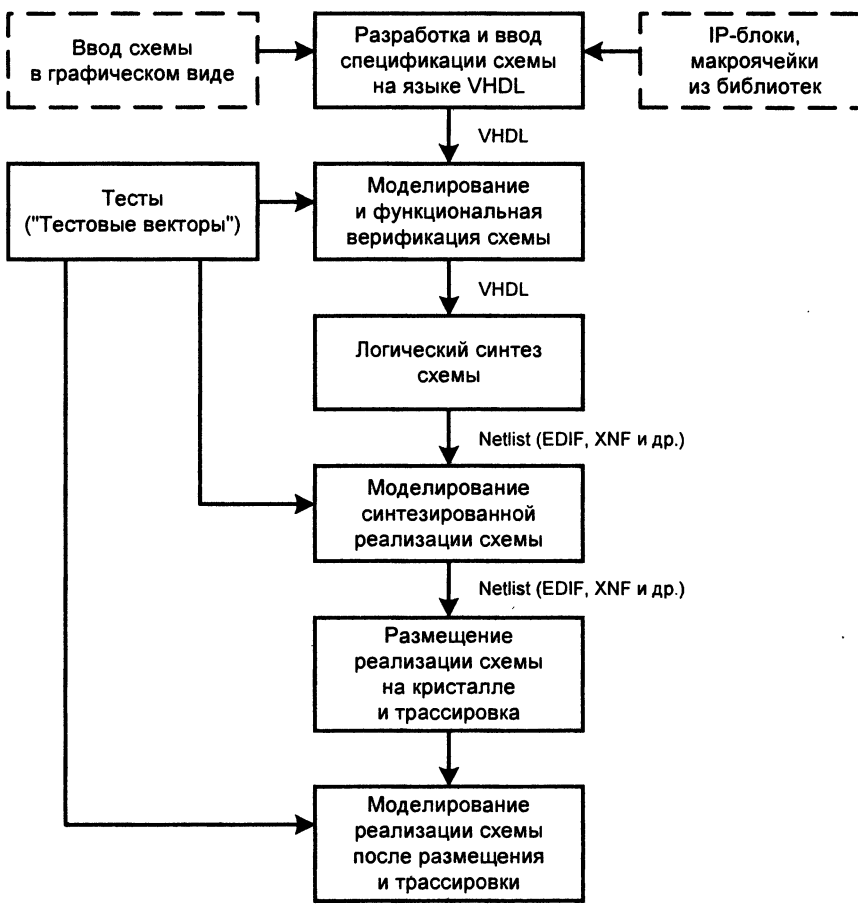


Рис. 1.3. Этапы разработки системы на СБИС при проектировании на языке VHDL

Синтез проектируемого устройства для его физической реализации в СБИС обычно выполняется поэтапно, несколькими компиляторами, синтезирующими программами, каждая из которых выполняет свой этап перевода с одного уровня представления проекта на другой (см. диаграмму Гайского-Канна, рис. 1.1). Предметом основного рассмотрения в данной книге является этап логического синтеза, с которым имеет дело разработчик устройств на СБИС. После этого этапа, дальнейшие стадии проектирования/синтеза выполняют уже другие специалисты — специалисты фирмы-изготовителя СБИС (рис. 1.2). Пакет Foundation Express, предназначенный для проектирования реализации устройств в FPGA, формирует реализацию устройства на вентильном уровне (gate level design), а также выполняет и ряд последующих этапов, но только для FPGA.

Разрабатываемый на языке VHDL проект устройства вводится, редактируется, проверяется. На этом этапе многие современные пакеты автоматизации проектирования позволяют вводить описание структуры проектируемого устройства (или его частей) и в альтернативной форме — в графическом виде, в виде структурных схем. Такие пакеты, как рассматриваемые в данной книге пакеты OrCAD Express и Foundation Express, автоматически транслируют графическое представление в структурное описание на языке VHDL и интегрируют его в общий проект на VHDL.

При формировании проекта устройства на VHDL также активно используются библиотеки описания компонентов.

Модель-спецификация проектируемого устройства на языке VHDL поступает на этап моделирования. На этом этапе производится функциональное моделирование проектируемого устройства, отладка проекта, формирование набора тестов и верификация на них устройства, спроектированного на языке высокого уровня.

Отлаженная модель-спецификация проектируемого устройства на языке VHDL поступает на этап Логического синтеза, где производится преобразование проекта устройства с функциональной модели на языке высокого уровня на уровень логического проекта, представление проектируемого устройства на вентильном уровне[1], т. е. на уровне, определяющем логическую схему устройства с учетом базиса реализации. На этом этапе выполняется и оптимизация логического проекта.

Список связей (Netlist). Логическое описание на вентильном уровне, формируемое этапом логического синтеза, представляется уже не на языке VHDL, а в другой форме, называемой списком связей.

Список связей (Netlist) является текстовой формой кодирования схемы, т. е. текстовым эквивалентом графического представления схемы, введенной раз-

[1] Об условности понятия «вентильное представление» мы говорили выше.

работчиком или синтезированной САПР по описанию на VHDL. Для представления списка связей используются как стандартные форматы текстового кодирования схемы, например, EDIF (Electronic Digital Interchange Format), так и различные специфические форматы производителей СБИС и САПР для них (например, формат XNF — Xilinx Netlist Format, используемый в пакете Xilinx Foundation Express, рассматриваемом в настоящей книге).

В качестве иллюстрации приведем вид описания логической схемы (рис. 1.4) в формате EDIF (рис. 1.5). Если бы ввели такую схему на этапе разработки проекта в графическом виде, то пакет OrCAD Express автоматически преобразовал бы ее в структурное описание на языке VHDL, показанное на рис. 1.6.

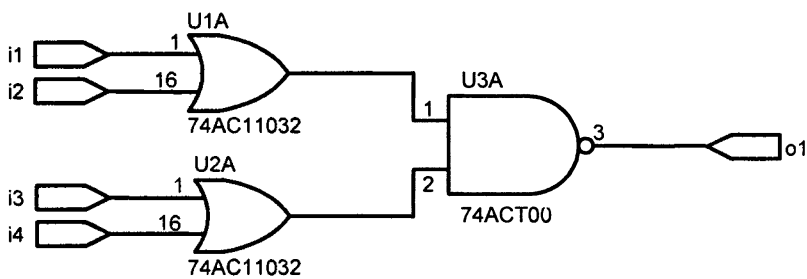


Рис. 1.4. Графическое представление схемы в OrCAD

Внимание

Комментарии к формату EDIF. В первой строке формата после ключевого слова `edif` указывается имя исходного объекта, для которого создавался список связей, затем указывается версия `edif`. И Foundation, и OrCAD используют одинаковую версию. Далее может идти статусная информация (время создания списка, программа, с помощью которой он был сгенерирован, путь к исходному файлу и др.).

Собственно описание включает в себя описание компонентов, входящих в состав схемы. Каждый компонент называется ячейкой (cell). Для каждого компонента указываются наименования и направления портов. В качестве отдельных компонентов рассматриваются входные и выходные порты, компоненты, с помощью которых реализуется собственно схема. Вся схема тоже рассматривается как компонент. Далее идет описание цепей (net). Для каждой линии связи указываются порты компонентов, с которыми она соединена. Имена линий связи генерируются автоматически (в OrCAD и Foundation они формируются по-разному).

```
edif SCHEMATIC1
(edifVersion 2 0 0)
(edifLevel 0)
(keywordMap (keywordLevel 0))
(status
 (written
  (timeStamp 2002 10 06 19 15 20)
  (program "CAPTURE.EXE"))
  (comment "Original data from
      OrCAD/CAPTURE schematic"))
 (external OrCAD_LIB
  (edifLevel 0)
  (technology
   (numberDefinition
    (scale 1 1 (unit distance))))
  (cell &74AC11032
   (cellType generic)
   (comment
   "From OrCAD library GATE.OLB")
(view NetlistView
 (viewType netlist)
 (interface
  (port &1 (direction INPUT))
  (port &16 (direction INPUT))
  (port &2 (direction OUTPUT))
  (port &13 (direction INPUT))
  (port &12 (direction INPUT))
  (port &4 (direction INPUT))
  (port &5 (direction INPUT))
  (port &15 (direction INPUT))
  (port &14 (direction INPUT))
  (port &3 (direction OUTPUT))
  (port &11 (direction INPUT))
  (port &10 (direction INPUT))
  (port &6 (direction OUTPUT))
  (port &9 (direction INPUT))
  (port &8 (direction INPUT))
  (port &7 (direction OUTPUT)))))
 (cell &74ACT00
   (cellType generic)
   (comment
   "From OrCAD library GATE.OLB")
   (view NetlistView
    (viewType netlist)
    (interface
     (port &1 (direction INPUT))
     (port &2 (direction INPUT))
     (port &3 (direction OUTPUT))
     (port &14 (direction INPUT))
     (port &7 (direction INPUT))
     (port &4 (direction INPUT))
     (port &5 (direction INPUT))
     (port &6 (direction OUTPUT))
     (port &9 (direction INPUT))
     (port &10 (direction INPUT))
     (port &8 (direction OUTPUT))
     (port &12 (direction INPUT))
     (port &13 (direction INPUT))
    (port &11 (direction OUTPUT))))))
(library MAIN_LIB
 (edifLevel 0)
 (technology
  (numberDefinition
   (scale 1 1 (unit distance))))
(cell SCHEMATIC1
 (cellType generic)
 (view NetlistView
  (viewType netlist)
  (interface
   (port i1 (direction INPUT))
   (port i2 (direction INPUT))
   (port i3 (direction INPUT))
   (port i4 (direction INPUT))
   (port o1 (direction OUTPUT)))
  (contents
   (instance U1
    (viewRef NetlistView
     (cellRef &74AC11032
      (libraryRef OrCAD_LIB))))
   (instance U2
    (viewRef NetlistView
     (cellRef &74AC11032
      (libraryRef OrCAD_LIB))))
   (instance U3
    (viewRef NetlistView
     (cellRef &74ACT00
      (libraryRef OrCAD_LIB))))
(net N00013
 (joined
 (portRef &1 (instanceRef U3))
 (portRef &2 (instanceRef U1))))
(net N00019
 (joined
 (portRef &2 (instanceRef U3))
 (portRef &2 (instanceRef U2))))
(net VCC
 (joined
 (portRef &12 (instanceRef U1))
 (portRef &12 (instanceRef U2))
 (portRef &13 (instanceRef U2))
 (portRef &14 (instanceRef U3))
 (portRef &13 (instanceRef U1))))
```

Рис. 1.5. Список связей в формате EDIF, сгенерированный в OrCAD (фрагмент)

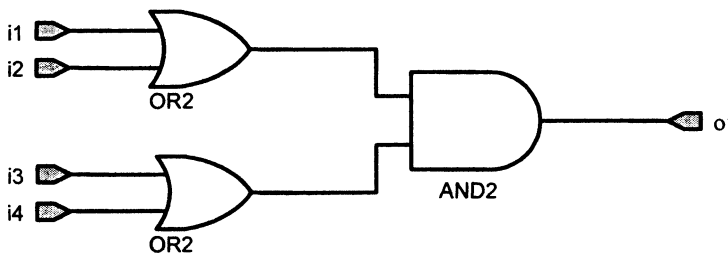
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

ENTITY SCHEMATIC1 IS PORT (
  i1 : IN std_logic;
  i2 : IN std_logic;
  i3 : IN std_logic;
  i4 : IN std_logic;
  o1 : OUT std_logic);
END SCHEMATIC1;
ARCHITECTURE STRUCTURE OF
SCHEMATIC1 IS
-- COMPONENTS
COMPONENT \74AC11032\
  PORT (
  I0_A : IN std_logic;
  I1_A : IN std_logic;
  O_A : OUT std_logic;
  VCC_A : IN std_logic;
  GND_A : IN std_logic;
  I0_B : IN std_logic;
  I1_B : IN std_logic;
  O_B : OUT std_logic;
  I0_C : IN std_logic;
  I1_C : IN std_logic;
  O_C : OUT std_logic;
  I0_D : IN std_logic;
  I1_D : IN std_logic;
  O_D : OUT std_logic );
  END COMPONENT;
COMPONENT \74ACT00\
  PORT (
  I0_A : IN std_logic;
  I1_A : IN std_logic;
  O_A : OUT std_logic;
  VCC : IN std_logic;
  GND : IN std_logic;
  I0_B : IN std_logic;
  I1_B : IN std_logic;
  O_B : OUT std_logic;
  I0_C : IN std_logic;
  I1_C : IN std_logic;
  O_C : OUT std_logic;
  I0_D : IN std_logic;
  I1_D : IN std_logic;
  O_D : OUT std_logic    );
END COMPONENT;
-- SIGNALS
SIGNAL N00013 : std_logic;
SIGNAL N00019 : std_logic;
SIGNAL VCC : std_logic;
SIGNAL GND : std_logic;
-- GATE INSTANCES
BEGIN
U1 : \74AC11032\  PORT MAP(
  I0_A => I1,
  I1_A => I2,
  O_A => N00013,
  VCC_A => VCC,
  GND_A => GND,
  I0_B => 'Z',
  I1_B => 'Z',
  O_B => OPEN,
  I0_C => 'Z',
  I1_C => 'Z',
  O_C => OPEN,
  I0_D => 'Z',
  I1_D => 'Z',
  O_D => OPEN);
U2 : \74AC11032\  PORT MAP(
  I0_A => I3,
  I1_A => I4,
  O_A => N00019,
  VCC_A => VCC,
  GND_A => GND,
  I0_B => 'Z',
  I1_B => 'Z',
  O_B => OPEN,
  I0_C => 'Z',
  I1_C => 'Z',
  O_C => OPEN,
  I0_D => 'Z',
  I1_D => 'Z',
  O_D => OPEN);
U3 : \74ACT00\  PORT MAP(
  I0_A => N00013,
  I1_A => N00019,
  O_A => O1,
  VCC => VCC,
  GND => GND,
  I0_B => 'Z',
  I1_B => 'Z',
  O_B => OPEN,
  I0_C => 'Z',
  I1_C => 'Z',
  O_C => OPEN,
  I0_D => 'Z',
  I1_D => 'Z',
  O_D => OPEN);
END STRUCTURE;
```

Рис. 1.6. Структурное описание на VHDL, сгенерированное OrCAD по схеме

В OrCAD и Foundation используются также различные библиотеки компонентов. Кроме того, в Foundation комбинационная схема сразу же отображается в LUT.

В пакете Foundation Express, кроме формата EDIF, используется и собственный формат XHF, общий для пакетов автоматизированного проектирования СБИС FPGA фирмы Xilinx, менее универсальный, но более компактный в представлении схем (рис. 1.7).



а

```
LCANET, 5
PROG, ACTIVE-CAD-2-XNF, 2.5.5.67, Mon Oct 07 14:21:37 2002
PART, V400-6-BG432
SYM, $I1, AND, SCHNM=AND2, LIBVER=2.0.0
PIN, I1, I, $Net00001_
PIN, I0, I, $Net00002_
PIN, O, O, O1
END
SYM, $I2, OR, SCHNM=OR2, LIBVER=2.0.0
PIN, O, O, $Net00001_
PIN, I1, I, I1
PIN, I0, I, I2
END
SYM, $I3, OR, SCHNM=OR2, LIBVER=2.0.0
PIN, O, O, $Net00002_
PIN, I1, I, I3
PIN, I0, I, I4
END
EXT, I1, I
EXT, I2, I
EXT, I3, I
EXT, I4, I
EXT, O1, O
EOF
```

б

Рис. 1.7. Списки связей в формате XNF: *а* — графическое представление схемы; *б* — список связей, сгенерированный Foundation Express

Форматы кодирования списков связей, хотя и имеют читаемый вид, не предназначены для чтения и ручного анализа пользователем, а ориентированы на формирование и использование пакетами автоматизации проектирования СБИС. Мы приводим их примеры для общего представления разработчика о том, что кроется за широко применяемыми терминами *список связей*, netlist, EDIF, XNF и др.

Модели в проектировании систем на СБИС

Высокая сложность современных СБИС и систем на их основе требует систематической методологии проектирования, сопровождаемой четкой спецификацией и документированием каждого этапа. Этапы можно представить следующим образом:

1. Спецификация требований к проектируемой системе и разработка абстрактной структуры, реализующей эти требования.
2. Декомпозиция каждого из компонентов этой абстрактной структуры в виде набора компонентов, связанных между собой и реализующих ту же функцию.
3. Декомпозиция уже этих компонентов, и так далее, до тех пор, пока не приведем структуру специфицируемого компонента к виду структуры из готовых, примитивных компонентов.

В результате получается иерархически организованная структура системы, которая позволяет совладать с растущей сложностью систем, проводить независимое проектирование компонентов и подсистем, с последующей их интеграцией в единую систему.

Такой подход к проектированию требует четкости и однозначности в спецификации системы, ее структуры, компонент и их функционирования. Решение этих задач базируется на разработке и использовании системы моделей проектируемого изделия и его компонентов.

Модель — это спецификация представления о системе, включающая все существенное и опускающая несущественные (на данном этапе) детали.

Можно указать основные группы задач в проектировании и использовании систем на СБИС, для решения которых используются модели разного уровня:

- *спецификация требований к системе* (полная, однозначная, непротиворечивая). Такая модель системы — это средство согласования и передачи информации между заказчиком системы и ее разработчиком;
- *спецификация системы для пользователя* (описание системы разработчиком для тех, кто будет ее использовать). Невозможно перечислить все сферы применения проектируемой системы словесно, с дополнительными рисунками и пояснениями, описать детальное ее поведение. Для

сложной системы никакое описание не будет исчерпывающим. Вместо этого, пользователю (дополнительно к общему описанию) дают *модель системы*, которую он может исследовать для получения ответов на неясные вопросы описания;

- *тестирование* (проверка методами эксперимента) *и верификация* (проверка по формальным методикам) проекта в ходе иерархического проектирования, а затем — в ходе производства системы. Важнейшей составляющей методологии проектирования является разработка и отработка тестов системы на модели для их применения при тестировании изготовленной системы. Тесты — это существенная часть (половина) результатов проектирования СБИС, передаваемых в изготовление;
- *автоматический синтез реализации системы* по ее описанию—модели, автоматическая генерация.

В процессе проектирования цифровой системы разработчик проходит целый ряд шагов, каждый из которых связан с формированием и анализом некоторого уровня модели проектируемого устройства [8]. Сначала разрабатывается функциональная (или, как часто говорят, поведенческая — behavioural спецификация проектируемого устройства), определяющая цели и правила его функционирования, интерфейс с окружением, в котором должно работать это устройство. Затем определяется общая архитектура устройства, отвечающая специфицированным требованиям к функционированию и интерфейсу. Здесь устройство представляется, как обобщенная структура, структурная схема из крупных блоков, часто соответствующих скорее концептуальной декомпозиции проектируемого изделия, чем продуманному разбиению на раздельно реализуемые физические блоки конечной схемы реализации. Каждый из блоков такой обобщенной структуры специфицируется своим интерфейсом и функциональным описанием, поведенческой моделью. Этот процесс может рекурсивно развертываться вглубь, насколько это окажется целесообразно на данной стадии проектирования.

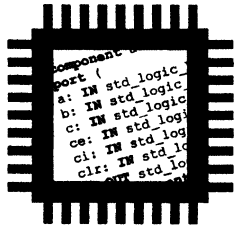
После отработки поведенческой спецификации проектируемого устройства, переходят к поэтапному синтезу устройства, отвечающего этой спецификации. Под *синтезом* обычно понимают процесс перехода от поведенческой спецификации устройства (или его компонента) к его структурному представлению, представлению как структуры из компонент меньшего уровня абстракции [8]. Если этими компонентами будут базовые, "библиотечные" компоненты, из которых и будет, в конечном счете, составляться реализация проектируемого устройства, то синтез выполнен. Если в составе сформированной структуры есть компоненты, не входящие в используемый перечень базовых, то процесс синтеза рекурсивно разворачивается вглубь [12].

Язык VHDL дает все необходимые средства для рекурсивного развертывания процессов спецификации, моделирования и анализа проектируемого устройства при иерархической декомпозиции, с использованием комбинации структурного и поведенческого описания проектируемого устройства и

его составляющих. VHDL исходно создавался и как язык для моделей-спецификаций, и как язык модели как средства представления и исследования, анализа поведения, функционирования проектируемого устройства [3]. VHDL предоставляет средства для описания функционирования и структуры проектируемой системы на СБИС на различных уровнях представления — от наиболее абстрактных, наиболее общих, до детального представления структуры устройства на вентильном уровне.

Как язык описания и моделирования цифровых систем, VHDL имеет существенные отличия от традиционных процедурных языков программирования. VHDL включает множество средств и понятий, встроенных в язык при его определении, предназначенных для моделирования цифровых систем, связаных с поведенческими, структурными, физическими свойствами этих систем: сигналы, время, параллельно функционирующие компоненты структуры, и др. [20].

Рассматривая проектирование систем на языке VHDL в нашей книге, мы постарались показать многообразие форм его применения в моделях цифровых систем на разных стадиях, на разных уровнях разработки систем — от моделей-спецификаций стандартов на интерфейсы цифровых систем, поведенческих моделей для функциональной верификации разрабатываемого устройства, до моделей-спецификаций для синтеза реализации устройства в заданном базисе.



Глава 2

Операторы и данные языка VHDL

Базовые элементы языка VHDL

Лексические элементы

При описании синтаксиса языка VHDL используются следующие правила:

- () — заключают в себе последовательность элементов, любой из которых может присутствовать в определении;
- | — отделяет отдельные элементы списка, любой из которых может присутствовать в конкретном определении;
- {} — заключают в себе последовательность элементов, которые не обязательно должны присутствовать (это, как правило, элементы, аналогичные предшествующим);
- жирным шрифтом выделяются зарезервированные слова.

В языке VHDL присутствуют следующие лексические элементы:

- *комментарии.* Часть строки после двух знаков "--" рассматривается как комментарий. Например:

```
I:=5 --переменной I присваивается значение 5
```

Внимание

Несмотря на то, что стандарт VHDL предусматривает использование кириллицы в комментариях, многие программы моделирования этого не поддерживают.

- *идентификаторы.* (Могут содержать символы алфавита от A до Z и от a до z, цифры, символы подчеркивания _). Идентификатор должен начинаться с символа алфавита, не должен заканчиваться символом подчеркивания. не должен содержать несколько символов подчеркивания подряд;

- *расширенные идентификаторы.* (Могут включать в себя любую последовательность символов, заключенную в / /. Например, /7004/ является допустимым идентификатором;
- *числа.* (Могут использоваться целые и действительные числа). 10, 0, 34 — целые десятичные числа. Число -10 рассматривается не как отрицательное целое, а как комбинация оператора и целого числа. Действительные числа могут записываться в экспоненциальной форме (записи 23.1, 231E-1, 2.31e 1 являются эквивалентными). Целые и действительные числа могут записываться по основанию, отличному от 10. Для оснований от 10 до 15 при записи цифр, больших 9, используются символы A—F (или a—f). Например:

```
2#1111101#  16#FD#    8#0.4#    16#4#E2
```

  Для удобочитаемости больших чисел между отдельными знаками могут включаться символы подчеркивания;
- *строки.* Содержимое строки заключается в двойные кавычки. Если внутри строки необходимо использовать двойные кавычки, то они удваиваются. Например:

```
"A string is a string: ""string"". "
```

  VHDL поддерживает битовые строки, строки восьмеричных и шестнадцатеричных символов. Перед такими строками соответственно ставятся символы: b, o, x.

  Например:

  - x"1101" рассматривается как шестнадцатеричное число (4353 — его десятичное представление);
  - b"1101" рассматривается как двоичное число (13 — его десятичное представление).

Описания констант, переменных, сигналов

Описание констант, переменных и сигналов имеет сходную структуру. В начале строки описания указывается ключевое слово, определяющее вид описываемого объекта, затем указывается одно или несколько имен описываемых объектов и их тип. Для констант должно быть указано значение, для переменных и сигналов указание начального значения возможно, но не обязательно.

Описание *константы* имеет следующий синтаксис:

```
constant name {,...}:subtype_indication := expression
```

Например:

```
constant address_length: integer:=4;
```

Описание *переменной* имеет следующий вид:

```
variable name{,...}: subtype_indication[:=expression];
```

Например:

```
variable counter: integer:=7;
```

Присваивание нового значения переменной:

```
[label:]name:=expression;
```

Например:

```
counter:=counter+1;
```

Описание *сигналов* имеет следующий синтаксис:

```
signal identifier {,...}:subtype_indication [:=expression];
```

Например:

```
signal flag1:bit;
signal flag2:bit:='1';
```

Внимание

В Foundation Express *(см. главу 7)* присваивание начальных значений переменным и сигналам игнорируется

Типы данных

Язык VHDL имеет разнообразный набор типов данных, некоторые из которых (такие как, например, числовые типы) привычны для специалистов, программирующих на алгоритмических языках высокого уровня. Другие, Физические типы имеют весьма специфический характер, работа с ними требует не всегда привычных для программиста методов формирования и использования значений данных. Их понимание и корректное использование должно базироваться не только на знании синтаксиса программных конструкций и чисто программных вопросов их семантики, но и на представлении о физических процессах (хотя бы на уровне логических сигналов и временных соотношений, которые происходят в аппаратуре проектируемых устройств).

Структура отношений между типами данных представлена на рис. 2.1 [3].

В данной главе мы рассмотрим все типы, кроме файлового.

Описание *типа* имеет следующий синтаксис:

```
type name is type_definition
```

Внимание

Если несколько имен типов имеют одинаковое описание, переменные этих типов не считаются принадлежащими к одному типу.

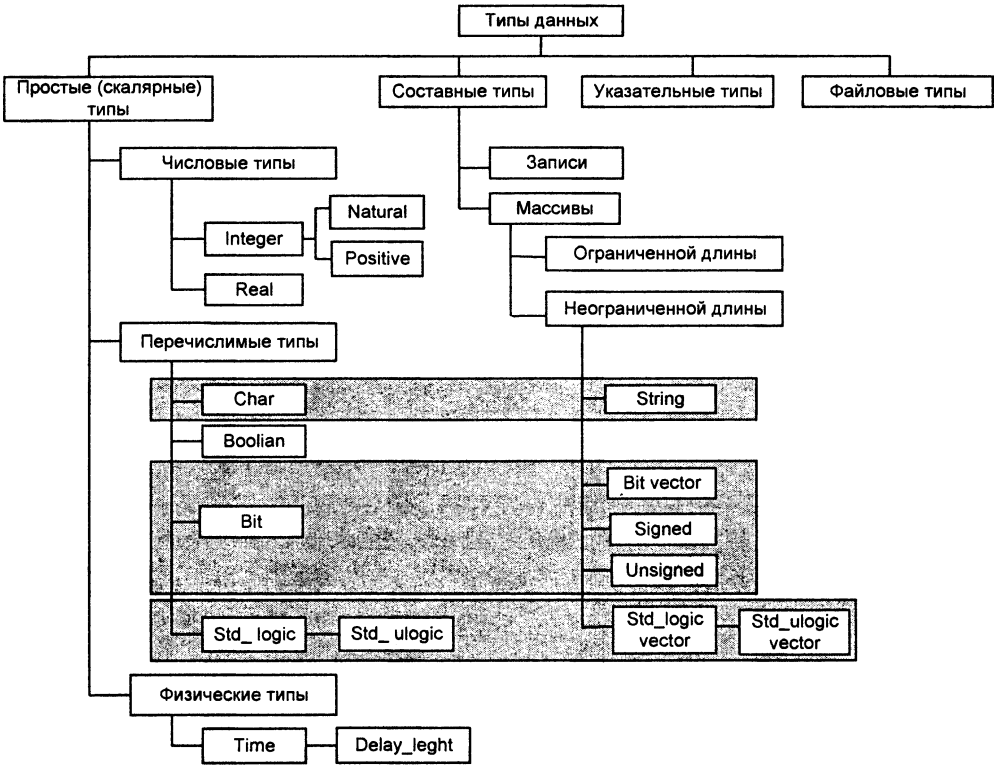


Рис. 2.1. Типы данных языка VHDL

Скалярные типы данных

Скалярные типы могут использоваться для описания следующих групп объектов: чисел, символов, значений сигналов и других физических объектов. При моделировании схем устройств на различных уровнях абстракции, в языке VHDL для описания сигналов могут использоваться различные типы данных и их сочетания. Так, на высоких уровнях абстракции, как правило, используются перечислимые типы, задаваемые списком значений, и числовые типы. А при моделировании на уровне регистровых передач, как правило, используется булевский тип, битовый тип, стандартный логический тип, а также типы, которые определяются на их базе.

Числовые типы

В VHDL поддерживается два основных типа для представления чисел: целочисленный и тип с плавающей запятой. Вследствие того, что компиляторы

языка VHDL существуют, в основном, в рамках систем автоматизированного проектирования, тип с плавающей запятой большинством из них или вообще не поддерживается или поддерживается не в полной мере. Это относится и к рассматриваемым в данной книге средам OrCAD Express и Foundation Express.

Тип *Integer*

Для представления целых чисел используется тип `Integer`. Этот тип позволяет представить числа в диапазоне от −2 147 483 647 до 2 147 483 647.

В среде Foundation Express этот тип автоматически преобразуется в битовый вектор, диапазон которого определяется как минимально допустимый для заданного диапазона значений, поэтому целесообразно использовать не тип `Integer` как таковой, а диапазон, заданный на базе этого типа.

Внутри среды Foundation Express целые числа представляются в двоичном формате. Для работы с ними используется тип `bit_vector`. Преобразование целых чисел осуществляется автоматически, без участия пользователя. Размер битового вектора определяется как минимально допустимый для заданного диапазона значений исходного целого типа. При просмотре результатов моделирования так же автоматически осуществляется обратное преобразование. В результате значения сигналов, которые имеют целый тип, можно просматривать в десятичном формате.

Описание типа на базе `Integer` имеет следующий вид:

```
range simple_expression (to|downto) simple_expression
```

Например:

```
type t1 is range 1 to 200;
type t2 is range 40 downto 5;
```

Когда переменная описывается таким типом, то по умолчанию ее начальное значение определяется равным левой границе интервала. Операции, выполнимые над множеством целых типов, приведены в табл. 2.1.

Таблица 2.1. *Операции над множеством целых типов*

Обозначение	Описание
+	Сложение
-	Вычитание
*	Умножение
/	Деление
mod	Деление по модулю

Таблица 2.1 (окончание)

Обозначение	Описание
Rem	Остаток от деления
abs	Модуль
**	Возведение в степень

Тип *Real*

Для представления действительных чисел используется тип `Real`. Он имеет диапазон от −1.0E+38 до 1.0E+38.

Физические типы данных

Для представления физических величин (таких как длина, масса, время), в языке VHDL используются так называемые *физические типы*. Данные, принадлежащие к физическому типу, определяются своим значением и единицей измерения. Для одного и того же физического параметра может использоваться множество единиц измерения.

Организация физических типов в VHDL позволяет установить соответствие между различными единицами измерения. Это освобождает пользователя от последующего написания множества функций преобразования данных из одних единиц измерения в другие.

Внимание

В Foundation Express физические типы не поддерживаются.

Определение физического типа включает в себя *первичный модуль*, в котором определяется основная единица измерения. Оно также может включать в себя *вторичные модули*, в которых определяются дополнительные единицы измерения и их связь с основной единицей.

Определение имеет следующий синтаксис:

```
type name is range simple_expression (to|downto) simple_expression
units
  ed;
  ked=X ed;
  ...
end units name;
```

Здесь:

- `ed` — имя первичного модуля;
- `ked` — имя вторичного модуля.

Например:

```
type length is range 0 to 1E9
units
  um;   -- micron
  mm=1000 um;
  m=1000 mm;
  mil=254 um;
  inch=1000 mil;
end units length;
```

Внимание

В OrCAD Express 9.1 после `end units` нельзя указывать имя типа, должна сразу же ставиться точка с запятой.

Значения величины этого типа можно записать следующим образом: `1 mm`, `200 mil`.

Величина, описанная типом `length`, может принимать значения от `0 um` до `1E9 um`. Для задания значения величины может использоваться и действительное число. Оно будет округлено до ближайшего целого в основных единицах измерения. Так, для типа `length` запись `0.1 inch`, `2.54 mm` и `2.540528 mm` будут интерпретированы как одно и тоже число `2540 um`.

Например, описание переменной с использованием этого типа может выглядеть следующим образом:

```
Variable leng1: length:=13 um;
```

В этом примере переменной `leng1` присваивается начальное значение 13 мкм.

Операции над физическими типами

К физическим типам может быть применено большинство арифметических операторов, но с некоторыми ограничениями:

- операции сложения и вычитания могут производиться только над данными одного и того же типа; при этом получается результат того же типа, что и операнды;
- значение физического типа может быть умножено на целое или действительное число; результату будет присвоен физический тип операнда.
- значение физического типа можно делить на целое или действительное число; будет получен результат того же физического типа;
- для операции деления оба операнда также могут быть одного и того же физического типа, в этом случае будет получен результат целого типа.

Например:

```
Leng1:=leng1 + 5 um;
```

Описание времени

VHDL поддерживает встроенный физический тип `time` для описания времени. Описание этого типа выглядит следующим образом:

```
type time is range implementation defined
units
  fs;
  ps=1000 fs;
  ns=1000 ps;
  us=1000 ns;
  ms=1000 us;
  sec=1000 ms;
  min=60 sec;
  hr=60 min;
end units;
```

При моделировании, `fs` является минимальным временем, которое может быть учтено. Однако для того, чтобы расширить диапазон значений времени, в качестве минимальной единицы может быть выбран один из больших модулей.

Приведенное выше описание типа `time` дано для пояснения его смысла. Поскольку тип `time` является встроенным, описывать его в своей программе не надо.

Перечислимые типы данных

Перечислимые типы могут задаваться двумя способами: диапазоном и списком значений.

Описание типа с использованием диапазона

Это описание имеет следующий синтаксис:

```
range simple_expression (to|downto) simple_expression
```

Примером такого описания могут служить рассмотренные выше типы на базе целочисленного.

Описание типа с использованием списка значений

При моделировании на абстрактном уровне значениям сигналов удобно сопоставлять *имена*, отражающие их смысл. В этом случае используется зада-

ние перечислимого типа списком значений. Описание типа выглядит следующим образом:

```
type name is (value1, value2,...,value_n)
```

Например:

```
type states is (enable, disable, idle);
type octal_digits is ('1', '2', '3', '4', '5', '6', '7');
```

Описания различных типов могут включать одни и те же имена (так называемая *перегрузка* имен значений). Например, допустимо следующее описание:

```
type states is (enable, disable, idle);
type sign1 is (enable, disable);
```

При присваивании объектам перегруженных значений, в языке VHDL, для облегчения читаемости, может использоваться следующая конструкция:

```
type_name'(value);
```

Например:

```
states'(enable)
sign1'(enable)
```

Символьный тип. Этот тип является перечислимым типом, задаваемым списком входящих в него значений. VHDL поддерживает весь набор восьмибитовых символов ISO. Определение типа имеет следующий вид:

```
type character is (
    nul, soh, stx, etx, eot, enq, ack, bel,
    bs,  ht,  lf,  vt,  ff,  cr,  so,  si,
    dle, dc1, dc2, dc3, dc4, nak, syn, etb,
    can, em,  sub, esc, fsp, gsp, rsp, usp,
    ' ', '!', '"', '#', '$', '%', '&', ''',
    '(', ')', '*', '+', ',', '-', '.', '/',
    '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', ':', ';', '<', '=', '>', '?',
    '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
    'X', 'Y', 'Z', '[', '\', ']', '^', '_',
    '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
    'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
    'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
    'x', 'y', 'z', '{', '|', '}', '~', del,
    c128, c129, c130, c131, c132, c133, c134, c135,
```

```
c136, c137, c138, c139, c140, c141, c142, c143,
c144, c145, c146, c147, c148, c149, c150, c151,
c152, c153, c154, c155, c156, c157, c158, c159,
' ', 'Ў', 'ў', 'Ј', '¤', 'Ґ', '¦', '§',
'Ё', '(c)', 'Є', '«', '¬', '-', '(r)', 'Ї',
'°', '±', 'І', 'і', 'ґ', 'µ', '¶', '·',
'ё', '№', 'є', '»', 'ј', 'Ѕ', 'ѕ', 'ї',
'А', 'Б', 'В', 'Г', 'Д', 'Е', 'Ж', 'З',
'И', 'Й', 'К', 'Л', 'М', 'Н', 'О', 'П',
'Р', 'С', 'Т', 'У', 'Ф', 'Х', 'Ц', 'Ч',
'Ш', 'Щ', 'Ъ', 'Ы', 'Ь', 'Э', 'Ю', 'Я',
'а', 'б', 'в', 'г', 'д', 'е', 'ж', 'з',
'и', 'й', 'к', 'л', 'м', 'н', 'о', 'п',
'р', 'с', 'т', 'у', 'ф', 'х', 'ц', 'ч',
'ш', 'щ', 'ъ', 'ы', 'ь', 'э', 'ю', 'я');
```

Булевский тип. Это перечислимый тип, задаваемый множеством своих значений. Он имеет следующее определение:

```
type boolean is (false, true);
```

Этот тип используется для представления результатов условных выражений. В состав этих выражений могут входить операторы сравнения =, /=, <, <=, >, >=, и логические операторы `and`, `or`, `nand`, `xor`, `xnor`, `not`. Операторы <, <=, >, >= могут применяться только к значениям упорядоченных типов.

Битовый тип. Это перечислимый тип, задаваемый множеством своих значений. Он имеет следующее определение:

```
type bit is ('0', '1');
```

Значения `'0'`, `'1'` являются перегруженными (они входят и в битовый, и в символьный типы). Логические операторы применимы только к битовому типу. Если логический оператор имеет два операнда, то они должны быть одного типа (выражение типа: `'0' and true` – не допустимо).

Как правило, битовый тип используется для моделирования на аппаратном уровне, а булевский тип — для моделирования на абстрактном уровне. Применение логических операторов к битовым значениям осуществляется в терминах позитивной логики (на абстрактном уроне `'0'` соответствует `false`, а `'1'` — `true`).

Стандартный логический тип std_ulogic. Это перечислимый тип, задаваемый множеством своих значений. Он используется для представления возможных значений электрического сигнала в соответствии со стандартом IEEE 1164.

Его определение имеет следующий вид:

```
type std_ulogic is ('U'   -- uninitialised
                    'X'   -- forcing unknown
                    '0'   -- forcing zero
                    '1'   -- forcing one
                    'W'   -- Weak unknown
                    'L'   -- Weak zero
                    'H'   -- Weak one
                    '-'); -- Don't care
```

Величины этого типа могут быть операндами логических операторов.

Для того чтобы включить тип `std_ulogic` в модель, необходимо перед ней разместить следующие строки:

```
library ieee;
use ieee.std_logic_1164.all;
```

Это связано с тем, что тип не является встроенным в VHDL, а расположен в отдельном пакете, находящемся в отдельной библиотеке ieee.

Атрибуты скалярных типов

Тип определяет множество значений и множество применимых к ним операций. При определении типа попутно определяется также множество *атрибутов типа*, которое позволяет получить информацию о диапазоне значений, входящих в этот тип. Обращение к атрибутам имеет следующий синтаксис:

```
type_name'attribute
```

Перечень атрибутов приведен в табл. 2.2. В ней `T` — имя объекта или имя типа.

Таблица 2.2. *Перечень атрибутов скалярных типов*

Атрибут	Описание
`T'left`	Первое значение в типе (левая граница интервала)
`T'right`	Последнее значение в типе (правая граница интервала)
`T'low`	Наименьшее значение в типе
`T'high`	Наибольшее значение в типе
`T'ascending`	`true`, если диапазон задан от меньшего к большему или этот тип определяется перечислением входящих в него значений, в противном случае — `false`
`T'image(x)`	Возвращает строковое представление значения `x`
`T'value(s)`	Возвращает значение указанного типа на базе строкового представления `s`

Например:

```
type index_range is range 30 downto 2;
index_range'left=30;
index_range'right=2;
index_range'low=2;
index_range_high=30;
index_range'ascending=false;
index_range'image(14)="14";

type logic_level is (unknown, low, undriv, high);
logic_level'left=unknown;
logic_level'right=high;
logic_level'low=unknown;
logic_level'hight=high;
logic_level'asccending=true;
logic_level'image(undriv)="undriv";
logic_level'value("low")=low;
```

Внимание

Атрибуты `image` и `value` в OrCAD Express не поддерживаются.

В табл. 2.3 приведены атрибуты, присущие физическим типам и перечислимым типам, `T` — имя типа.

Таблица 2.3. *Атрибуты физических типов*

Атрибут	Описание
`T'pos(x)`	Номер позиции `x` в `T`
`T'val(n)`	Значение в `T` на позиции `n`
`T'succ(x)`	Значение в `T` на позиции, на 1 большей, чем `x`
`T'pred(x)`	Значение в `T` на позиции, на 1 меньшей, чем `x`
`T'leftof(x)`	Значение в `T` на позиции, на 1 левее, чем `x`
`T'rightof(x)`	Значение в `T` на позиции, на 1 правее, чем `x`

Например:

```
logic_level'pos(unknown)=0;
logic_level'val(3)=high;
```

```
logic_level'succ(unknown)=low;
logic_level'pred(undriven)=low;
```

Если тип имеет возрастающий диапазон, то `T'succ(x) = T'rightof(x)` и `T'pred(x) = T'leftof(x)`, в противном случае `T'succ(x) = T'leftof(x)` и `T'pred(x) = T'rightof(x)`.

Для целых типов номер позиции совпадает со стоящим на нем значением, а тип номера позиции называется `universal integer`. Для физических типов номер позиции определяет значение объекта, выраженное в базовых единицах измерения. Для физических типов номер позиции совпадает со стоящим на нем значением, выраженным в базовых единицах измерения.

Например:

```
time'pos(4 ns)=4000000
```

Поскольку операция умножения двух объектов физического типа выполняться не может, можно перемножать их номера позиций в типе, а затем преобразовывать результат в нужный тип. Например:

```
type length is range integer'low to integer'high
  units  mm
end units length;
type area is range integer'low to integer'high
  units  sq_mm
end units area;
```

Например, пусть имеются следующие описания типов: `L1`, `L2` — объекты типа `length`; `S` — объект типа `area`.

Выражение

```
A:=L1*L2
```

приведет к ошибке.

Допустимое выражение имеет следующий вид:

```
A:=area'val(length'pos(L1)* length'pos(L1));
```

Определения типов могут меняться в ходе проектирования. При этом в некоторых случаях использование атрибутов позволяет избежать редактирования модели.

Например, в ходе разработки конечного автомата набор его состояний может меняться. Но разработчики могут договориться о том, чтобы в описании перечислимого типа состояние, соответствующее ошибке, при любых изменениях стояло последним в списке состояний. Тогда, независимо от его имени, обращаться к нему можно будет с использованием атрибута `right`.

Преобразование скалярных типов

Для выполнения ряда операций требуется, чтобы операнды были одного и того же типа. Преобразование типов integer в real и наоборот, осуществляется следующим образом:

```
real(integer_value); integer(real_value)
```

При преобразовании действительного числа к целому производится округление до ближайшего целого.

Для преобразования данных из одного перечислимого типа в другой, могут использоваться атрибуты pos и val. Например:

```
Variable i2:bit;
i2:=bit'val(boolean'pos(true));
```

Атрибут image может быть использован при сохранении результатов работы в файле, а атрибут value – при чтении данных из файла.

Подтипы

Нередко модель содержит объекты, которые могут принимать значения только в строго ограниченной области из множества возможных значений какого-либо типа. Тип таких объектов можно определить на базе основного типа. Он будет называться *подтипом* этого типа. Использование подтипов облегчает понимание модели.

Структура описания подтипа имеет следующий вид:

```
subtype name_of_subtype is name_of_base_type range simple_expression
(to|downto) simple_expression;
```

Например:

```
subtype short is integer range 0 to 255;
```

Все операции, применимые к объектам базового типа, применимы и к объектам его подтипов. В таких операциях как сложение, вычитание, умножение, деление и т. д. могут смешиваться операнды базового типа и его подтипов. Однако при попытке присваивания переменной значения, выходящего за рамки, определенные ее подтипом, будет возникать ошибка.

Допустима ситуация, когда базовый тип имеет диапазон, заданный в одном направлении (например, от меньшего к большему), а подтип имеет диапазон, определенный в противоположном направлении (от большего к меньшему).

Встроенные подтипы

В VHDL имеются следующие встроенные подтипы:

```
subtype natural is integer range 0 to biggest_integer;
subtype positive is integer range 1 to biggest_integer;
```

Такие подтипы используются для объектов, которые, согласно работе модели, не должны принимать отрицательные значения. Это позволяет отслеживать ошибки. Для тех же целей используется физический подтип времени:

```
subtype delay_length is time range 0 fs to biggest_time;
```

Стандартный логический тип `std_ulogic` также имеет стандартный подтип `std_logic`, который имеет то же самое множество значений, что и `std_ulogic`, но для него по-другому определена решающая функция. (Это будет рассматриваться в *главе 3*).

Для подтипов могут использоваться все атрибуты, применяемые для их базовых типов, а также один добавочный, позволяющий программе определить базовый тип для этого подтипа: `T'base`. Этот атрибут может использоваться в качестве префикса для других атрибутов. Например:

```
type opcode is (nop, load, store, add, substract,branch);
subtype sm_op is opcode range load to substract;
sm_op'base'left=nop;
sm_op'base'succ(substract)=branch;
```

Составные типы данных

Массивы

Массив представляет собой набор элементов одного и того же типа. Позиция каждого элемента задается скалярным значением — *индексом*. Определение типа одномерного массива имеет следующий вид:

```
type name is array (diskrete_subtype_indication range
  simple_expression to|downto simple_expression) of
  element_sbtype_indication;
```

Например:

```
type word is array (0 to 31) of bit;
type state is (initial, idle, active, error);
type state_count is array (state range idle to error) of natural;
subtype coeff is integer range 0 to 63;
type coeff_array is array (coeff) of real;
```

Обращение к элементу массива происходит по его индексу, который стоит в круглых скобках.

Внимание

В Foundation Express индекс может принадлежать только типу, определенному на базе `integer`.

Определение типа многомерного массива отличается от определения одномерного тем, что количество определений индексов равно размерности массива. Определения индексов отделяются друг от друга запятыми. Например:

```
type matrix is array (1 to 3, 1 to 3) of integer
```

Обращение к элементу многомерного массива осуществляется указанием всех его индексов, заключенных в круглые скобки, через запятую. Например:

```
variable matrix1: matrix;
Matrix1(1,3):=15;
```

В результате элементу массива с индексами 1, 3 будет присвоено значение 15.

Внимание

В Foundation Express многомерные массивы в явном виде (с указанием в скобках нескольких диапазонов через запятую) не поддерживаются, но можно описать массив, элементы которого, в свою очередь, будут массивами.

Например, описание рассмотренного выше массива `matrix` в Foundation Express может выглядеть следующим образом:

```
type ell_matrix is array (1 to 3) of integer;
type matrix is array (1 to 3) of ell_matrix;
```

Обращение к элементу такого массива может иметь следующий вид:

```
Matrix1(1)(3):=15;
```

Определение начальных значений

Начальные значения объектам типа массив могут задаваться тремя следующими способами [3].

Прямое определение. Начальные значения элементов могут задаваться списком, в котором первое значение соответствует первому элементу массива, а каждое последующее — последующему элементу. Таким образом, номер определяемого элемента в массиве ассоциируется с позицией значения в списке начальных значений.

Например:

```
type points is array (1 to 3) of real;
```

объекту этого типа можно присвоить значения следующим списком:

```
(2.3, 3.2, 5.0)
```

Для многомерного массива список состоит из подсписков, каждый из которых заключен в круглые скобки и отделен от других запятыми. Для двумерных массивов каждый подсписок включает в себя элементы, которые соответствуют одному и тому же значению первого индекса, определяемого номером подсписка в списке. Если массив трехмерный, то список для задания его значений состоит из подсписков, формируемых так же, как список для двумерного массива, и т. д.

Пример:

```
type points is array (1 to 3, 1 to 4) of real;
```

объекту этого типа можно присвоить значения следующим списком:

```
((2.3, 3.2, 5.0,7.0), (2.1, 6.2, 5.1,7.9), (6.3, 3.4, 1.0, 8.1))
```

Ассоциирование. Выполняется так называемое ассоциирование значения индекса со значением элемента, т. е. сопоставление элементу с указанным индексом заданного значения элемента. Это определение имеет следующий синтаксис:

```
({index_values=>element_value},...);
```

где `index_value` — может представлять:

- простое выражение;
- интервал (в этом случае все элементы, индексы которых попадают в этот интервал, будут иметь одно и то же значение);
- список значений индексов;
- **others** (в этом случае все элементы, индексы которых не вошли ни в одно из предыдущих определений, будут иметь указанное значение). **Others** должно быть последним в списке.

Если задается список значений индексов, то его элементы отделяются друг от друга вертикальной чертой. Например:

```
type symbol is ('a', 't', 'd', 'h', cr);
type state is range 0 to 2;
type matrix is array(state,symbol) of state;
constant n_state:matrix:=(0=>('a'=>1, 'd'=> 2, others =>0),
                          1=>('a'|'h'=>2, others=>0),
                          2=>('a' to 'h'=>1, others=>2));
```

В результате матрица будет иметь следующий вид:

```
1   0   2   0   0
2   0   0   2   0
1   1   1   1   2
```

Агрегаты. Могут использоваться для задания значений нескольким переменным или сигналам одновременно. Это определение имеет следующий синтаксис:

```
(name1, name2, ..., name_n)<=name_aggregate
```

Например, пусть имеются четыре переменных типа `bit` и переменная типа `bit_vector` длины 4. Для того чтобы присвоить переменным значения элементов этого вектора, можно выполнить следующее действие:

```
(z_flag, n_flag, v_flag, c_flag)<=flag_reg;
```

Атрибуты данных типа "массив"

Язык VHDL поддерживает набор атрибутов, применимых к объектам типа массив и позволяющих получить информацию о типе. Для обращения к значениям этих атрибутов используется тот же синтаксис, что и для обращения к атрибутам неструктурированных типов. Перечень атрибутов приведен в табл. 2.5 (A — имя типа массив или объекта массив).

Таблица 2.5. *Перечень атрибутов типа массив*

Имя атрибута	Описание
`A'left(N)`	Левая граница массива по указанному измерению N
`A'right(N)`	Правая граница массива по указанному измерению N
`A'low(N)`	Нижняя граница массива по указанному измерению N
`A'hight(N)`	Верхняя граница массива по указанному измерению N
`A'range(N)`	Интервал индекса массива по указанному измерению N
`A'reverse_range(N)`	Интервал индекса массива по указанному измерению N в обратном порядке
`A'length(N)`	Длина интервала индекса массива по указанному измерению N
`A'ascending(N)`	Принимает значение `true`, если интервал индекса массива задан от меньшего к большему

Например:

```
type A is array(1 to 4, 31 downto 0) of boolean;
A'left(1)=1;
A'low(1)=1;
A'right(2)=0;
A'hight(2)=31;
A'range(1) is 1 to 4;
A'range(2) is 0 to 31;
A'length(1)=4;
A'length(2)=32;
A'ascending(1)=true;
A'ascending(2)=false;
```

Если номер измерения не указан, то по умолчанию он принимается равным 1.

Массивы неограниченной длины

Для массивов неограниченной длины не задается конкретный диапазон индекса, а только его тип. Описание таких типов имеет следующий синтаксис:

```
type name is array (name_of_type_of_index range <>) of element_type;
```

Например:

```
type sample is array (natural range <>) of integer;
```

Когда описывается объект такого типа, то необходимо задать конкретное значение интервала для индекса, например:

```
variable sh_sam: sample(0 to 63);
```

На базе типа, описанного таким образом, можно определять подтипы, в которых также должно быть задано конкретное значение интервала для индекса.

Если таким типом описывается константа, то это делается на базе агрегатного описания. В таком описании могут быть просто перечислены значения элементов (тогда они присваиваются элементам с номерами, начиная с наименьшего значения индекса, в соответствии с типом, по порядку, а их количество определяет размер интервала индекса), или же могут быть номера элементов, которым присваиваются значения, а в агрегате задаются напрямую.

Например:

```
constant cc_dam:=(127,63,0,-63);
constant dd_sam:=(1=>23, 3=>-4, 2=>100);
```

Битовые векторы. Для представления битовых векторов язык VHDL имеет встроенный тип — массив неограниченной длины с битовым типом элементов и положительным типом индекса:

```
type bit_vector is array (positive range <>) of bit;
```

Кроме того, в стандартном пакете numeric_std *(см. приложение 4)* аналогичным образом определены типы `Signed` и `Unsigned`. Это также массивы неограниченной длины, элементы которых имеют тип `bit`, но для которых определен расширенный набор операций и функций преобразования.

Строки. Для представления строк, VHDL имеет встроенный тип-массив неограниченной длины с символьным типом элементов и положительным типом индекса:

```
type string is array (positive range <>) of character;
```

Физические типы неограниченной длины. Пакет стандартной логики std_logic_1164 *(см. приложение 3)* предлагает следующие типы — массивы неограниченной длины:

```
type std_ulogic_vector is array (natural range <>) of std_ulogic;
```

Например:

```
variable c_t:std_ulogic_vector (0 to 13):="ZZZZZ11Z00---";
constant f_f:std_ulogic_vector (15 downto 0):=X"FFFF";
```

Массивы неограниченной длины находят широкое применение при описании входных и выходных сигналов проектируемого устройства.

В качестве элементов массивов могут быть использованы массивы. Например:

```
type big_array is array (0 to 63) of std_ulogic_vector(15 downto 0);
variable n_array:big_array;
variable n_vec:std_ulogic_vector(15 downto 0);
variable n_1:std_ulogic;
N_vec:=n_array(1);
N_1:=n_array(1)(15);
```

Операции над массивами

Большинство операций, производимых над массивами, осуществляются поэлементно.

Однако есть ряд операций, которые могут выполняться над целыми одномерными массивами [3]. К операциям, выполняемым над одномерными массивами целиком, относятся: логические операторы, операторы сдвига, операторы отношения, конкатенация.

Логические операторы (**and**, **or**, **nand**, **nor**, **xor**, **xnor**) могут быть применимы к двум одномерным массивам, имеющим одинаковую длину и тип. Элементы таких массивов должны иметь тип `bit` или `boolean`. Логическая операция применяется к парам элементов массивов, имеющим одинаковый индекс, а результат присваивается элементу результирующего массива с тем же номером. Например:

```
type wword is array(positive range<>) of bit;
variable bb1,bb2,bb3:wword(0 to 5);
    bb1:=(others=>'1');
    bb2:=(0=>'1',3=>'1',others=>'0');
    bb3:=bb1 and bb2;
```

Операторы сдвига (**sll**, **srl**, **sla**, **sra**, **rol**, **ror**) также могут применяться к одномерным массивам с типами элементов `bit` или `vector`. Массив является левым операндом, результирующий массив должен иметь тот же тип.

Операторы отношения могут применяться к одномерным массивам, элементы которых имеют любой дискретный тип. Оба массива-операнда не обязательно должны быть одинаковой длины, но должны иметь одинаковый тип элементов. Рассмотрим, как осуществляется сравнение, на примере опера-

тора <. Предположим, что надо определить результат операции a<c (где: a и c — одномерные массивы). Если a и c имеют длину 0, то a<c is false. Если a имеет длину 0, а длина c отлична от 0, то считается, что a меньше c. Если и a, и c имеют длину, отличную от 0, то a<c, если a(1)<c(1) или a(1)=c(1), но при этом a(2)<c(2), и т. д.

Конкатенация. К одномерным массивам может применяться конкатенация, в результате которой элементы правого операнда добавляются в конец левого. Например: "abc" & "df" = "abcdf".

Фрагменты массивов

Нередко возникает необходимость работы не со всем массивом, а с некоторым его фрагментом — набором элементов с индексами, лежащими в заданном интервале. При этом порядок следования индексов во фрагменте может быть обратным порядку следования индексов в массиве.

Например:

```
type array1 is array ( 1 to 100) of integer;
type array2 is array (100 downto 1) of integer;
variable a1:array1;
variable a2:array2;
```

Следующие определения фрагментов массивов являются допустимыми:

```
a1(20 to 50)
a2(70 downto 30)
a1(50 downto 20)
a2(30 to 70)
```

К фрагментам массивов могут применяться те же операции, которые применимы к массивам в целом. Например:

```
type wword is array (0 to 5) of bit;
variable bb1, bb2 : wword;
bb1(0 to 2):= bb2(2 to 4);
```

Преобразование типов

Объекты массивов различных типов могут быть преобразованы друг в друга, если массивы имеют: одинаковый тип элементов, одинаковую размерность и одинаковые типы индексов. Может быть выполнено преобразование объекта типа массив в объект другого типа массив. При этом исходный и новый тип должны иметь одинаковый тип элементов и одинаковую размерность.

Преобразование типов имеет следующий синтаксис:

```
object_of_type1:= name_of_type1(object_of_type2)
```

Стандартный пакет std_logic_arith содержит следующие функции преобразования стандартных векторных типов:

```
function CONV_INTEGER(ARG: INTEGER)     return INTEGER;
function CONV_INTEGER(ARG: UNSIGNED)    return INTEGER;
function CONV_INTEGER(ARG: SIGNED)      return INTEGER;
function CONV_INTEGER(ARG: STD_ULOGIC) return SMALL_INT;

function CONV_UNSIGNED(ARG: INTEGER;SIZE: INTEGER)     return UNSIGNED;
function CONV_UNSIGNED(ARG: UNSIGNED; SIZE: INTEGER)  return UNSIGNED;
function CONV_UNSIGNED(ARG: SIGNED;SIZE: INTEGER)      return UNSIGNED;
function CONV_UNSIGNED(ARG: STD_ULOGIC;SIZE: INTEGER) return UNSIGNED;

function CONV_SIGNED(ARG: INTEGER;SIZE: INTEGER)        return SIGNED;
function CONV_SIGNED(ARG: UNSIGNED;SIZE: INTEGER)       return SIGNED;
function CONV_SIGNED(ARG: SIGNED; SIZE: INTEGER)        return SIGNED;
function CONV_SIGNED(ARG: STD_ULOGIC; SIZE: INTEGER)    return SIGNED;

function CONV_STD_LOGIC_VECTOR(ARG: INTEGER;SIZE: INTEGER) return
STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: UNSIGNED;SIZE: INTEGER) return
STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: SIGNED; SIZE: INTEGER) return
STD_LOGIC_VECTOR;
function CONV_STD_LOGIC_VECTOR(ARG: STD_ULOGIC; SIZE: INTEGER) return
STD_LOGIC_VECTOR;
```

Стандартный пакет numeric_std содержит следующие функции преобразования стандартных векторных типов:

```
function to_integer (arg: unsigned)                 return natural;
function to_integer (arg: signed)                   return integer;
function to_unsigned (arg: integer; size: natural) return unsigned;
function to_signed (arg: integer; size natural)     return signed;
```

Поддерживаются следующие функции изменения размера:

```
function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL)   return SIGNED;
function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL) return UNSIGNED;
```

Записи

Описание типа записи имеет следующий синтаксис:

```
type name is record
  name_of field1:name_of_type
```

```
  name_of field2:name_of_type
...
  name_of field_n:name_of_type
end record name;
```

Например:

```
type big_time is record
  seconds:  integer range 0 to 59;
  minutes:  integer range 0 to 59;
  hours:    integer range 0 to 23;
end record big_time;
type in_port is record
  address:  std_logic_vector(7 downto 0);
  data:     std_logic_vector(16 downto 0);
  wr_en:    std_logic;
end record in_port;
```

Возможно использование агрегатов для задания значений объектов такого типа. Например:

```
constant time_counter_1:big_time:=(2,2,12);
constant time_counter_2:big_time:=(hours=>12, minutes=>0,seconds=>2);
```

Значения присваиваются полям в порядке их описания в типе, или же по именам полей.

Агрегаты могут использоваться также для определения начальных значений переменных (записей), при их описании в декларативной части, но не могут использоваться для определения их значений в других частях модели.

Если выполняется присваивание значения конкретному полю записи, то сначала указывается имя объекта-записи, а затем, через точку, указывается имя поля:

```
variable my_time:big_time;
My_time.seconds:=30;
```

Указательные типы данных (access)

Назначение указательных типов

Скалярные и составные типы данных позволяют представлять как отдельные элементы данных, так и регулярные структуры, из них состоящие. Однако в некоторых приложениях возникает необходимость в создании набора данных, размер которых заранее не известен, или организации сложной

структуры отношений между индивидуальными объектами данных. В этих случаях используются *указательные типы*.

Указательные типы позволяют создавать сложные структуры данных в процессе моделирования. Они имеют много общего с указательными типами в других языках программирования.

Описание данных указательного типа

Описание указательного типа имеет следующий синтаксис:

```
type name is access subtype_indication;
```

Например:

```
type natural_ptr is access natural;
```

Объект, принадлежащий типу `natural_ptr`, может содержать значение-ссылку на объект данных типа `natural`, но не на объекты других типов.

Ссылки могут быть организованы на объекты любых типов, кроме файлового.

Когда переменная описана как объект указательного типа, ее начальное значение по умолчанию — `null`. Переменную можно связать с конкретным объектом данных, размещенным в памяти. Оператор связи имеет следующий синтаксис:

```
name_of_object:=allocator_expression;
```

Здесь `allocator_expression` имеет следующий синтаксис:

```
new subtype_indication | new qualified_expression
```

В результате выполнения такого оператора связи в памяти создается объект данных указанного типа и ему присваивается значение 0. Указатель на вновь созданный объект присваивается объекту `name_of_object`. Например:

```
variable count:natural_ptr;
count:=new natural;
```

Для работы с объектом, на который ссылается переменная-указатель, используется ключевое слово **all**. Например:

```
count.all:=10;
```

Для того чтобы при создании указателя на новый объект сразу же присвоить этому объекту значение, используется `qualified_expression`. Оно имеет следующий синтаксис:

```
type_mark'(expression)| type_mark'agregate
```

Пример 1:

```
count:=new natural'(10);
```

Пример 2:

```
type stimulus_record is record
  stimulus_time:time;
  stimulus_value:bit_vector(0 to 3);
end record stimulus_record;
type stimulus_ptr is access stimulus_record;
variable bus_stimulus:stimulus_ptr;
bus_stimulus:=new stimulus_record(20 ns, B'0011');
```

Работу с указателями поясним на следующем примере. Пусть:

```
variable count1, count2: natural_ptr;
count1:=new natural'(5);
count2:=new natural'(10);
```

После выполнения этих операторов в памяти будет наблюдаться следующее (рис. 2.2).

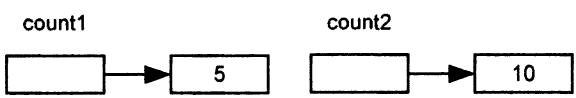


Рис 2.2. Иллюстрация состояния памяти 1

Затем, выполняя:

```
count2:=count1;
count2.all:=20;
```

получим результат, представленный на рис. 2.3. В данном случае `count1=count2` и они оба указывают на объект со значением 20.

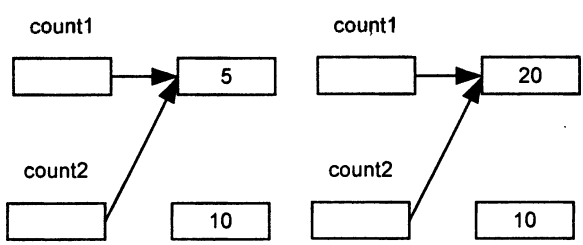


Рис. 2.3. Иллюстрация состояния памяти 2

Другой пример. Если выполнить следующую последовательность действий:

```
count1:= new natural'(30);
count2:= new natural'(30);
```

то получим два разных указателя, count1<>count2, указывающих на разные объекты, хотя эти объекты в данный момент будут иметь и одинаковые значения (рис. 2.4).

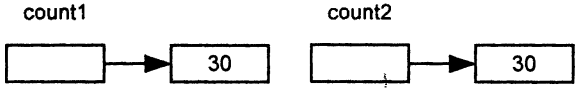


Рис. 2.4. Иллюстрация состояния памяти 3

Указатели на массивы

Работа с указателями на массивы постоянной длины. Если организуется указатель на массив постоянной длины, то, для обращения к конкретному элементу этого массива, его индекс указывается в скобках после ключевого слова **all**. Например:

```
type coordinate is array (1 to 3) of real;
type coordinate_ptr is access coordinate;
variable origin:coordinate_ptr:=new coordinate'(0.0,0.0,0.0);
origrin.all(1):=1.0;
```

Работа с указателями на массивы переменной длины. Если организуется указатель на массив неопределенной длины, то длина массива определяется при первом же присваивании ему значений и определяется количеством присваиваемых значений. Например:

```
type time_array is array (positive range<>) of time;
type time_array_ptr is access time_array;
variable activ_times:time_array_ptr;
activ_times:=new time_array_ptr(10 us, 15 us, 20 us);
```

После выполнения последнего действия размер массива, связанного с переменной, определен. Значения элементов массива можно изменять с помощью оператора присваивания. При работе с массивом через указатели, для изменения размера самого массива его необходимо создать заново. Например, чтобы добавить в него два элемента, необходимо выполнить следующее действие:

```
activ_times:=new time_array'(activ_times.all&time_array'(70 us, 100 us));
```

В результате массив будет состоять из пяти элементов. Первые три из них будут содержать значения из исходного массива, а два новых значения будут добавлены в конец:

```
(10 us, 15 us, 20 us, 70 us, 100 us)
```

Для того чтобы создать указатель на объект определенной длины на базе типа с незаданной длиной, можно воспользоваться прямым указанием длины, а не агрегатом, на базе которого она вычисляется. Например:

```
activation_times:=new time_array (1 to 10);
```

Организация связанных структур данных

Частичное предописание

Внимание

В OrCAD Express 9.1 частичное предописание не поддерживается (выдается сообщение Syntax Error). В Foundation Express наличие частичного предописания не приводит к возникновению синтаксической ошибки, но при синтезе это описание игнорируется.

В связанной структуре данных каждый объект состоит из двух частей. В первой из них располагаются собственно данные, а вторая включает в себя указатели на другие объекты структуры.

Например, однонаправленный список может быть построен на базе объектов, имеющих следующую структуру:

```
type value_cell is record
  value: bit_vector( 0 to 3)
  next_cell:value_ptr;
end record value_cell;
type value_ptr is access value_cell;
```

При таком описании подобной структуры возникают проблемы, связанные с тем, что каждый из описанных типов определяется при помощи другого. Для разрешения этой проблемы используется частичное предописание. В нем сначала описывается тип запись, на базе которого будет создаваться список, затем описывается указатель на этот объект, а уже после этого описывается структура самого объекта. Например:

```
type value_cell;
type value_ptr is access value_cell;
type value_cell is record
  value: bit_vector( 0 to 3)
  next_cell:value_ptr;
end record value_cell;
```

Организация списка. На базе описанных выше объектов организуем список, в котором объекты добавляются в самое начало.

```
variable value_list: value_ptr;
value_list:=new value_ptr;
value_list:=new value_cell'(B"1000",value_list);
```

В результате выполнения этих действий список будет иметь вид, показанный на рис. 2.5.

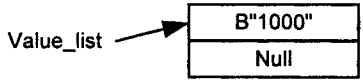


Рис 2.5. Состояние списка 1

Добавим в список еще один элемент:

```
value_list:=new value_ptr'(B"0000",value_list);
```

В результате список примет вид, представленный на рис. 2.6.

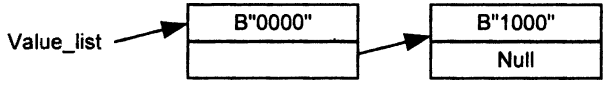


Рис 2.6. Состояние списка 2

Для того чтобы выполнять действия над всей последовательностью элементов списка, можно воспользоваться следующей циклической конструкцией:

```
current_cell:=value_list
while current_cell /= null  loop
  -- Выполнение необходимых действий с текущим элементом списка.
  current_cell:=current_cell.next_cell;
end loop;
```

Удаление объектов-указателей из памяти. Для удаления из памяти объекта, ранее распределенного с помощью механизма указателей, служит процедура **deallocate**. Она генерируется автоматически для каждого указательного типа.

Пусть определен тип T.

```
type T_ptr is access T;
```

Для типа T_ptr процедура **deallocate** имеет следующий заголовок:

```
deallocate(P: inout T_ptr);
```

В программе, например, она может быть использована следующим образом:

```
cell_to_be_deleted:=value_list;
value_list:=value_list.next_cell;
deallocate(cell_to_be_deleted);
```

После выполнения этой последовательности действий в списке останутся только элементы, начиная с того, на который теперь указывает `value_list`. Объект, на который указывает `cell_to_be_deleted`, корректно удаляется из памяти. Если он был не первым объектом в списке, то все предшествующие ему объекты превращаются в "мусор".

Обращение к объекту, который уже удален из памяти, может иметь непредсказуемые последствия (от возвращения в качестве результата случайных данных до полной остановки процесса моделирования).

Абстрактные типы данных

На базе указательных структур могут создаваться так называемые абстрактные типы данных (Abstract Data Types, ADT). ADT включает в себя тип данных, а также набор операций по созданию объектов этого типа и работы с ними. В идеале, при работе с таким типом пользователю не должна быть видна его внутренняя структура. Однако язык VHDL не предоставляет механизма, позволяющего скрыть структуру объекта.

Наиболее подходящим методом реализации ADT в VHDL является создание пакетов. В декларации пакета производится описание структуры данных, декларируются функции и процедуры для работы с этой структурой. В теле пакета размещается реализация этих процедур и функций. Но и здесь, поскольку в VHDL, в отличие, например, от C++, нет средств сокрытия описаний, ничто не мешает пользователю иметь свободный доступ к структуре данных без использования процедур и функций, для этого предназначенных.

Механизм шаблонов пакетов

Для выполнения таких операций, как добавление элемента в список, не важен тип элементов, из которых состоит список. VHDL не имеет механизмов для параметризации по имени типа, поэтому для решения этой проблемы используется механизм создания шаблонов пакетов. Каждый раз, когда требуется работать с конкретным типом данных, делается очередная копия шаблона пакета и в нее подставляются конкретные имена в соответствии с этим типом.

Ограничения на практическое применение указательных типов

Во многих системах автоматизированного проектирования на ПЭВМ компиляторы VHDL не поддерживают указательные типы (частично или полностью).

В OrCAD Express собственно указательные типы поддерживаются, но не поддерживается организация ссылочных структур на этой основе. Это делает их использование практически бесполезным.

В Foundation Express указательные типы не поддерживаются вовсе, что связано с трудностью их интерпретации при реализации физических структур.

Операторы

Оператор присваивания

Оператор присваивания нового значения переменной имеет традиционный для процедурных языков синтаксис — метка, объект, которому присваивается новое значение, знак присваивания, а в левой части — выражение.

```
[label:]name:=expression;
```

Операции над данными в выражениях

Перечень операций, которые могут встречаться в выражениях, определяющих значения переменных, приведен в табл. 2.4.

Таблица 2.4. *Перечень операторов*

Оператор	Операция	Тип левого операнда	Тип правого операнда	Тип результата
**	Возведение в степень	integer, real	integer	Тот же, что и у левого операнда
abs	Абсолютное значение	Числовой	integer	Тот же, что и у операнда
not	Отрицание	bit, boolean или одномерный массив из элементов этих типов	integer	Тот же, что и у операнда
*	Умножение	integer или real	Тот же, что и у левого операнда	Тот же, что и у операндов
		Физический	integer или real	Тот же, что и у левого операнда

***Таблица 2.4** (продолжение)*

Оператор	Операция	Тип левого операнда	Тип правого операнда	Тип результата
		integer или real	Физический	Тот же, что и у правого операнда
/	Деление	integer или real	Тот же, что и у левого операнда	Тот же, что и у операндов
		Физический	integer или real	Тот же, что и у левого операнда
		Физический	Тот же, что и у левого операнда	Universal integer
mod	Деление по модулю	integer	Тот же, что и у левого операнда	Тот же, что и у операндов
rem	Остаток от деления	integer	Тот же, что и у левого операнда	Тот же, что и у операндов
+	Увеличение на 1	Числовой	Тот же, что и у левого операнда	Арифметический сдвиг влево
-	Уменьшение на 1	Числовой	Тот же, что и у левого операнда	Тот же, что и у левого операнда
+	Сложение	Числовой	Тот же, что и у левого операнда	Тот же, что и у операндов
		Физический	Физический	Физический
-	Вычитание	Числовой	Тот же, что и у левого операнда	Тот же, что и у операндов
		Физический	Физический	Физический
&	Конкатенация	Одномерный массив	Тот же, что и у левого операнда	Тот же, что и у операндов

Таблица 2.4 (продолжение)

Оператор	Операция	Тип левого операнда	Тип правого операнда	Тип результата
+	Сложение	Числовой	Тот же, что и у левого операнда	Тот же, что и у операндов
-	Вычитание	Числовой	Тот же, что и у левого операнда	Тот же, что и у операндов
&	Конкатенация	Одномерный массив	Тот же, что и у левого операнда	Тот же, что и у операндов
sll	Логический сдвиг влево	Одномерный массив с элементами типа bit или boolean	integer	Тот же, что и у левого операнда
srl	Логический сдвиг вправо	Одномерный массив с элементами типа bit или boolean	integer	Тот же, что и у левого операнда
sla	Арифметический сдвиг влево	Одномерный массив с элементами типа bit или boolean	integer	Тот же, что и у левого операнда
sra	Арифметический сдвиг вправо	Одномерный массив с элементами типа bit или boolean	integer	Тот же, что и у левого операнда
rol	Циклический сдвиг влево	Одномерный массив с элементами типа bit или boolean	integer	Тот же, что и у левого операнда
ror	Циклический сдвиг вправо	Одномерный массив с элементами типа bit или boolean	integer	Тот же, что и у левого операнда

***Таблица 2.4** (продолжение)*

Оператор	Операция	Тип левого операнда	Тип правого операнда	Тип результата
=	Равенство	Все типы, кроме `file`	Тот же, что и у левого операнда	`Boolean`
/=	Неравенство	Все типы, кроме `file`	Тот же, что и у левого операнда	`Boolean`
<	Меньше	Скалярный тип или одномерный массив с элементами дискретного типа	Тот же, что и у левого операнда	`Boolean`
<=	Меньше или равно	Скалярный тип или одномерный массив с элементами дискретного типа	Тот же, что и у левого операнда	`Boolean`
>	Больше	Скалярный тип или одномерный массив с элементами дискретного типа	Тот же, что и у левого операнда	`Boolean`
>=	Больше или равно	Скалярный тип или одномерный массив с элементами дискретного типа	Тот же, что и у левого операнда	`Boolean`
and	Логическое "И"	`bit`, `boolean` или одномерный массив с элементами этих типов	Тот же, что и у левого операнда	Тот же, что и у операндов
or	Логическое "ИЛИ"	`bit`, `boolean` или одномерный массив с элементами этих типов	Тот же, что и у левого операнда	Тот же, что и у операндов

Таблица 2.4 (окончание)

Оператор	Операция	Тип левого операнда	Тип правого операнда	Тип результата
Nand	"Не-И"	bit, boolean или одномерный массив с элементами этих типов	Тот же, что и у левого операнда	Тот же, что и у операндов
nor	"Не-ИЛИ"	bit, boolean или одномерный массив с элементами этих типов	Тот же, что и у левого операнда	Тот же, что и у операндов
xor	Исключающее "ИЛИ"	bit, boolean или одномерный массив с элементами этих типов	Тот же, что и у левого операнда	Тот же, что и у операндов
xnor	Отрицание исключающего "ИЛИ"	bit, boolean или одномерный массив с элементами этих типов	Тот же, что и у левого операнда	Тот же, что и у операндов

Внимание

В Foundation Express существуют следующие ограничения на использование операций:

- операции деления, деления по модулю, остатка от деления поддерживаются только в случае, если оба операнда являются константами или делитель является степенью 2;
- операции возведения в степень поддерживается только в случае, если оба операнда являются константами, или если основание степени равно 2.

Управляющие операторы

Как для всякого алгоритмического языка, для языка VHDL определена естественная последовательность выполнения исполняемых операторов. Это — последовательность выполнения в порядке записи операторов программы.

Операторы управления последовательностью действий позволяют программно изменять эту естественную последовательность, устанавливать зависимость хода выполнения программы от значений обрабатываемых данных.

Операторы условного перехода

Оператор if

Оператор **if** имеет следующий синтаксис:

```
if boolean_expression then
  {seqential_statement}
[{elsif boolean_expression then
  {sequential_statement}]
[else
  {sequential_statement}]
end if;
```

Оператор может содержать только секцию **if**, что соответствует пустому оператору по ветке **then**.

Секций **elsif** может быть несколько. Например:

```
if data1="1111" then
  data2:=data3;
else data2:=data4;
end if;
if (data1=5) and (data2=4) then
  dataout:=7;
elsif (data1=3) and (data2=9) then
dataout:=1;
end if;
```

Оператор case

Оператор **case** имеет следующий синтаксис:

```
case expression is
  {when (simple_expression | discrete range | others) =>
  {sequential_statement}}
end case;
```

Выражение, стоящее после **case** (селектор), должно принимать дискретный набор значений. Значение этого выражения сравнивается со значениями, стоящими после when (их тип должен совпадать). Выполняется последовательность действий, стоящая после первого же **when**, для которого было обнаружено совпадение. Если необходимо, чтобы выполнялась некоторая последовательность действий при отсутствии совпадений, то используется конструкция **when others**, после которой и задается эта последовательность. Эта конструкция должна быть последней в операторе.

Например:

```
type alu_func is (pass1, pass2, pass3, pass4);
variable func: alu_func;
case func is
  when pass1 => result:=operand1;
  when pass2 => result:=operand2;
  when others => result:=0;
end case;
```

Если нескольким значениям селектора соответствует одна и та же последовательность действий, они могут быть перечислены после одного when. Разделителем служит вертикальная черта.

Например:

```
when load | add | subs operand:= mem_operand;
```

Если одна и та же последовательность действий соответствует некоторому интервалу значений селектора, то после when можно задать интервал. Это имеет следующий синтаксис:

```
when simple_expression to | downto simple_expression =>
```

Значения выражений, стоящих после when, вычисляются только на этапе компиляции, поэтому они не должны содержать объектов, значения которых могут изменяться в ходе моделирования.

Циклы

Оператор цикла loop

Простейшая форма оператора **loop**, которая позволяет организовать постоянно повторяющийся цикл, имеет следующий синтаксис:

```
[loop_label:] loop
  {sequential_statement}
end loop [loop_label];
```

Циклы **loop** могут быть вложенными.

Оператор **loop** сам по себе организует бесконечный цикл. Тело цикла будет выполняться неограниченное число раз, если выполнение цикла не будет прервано другими, специальными управляющими операторами.

Оператор завершения цикла exit

Для того чтобы организовать цикл, который бы не был бесконечным, используется специальный оператор **exit**. Он имеет следующий синтаксис:

```
exit [loop label] [when boolean_expression]
```

Если условие **when** не указано, то выполнение оператора однозначно приводит к завершению цикла; в противном случае цикл завершается только при выполнении условия, стоящего после **when**.

Если не указана метка цикла, то оператор exit осуществляет выход из того цикла, в котором он непосредственно находится. Указание метки цикла после ключевого слова exit позволяет использовать этот оператор для завершения сразу нескольких вложенных друг в друга циклов. Например:

```
outer: loop
  inner: loop
    exit outer when condition_1;
         -- выход из внутреннего и внешнего циклов
    exit when condition_2;
         -- выход из внутреннего цикла
  end loop inner;
  exit when condition_3; -- выход из внешнего цикла
end loop outer;
```

Оператор прерывания текущей итерации цикла next

Для того чтобы на очередной итерации цикла не выполнять всю последовательность действий, включенных в тело цикла, может использоваться оператор next:

```
next [loop label] [when boolean_expression]
```

Этот оператор отличается от оператора exit тем, что он прерывает не выполнение цикла, а только эту итерацию. После выполнения next прекращается выполнение текущей итерации и происходит переход на следующую итерацию цикла.

Как и в операторе **exit**, этот оператор может быть снабжен условием выполнения, однако, в отличие от оператора **exit** (который прерывает выполнение, если условие истинно), **next** прерывает выполнение, если условие ложно.

Циклы с условием while

Циклы с предусловием **while**. Условие выполнения цикла можно вынести в его заголовок. Тогда, если перед началом очередной итерации условие истинно, то она выполняется, в противном случае цикл завершается. Такой цикл имеет следующий синтаксис:

```
[loop_label] while condition loop
  {sequential_statement}
end loop [loop_label];
```

При подобной организации цикла предполагается, что в процессе его выполнения объекты, входящие в условие заголовка цикла, могут изменить свое значение в теле цикла.

```
while data="0000" loop
  data <= data_in;
end loop;
```

Циклы с указанием количества повторений for.

В заголовке цикла можно указать количество его повторений. Это имеет следующий синтаксис:

```
[loop_label] for identifier in
(simple_expression to | downto simple_expression)
loop
  {sequential_statement}
end loop [loop_label];
```

Объект, указанный идентификатором `identifier`, может быть любого дискретного типа и не требует предварительного описания в модели. Идентификатор считается определенным только на время выполнения цикла. Хотя его имя может совпадать с именем другого объекта в модели, данный идентификатор в заголовке цикла никак не будет с ним связан; его значение не будет влиять на значение другого объекта в модели, имеющего то же имя, и наоборот.

Перед первым выполнением цикла идентификатору, указанному в заголовке, присваивается значение левой границы интервала, заданного в заголовке. При каждом выполнении цикла оно увеличивается (уменьшается) на 1 (для типов, задаваемых перечислением элементов, идентификатору присваивается значение следующего элемента). Количество повторений цикла равно количеству элементов, входящих в заданный в заголовке интервал.

Пример:

```
variable a, b: integer;
begin
  a:=10;
  for a in 0 to 7 loop
    b:=a;
  end loop;
```

После завершения выполнения этого цикла переменные имеют следующие значения: `b=7`, `a=10`.

В теле цикла **for** нельзя изменять значения идентификатора, указанного в заголовке цикла.

Если цикл задан следующим образом:

```
for a in expr1 to expr2 loop:
```

при expr1>expr2 цикл не будет выполняться ни разу. Аналогичный эффект получим и для следующего примера:

```
for a in expr1 downto expr2 loop:
```

при expr1<expr2.

Использование атрибутов в циклах работы с массивами.

Циклы нередко используются при работе с массивами. Рассмотрим пример работы с циклами, в которых будут использованы атрибуты типов. В ходе работы над проектами определения типов могут претерпевать изменения. Крайне нежелательно, чтобы при этом приходилось редактировать большой объем кода. Во избежание такой ситуации могут быть использованы атрибуты типов. Пусть, например, входные данные представляются в виде вектора, элементы которого имеют тип std_ulogic , а его длина составляет 16:

```
type my_vector_type is std_ulogic_vector(0 to 15);
```

Пусть в программе имеется переменная my_vector, описанная с использованием этого типа. Необходимо определить количество единиц в этом векторе. Фрагмент кода может выглядеть следующим образом:

```
Sum:=0;
for i in 0 to 15 loop
  if my_vector(i)='1' then sum:=sum+1; end if;
end loop;
```

Если при дальнейшей работе с проектом выяснится, что вектор должен иметь длину не 16, а 32 разряда, то заголовок цикла придется редактировать.

Однако, решая ту же задачу, заголовок цикла можно написать с использованием атрибутов типа:

```
Sum:=0;
for i in my_vector_type'range loop
  if my_vector(i)='1' then sum:=sum+1; end if;
end loop;
```

В этом случае текст программы цикла не придется редактировать, даже если изменится не только верхняя и нижняя граница диапазона, но и направление интервала.

Пустой оператор

Пустой оператор имеет следующий синтаксис:

```
Null;
```

Как правило, он используется на этапе отладки модели, когда известно, что в определенном месте модели должна выполняться некая последовательность действий, но какая именно, еще неясно. Например:

```
if a<b then Null;
else a:=b;
end if;
```

Операторы управления сбором информации в ходе моделирования

Операторы управления сбором информации в ходе моделирования входят в стандарт языка VHDL.



Все, что описано в этом параграфе, может быть использовано только в OrCAD Express. В Foundation Express эти операторы игнорируются.

Оператор assert

При отладке модели полезно контролировать значения, которые принимают объекты, и заносить информацию, получаемую в ходе работы модели, в отчет о проведенном моделировании. Для этого в программу на языке VHDL вводятся специальные операторы, управляющие сбором и сохранением информации о прогоняемой модели.

Для этих целей используется оператор **assert**, который имеет следующий синтаксис:

```
assert boolean_expression [report expression] [severity expression]
```

Если логическое выражение, стоящее в операторе **assert**, принимает в ходе моделирования значение **false**, моделирующая программа немедленно выдает уведомление об этом. Выражение, стоящее в секции **report** (если она имеется), при этом заносится в отчет. Это выражение может содержать текстовые строки, в качестве которых могут быть использованы атрибуты объектов, приводящие их значения к текстовому типу. Выражение, стоящее в секции **severity**, позволяет определить уровень ошибки. Оно должно принимать одно из значений, принадлежащих типу `severity_level`, определение которого выглядит следующим образом:

```
type severity_level is (note, warning, error, failure);
```

Значение `note` соответствует ситуации, когда необходимо просто поместить информативное сообщение в отчет. Значение `warning` соответствует ситуации, когда в ходе выполнения модели возникло нечто, не характерное для нее (выполнение модели может продолжаться, но результат возможен не-

обычный). Уровень `error` используется, когда возникшая ситуация свидетельствует об ошибке (например, ширина тактового импульса оказалась меньше допустимой).

Уровень `failure` свидетельствует о нарушении целостности (например, разность значений указателей последней и первой позиций в буфере, увеличенная на 1, оказалась не равна количеству объектов в буфере). Как правило, такие ситуации не приводят к аварийному завершению моделирования. Однако после такой ситуации модель может начать вести себя неадекватно, значения сигналов модели перестанут соответствовать тому, что могло бы быть в реальной системе. Видимо, эти операторы и были созданы для того, чтобы проще было отслеживать ситуации, которые на временной диаграмме можно и не заметить.

В OrCAD Express иногда возникают ситуации, когда моделирование завершается аварийно самой средой. Это может быть не связано с наличием оператора **`assert`**.

Данное определение `severity_level` входит в пакет Standard, который подключается к модели по умолчанию, поэтому пользователю в явном виде вводить его в своей программе не надо.

Например:

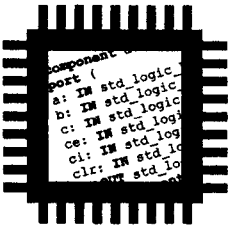
```
assert free_mem<=low_limit report "memory is small" severity warning;
```

Если в операторе отсутствует секция `report`, то по умолчанию в файл отчета записывается фраза "Assertion violation". Если в операторе отсутствует секция `severity`, то по умолчанию ситуация имеет уровень `error`.

Оператор report

Если информация должна заноситься в отчет при любых условиях, то используется оператор `report` безусловного занесения информации в отчет. Он имеет следующий синтаксис:

```
report expression [severity expression];
```



Глава 3

Базовые конструкции моделей на языке VHDL

Сигналы

Сигналы являются отдельным классом объектов языка VHDL, отличным от классов переменных и констант, достаточно традиционных для языков высокого уровня. Понятие сигнала не имеет аналогов в языках программирования типа FORTRAN, C, PASCAL и др.

Цифровые системы работают во времени, преобразуя и передавая сигналы. Естественно, что понятие сигнала является базовым в языке описания аппаратуры — языке VHDL. Сигналы являются абстракцией, представлением в модели на VHDL состояния проводников в структуре цифрового устройства.

О понятии Время. Рассуждая о языках и системах моделирования, мы имеем дело с несколькими понятиями, которые соотносятся с термином "время":

- *физическое время моделируемого устройства.* Оно непрерывно; в нем протекают реальные физические процессы, изменяются электрические сигналы в цифровых системах, наблюдаемые на логическом уровне как изменения логических состояний.
- *модельное время.* Это время в модели устройства; оно является обобщенным представлением, на уровне модели устройства, физического времени, в котором работает моделируемое устройство. Модельное время дискретно;
- *время моделирования.* Это время выполнения программы на VHDL. В семантике программы на VHDL впрямую не наблюдается, однако, проявляется через упорядочение операторов программы по времени выполнения. Не имея для программиста на VHDL числовых значений, время работы системы моделирования позволяет определить отношение поряд-

ка над событиями — выполнением операторов программы на VHDL. Для каждой пары исполненных операторов программы определены отношения "выполнен раньше"/"выполнен позже".

Важно различать эти три разных вида времени, с которыми имеет дело проектировщик, работающий на языке VHDL.

Сигналы в цифровых устройствах. Каждое изменение состояния элемента схемы, его входов, является воздействием, которое может привести к изменению некоторого сигнала. Например, подача '1' на вход вентиля может привести к изменению сигнала на его выходе, но не мгновенно, а через интервал времени, определяемый задержкой вентиля ($\tau_в$, скажем, 5 нс.). Так что, подавая '1' на вход вентиля в момент t_2, мы как бы "запланировали" изменение сигнала на его выходе на время $t_3 = t_2+\tau_в$.

А что будет на выходе вентиля в момент $t_2 = t_2+\tau_в - 2$ нс.? Это определяется тем, что было на входах вентиля в момент $t - 2$ нс. (т. е. — что было запланировано, но еще не успело выполниться, находится в процессе выполнения). Таким образом, в момент t_2 в схеме идет множество процессов, которые "запланировали" значения сигналов на каждый момент времени, от момента t_2 на некоторый промежуток времени вперед. В текущем состоянии физического устройства как бы заложена временная диаграмма рассматриваемого сигнала на некоторый период времени вперед.

В реальном цифровом устройстве, которое моделируется программой на VHDL, воздействий на сигнал, приводящих к его изменению, может быть несколько. Если мы посмотрим с этой точки зрения на сигнал и на устройство, то увидим, что в текущий момент физического времени в схеме одновременно идет множество процессов, в разных ее элементах и на связях между ними. Процессы в устройстве, которые приведут к изменению сигнала, уже идут, но в измененном состоянии сигнала они проявятся через некоторые промежутки времени (возможно, разные процессы — через разные промежутки).

Сигналы в моделях устройств на VHDL. Сигналы, как и переменные, имеют некоторые значения, которые им присваиваются. Однако если переменную характеризует только *значение*, то сигнал характеризуется и *моментом модельного времени* [1], в который этот сигнал имеет данное значение. Можно сказать, что состояние сигнала — это пара: момент модельного времени / значение, которое сигнал имеет в данный момент.

Именно ассоциация сигнала с моментами модельного времени является его основным отличием от переменных. Сигнал существует в модельном времени, в каждый момент которого он может иметь свое значение, а переменная всегда имеет одно текущее значение. Таким образом, сигнал — это не тип

[1] Не путать со временем выполнения программы моделирования!

переменной, это совсем другой, принципиально отличный класс программных объектов. И разница здесь значительно существенней, чем между переменной и константой.

По ходу модельного времени сигнал может менять свое значение. Последовательность значений сигнала в привязке к моментам модельного времени в течение некоторого промежутка времени формирует *временную диаграмму* сигнала (waveform).

В модели устройства на языке VHDL воздействия, изменяющие состояние сигнала, принимают форму *присваивания значения сигналу.* Проявляется воздействие в изменении значения сигнала, как правило, не в тот момент модельного времени, когда происходит это воздействие, а в другой, отстоящий от текущего на некоторый промежуток времени (задержку) по оси модельного времени.

Как и в физическом устройстве, в его модели на языке VHDL параллельно идет множество процессов, определяющих значения сигналов в каждый момент модельного времени. Однако из прагматических соображений, чтобы сократить время прогона модели, система моделирует работу компонентов схемы не во всех моментах модельного времени, а только в тех, на которые запланированы действительные изменения каких-либо сигналов. Между такими моментами модельного времени сигналы не меняют своих значений, что мы и можем наблюдать при визуализации результатов моделирования на временных диаграммах.

События изменения сигнала. Изменение сигнала, реализуемое в какой-то момент модельного времени, называют *событием* (event). Принцип продвижения модельного времени системой моделирования формулируется так: от момента события — до следующего момента модельного времени, в который происходит хотя бы одно событие, пропуская промежуточные моменты, в которые не происходит никаких событий. В системах моделирования это называют *событийным моделированием.*

Транзакция. Когда мы осмысливаем понятие переменных в языках программирования, обычно представляем себе определенные аспекты их реализации после трансляции, мыслим о них как о выделенных под переменные участках памяти. Также и при обсуждении сигналов и механизмов языка VHDL полезно представлять, хотя бы в общем виде, как сигналы реализуются в системах моделирования, выполняющих программы на VHDL.

В реализации сигналов, изменяемых в модельном времени, основным становится планирование в этом времени изменений сигналов. При выполнении операторов присваивания значения сигналу, системой моделирования формируется специальная структура данных, пара: значение сигнала /момент модельного времени, когда сигнал примет это значение. Такая пара в описаниях языка VHDL называется *транзакцией* (transaction). Транзак-

ция — это внутренняя инструкция системы моделирования по изменению указанного сигнала в заданный момент модельного времени, Система моделирования, работающая по принципам событийного моделирования, ведет *список транзакций*, упорядоченных по меткам модельного времени.

Транзакция, как и список транзакций, является механизмом реализации VHDL, внутренним делом системы моделирования. Однако эти понятия часто привлекаются при описании и разъяснении различных механизмов языка VHDL.

Источник сигнала (драйвер). Еще одним понятием, связанным с сигналами и часто используемым в описании VHDL, является понятие *драйвера* (driver) — *источника сигнала.*

В реальных цифровых устройствах выход элемента схемы формирует сигнал на проводнике, подсоединенном к этому выходу, он является источником сигнала в этой точке схемы. Каждый выход каждого элемента схемы — источник для одного сигнала. Множество элементов схемы функционируют параллельно в физическом времени, причем элемент, имеющий несколько выходов, формирует сигналы на них также параллельно во времени. Когда несколько выходов элементов схем подсоединяются проводниками в одну точку схемы, то получается, что для этой точки схемы имеется несколько источников сигналов. Результирующий сигнал в этой точке схемы формируется как результат выполнения (на физическом уровне) некоторых преобразований над сигналами, сформированными всеми источниками. Вид выполняемого преобразования зависит от элементной базы, используемой в устройстве.

Работа с сигналами в модели устройства на языке VHDL достаточно точно воспроизводит описанные ситуации в реальном цифровом устройстве.

Каждый сигнал существует параллельно с другими сигналами в модельном времени.

Источники сигналов в программе на VHDL представлены специальными операторами присваивания значений сигналам. Один источник сигнала и в реальном устройстве, и программе на VHDL, формирует один сигнал.

Однако в языке VHDL источник сигнала может не только определять значение сигнала в какой-то один момент модельного времени, но и задавать целую последовательность значений сигнала в разные моменты модельного времени, формировать некоторую *планируемую временную диаграмму* сигнала (projected output waveform).

В реализации языка VHDL запланированная источником временная диаграмма сигнала представляется упорядоченным (по моментам модельного времени) списком транзакций, фиксирующих моменты изменения сигнала на временной диаграмме. Часто в описании VHDL именно этот список

транзакций, связанный с одним источником сигнала, называют термином *драйвер* (driver) [19].

Поскольку каждый источник сигнала в реальной схеме — выход некоторого элемента, формирует свой сигнал параллельно во времени с другими источниками сигналов, то и в модели устройства на VHDL источники сигналов работают параллельно в модельном времени.

Синтаксически, в структуре программы на VHDL, параллельно работающий источник сигнала может быть оформлен по-разному. Это может быть отдельный параллельный оператор присваивания значения сигналу, может быть и последовательный оператор присваивания значения сигналу среди операторов тела процесса (параллельная конструкция программы на VHDL). Но в любом варианте, осмысливая соответствующие программные конструкции и фрагменты программ на VHDL, представляющие источники сигналов, мы должны стараться соотносить их с реальной работой схем и формируемых ими сигналов.

Структура описания объекта моделирования

Описание объектов моделирования состоит из *декларативной части* и *описания архитектуры.*

В декларативной части описываются связи объекта с внешним миром — входы и выходы объекта. Это, прежде всего, спецификация интерфейса описываемого объекта. В описании архитектуры определяется функция специфицируемого объекта, осуществляемого им формирования выходных сигналов на основании входных сигналов и внутреннего состояния объекта.

Декларативная часть

Полный формат декларативной части описания объекта моделирования имеет следующий синтаксис:

```
entity entity_name is
  [generic (generic_interface_list);]
  [port (port_interface_list);]

[begin {concurrent_assertion_statement |
  passive_concurrent_procedure_call_statement |
  passive_process_statement}]
  {entity_declarative_item}
end [entity] [entity_name];
```

После ключевого слова **entity** указывается некоторый идентификатор entity_name — имя объекта моделирования. Поскольку оно используется для идентификации объекта в рамках проекта, имя объекта моделирования должно быть уникальным.

Секция **generic** предназначена для описания констант, определяющих изменяемые параметры объекта моделирования. Например, таким образом могут определяться времена задержек, размер внутренних буферов, и др. Структура этой секции будет рассмотрена подробнее в дальнейшем. Секция **generic** не является обязательной.

В терминах языка VHDL входы и выходы проектируемой схемы называют *портами*. Порты — специальные программные объекты, являющиеся сигналами, а не переменными. Подобно переменным в традиционных языках программирования, в программе на VHDL порты должны быть определены (декларированы) с указанием типа соответствующих им сигналов. Для определения входных и выходных портов используется секция **port** в декларативной части спецификации объекта моделирования.

После ключевого слова **port** в круглых скобках располагается список описаний сигналов — port_interface_list. Он имеет следующий синтаксис:

```
{identifier, {...}:[mode] subtype_indication [:=expression]}, {...}
```

Описание каждого сигнала состоит из имени, вида сигнала (**mode**) и типа сигнала. Вид сигнала может иметь одно из следующих значений:

- in — входной сигнал;
- out — выходной сигнал;
- inout — сигнал, являющийся и входным, и выходным.

Все типы, которые указываются в описании портов, должны быть уже известны. Как видно из общей формы синтаксиса декларации **entity**, здесь нет места описанию типов. Если это не стандартные, встроенные в VHDL типы, то они должны быть описаны до декларации **entity**. Эти описания могут располагаться в том же файле, выше по тексту, или (как часто делается), вынесены в отдельный пакет или библиотеку.

В рамках описания, в секции **port** сигналам могут присваиваться значения по умолчанию. Если для сигнала определено значение по умолчанию, то оно используется внутри объекта моделирования в том случае, если сигналу не будет присвоено другое значение извне. Идентификаторы нескольких сигналов, имеющих одинаковое дальнейшее описание (вид, тип, значение по умолчанию, если оно есть) могут следовать в одном описании, через запятую.

Секция **port** также не является обязательной составляющей декларативной части описания объекта. Объект моделирования может не иметь входов и

выходов. Такие объекты, как правило, расположены на верхнем уровне иерархии.

После ключевого слова **begin** следуют секции параллельно выполняющихся действий (не являются обязательной частью описания), которые могут использоваться для проверки правильности функционирования объекта и для документирования процесса функционирования. Организация этих секций более подробно будет рассмотрена ниже.

Последней располагается секция внутренних деклараций, {entity_declarative_item} описываемого объекта. В этой секции может содержаться декларация констант, переменных, сигналов и типов, являющихся внутренними для данного объекта моделирования, — т. е. доступных только внутри этого объекта.

Завершается описание объекта ключевым словом **end**, за которым следует слово **entity** (по стандарту VHDL'93 — обязательно); потом указывается имя описанного объекта (рекомендуется).

Примеры простейшего описания декларативной части объектов моделирования приведены в листингах 3.1 и 3.2.

В листинге 3.1 описан объект моделирования с именем adder. Он имеет три входных сигнала — a, b, c, имеющих тип word, и один выходной сигнал sum того же типа.

Листинг 3.1

```
entity adder is
port(  a,b,c: in word;
    sum: out word);
end entity adder;
```

В листинге 3.2 описан объект моделирования с именем adder1. Он имеет три входных сигнала — a, b, c, имеющих тип **integer**, и один выходной сигнал sum типа **word**. Одному из трех его входных сигналов присваивается значение по умолчанию, равное 1.

Листинг 3.2

```
entity adder1 is
port(  a: in integer;
    b: in integer:=1;    -- значение по умолчанию
    c: in integer;
    sum: out word);
end entity adder1;
```

Описание архитектуры объекта моделирования

В языке VHDL под описанием архитектуры понимается описание функционирования специфицируемого объекта. Если декларативная часть описания объекта определяет его внешнее представление, задает интерфейсную спецификацию объекта, вводит имя объекта и входы/выходы ("порты", в терминах VHDL), то описание архитектуры задает его содержательное наполнение, спецификацию функциональной и временной работы описываемого объекта.

Отметим, что используемый в VHDL термин "архитектура" не вполне соответствует общепринятому в вычислительной технике смыслу (описание ЭВМ для программирующего на уровне машинных кодов или их символьного представления, на ASSEMBLER). "Архитектурное описание" на VHDL может, конечно, в конкретной программе получить и такое смысловое наполнение. Мы можем, например, в архитектурной части описания объекта моделирования — процессора, действительно дать такое описание, какое принято давать при описании архитектуры (в общепринятом смысле). Но в архитектурном описании на языке VHDL может быть, с одной стороны, дана и детальная функциональная или принципиальная схема процессора, а с другой стороны, наоборот, описание может идти на более абстрактном уровне, и программно доступные элементы архитектуры процессора не найдут отражения в архитектурном описании на VHDL.

Описывать архитектуру объекта на языке VHDL можно следующими способами:

1. Описать поведение объекта, преобразование информации и его внутреннего состояния, формирование выходных сигналов при поступлении входных, задать алгоритмическое описание поведения специфицируемого объекта. Внутренняя структура описываемого объекта при этом не специфицируется. Такое описание называется *поведенческим описанием* архитектуры объекта.
2. Описать структуру объекта, как состоящего из некоторых других объектов, указывая их перечень и связи между ними. Такое описание называется *структурным описанием* архитектуры объекта. Поведение объекта, преобразование информации и формирование выходных сигналов при поступлении входных, определяется здесь составом и связями объектов, формирующих заданную структуру. Сами объекты-компоненты описываются отдельно, с использованием структурного или поведенческого описания, и т. д. В конечном итоге, на каком-то уровне вложенности таких описаний мы доходим либо до предопределенных компонентов, либо останавливаемся на алгоритмическом, поведенческом описании объекта-компонента.
3. Допускается и *смешанное структурно-поведенческое описание*, являющееся комбинацией первых двух видов описаний.

Описание архитектуры объекта моделирования имеет следующий синтаксис:

```
architecture identifier of entity_name is
  {block_declarative_item}
begin
  {concurrent_statements}
end [architecture] [identifier];
```

Спецификатор `entity_name` позволяет связать декларативную и архитектурную части описания объекта моделирования. После ключевого слова **architecture** указывается уникальный идентификатор **identifier** данной архитектуры.

После ключевого слова **begin** следуют параллельные операторы, задающие в алгоритмическом виде функционирование описываемой архитектуры объекта.

Завершается описание архитектуры объекта ключевым словом **end**, за которым следует слово **architecture** (по стандарту VHDL'93); потом указывается идентификатор описанной архитектуры объекта (рекомендуется).

Объект моделирования может иметь несколько различных видов архитектурного описания, однако одновременно в модели может использоваться только один из них. Выбор конкретного вида архитектурного описания осуществляется с использованием конфигурационной спецификации, о которой будет сказано в дальнейшем.

Вариант архитектурного описания для сумматора, декларативная часть которого представлена ранее в листинге 3.2, приведен в листинге 3.3.

Листинг 3.3

```
architecture abstract of adder is
begin
add_a_b_c: process(a,b,c) is
begin
  sum<=a+b+c;
end process add_a_b_c;
end architecture abstract;
```

Библиотеки

Для облегчения процесса проектирования описание объекта моделирования и различные варианты описания его архитектуры обычно размещают в библиотеках (они организованы как отдельные файлы). Это позволяет редактировать описание одних объектов, не затрагивая файлы, в которых расположены описания других объектов. Кроме того, библиотеки могут быть

использованы в различных проектах, что позволяет лучше организовать повторное использование уже разработанных объектов.

В библиотеках размещают описания объектов, которые могут использоваться одним или несколькими пользователями в рамках одного или нескольких проектов. В библиотеках также могут размещаться описания констант, переменных, типов, процедур, функций, а также декларации конфигураций. Эти описания тоже могут формироваться в отдельные пакеты в соответствии с тематикой, а потом объединяться в библиотеки.

Если описание архитектуры включает объекты, размещенные в библиотеках, то непосредственно перед описанием архитектуры необходимо указать имена библиотек, которые используются. Это имеет следующий синтаксис:

```
library library_name {,...}
```

Здесь `library_name` — идентификатор, имя библиотеки.

Примечание

В OrCAD Express библиотеки реализуются как файлы с расширением vhd, имя библиотеки – имя файла.

Например, пусть имеются библиотеки с именами `widget_cells` и `wasp_lib`. Тогда, если модуль использует некоторые объекты из указанных библиотек, это можно показать на примере листинга 3.4.

Листинг 3.4

```
library widget_cells, wasp_lib;
architecture cell of filter is
begin
clk_pad: entity wasp_lib.in_pad -- используется элемент с именем in_pad
                                -- из библиотеки wasp_lib
  port map...
accum: entity widget_cells.reg32
  port map...
end architecture cell;
```

Для того чтобы в тексте модуля каждый раз не указывать имя библиотеки часто используемого объекта, можно описать *условную ссылку*. Она имеет следующий синтаксис:

```
use library_name.(identifier | all)
```

Здесь `identifier` — имя объекта из библиотеки `library_name`.

Предыдущий пример (листинг 3.4) можно переписать в виде, представленном листингом 3,5:

Листинг 3.5

```
library widget_cells, wasp_lib;
use widget_cells.reg32;
architecture cell of filter is
begin
clk_pad: entity wasp_lib.in_pad -- используется элемент с именем in_pad
                                -- из библиотеки wasp_lib
  port map...
accum: entity reg32 -- используется элемент с именем reg32 из
                    -- из библиотеки widget_cells
  port map...
end architecture cell;
```

Если в описании указано ключевое слово **all**, то все объекты, содержащиеся в библиотеке, можно использовать в теле использующего эту библиотеку объекта без указания имени библиотеки (все они становятся непосредственно видимыми).

Пакеты

При построении полноразмерных моделей проектируемых устройств мы имеем дело с большим числом программных объектов на языке VHDL. Их описания приходится многократно использовать как внутри одной модели, так и в разных моделях. Механизм пакетов позволяет сгруппировать некоторые описания в единую совокупность — пакет, который далее может многократно использоваться в проектах. Пакет может использоваться как в различных частях одной модели, так и в разных моделях.

Обычно имеется в виду, что описания, объединяемые в пакет, некоторым образом логически между собой связаны, имеют некий содержательный критерий для их объединения. Однако это, конечно, вопрос стиля программирования, а не собственно языка VHDL.

Механизм пакетов позволяет объединить описания типов, констант, процедур, функций, компонентов. Сами эти описания выполняются так же, как в различных частях объекта моделирования.

Описание пакета состоит из декларативной части и тела пакета.

Описание декларативной части пакета имеет следующий синтаксис:

```
package name is
  {package_declarative_item}
end [package] [name];
```

Описание тела пакета имеет следующий синтаксис:

```
package body name is
  {package_body_declarative_item}
end [package body] [name];
```

Описание пакета может состоять только из декларативной части (например, если он содержит только описание типов, констант, переменных). Пример такого описания — в листинге 3.6:

Листинг 3.6

```
package cpu_types is
constant word_size:positive:=16;
subtype word is bit_vector (word_size-1 downto 0);
end package cpu_types;
```

Описание одного или нескольких пакетов размещается в библиотеке. Обращение к объекту, расположенному в пакете, имеет следующий синтаксис:

```
library_name.packet_name.object_name,
```

где: `library_name` — имя процедуры, `packet_name` — имя пакета, `object_name` — имя объекта.

Некоторую специфику в пакетах имеет описание констант. Для констант в декларативной части пакета может использоваться сокращенное описание, содержащее только имя и тип константы. Оно имеет следующий синтаксис:

```
constant name: type_name;
```

Константе, описанной таким образом, в теле пакета необходимо присвоить значение.

Например, если в декларативной части пакета имеется описание:

```
constant max_size: positive;
```

то в теле пакета необходимо выполнить полное описание:

```
constant max_size:positive := 40;
```

Этот механизм применяется, когда в пакете важно определить не столько конкретное значение константы, сколько диапазон значений, которые могут ей соответствовать.

Однако область применения констант, описанных таким образом, имеет ограничения. Их значения являются неопределенными на этапе компиляции, когда программа моделирования анализирует исходную программу на языке VHDL. Поэтому они могут использоваться только в тех местах конструкций языка VHDL, где могут использоваться переменные. Например, константы с сокращенным описанием не могут быть использованы в выражениях после ключевого слова **when** в операторе **case**.

Описание тела пакета может содержать описание дополнительных типов, подтипов, констант и подпрограмм. Все описания, выполненные в декларативной части пакета, автоматически видны в его теле.

Описания *сигналов* в тело пакета включаться не могут.

Для того чтобы каждый раз при использовании объектов, расположенных в пакетах, не указывать их полное имя можно (как и для библиотек) использовать *условные ссылки.* В этом случае условная ссылка имеет следующий синтаксис:

```
use library_name.packet_name.(identifier | character_literal
                              |operator_symbol | all);
```

где: `library_name` — имя библиотеки. Далее следует имя пакета packet_name, затем имя `identifier` внутри пакета.

Описание поведения объекта моделирования

Сигнал, как элемент программы на VHDL, характеризуются парой: значение + время (модельное), в котором сигнал имеет это значение. В программе на VHDL сигнал меняет свое состояние в результате выполнения специальных операторов присваивания значений сигналам, определяющих новое значение сигнала и момент модельного времени, в который это изменение произойдет. Несколько утрируя, можно сказать, что все функционирование объекта, специфицируемого на языке VHDL, сводится к формированию этих пар (значение + время) для сигналов, определенных в системе.

Цифровые устройства функционируют непрерывно и параллельно. Множество компонентов схемы работают одновременно, формируя значения определенных сигналов. В результате в моделируемом устройстве параллельно, в физическом времени, происходит множество изменений состояний сигналов. Соответственно, и в модели устройства изменения значений соответствующих сигналов должны производится параллельно, в модельном времени. Это определяет построение VHDL как языка, параллельного по своей природе.

Тело архитектурного описания специфицируемого объекта, заключенное между **begin** и **end**, содержит совокупность параллельных операторов, иначе говоря операторов, выполняющихся параллельно друг с другом. Работая на языке VHDL, мы постоянно должны помнить, что работаем на языке параллельного программирования.

Здесь можно провести аналогию с языком параллельного программирования OKKAM для транспьютеров. В OKKAM также базовым понятием является процесс, работающий параллельно с другими процессами. Для того чтобы задать последовательное выполнение действий, исходно параллель-

ные процессы в OKKAM выстраиваются в нужную последовательность с помощью специальной языковой конструкции SEQ ("выполнять последовательно" [23]).

Последовательность записи параллельных операторов в теле архитектурного описания значения не имеет. Порядок выполнения (или, как говорят обычно, в параллельных вычислениях — порядок срабатывания параллельных операторов) определяется не порядком их текстуальной записи операторов между **begin** и **end**, а другими правилами. Основным принципом здесь является управление от потока изменений сигналов, входных для параллельного оператора. Событие изменения значения сигнала, являющегося входным для параллельного оператора, запускает срабатывание данного оператора (т. е. — акт выполнения оператора). Для параллельного оператора типа **process** *(см. следующий раздел)* имеется возможность явно указывать сигналы, изменение которых запускает процесс. Для других параллельных операторов присваивания значений сигналам изменение любого входного сигнала инициирует их срабатывание.

В общем формате описания архитектуры объекта моделирования, секция параллельных операторов {concurent_statement} содержит один или несколько параллельно выполняющихся операторов, с помощью которых задается зависимость выходных сигналов от входных, как по значению (состоянию), так и по временным соотношениям. В языке VHDL введен целый ряд параллельных операторов, основными из которых являются процессы и операторы параллельного присваивания значений сигналам.

Процессы для описания архитектуры

Одной из форм параллельных операторов являются *процессы*. Процессы являются базовыми конструкциями для задания поведения описываемой архитектуры с учетом параллельности выполняемых действий.

Процессы выполняются параллельно относительно друг друга, а действия внутри тела процесса выполняются последовательно. Выполнение процессов осуществляется в соответствии с изменениями значений сигналов.

Описание процесса имеет следующий синтаксис:

```
[process_label:] process [(signal_name {,...})] [is]
  {process_declarative_item}
begin
  {sequential_statement}
end process [process_label];
```

Список сигналов в скобках после ключевого слова **process** называется *списком чувствительности*. Если в описании процесса задан список чувствительности, то процесс активируется (начинается выполнение действий,

описанных внутри процесса) при изменении значения любого из этих сигналов. Если список чувствительности отсутствует, то процесс активируется при изменении любого сигнала в модели, если не указано иное с помощью команд ожидания внутри самого процесса.

В секцию `process_declarative_item` могут включаться описания локальных констант, типов и переменных, используемых в описываемом процессе.

Сигналы не могут быть описаны как локальные данные процесса.

Когда в ходе моделирования процесс активируется, его выполнение начинается с первого оператора, указанного в теле процесса (в секции `sequential_statement`). В этой секции располагаются последовательные операторы, которые определяют будущие значения сигналов на базе текущих значений. Кроме того, в теле процесса, как и в обычных последовательных программах на языках высокого уровня, для изменения последовательности выполнения могут использоваться управляющие операторы (описаны в предыдущей главе).

Наличие метки процесса `process_label` (фактически — имени процесса) облегчает отладку программы на VHDL, включающей множество процессов.

Последовательный оператор присваивания значения сигналу

Последовательные операторы присваивания значения сигналу могут располагаться только внутри тела процесса, в секции `{sequential_statement}`. Такие операторы имеют следующий синтаксис:

```
name <= [delay_mechanism] (value_expression [after time_expression]),
{,...};
```

Оператор присваивания значения сигналу (указывается именем `name`) включает в себя как определение *нового значения* сигнала (задается выражением `value_expression`), так и определение *момента времени*, в который сигнал примет это новое значение (задается конструкцией **after** `time_expression`).

Время изменения значения сигнала, задаваемое в этом операторе, определяется относительно модельного времени. В момент запуска модели модельное время равно 0. Оно увеличивается дискретно, по мере возникновения событий в модели. Эта техника носит название моделирование дискретных событий.

Когда выполняется оператор присваивания нового значения сигналу, указанное в нем время (определяется выражением `time_expression` после **after**) добавляется к текущему модельному времени; этим и определяется момент мо-

дельного времени, при наступлении которого сигнал примет новое значение. Выражение time_expression должно вычислять значение типа **time**.

Для того чтобы осуществить изменение сигнала в нужное время, программа моделирования автоматически планирует транзакцию на время принятия сигналом нового значения. Когда в модели наступает момент времени, на который запланирована эта транзакция, она выполняется и сигнал принимает новое значение. В том цикле моделирования, в котором он принимает новое значение, сигнал считается *активным*.

Например, возьмем оператор y<= **not** x **after** 5 ns. Сигнал y примет значение, инверсное значению сигнала x (на момент выполнения данного оператора); изменение y произойдет через 5 нс. после момента модельного времени, соответствующего выполнению этого оператора.

Один оператор позволяет задать несколько значений, которые сигнал будет принимать последовательно в указанные моменты времени. Список моментов времени должен быть возрастающим, чтобы порядок следования планируемых на его основе транзакций соответствовал порядку их выполнения. Например:

```
y<= not x after 5 ns, x after 10 ns;
```

Если в момент выполнения этого оператора модельное время равно 7 нс., то в момент времени 13 нс. y примет значение равное not x, а в момент времени 17 нс. — значение равное x (где x — в момент времени 7 нс.).

Процесс, например, предназначенный для генерации тактового сигнала, можно проиллюстрировать листингом 3.7.

Листинг 3.7

```
entity my_clk is
...
generic (t:natural:=2 ns);
end entity;

architecture rtl of  my_clk is
signal clk:std_logic:='0';
begin
L_clk: process (clk)
begin
if clk='0' then
  clk<='1' after t,'0' after 2*t
end if;
end process;
```

Выполнение этого процесса начинается каждый раз при изменении значения сигнала `clk`, стоящего в списке чувствительности. Если сигнал `clk='0'`, то выполняется последовательный оператор присваивания значения сигналу. Пусть в момент времени `t1` сигнал `clk` получил значение `'0'`. Тогда в результате выполнения оператора присваивания на момент времени `t1+t` будет запланирована транзакция, присваивающая сигналу `clk` значение `'1'`, а на момент времени `t1+2*t` будет запланирована транзакция, присваивающая сигналу `clk` значение `'0'`. Выполнение каждой из этих транзакций приведет к активизации процесса `L_clk`. В результате выполнения второй транзакции сигнал примет значение, удовлетворяющее условию, и оператор присваивания будет выполнен снова. Транзакция (и, соответственно, оператор присваивания) будет выполняться в момент времени `t1+2*t`, она приведет к планированию транзакций на время `t1+2*t+t=t1+3*t` и `t1+2*t+2*t=t1+4*t`. Таким образом, процесс позволяет сгенерировать симметричный тактовый сигнал с полу периодом, равным t. Временная диаграмма сигнала тактирования при t=2 представлена на рис. 3.1.

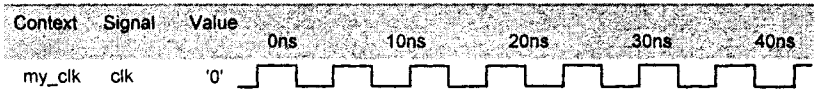


Рис. 3.1. Временная диаграмма сигнала тактирования (для примера с листинга 3.7)

Внутри тела процесса может быть несколько операторов присваивания значений сигналам, соответствующих разным выходам элемента схемы, но для каждого сигнала в теле процесса может стоять только один оператор присваивания значения сигналу. Иными словами, *в теле процесса для одного сигнала может быть задан только один источник сигнала.*

Задержки в модели устройства

Представление цифрового устройства на языке VHDL требует аккуратного и корректного представления в модели устройства задержек распространения сигналов в схеме. Язык VHDL включает различные модели задержек, отражающие разные аспекты функционирования реальных схем, которые связаны с временными факторами.

Описание механизма задержек, `delay_mechanism`, в операторах присваивания значений сигналам имеет следующий синтаксис:

```
[reject time_expression] inertial | transport
```

Инерционная задержка

Цифровые схемы обладают определенной инерционностью. Для формирования сигнала на выходном контакте, в ответ на изменение входного сигнала, требуется некоторое количество энергии и определенное время. Чтобы на выходе сформировался устойчивый сигнал, входной сигнал должен продержаться в новом состоянии не менее некоторого промежутка времени. Если же входной сигнал не простоит в этом состоянии нужное время, то вызванные им изменения состояния схемы не успеют распространиться до рассматриваемого выхода, и мы не можем быть уверены в формировании на данном выходе схемы ожидаемого значения. Например, если вентиль имеет задержку 6 нс., то любой импульс меньшей длительности (скажем, 4 нс.), не пройдет через него и не появится на выходе вентиля. Про импульсы, длительность которых меньше задержки, необходимой для распространения сигнала через схему, говорят, что они отбрасываются, т. е. отфильтровываются схемой. Именно такая логика работы с распространением сигналов по схеме используется в семантике языка VHDL (по умолчанию).

Для представления этого вида задержек распространения сигналов в языке VHDL используется понятие *инерционной задержки* (inertial delay), в операторе присваивания — ключевое слово **inertial**.. По умолчанию, если в программе не указано иное, при спецификации задержек имеется в виду именно модель инерционной задержки. Минимальная длительность сохранения установившегося значения входного сигнала (иными словами — минимальная длительность импульса на входе) по умолчанию считается равной времени распространения сигнала с этого входа до указанного выхода.

До тех пор, пока входной сигнал изменяется не чаще, чем время, указанное в секции **after**, изменения выходного сигнала происходят в соответствии с изменениями входного, но с учетом указанной задержки. Если же изменения входного сигнала происходят чаще, чем время, указанное в секции **after**, они игнорируются.

Однако в реальных схемах такая прямая зависимость между задержкой распространения сигнала и минимальной длительностью входного импульса выполняется не всегда, даже для простых схем. На практике длительность импульсов, которые отфильтровываются схемой, зависит от многих факторов (физического проекта схемы в СБИС, параметров процесса изготовления СБИС и др.), и определить их точные значения бывает непросто.

Если разработчик схемы имеет более детальную информацию об инерционных задержках и минимальных длительностях импульсов на входах, он может явно указать минимальную длительность импульсов, отличную от спецификированной задержки формирования выходного сигнала. Когда минимальная длительность входного сигнала, приводящая к изменению выходного сигнала, меньше заданной задержки, для ее указания используется секция **reject**.

Например, в операторе присваивания значения сигналу можно указать[1]:

```
z <= reject 3 ns inertial (x xor y) after 7 ns;
```

Здесь минимальная длительность импульса на входах x и y установлена равной 3 нс., в то время как задержка формирования выходного сигнала z равна 7 нс. При длительности, меньшей 3 нс., импульс отфильтровывается, отбрасывается системой моделирования и не приводит к формированию нового значения выходного сигнала z.

Если в операторе присваивания значения сигналу присутствует секция **inertial** и несколько секций **after**, то секция **inertial** применяется только к первой секции **after**, а к остальным секциям **after** применяются правила работы с транспортными задержками (описаны в следующем разделе).

Транспортная задержка

И в проектах, и в реальных устройствах на СБИС, на определенных этапах проектирования мы сталкиваемся с другими видами задержек, отличными от задержек инерционных. Как видно из предыдущего раздела, для механизма инерционных задержек характерно понятие минимальной длительности импульса; импульсы меньшей длительности отфильтровываются, отбрасываются.

Однако в проектах устройств на СБИС возникают ситуации, когда это ограничение мешает, не отражает нужную ситуацию в моделируемом устройстве. Часто в модели необходимо, чтобы изменения сигналов любой длительности не отбрасывались, а отрабатывались системой моделирования и влияли на формирование выходных сигналов.

Здесь можно указать два примера такой ситуации.

Во-первых, это учет задержек на распространение сигналов по линиям связи. Распространение сигналов по линиям между логическими элементами имеет некоторую конечную задержку, пропорциональную длине линии. В современных субмикронных интегральных технологиях, по мере уменьшения проектных норм, этот фактор оказывает все большее влияние, задержки в линиях начинают преобладать над задержками в логических элементах. Понятно, что в таких схемах, при детальном моделировании проектируемого устройства, задержками распространения сигналов по линиям связи нельзя пренебрегать, их надо отражать в модели.

Однако задержки в линиях связи по своей природе существенно отличаются от задержек в логических элементах. Они имеют малую инерционность, могут передавать импульсы очень малой длительности. Обычно линию связи

---

[1] Только в VHDL'93. В стандарте VHDL'87 конструкция указания минимальной длительности импульса отсутствует.

рассматривают как среду с конечной задержкой, но с нулевой инерционностью, способную передавать сигналы без ограничений на минимальную длительность импульса.

Для отражения такого рода задержек, в VHDL введен другой вид задержек — *транспортные задержки* (transport delays). В отличие от инерционных задержек, транспортные задержки не накладывают ограничений на минимальную длительность импульса, не отфильтровывают короткие входные импульсы, а пропускают в схему любые входные сигналы.

Второй класс ситуаций, когда удобны как раз транспортные задержки, возникает при проектировании и моделировании устройств на верхнем уровне иерархии. Здесь задержка формирования сигналов на выходе сложного устройства может быть весьма значительна, однако это не повод отфильтровывать короткие импульсы входных сигналов. Скажем, если мы моделируем устройство с памятью, время доступа к которой 50 нс., то это вряд ли будет основанием требовать минимальную длительность импульса для всех сигналов, используемых при чтении из памяти, равную 50 нс.

Кроме того, при анализе устройства, особенно при поведенческом описании его архитектуры, у нас часто нет достоверной информации о минимальных длительностях сигналов. Она появляется только на последующих этапах проектирования системы на СБИС.

В таких случаях при построении модели на VHDL предпочтительно пользоваться транспортными, а не инерционными задержками.

Разрешение неоднозначности установления значения сигнала

При использовании механизма задержек могут возникать неоднозначные ситуации. Значение одному и тому же сигналу могут устанавливаться несколькими операторами присваивания значений. Эти операторы могут устанавливать разные значения, определять их с разными задержками. В результате в программе на VHDL может создаваться неоднозначная ситуация, когда на каком-то интервале модельного времени значения, устанавливаемые разными операторами присваивания, входят в противоречие друг с другом.

В семантике языка VHDL приняты специальные правила для разрешения такой неоднозначности. В основе этих правил лежит сопоставление между последовательностью выполнения операторов присваивания значений сигналам и заданных ими моментов изменения сигналов. Знание этих правил важно для правильного понимания и отладки модели устройств, написанных на VHDL.

В VHDL принято, что если возникает ситуация, когда транзакция изменения значения сигнала, порожденная позже, планируется на более раннее

время, то транзакция, порожденная раньше по ходу исполнения программы, но запланированная на более позднее время, уничтожается.

Пример коллизии, возникающей в ситуации, когда одному сигналу присваиваются разные значения, последовательно по ходу выполнения программы, несколькими операторами, демонстрирует листинг 3.8.

Листинг 3.8

```
process (a)
  constant T_01:time:=800 ns;
  constant T_10:time:=500 ns;
begin
  if a='1' then
    z<=transport a after T_01;
  else
    z<=transport a after T_10;
end if;
end process;
```

Пусть входной сигнал a ведет себя следующим образом:

```
a='0' с 0 ns по 200 ns
a='1' с 200 ns по 400 ns
a='0' с 400 ns по 1100 ns
```

В момент времени 200 будет запланирована транзакция, которая в момент модельного времени 1000 должна будет установить значение сигнала z в 1. В момент модельного времени 400 будет запланирована транзакция, которая должна будет в момент времени 900 установить значение сигнала z в 0. Поскольку эта транзакция планируется на модельное время, меньшее чем предыдущая, то предыдущая транзакция будет уничтожена. Таким образом, если значение сигнала z в начальный момент модельного времени было равно 0, оно не претерпит никаких изменений. Временная диаграмма, соответствующая этому примеру, представлена на рис. 3.2.

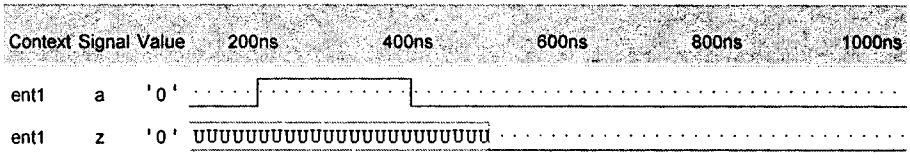


Рис. 3.2. Временная диаграмма, соответствующая примеру листинга 3.8

На этой диаграмме до момента времени 500 выходной сигнал имеет неопределенное значение. В момент времени 500 ему присваивается значение 0, которое остается без изменений.

На эти правила разрешения неоднозначности через удаление ранее спланированных транзакций накладываются и ограничения на длительность импульсов, устанавливаемые инерционными задержками.

Когда в очередной раз выполняется оператор присваивания нового значения сигналу, содержащий секцию **inertial**, выполняются следующие действия. Например, пусть выполняющийся оператор добавляет новую транзакцию изменения некоторого сигнала на момент времени `tnew` при ограничении ширины импульса, равном `tr`. Сначала проверяется список отложенных транзакций, изменяющих этот сигнал. Если в нем присутствуют транзакции, запланированные на время большее, чем `tnew`, то они сразу же удаляются. Затем новая транзакция добавляется в список. После этого проверяются транзакции, запланированные на интервал времени от (`tnew` — `tr`) до `tnew`. Из них в списке оставляются только те транзакции, которые непосредственно предшествуют добавляемой (между ними и добавляемой транзакцией нет других транзакций) и которые устанавливают сигнал в то же значение, что и добавляемая. Таким образом, если время между изменениями сигналов оказывается меньше, чем `tr`, то соответствующее значение не отражается на выходе.

Другой пример приведен в листинге 3.9.

Листинг 3.9

```
process (a)
begin
  y <= inertial not a after 3 ns;
end process;
```

Пусть входной сигнал `a` изменяется в модельном времени следующим образом:

```
a='0' от 0 ns до 1 ns
a='1' от 1 ns до 6 ns
a='0' от 6 ns до 8 ns
a='1' от 8 ns до 12 ns
```

В момент модельного времени 1 будет запланирована транзакция установки `y` в '0' на момент модельного времени 4. (В момент времени 1 сигнал `a` меняет свое значение на '1'. В соответствии с указанной в операторе задержкой 3 нс., изменение сигнала `y` планируется на момент времени 1+3=4 нс. К моменту времени 4 сигнал `a` по-прежнему будет иметь значение 1, поэтому транзакция будет выполнена. В момент времени 6 сигнал a поменяет

свое значение на '0', и будет запланирована транзакция установки y в '1' на момент модельного времени 9. Однако в момент времени 8 сигнал a поменял свое значение. В результате транзакция, запланированная на момент времени 9, будет отменена. На рис. 3.3 представлена иллюстрация данной ситуации.

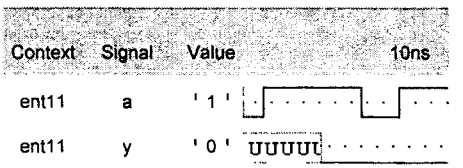


Рис. 3.3. Временная диаграмма к листингу 3.9

Третий пример. Пусть в момент модельного времени t2=10 нс. выполняется оператор

```
s <= reject 5 ns inertial '1' after 8 ns.
```

При этом пусть список транзакций изменения сигнала s, уже запланированных к началу выполнения этого оператора, имеет вид:

```
11 ns – ('1'); 12 ns – ('x'); 14 ns – ('1'); 15 ns – ('0');
16 ns – ('1'); 17 ns – ('1'); 20 ns – ('1'); 25 ns – ('0');
```

После выполнения оператора список запланированных транзакций примет вид:

```
11 ns – ('1'); 12 ns – ('x');); 16 ns – ('1'); 17 ns – ('1');
18 ns – ('1').
```

Транзакции, запланированные на моменты 20 и 25 нс. на оси модельного времени, будут удалены из списка, поскольку они запланированы на время большее, чем новая транзакция, запланированная на момент 18 нс. и добавленная выполняющимся оператором. Транзакции, запланированные на 14 и 15 нс., удалены из списка, так как они попадают в интервал от 13 (13=18 – 5, где: 5 — значение, стоящее после **reject**) до 18 нс. и значение, устанавливаемое транзакцией, запланированной на 15 нс. (транзакцией, непосредственно предшествующей вновь добавляемой), не совпадает со значением, устанавливаемым новой транзакцией. Все транзакции, находящиеся в этом интервале левее самой правой транзакции (и она сама в том числе), значение которой не совпало со значением, устанавливаемым новой транзакцией, должны быть удалены, даже если их значения и совпадают со значением добавляемой транзакции, как и произошло в этом примере.

Дельта-задержка сигналов

Особая ситуация складывается в модели устройства на языке VHDL, если оператор присваивания нового значения сигналу устанавливает нулевую за-

держку. Например, если в операторе присваивания нового значения сигналу отсутствует секция **after**, то считается, что задержка изменения сигнала составляет 0 нс., т. е. сигнал принимает новое значение в текущее время моделирования.

В реальных устройствах такие ситуации невозможны. Новое значение сигнала y, формируемое на основе текущего значения некоторого сигнала x, всегда появляется с некоторой конечной задержкой. Любое событие, любое значение сигналов в текущий момент проявится в изменениях каких-либо сигналов не мгновенно, а по прошествии некоторого, может быть, очень небольшого, но никак не нулевого времени.

В модели мы можем попытаться этими временными задержками пренебречь (например, когда важно только функциональное поведение модели, а ее детальное временное поведение не рассматривается). Часто, на определенных этапах проектирования системы, у нас просто нет еще достоверной информации о задержках в компонентах схемы, и тогда нашей задачей является соблюдение функциональной корректности модели устройства.

Проблема, которая возникает при нулевых задержках в изменениях сигналов, связана чисто с внутренними проблемами организации системы моделирования.

Моделирование идет по дискретам модельного времени. Выполняя моделирование в момент $t1$ модельного времени, система моделирования производит все изменения сигналов, запланированные на текущий момент времени, на основании имеющихся значений сигналов устройства, наблюдаемых в начале текущего цикла моделирования. Выполнив все заданные для этого момента изменения сигналов, программа моделирования заканчивает текущий цикл моделирования и переходит к следующему моменту модельного времени, который хотя бы на единицу больше текущего.

Однако если, по программе на VHDL, сигналы поменялись с нулевой задержкой, изменения должны произойти в тот же момент $t1$ модельного времени. Система же моделирования их не увидит, так как уже закончила цикл моделирования для момента $t1$. Таким образом, окажутся нарушенными не только временные соотношения, но и логика работы схемы.

Для решения проблемы, типовой для систем дискретного моделирования, в семантике языка VHDL введен специальный механизм разрешения подобных коллизий. Суть его в следующем: имеется в виду, что система моделирования, закончив текущий цикл моделирования для момента $t1$ модельного времени, не сразу переходит к следующему моменту модельного времени $t2 > t1$ (например, к $t2 = t1 + 1$). Она проверяет, имеются ли изменения сигналов, вновь запланированные на момент $t1$. Это может произойти при указании в программе на VHDL нулевых задержек. Если выявлены новые изменения сигналов на тот же момент $t1$ модельного времени, то система моделирования выполняет новый цикл моделирования, отрабатывая эти

изменения. Новые изменения, в свою очередь, могут привести к изменению еще каких-то сигналов, что при нулевой задержке снова приведет к появлению еще каких-то сигналов, изменения которых запланированы на $t1$. Значит, система выполнит еще один цикл моделирования, не сдвигаясь на момент $t2$ модельного времени. И так далее, пока не будет определено, что по результатам текущего цикла моделирования не появилось новых сигналов, запланированных на момент $t1$, после чего система переходит к моделированию момента $t2$ модельного времени.

Если первый из рассмотренных нами циклов работы системы моделирования соответствовал моменту $t1$ модельного времени, то каким точкам на оси модельного времени соответствуют следующие циклы? С точки зрения модели на VHDL, которая указывает, что некоторый сигнал изменяется с нулевой задержкой, мы бы сказали, что система моделирования "крутится", оставаясь в том же моменте $t1$ модельного времени.

Однако можно рассматривать ситуацию и по-иному. Временная последовательность таких дополнительных циклов отражает, в определенной степени, и зависимости сигналов, логику влияния одних сигналов на формирование других, и последовательность этих влияний. Выстроить в нашем представлении эту последовательность циклов работы системы моделирования позволяет введенное в языке VHDL понятие *дельта-задержки* (delta-delay); обозначим ее Δ. Дельта-задержке не приписывается никакого числового значения. Эта абстракция используется только для упорядочивания последовательности событий в модели и отрабатывающих их циклов работы системы моделирования.

В момент $t1$ будут сделаны все изменения сигналов, которые исходно были запланированы на $t1$. В момент $t1+\Delta$ будут выполнены изменения, инициированные в момент $t1$, для которых указана нулевая задержка. В момент $t1+2\Delta$ будут выполнены изменения с нулевой задержкой, инициированные в момент $t1+\Delta$, и т. д. Эти циклы называют *дельта-циклами* (delta-cycles). И только когда все возникающие на тот же момент $t1$ изменения сигналов будут отработаны, система моделирования перейдет к циклу обработки момента $t2$.

Дельта-задержка — задержка условная, не существующая ни в реальных схемах, ни на оси модельного времени, но позволяющая отразить зависимость изменения сигналов при нулевых задержках и соответствующую им последовательность дельта-циклов — циклов внутренней работы системы моделирования. На формируемых системой моделирования временных диаграммах, на оси модельного времени дельта-циклы не отражаются. Все соответствующие значения сигналов выстраиваются на диаграммах в одной вертикали, на момент $t1$. Однако, если в процессе присутствует оператор присваивания, в правой части которого используется сигнал, значение которого определяется в этом же процессе, дельта-задержки уже можно наблюдать.

Рассмотрим этот механизм на следующем примере, показанном в листинге 3.10.

Листинг 3.10

```
Library IEEE;
Use IEEE.std_logic_1164.all;
Entity ent1 is
Port (in1,in2:in std_logic;
      Out1:out std_logic);
End entity ent1;

Architecture rtl of ent1 is
Signal s1:std_logic;
Begin
Process (in1,in2)
Begin
s1<=in1 and in2;
Out1<=s1 ;
End process;
End architecture;
```

Пусть оператор присваивания нового значения сигналу `s1` выполняется в момент времени $t1$. В результате, на момент времени $t1$ планируется транзакция изменения значения `s1`. Однако, в результате дельта-задержки, сигналу `out1` присваивается прежнее значение сигнала `s1`. Только при следующем выполнении процесса сигналу `out1` будет присвоено значение сигнала `s1`, которое тот получил в момент времени $t1$. Это иллюстрирует рис. 3.4.

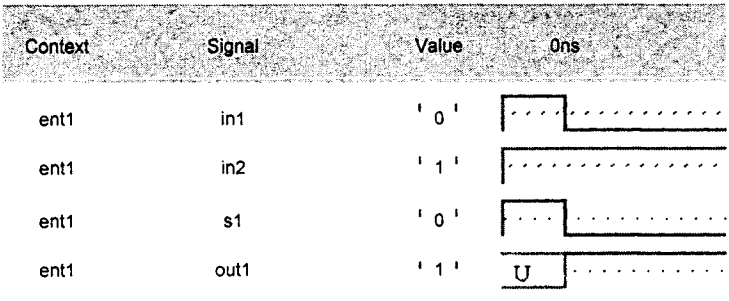


Рис. 3.4. Пример временной диаграммы функционирования объекта, описанного в листинге 3.10

Параллельные операторы присваивания значения сигналу

Параллельные операторы присваивания (concurrent assignment statement — CSA) включают в себя простой параллельный оператор присваивания и ряд параллельных операторов присваивания — оператор условного присваивания и оператор селективного присваивания.

В отличие от последовательных операторов присваивания, эти операторы могут использоваться не только внутри процессов, но в любом месте тела архитектурного описания.

Простой параллельный оператор присваивания

Ранее мы представили формат последовательного оператора присваивания, который используется внутри тела процессов. Простой параллельный оператор присваивания имеет практически тот же формат, однако указывается непосредственно в архитектурном описании, между **begin** и **end**.

Например:

```
architecture hadd1 of SM is
begin
s<=(a xor b) after 4 ns;
c<=(a and b) after 4 ns;
end architecture hadd1;
```

В этом фрагменте заданы два простых параллельных оператора, активизируемые при изменениях любого из входных сигналов a и b. Оба оператора активизируются и выполняются параллельно в модельном времени.

Оператор условного присваивания

Оператор условного присваивания имеет следующий синтаксис:

```
name<=[delay_mechanism] {waveform when boolean_expression else}
        waveform [when boolean_expression];
```

Этот оператор позволяет определить, которая из форм, в зависимости от значений логических условий, будет присвоена сигналу. Примеры приведены в листингах 3.11 и 3.12.

Листинг 3.11

```
zmux:   z<=d0 when sel1='0' and sel0='0' else
     d1 when sel1='0' and sel0='1' else
     d2 when sel1='1' and sel0='0' else
     d3 when sel1='1' and sel0='1';
```

Этот фрагмент аналогичен фрагменту листинга 3.12.

Листинг 3.12

```
zmux: process
begin
  if    sel1='0' and sel0='0' then z<=d0;
  elsif sel1='0' and sel0='1' then z<=d1;
  elsif sel1='1' and sel0='0' then z<=d2;
  elsif sel1='1' and sel0='1' then z<=d3;[1]
end if;
wait on d0,d1,d2,d3,sel0,sel1;
end process zmux;
```

В последнем примере список чувствительности для процесса отсутствует. Вместо него используется конструкция **wait on** с последующим списком сигналов. Это работает следующим образом: процесс выполняется один раз до этой секции, далее он приостанавливается, пока не произойдет изменение какого-либо из сигналов в списке чувствительности, а затем процесс выполняется сначала. Более подробно использование оператора **wait** будет рассмотрено далее (в этой же главе).

На рис. 3.5 представлен пример диаграммы работы zmux.

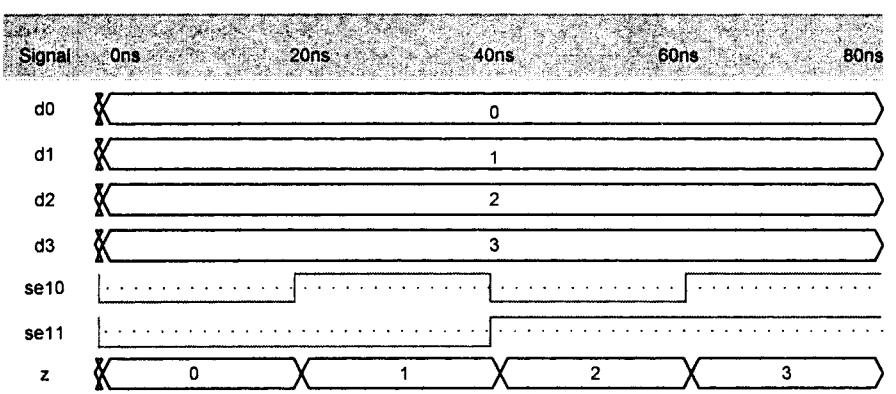


Рис. 3.5. Пример диаграммы работы zmux

В записи оператора условного присваивания имеет значение порядок записи специфицируемых условий. Условия вычисляются и проверяются в том порядке, в каком они записаны в операторе. Срабатывает первое же из ус-

[1] Эта строка может быть заменена на else z<=d3;

ловий, для которого обнаружено истинное значение, и на выход будет передано соответствующее ему значение.

Оператор условного присваивания позволяет также использовать описанный выше механизм задержек.

Механизм условных присваиваний в некоторых случаях имеет преимущества перед механизмом процессов. Так, если в последней секции оператора условного присваивания не будет условия, этот оператор в любом случае будет присваивать сигналу какое-либо значение. Процесс же, если ни один из сигналов, включенных в список его чувствительности, не изменит своего значения, никогда не выполнится, в результате чего сигнал может не получить никакого значения. Примеры приведены в листингах 3.13 и 3.14.

Листинг 3.13

```
reset_gen: reset<='1', '0' after 200 ns when extended_reset else
      '1', '0' after 50 ns ;
```

Листинг 3.14

```
reset_gen: process
begin
if extended_reset then
  reset<='1', '0' after 200 ns;
else
  reset<='1', '0' after 50 ns;
end if;
wait;
end process reset_gen;
```

Если за время выполнения модели сигнал `extended_reset` ни разу не изменит своего значения, то процесс (листинг 3.14) не выполнится ни разу, в итоге значение сигнала `reset` останется неопределенным. Напротив, условный оператор (листинг 3.13) присвоит в этом случае ему вторую форму волны (`'1'` — в момент начала моделирования и `'0'` через 50 нс.).

Кроме того, если при выполнении некоторого логического условия не надо выполнять никаких действий с сигналом, вместо формы сигнала пишется слово **unaffected**[1]. Например:

```
st: r<=a when s1='1' and s2='3' else
  unaffected when s1='1' and s2='1' else
  b;
```

[1] Только в VHDL'93.

При равенстве единице обоих сигналов, s1 и s2, оператор не будет менять значение сигнала z.

Оператор селективного присваивания

Оператор селективного присваивания имеет следующий синтаксис:

```
with expression select
name <= [ delay_mechanism] {waveform when choices,}
    waveform when choices;
```

Оператор селективного присваивания позволяет делать выбор между несколькими возможными формами сигнала в зависимости от значения выражения, стоящего в заголовке оператора. Этот оператор чувствителен к изменению всех сигналов (и входящих в выражение заголовка, и входящих в формы, определяющие выходной сигнал).

На выражение в заголовке и выражения, стоящие после **when**, накладываются те же условия, что и на аналогичные выражения в операторе **case**. В операторе селективного присваивания также возможно использование слова **unaffected**[1].

Оператор селективного присваивания схож с оператором условного присваивания, однако имеются существенные различия в их семантике.

В операторе селективного присваивания проверяются все указанные условия, в то время как в операторе условного присваивания условия вычисляются последовательно, в порядке записи, до первого истинного условия. Программист должен следить за тем, чтобы в селективном операторе присваивания все условия были бы различными, но только одно из них — истинным. Кроме того, специфицированный в операторе селективного присваивания набор условий должен быть полным, т. е. охватывать весь набор возможных значений, указанных в условиях сигналов. Избежать в условиях полного перечисления возможных значений проверяемых сигналов можно, используя ключевое слово **others** (все другие значения).

Пример приведен в листинге 3.15.

Листинг 3.15

```
entity sell is port (data_a, data_b, data_c: in std_logic_vector(0 to 3);
      isr : in integer:=1;
      data_out: out std_logic_vector(0 to 3));
  end;
architecture behavior of sell is
```

[1] Только в VHDL'93

```
  begin
    with isr select
    data_out<= data_a when 1,
        data_b when 2,
        data_c when others;
  end;
```

На рис. 3.6 приведен пример временной диаграммы работы такого устройства.

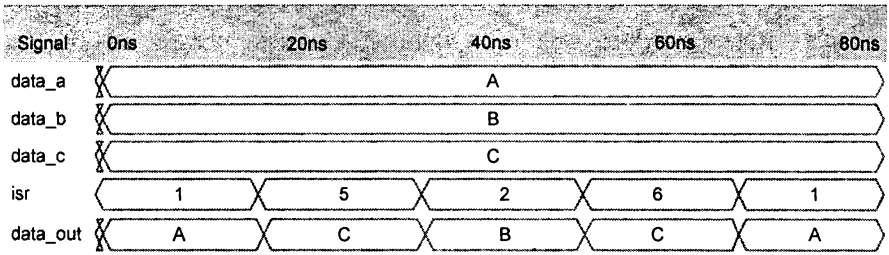


Рис 3.6. Пример временной диаграммы работы устройства, описанного в листинге 3.15

Примеры присваивания значений сигналам

Пусть имеется объект, описание которого имеет вид, представленный в листинге 3.16.

Листинг 3.16

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
entity rtl is
  port(data_a, data_b : in integer range 0 to 20;
       data_out       : out integer range 0 to 20);
end entity rtl;
architecture behaviour of rtl is
begin
process (data_a, data_b)
            begin  data_out <= data_a after 1 ns, data_b after 5 ns;
end process;
end architecture behaviour;
```

Пусть значения входных сигналов определены следующим образом:

```
data_a: 0 -0ns, 1-3ns, 3- 7ns
data_b: 7 – 0ns, 2- 2ns
```

Результат моделирования представлен на рис. 3.7, *а*.

Сигнал `data_out` принимает значение сигнала `data_a` через 1 нс. после его изменения, в моменты времени 4 и 8 нс. В момент модельного времени 7 нс сигнал `data_out` должен принять значение сигнала `data_b`, но этого не происходит, поскольку в этот момент модельного времени изменяется значение сигнала `data_a`, стоящего в списке присваивания первым.

Время, указанное в операторе присваивания после всех ключевых слов **`after`**, отсчитывается от момента изменения объекта, стоящего в списке перед первым словом **`after`**. Для сравнения изменим значения входных сигналов следующим образом:

```
data_a: 0 -0ns, 1-3ns, 3- 7ns
data_b: 7 – 0ns, 2- 2ns
```

Результат моделирования представлен на рис. 3.7, *б*. Теперь выходной сигнал успевает принять новые значения в зависимости не только от сигнала `a`, но и от сигнала `b`.

Если перед словами **`after`** стоят объекты, не являющиеся сигналами, то при первом изменении любого сигнала, находящегося в списке чувствительности процесса, выполняется первая секция оператора присваивания значения, при втором изменении любого сигнала — вторая, и так далее, до последней секции; после чего вновь выполняется первая секция.

Заменим оператор присваивания в предыдущем примере на следующий:

```
data_out<=1 after 1 ns, 5 after 5 ns;
```

Результат моделирования представлен на рис. 3.7, *в*. Из этого рисунка видно, что новые значения на выходе появляются в моменты времени, отсчитываемые от моментов изменения входных сигналов, указанных в списке чувствительности процесса.

Если было запланировано выполнение очередной секции, но прежде произошло изменение одного из сигналов в списке чувствительности, то ее выполнение и переход к обработке последующих секций откладывается до очередного изменения сигнала, который изначально инициировал работу этой секции (рис. 3.7, *г*).

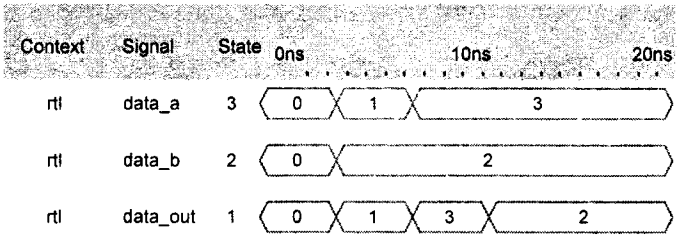
Необходимо учитывать, что оператор присваивания, включающий в себя несколько секций, не является эквивалентом нескольких операторов, каждый из которых включает в себя по одной его секции. В последнем случае будет выполняться только последний из этих операторов. Последовательности

```
data_out <= 1 after 1 ns; data_out <= 5 after 5 ns;
```

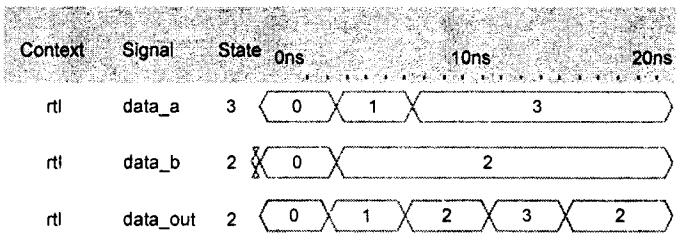
соответствует диаграмма, представленная на рис. 3.8, *а*, а последовательности

```
data_out <= 5 after 5 ns; data_out <= 1 after 1 ns;
```

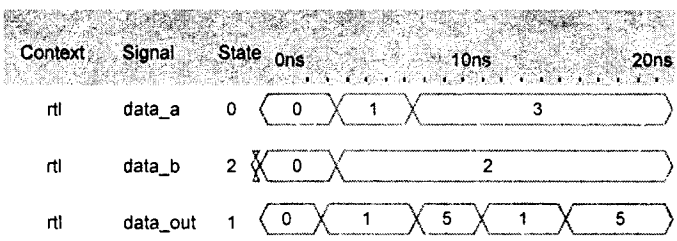
соответствует диаграмма, представленная на рис. 3.8, *б*.



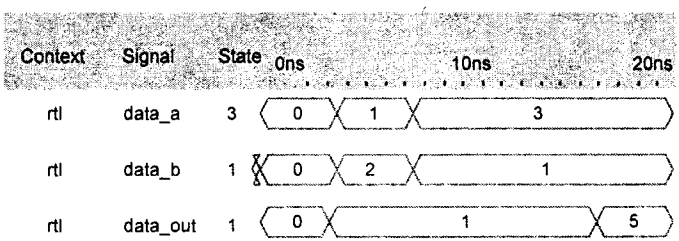
а



б



в



г

Рис. 3.7. Пример временной диаграммы работы устройства, описанного в листинге 3.16, и его модификаций

Если используется механизм **transport**, то при нескольких секциях в операторе присваивания этот механизм используется для каждой секции. Если перед **after** указан объект — сигнал, то относительно него и производятся изменения (в отличие от случая, когда этот механизм не используется).

Например, если в исходном примере оператор присваивания имеет следующий вид:

```
data_out <=transport data_a after 1 ns, data_b after 5 ns;
```

то будет иметь место диаграмма, представленная на рис. 3.8, *в*.

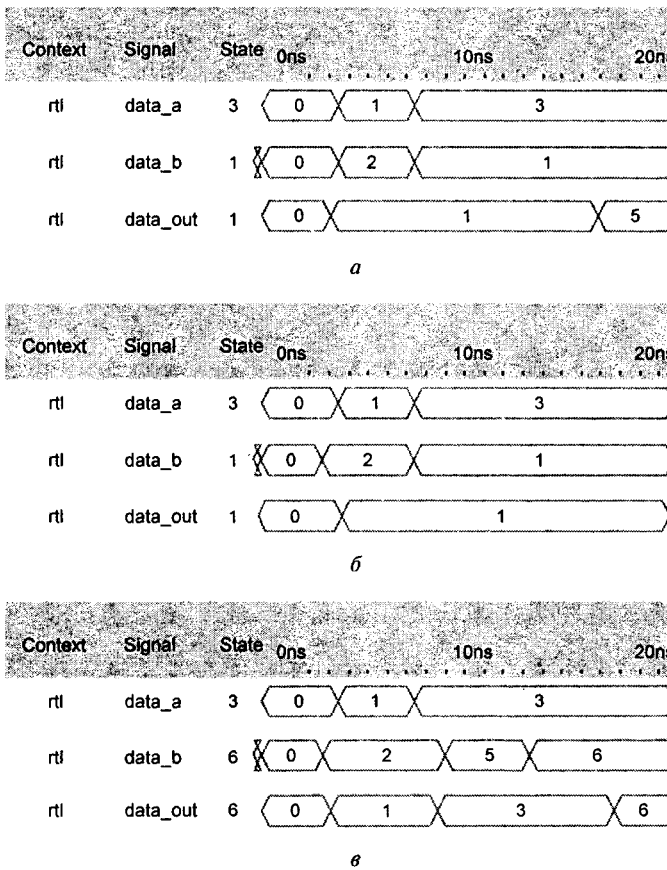


Рис. 3.8. Временные диаграммы, иллюстрирующие различия при использовании одного оператора присваивания с несколькими секциями `after` и нескольких операторов присваивания

Атрибуты сигналов

Атрибуты сигналов используются для получения информации о событиях, которые происходят с сигналами. Обращение к атрибутам имеет следующий синтаксис:

```
signal_name'attribute_name
```

Перечень предопределенных в языке VHDL атрибутов сигналов приведен в табл. 3.1 (где `S` — имя сигнала).

Обратим особое внимание на атрибуты (**`delayed`**, **`stable`**, **`quiet`**, **`transaction`**), значениями которых являются *сигналы*. Эти атрибуты создают новые сигналы в модели. В отличие от обычных, явно декларированных в тексте на VHDL сигналов (explicit signals), генерируемые атрибутами новые сигналы — неявные (implicit signals). Тем не менее, эти сигналы фактически генерируются в модели и могут быть использованы наравне с явно декларированными сигналами. Так, например, их можно использовать в правой части операторов присваивания значений сигналам.

Таблица 3.1. *Перечень атрибутов сигналов*

Название атрибута	Описание
`S'delayed (T)`	Сигнал, имеет то же значение, что и `S`, но задержанное на время `T` (если значение задержки не задано, то принимается дельта-задержка)
`S'stable(T)`	Сигнал, имеет значение `true`, если в течение времени `T` сигнал не изменялся
`S'quiet(T)`	Сигнал, имеет значение `true`, если в течение времени `T` к сигналу не было обращений
`S'transaction`	Сигнал типа `bit`, переключается из `"0"` в `"1"` (или, наоборот) при каждом обращении к `S`
`S'event`	Принимает значение `true`, если в текущем цикле моделирования происходило изменение сигнала
`S'active`	Принимает значение `true`, если в текущем цикле моделирования есть обращение к этому сигналу
`S'last_event`	Интервал времени, прошедший с последнего изменения сигнала
`S'last_active`	Интервал времени, прошедший с последнего обращения к этому сигналу
`S'last_value`	Предыдущее значение сигнала

Операторы ожидания *wait*

Как мы уже говорили, основным принципом управления запуском параллельных операторов, процессов в VHDL является управление по событиям изменения сигналов, инициирующих срабатывание параллельного оператора, для которого данный сигнал является входным. Этот принцип хорошо согласуется с характером работы комбинационных схем, где выходные сигналы зависят только от входных сигналов [27], и любое изменение входного сигнала может привести к изменениям сигналов выходных.

Однако, например, для синхронных последовательностных схем, схем с памятью тактовый сигнал определяет моменты, в которые только и могут быть считаны входные сигналы, влияющие на функционирование схемы, или же выданы новые значения выходных сигналов. Программные конструкции, соответствующие в модели схемам такого рода, их параллельные операторы могут запускаться на выполнение только в некоторые моменты модельного времени (или при определенных событиях, например, при выполнении условий, определяемых для некоторой совокупности входных сигналов).

Конечно, и здесь можно было бы применять тот же принцип управления запуском параллельных операторов, но это часто приводило бы к выполнению очевидно лишних циклов моделирования, тем самым существенно его замедляя. Введение более развитого, программно управляемого механизма запуска параллельных операторов позволяет решить эту проблему, причем в терминах, близких к содержательной стороне функционирования моделируемых устройств.

Для этого в языке VHDL введены *операторы ожидания*, операторы **wait**.

Операторы ожидания позволяют управлять моментами времени, в которые параллельные операторы будут реагировать на изменение сигналов. Операторы ожидания имеет следующий синтаксис:

```
wait   [ on signal_name {,...} ]
  [until boolean_expression]
  [for time_expression];
```

Оператор **wait** позволяет приостанавливать выполнение параллельного оператора в модели и программно задавать условия, при которых его выполнение может быть возобновлено.

Выполнение оператора **wait** приводит к приостановке выполнения параллельного оператора на некоторое время или до выполнения некоторого условия. Как только заданные условия будут выполнены, процесс продолжит свое выполнение.

Секция **on** позволяет определить список сигналов, на изменение которых будет реагировать процесс. Когда выполнение процесса доходит до операто-

ра **wait**, оно приостанавливается до момента, когда какой-либо из указанных в списке сигналов изменит значение.

Секция **until** позволяет определить булевское выражение, которое должно иметь значение `true` (для того, чтобы процесс продолжил выполнение после этой инструкции). Если же в момент выполнения оператора **wait** условие уже имело значение `true`, то выполнение процесса будет продолжено только после того, как заданное условие сначала примет значение `false`, а потом вновь значение `true`. Если команда содержит и секцию **on**, и секцию **until**, то булевское выражение, стоящее в **until**, проверяется только при условии, что произошло изменение какого-либо сигнала, стоящего в списке секции **on**.

Секция **for** позволяет определить интервал времени, на который процесс приостанавливается в модельном времени. Эта секция может задаваться вместе с двумя предыдущими. В таком случае процесс возобновит исполнение, как только условие, заданное одной из секций будет выполнено. Если присутствуют все три секции, то условие секции **until** проверяется только при изменении одного из сигналов, указанных в секции **on**.

Оператор **wait** может использоваться и без секций **on**, **until**, **for**. В этом случае выполнение **wait** приведет к остановке выполнения процесса до конца моделирования.

В теле процесса может быть несколько операторов **wait**.

Примечание

В описании процесса нельзя одновременно использовать и список чувствительности, и оператор **wait**.

Примечание

В OrCAD Express 9.1 и в Foundation 2.1 не поддерживается возможность использования нескольких операторов **wait** в одном процессе.

Параллельный оператор контроля в ходе моделирования *Assert*

Оператор **assert** является общим механизмом выявления существенных ситуаций (с точки зрения разработчика модели устройства), а также выдачи о них сообщений. Оператор **assert** заменяет процесс, единственный оператор в теле которого выполняет контроль значения объекта и, возможно, заносит результат в отчет. Параллельный оператор **assert** имеет следующий синтаксис:

```
[label:] assert boolean_expression [report expression] [severity expression];
```

При использовании этого оператора, в отличие от аналогичного процесса, логическое условие проверяется постоянно, а не только при изменении значений объектов, входящих в него, что позволяет организовать контроль не только значений сигналов, но и временных ограничений.

Выражение, значение которого проверяется, ставится за ключевым словом **assert**(Утверждение), суть которого состоит в том, что заданное булевским выражением boolean_expression условие должно быть истинным. Если обнаруживается, что условие ложно, то это и есть "нарушение утверждения", о чем выдается сообщение.

Секция **report** задает выражение, значение которого (это может быть и текстовая строка) выдается при обнаружении нарушения проверяемого условия (например, на экране системы моделирования). Если секция **report** не указана, то по умолчанию выдается стандартное сообщение: "Assert violation" (нарушение утверждения).

Секция **severity** позволяет программисту самому определять уровень "серьезности" нарушения утверждений и, соответственно, далее управлять реакцией на него в соответствии с указанной категорией. Определены 4 категории нарушений: **note** (заметка), **warning** (предупреждение), **error** (ошибка) и **failure** (отказ).

Структурное описание объекта моделирования

Компоненты

Другим, альтернативным поведенческому описанию архитектуры объекта на VHDL, является структурное описание архитектуры.

Структурное описание позволяет описать систему как совокупность компонентов — подсистем, объединенных сигналами. Подсистемы также являются объектами моделирования, но находятся на более низком уровне иерархии структурирования и представления проектируемого устройства. Каждая подсистема, в свою очередь, может быть представлена совокупностью подсистем, и так далее, пока на каком-то уровне не будет задано поведенческое описание архитектуры компонента, или не будет использован предопределенный компонент.

Рассмотрим конструкцию структурного описания объекта моделирования. Пусть описание имеет два уровня иерархии. Объект моделирования, находящийся на верхнем уровне иерархии, включает в себя 3 компонента (рис. 3.9).

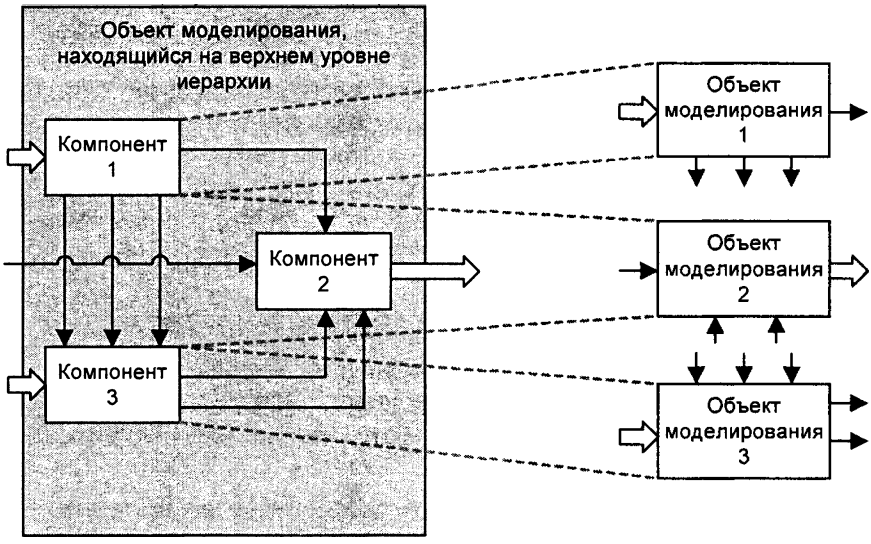


Рис. 3.9. Конструкция структурного описания объекта моделирования

Каждый компонент, используемый в структурном описании, должен быть сам описан как объект моделирования (entity). Каждое такое описание может быть размещено в отдельном файле или библиотеке.

Декларация компонента

Для того чтобы один объект моделирования мог быть включен в состав другого объекта, его необходимо декларировать как *компонент* — **component**.

Декларация компонента должна полностью совпадать с декларацией соответствующего ему объекта моделирования, но ключевое слово **entity** заменяется ключевым словом **component**.

Декларация компонента имеет следующий синтаксис:

```
component component_name is
  [generic (generic_interface_list);]
  [port (port_interface_list);]
end component [component_name];
```

Идентификатор component_name вводит имя компонента описываемого вида. Мы будем говорить о нем как о *типе компонента*. Как показано далее, в структурном описании архитектурного тела могут быть несколько компонентов (назовем их компонентами-экземплярами), к которым применима эта декларация.

Декларация компонентов может размещаться в декларативной части объектов моделирования (наряду с описаниями типов, подтипов), но может быть

расположена и в тех файлах, в которых описаны объекты моделирования, используемые в качестве компонентов.

Здесь можно провести аналогию с традиционными языками программирования — Fortran, C, Pascal и др. Там мы декларируем переменные, структуры данных, функции прежде, чем использовать их в программе. Так и в структурном описании объекта (архитектуры объекта) на VHDL мы декларируем компоненты и сигналы прежде, чем специфицировать, как они связаны между собой в структуру, составляющую описываемый объект.

Включение компонента в модель объекта (instantiation)

Структурное описание объекта моделирования, находящегося на верхнем уровне иерархии, состоит из так называемых *назначений компонентов*. В рамках этих назначений определяются фактические значения обобщающих констант и связи с другими объектами.

В структурном описании объекта мы как бы собираем его из других компонентов. Задав декларацию компонентов, мы как бы написали спецификацию проектируемого объекта, определив, какого типа компоненты в него входят. С помощью конструкции `instantiation` — оператора назначения компонента, мы устанавливаем компоненты в проектируемый объект и связываем их в некоторую структуру.

Оператор назначения компонента имеет следующий синтаксис:

```
instantiation_label:
  [component] component_name
  [generic map (generic_association_list);]
  [port map (port_association_list)];
```

Операторы назначения компонентов, наряду с процессами и параллельными операторами присваивания значений сигналам, являются *параллельно выполняемыми*.

Прежде всего обратим внимание на то, что в программной конструкции назначения компонента *метка является обязательной*. В декларации компонента мы указали лишь тип используемого компонента. Компонентов такого типа в структуре описываемого объекта может быть несколько. Именно метка `instantiation_label`, уникальная для данного структурного описания архитектуры объекта, фактически задает однозначное имя экземпляра компонента в специфицируемой структуре. Другому компоненту того же типа, также устанавливаемому (включаемому) в состав описываемого объекта, будет соответствовать свой оператор назначения компонента, с другой меткой.

Имя компонента `component_name` — это, фактически, имя *типа* компонента (определено декларацией компонента), а не имя *экземпляра* компонента в

структуре описываемого объекта. Ключевое слово component в операторе назначения компонента может ставиться, но не является обязательным.

Секция связей портов компонента **port map** специфицирует связи данного компонента с сигналами объекта и портами других компонентов. Эти связи задаются списком port_association_list, который содержит список фактических сигналов, связанных с портами компонента. Он имеет следующий синтаксис:

```
{ [port_name=>] (signal_name | expression | open)} {,...}
```

Каждый элемент в списке (сигнал) ассоциируется с портом объекта моделирования, описанного на один уровень выше, или с его внутренним сигналом, или является независимым, что помечается ключевым словом **open**. Так, ключевым словом **open** следует помечать выходные порты компонента, которые не соединяются ни с одним сигналом или входом другого компонента.

Список Generic_association_list содержит фактические значения обобщающих констант и имеет следующий синтаксис:

```
{constant_name:=] value } {,...}
```

Оператор назначения является параллельным оператором. Он задает компонент-экземпляр, который функционирует параллельно (в модельном времени) с другими компонентами-экземплярами и другими параллельными операторами в теле описания архитектуры объекта.

Рассмотрим, например, как на базе объектов моделирования — четырехразрядных регистров — может быть сконструирован новый объект моделирования — шестнадцатиразрядный регистр.

Пример описания четырехразрядного регистра приведен в листинге 3.17.

Листинг 3.17

```
Entity register_4 is
Port(data_in:in std_logic_vector(3 downto 0);
     Data_out: out std_logic_vector(3 downto 0);
     Wren:in std_logic;
     Rden:in std_logic;
     Clk:in std_logic);
End register_4;
Architecture rtl of register_4 is
Signal data:std_logic_vector(3 downto 0);
Begin
Process (clk)
```

```
Begin
If clk'event and clk='1' then
If wren='1' then data<=data_in; end if;
If Rden='1' then data_out<=data else data_out<="ZZZZ"; End if;
Process process;
End architecture rtl;
```

Пример описания шестнадцатиразрядного регистра, использующего четырехразрядные регистры как компоненты, приведен в листинге 3.18.

Листинг 3.18

```
Entity register_16 is
Port(data16_in: in std_logic_vector(15 downto 0);
     Data16_out: out std_logic_vector(15 downto 0);
     Wren16:in std_logic;
     Rden16:in std_logic;
     Clk16:in std_logic);
End register_16;
Architecture struct of register_16 is
Component register_4 is
Port(data_in:in std_logic_vector(3 downto 0);
     Data_out: out std_logic_vector(3 downto 0);
     Wren:in std_logic;
     Rden:in std_logic;
     Clk:in std_logic);
End component register_4;

Begin
Reg_0: register_4
Port map(data_in => data16_in(3 downto 0),
         data_out=>data16_out(3 downto 0),
         Wren=>wren16, Rden=>rden16, clk=>clk16);
Reg_1: register_4
Port map(data_in => data16_in(7 downto 4),
         data_out=>data16_out(7 downto 4),
         Wren=>wren16, Rden=>rden16, clk=>clk16);
Reg_2: register_4
Port map(data_in => data16_in(11 downto 8),
         data_out=>data16_out(11 downto 8),
         Wren=>wren16, Rden=>rden16, clk=>clk16);
```

```
Reg_3: register_4
Port map(data_in=>data16_in(15 downto 12),
         data_out=>data16_out(15 downto 12),
         Wren=>wren16, Rden=>rden16, clk=>clk16);
End architecture struct;
```

Структурная схема `register_16` приведена на рис. 3.10.

В этом примере объект моделирования (`register_16`), находящийся на верхнем уровне иерархии, строится на базе четырех однотипных компонентов типа `register_4`, поэтому в его описании присутствует только одна декларация компонента. Четыре оператора назначения компонента создают в структурном описании архитектуры `struct` для описываемого объекта `register_16` четыре компонента-экземпляра одного и того же типа: `Reg_0`, `Reg_1`, `Reg_2` и `Reg_3`. Назначение выполняется для каждого компонента в отдельности.

Можно ассоциировать отдельные самостоятельные сигналы с элементами сигналов, совпадающих с ними по типу или, наоборот, ассоциировать элементы сигналов с отдельными сигналами, совпадающими с ними по типу, как это продемонстрировано в листинге 3.18. Сигналы `data_in` и `data_out` ассоциируются с фрагментами сигналов `data16_in` и `data16_out`.

При описании объекта моделирования можно задать *значения входных сигналов по умолчанию*. Тогда, при использовании этого объекта в качестве компонента другого объекта, значения этих сигналов для него можно не задавать, а указать слово **open**. В этом случае для них будет использоваться значение по умолчанию.

Для выходных и двунаправленных сигналов также может использоваться ключевое слово **open**. Если это описание применено к выходному сигналу, то значение, принимаемое им, в модели игнорируется. Этот сигнал можно вообще никак не описывать, однако его описание облегчает понимание модели.

Если описание **open** применено к двунаправленному сигналу, то внутри объекта используется значение, которое этот сигнал имеет по умолчанию, а вне объекта этот сигнал не используется.

Оператор генерации (Generate)

Если объект моделирования включает в себя много однотипных компонентов, назначения для них имеют сходную структуру. В модели они занимают много места и затрудняют ее понимание. Оператор генерации **generate** позволяет в этом случае более компактно описать модель. Он позволяет организовать цикл с параметром, с однократным параметризованным описанием внутри, на базе которого будут сгенерированы описания для всей группы однотипных компонентов. Фактически это некоторая форма макрооператора в языке VHDL.

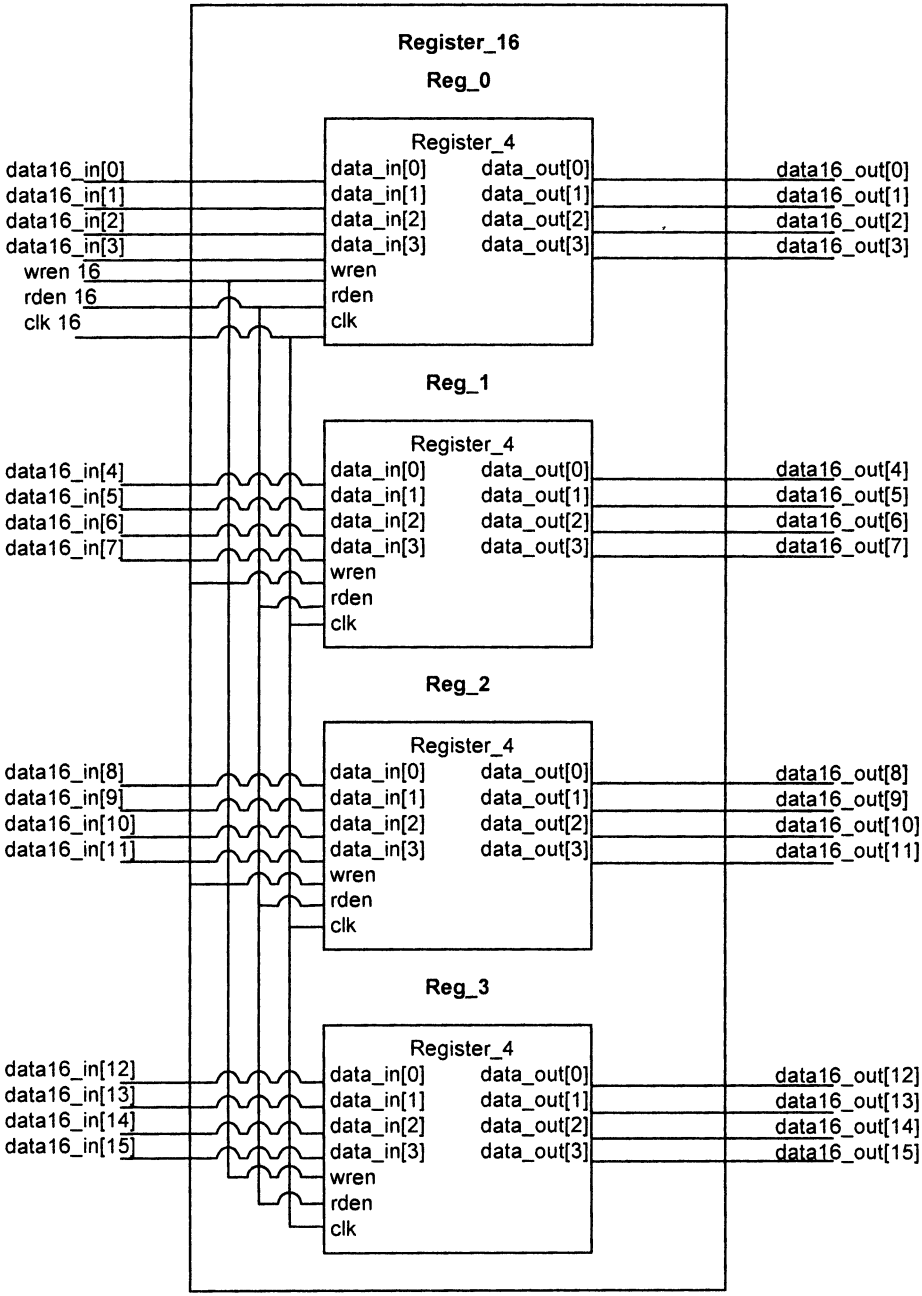


Рис. 3.10. Структурная схема

Оператор генерации имеет следующий синтаксис:

```
Group_label: for index in range generate
Element_label: component_name [generic map (generic_accosiation_list)]
                              [port map (port_accosiation_list)]
  end generate [Group_label];
```

Допускается вложенность операторов генерации.

С использованием этого оператора описание того же шестнадцатиразрядного регистра `register_16` может принять следующий вид (листинг 3.19).

Листинг 3.19

```
Entity register_16 is
Port(data16_in:in std_logic_vector(15 downto 0);
     Data16_out: out std_logic_vector(15 downto 0);
     Wren16:in std_logic;
     Rden16:in std_logic;
     Clk16:in std_logic);
End register_16;
Architecture struct of register_16 is
Component register_4 is
Port(data_in:in std_logic_vector(3 downto 0);
     Data_out: out std_logic_vector(3 downto 0);
     Wren:in std_logic;
     Rden:in std_logic;
     Clk:in std_logic);
End component;
Begin
Registers: for i from 0 to 3 generate
Reg: register_4
Port map(data_in =>data16_in((i+1)*4-1 downto i*4),
         data_out =>data16_out((i+1)*4-1 downto i*4),
         Wren=>wren16, Rden=>rden16, clk=>clk16);
End generate Registers;
End architecture struct;
```

Как видно из этого примера, оператор **generate** позволил сделать модель существенно более компактной (сравните с листингом 3.18).

Рассмотрим использование **generate** в этом примере. Группе четырехразрядных регистров присвоена метка `Registers`. Типовому оператору назна-

чения регистра внутри группы присвоена метка `Reg`. В операторе **`generate`** организован цикл от 0 до 3 (по количеству регистров, которое необходимо). Переменная цикла — `i`. На базе этого значения определяются конкретные фрагменты входного и выходного информационного сигнала шестнадцатиразрядного регистра, который связывается с входными и выходными информационными сигналами конкретных четырехразрядных регистров. На базе значения переменной цикла определяется конкретный фрагмент сигнала шестнадцатиразрядного регистра, с которым ассоциируется соответствующий сигнал четырехразрядного регистра. В результате такого описания на информационный вход четырехразрядного регистра с номером 0 будет подаваться `data_in(3 downto 0)`, на вход регистра с номером 1 — `data_in(7 downto 4)` и т. д.

При таком способе генерации компонентам-экземплярам средой моделирования присваиваются уникальные идентификаторы, состоящие, как правило, из общей метки компонента (в рассмотренном примере — `Reg`) и порядкового номера. Однако единого стандарта не существует, и конкретная форма идентификатора зависит от используемого инструментария.

Существует также условная форма оператора **`generate`**.

```
label: if boolean_expression generate
{concurrent_statements}
  end generate [label];
```

Секций **`else`** и **`elseif`** условная форма оператора **`generate`** не имеет.

Как и обычный оператор назначения компонентов, оператор **`generate`** является параллельным оператором. В теле самого оператора **`generate`** также могут быть указаны только параллельные операторы.

Пример структурного описания в OrCAD Express 9.1

В OrCAD Express существует возможность автоматической генерации структурного описания на языке VHDL на базе имеющегося рисунка схемы *(см. гл. 6)*.

Описание объектов, соответствующих компонентам, может располагаться в том же файле, что и описание структуры объекта моделирования. Однако оно может быть расположено и в отдельных файлах, которые могут иметь структуру пакетов или самостоятельных моделей. В этом случае такие файлы должны включаться в состав проекта.

Рассмотрим схему, включающую в себя два одинаковых элемента — последовательно соединенных инвертора (рис. 3.11). Порты входных и выходных сигналов не определены, поэтому схема рассматривается как находящаяся на верхнем уровне иерархии.

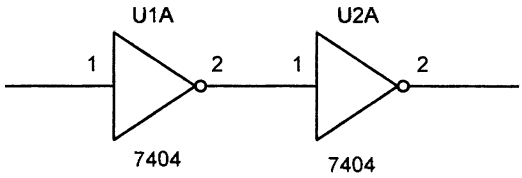


Рис. 3.11. Принципиальная схема, структурное описание которой, сгенерированное с использованием OrCAD Exxpress 9.1, приведено в листинге 3.20

Описание схемы, приведенной на рис. 3.11, сгенерированное с использованием OrCAD Express 9.1, будет иметь следующий вид.

Листинг 3.20

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;

-- Model of 2 invertors
ENTITY SCHEMATIC1 IS END SCHEMATIC1;
ARCHITECTURE STRUCTURE OF SCHEMATIC1 IS

-- COMPONENTS
COMPONENT \7404\
  PORT (
  I_A : IN std_logic;  O_A : OUT std_logic;
VCC : IN std_logic;  GND : IN std_logic;
I_B : IN std_logic;  O_B : OUT std_logic;
  I_C : IN std_logic;  O_C : OUT std_logic;
I_D : IN std_logic;  O_D : OUT std_logic;
I_E : IN std_logic;  O_E : OUT std_logic;
  I_F : IN std_logic;  O_F : OUT std_logic
  ); END COMPONENT;
-- SIGNALS
SIGNAL orcad_unused:std_logic;
SIGNAL N00009 : std_logic;
SIGNAL N00033 : std_logic;
SIGNAL N00036 : std_logic;
SIGNAL GND : std_logic;
SIGNAL VCC : std_logic;
```

```
-- GATE INSTANCES
BEGIN
U1 : \7404\
PORT MAP(
I_A => N00033,    O_A => N00009,
  VCC => VCC,    GND => GND,
I_B => orcad_unused,    O_B => OPEN,
I_C => orcad_unused,    O_C => OPEN,
  I_D => orcad_unused,    O_D => OPEN,
  I_E => orcad_unused,    O_E => OPEN,
I_F => orcad_unused,    O_F => OPEN
);

U2 : \7404\
PORT MAP(
I_A => N00009,    O_A => N00036,
  VCC => VCC,    GND => GND,
I_B => orcad_unused,    O_B => OPEN,
I_C => orcad_unused,    O_C => OPEN,
  I_D => orcad_unused,    O_D => OPEN,
I_E => orcad_unused,    O_E => OPEN,
I_F => orcad_unused,    O_F => OPEN
);
END STRUCTURE;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity  \7404\ is
PORT (
  I_A : IN std_logic;  O_A : OUT std_logic;
VCC : IN std_logic;  GND : IN std_logic;
I_B : IN std_logic;  O_B : OUT std_logic;
  I_C : IN std_logic;  O_C : OUT std_logic;
I_D : IN std_logic;  O_D : OUT std_logic;
I_E : IN std_logic;  O_E : OUT std_logic;
  I_F : IN std_logic;  O_F : OUT std_logic
  ) ;
end ;
```

```
architecture behavior  of \7404\ is
 -- place component declarations in architecture bodies before begin
begin
process (I_A)
  begin   O_A<= not (I_A);
end process;
end;
```

Обратите внимание на описание объекта моделирования: **ENTITY** SCHEMATIC1 **IS END** SCHEMATIC1; — в нем отсутствует секция портов. Такое описание допустимо только на верхнем уровне иерархии.

Поскольку внутренним линиям связей не были присвоены специальные имена, в схеме используются имена, присвоенные OrCAD Express автоматически, по умолчанию: N00033 — входной сигнал компонента U1, N00009 — выходной сигнал компонента U1 и входной сигнал компонента U2, N00036 — выходной сигнал компонента U2.

Задание конфигурации компонентов

В общем случае объект моделирования может иметь несколько различных описаний архитектуры (или, как иногда говорят, "архитектурных тел"). Это может быть связано как с разным уровнем детализации представления моделируемого объекта — от алгоритмического поведенческого описания до детальной схемы в базисе конкретной технической реализации устройства, так и с наличием многовариантных реализаций проектируемого устройства, его узлов и их моделей на VHDL.

Тогда возникает вопрос, какое из этих описаний должно использоваться в модели, содержащей данный объект в качестве компонента. Для указания этого используется *конфигурация* — ассоциация (связывание) объекта моделирования с одним из его описаний архитектуры. Использование конфигурации позволяет легко менять описания архитектуры, связанные с объектами моделирования, создавать очень гибкие модели.

Конфигурация может быть задана или в форме конфигурационной декларации, или в форме конфигурационной спецификации.

Конфигурационная спецификация

При структурном описании архитектуры объект описывается как структура из объектов-компонентов. Компонент включается в эту структуру оператором назначения компонента (`instantiation`), создающим экземпляр компонента в описываемой структуре. Тип генерируемого компонента-экземпляра предварительно декларируется конструкцией декларации компонента `component ... is`. Однако, как видно из описания этих конструк-

ций в предыдущих параграфах, ни декларация компонента, ни оператор назначения компонента не задают его архитектуру. Это логично, так как по самой идее компонентов, они являются отдельно описанными моделями некоторых устройств, узлов. В этом отдельном описании и определяется архитектура объекта, используемого здесь как компонент.

Соответственно, во-первых, нужно иметь средства для того, чтобы связать тип включаемого компонента с моделью этого компонента, которая задает не только внешнюю декларацию, но и описание архитектуры этого компонента. Во-вторых, как неоднократно подчеркивалось, одному объекту может соответствовать несколько альтернативных описаний архитектуры, и для компонента, включаемого в структурное описание, нужно выбрать, какое конкретное описание архитектуры здесь использовать.

Конфигурационная спецификация используется в архитектурном теле описываемого объекта для определения связи между компонентами и парами (декларация объекта моделирования / архитектурное тело объекта моделирования), соответствующими этим компонентам.

Предполагается, что модели объектов, соответствующих компонентам, расположены в библиотеках. Конфигурационная спецификация позволяет определить, в какой библиотеке расположена модель объекта, соответствующая компоненту, имя объекта-компонента в библиотеке, а также позволяет задать имя его архитектурного тела, которое должно быть использовано в данном контексте (при включении такого объекта как компонента в описываемый объект более высокого уровня).

Выполнение конфигурационной спецификации называют *связыванием* (binding).

Конфигурационная спецификация имеет следующий синтаксис:

```
for component_specification binding_indication; end for;
```

Секция component_specification задает идентификатор компонента-экземпляра или компонентов-экземпляров, к которым относится данная конфигурационная спецификация. Она имеет следующий синтаксис:

```
(instantiation_label {,...} | others | all): component_name
```

Здесь: `instantiation_label` — имя-метка экземпляра компонента в структурном описании объекта (выше мы подчеркивали необходимость наличия уникальной метки при каждом операторе назначения). Конфигурационная спецификация относится ко всем компонентам-экземплярам, имена-метки которых перечислены в этом списке. Ключевое слово **all** говорит о том, что конфигурационная спецификация относится ко всем экземплярам компонентов этого типа, а **others** — ко всем остальным, кроме отдельно специфицированных, экземплярам компонентов этого типа. Тип компонентов указывается идентификатором `component_name` (введен декларацией компонента в данном структурном описании архитектурного тела).

Секция binding_indication позволяет определить объект моделирования, который будет соответствовать данному компоненту. Она имеет следующий синтаксис:

```
use entity entity_name [(arcitecture_identifier)]
```

Поле entity_name задает идентификатор — имя модели, которая берется как модель конфигурируемого компонента; в этом поле могут быть указаны и имя библиотеки, и имя пакета, в которых помещено описание модели с указанным именем. Если в описании модели имеется несколько альтернативных архитектурных тел, то далее, в круглых скобках, указывается идентификатор архитектурного тела, которое должно использоваться для конфигурируемого компонента.

Пример приведен в листинге 3.21.

Листинг 3.21

```
for bit0, bit1: flipflop
  use entity work.edge_triggered_dff(basic)
end for;
```

В этом примере конфигурационная спецификация относится к двум компонентам — bit0 и bit1. Эти два конкретных компонента соответствуют одному типу компонента — flipflop. Он сопоставлен объекту, описание модели которого (идентификатор — edge_triggered_dff) расположено в библиотеке work. Для конфигурируемых компонентов нужно использовать архитектурное описание с именем basic.

Конфигурационная спецификация компонента всегда является частью описания структурной формы описания архитектуры некоторого объекта моделирования, в который включается конфигурируемый компонент.

Конфигурационная декларация

Поскольку конфигурационная спецификация компонента располагается внутри архитектурного тела, ее изменение приводит к необходимости редактировать текст модели объекта, включающего этот компонент, и перекомпилировать ее.

Другая форма задания конфигурации компонентов структурного описания объектов — *конфигурационная декларация* (**configuration**). Она содержит ту же информацию, что и конфигурационная спецификация, но организована как отдельный модуль программы на VHDL, может быть расположена в отдельном файле и использоваться независимо, наравне с модулями **entity** декларативной части описания объекта моделирования и модулями **architecture** описания архитектуры объекта моделирования.

Конфигурационная декларация должна иметь уникальное имя. Конфигурационная декларация пишется для конкретного объекта моделирования, для его конкретной архитектуры. Конфигурационная декларация имеет следующий синтаксис:

```
configuration configuration_name of entity_name is
  for architecture_name -- имя конфигурируемого описания архитектуры
    {for component_specification binding_indication;
    end for;}
  end for;
end [ configuration] [configuration_name];
```

Здесь:

- идентификатор `configuration_name` — имя конфигурационной декларации;
- идентификатор `entity_name` — имя объекта моделирования, для которого она предназначена;
- идентификатор `architecture_name` — имя конкретного архитектурного описания объекта моделирования.

Внутри конфигурационной декларации указываются конфигурационные спецификации для компонентов, входящих в состав данного архитектурного описания.

Например, конфигурационная спецификация (листинг 3.22) может быть включена в состав конфигурационной декларации следующим образом:

Листинг 3.22

```
configuration conf_1 of entity_bits is
for entity_bits_struct
       for bit0, bit1: flipflop
  use entity work.edge_triggered_dff(basic);
  end for;
end for;
end configuration conf_1;
```

В этом примере конфигурация `conf_1` определяется для объекта моделирования с именем `entity_bits`, для его архитектуры с именем `entity_bits_struct`.

Конфигурирование моделей с многоуровневой иерархией

Конфигурации являются мощным механизмом языка VHDL, позволяющим задавать конкретную реализацию моделей проектируемых устройств

в условиях, когда на различных уровнях модели есть множество вариантов их реализации: и по форме (поведенческая/структурная), и по уровню детализации, и по ориентации на разные технологии воплощения в СБИС, и др.

При конфигурировании моделей с многоуровневой иерархией возникает необходимость структурировать сами конфигурационные декларации, создавать многоуровневую иерархию конфигураций.

Для этого в секции `binding_indication` конфигурационной спецификации может включаться указание на имена используемых конфигураций. Это указание имеет следующий синтаксис:

```
use configuration configuration_name;
```

Например:

```
for flag_reg:reg44
  use configuration work.reg4_gate_level;
end for;
```

Такая форма связывания компонента в модели позволяет присоединить предварительно сконфигурированные пары объект/архитектура, просто называя конфигурационную декларацию, где они специфицированы. Обратите внимание, что в этой конфигурационной спецификации мы указали `configuration_name` — имя конфигурационной декларации, которая, в отличие от конфигурационной спецификации, является самостоятельным программным модулем на VHDL и имеет имя.

Для того чтобы конфигурировать объект, содержащий множество уровней иерархии, используется *комплексная конфигурационная декларация*, которая имеет следующий синтаксис:

```
configuration identifier of entity_name is
  block_configuration
  end [configuration] identifier];
```

Здесь секция `block_configuration` имеет следующий синтаксис:

```
  for architecture_name
    {for component_specification
      binding_indication;
    [block_configuration]
    end for;]
  end for;
```

Комплексная конфигурационная декларация позволяет сконфигурировать компоненты, входящие в состав объекта моделирования.

Использование полностью сконфигурированного объекта моделирования

При включении компонента-экземпляра в структурное описание архитектуры объекта можно использовать в качестве компонента полностью сконфигурированный объект моделирования. В этом случае оператором назначения компонента сразу указывается имя используемой конфигурации. Такая конструкция имеет следующий синтаксис:

```
instantiation_label:
  configuration configuration_name
  [generic map (generic_associstion_list)]
  [port map (port_association_list)];
```

Конфигурация с именем `configuration_name` включает спецификации объекта моделирования и соответствующего ему архитектурного тела. Секции **`generic map`** и **`port map`** позволяют задать действительные значения обобщающих констант, входных и выходных сигналов.

Включение **`generic map`** и **`port map`** в секцию `binding_indication` конфигурационной спецификации предоставляет очень гибкие возможности по присоединению компонентов. Синтаксис такой конструкции имеет следующий вид:

```
use entity entity_name[(architecture_identificator)] | configuration
configuration_name
[generic map (generic_association_list)]
[port map (port_association_list)]
```

Одним из возможных применений данного механизма является отделение спецификации обобщающих констант (используемых для задания временных характеристик) от спецификации собственно структуры схемы. Можно написать декларацию компонента структурно, не включая обобщающие константы, и только потом, включая его экземпляр в модель, указать значения этих констант. Пример приведен в листинге 3.23.

Листинг 3.23

```
library ieee; use ieee.std_logic_1164.all;
-- декларативная часть описания объекта reg
entity reg is
  generic (t_setup, t_hold, t_pd:delay_length;
      width:positive);
  port (clock:in std_logic;
    data_in: in std_logic_vector (0 to width-1);
```

```
    data_out: out std_logic_vector(0 to width-1));
end entity reg;
-- описание архитектуры (вариант structural) для объекта controller
architecture structural of controller is
    component reg is
      generic (width: positive);
      port (clock: in std_logic;
        data_in: in std_logic_vector (0 to width-1);
        data_out: out std_logic_vector (0 to width-1));
    end component reg;
begin
  state_reg: component reg
    generic map (width=>state_type'length)
    port map (clock=> clock_phase1,
        data_in=>next_state,
        data_out=> current_state);
end architecture structural;
      -- конфигурационная декларация
configuration controller_with_timing of controller is
  for structural
    for state_reg: reg
      use entity work.reg(gate_level)
      generic map (t_setup=>200 ns, t_hold=>150 ns,
          t_pd=>150 ns, width=>width);
    end for;
  end for;
end configuration controller_with_timing;
```

Более того, это позволяет решать в модели проблемы, связанные с несовпадением не только имен констант и сигналов, но даже их количества. Если у реального объекта их меньше, чем у присоединяемого, то задаются значения по умолчанию.

Связывание по умолчанию

Предусмотрены также правила связывания компонентов-экземпляров с объектами и их архитектурными телами в случаях, когда в конфигурационных спецификациях и декларациях модели отсутствуют явные указания для связывания. Это правила связывания по умолчанию.

Если для компонента, декларированного в структурном описании архитектурного тела, нет явных указаний для связывания, то автоматически ищется

объект моделирования с тем же именем, что и компонент. Если он найден, то связывание компонента производится с ним. Если для найденного объекта имеется несколько архитектурных тел, то для связывания используется то, которое компилировалось последним.

Если по указанным правилам не удается найти объект для связывания с ним компонента, то связывание считается отложенным.

Отложенные присоединения компонентов

Отложенные (deferred) присоединения компонентов используются, когда на момент моделирования разработаны еще не все компоненты. Это нередко происходит при моделировании сверху вниз.

В этом случае при моделировании следует учитывать, что входы такого компонента не используются, а выходы остаются не присоединенными.

Отложенное присоединение компонентов указывается в секции `binding_indication` и имеет следующий синтаксис: **`use open`**.

Вычисляемые сигналы. Разделяемые сигналы и функции разрешения коллизий

В схемах могут возникать ситуации, когда несколько выходных сигналов объединяются в один. В результате получается сигнал, формируемый множеством источников, причем разные источники могут присваивать сигналу различные значения. В этом случае возникает необходимость определения результирующего значения для этого сигнала. Например, в реальных цифровых устройствах на уровне физических сигналов реализуется формирование результирующего значения сигнала, что на уровне логических сигналов проявляется как выполнение над сигналами-источниками некоторой логической операции, называемой операцией монтажной логики, (в англоязычной литературе термин wired logic). В модели на VHDL мы должны запрограммировать такие логические операции, для чего и используют механизм вычисляемых сигналов.

В модели на VHDL, когда одному и тому же сигналу соответствует несколько источников, говорят, что эти источники разделяют (в смысле совместно используют) данный сигнал. Такие сигналы называют разделяемыми сигналами (shared signals). Источники одновременно присваивают сигналу значения, возможно, различные. Необходим механизм однозначного разрешения таких коллизий и определения результирующего значения разделяемого сигнала.

В языке VHDL предусмотрено разрешение коллизий, связанных с разделяемыми сигналами. Как разрешается коллизия, какое формируется результирующее значение разделяемого сигнала, определяется специальной функцией, называемой *функцией разрешения (коллизий)* — *resolution function*. Разделяемые сигналы в модели объекта должны быть декларированы как сигналы специального типа — *resolved type*, тип сигнала с разрешением коллизий. Назовем такие сигналы *вычисляемыми сигналами* (прямой перевод, т. к. термин "разрешаемый сигнал" кажется нам неудачным).

В отличие от обычных сигналов, при декларации вычисляемого сигнала указывается не только *тип*, но и *функция*, на базе которой будет определяться значение сигнала.

Описание вычисляемых сигналов

Описание вычисляемого сигнала имеет следующий синтаксис:

```
signal name: [resolution_function_name] type_mark
    [range (range_attribute_name |
            simple_expression (to | downto) simple_expression) |
                              (discrete range {,...})]
```

Здесь:

- идентификатор `type_mark` задает имя типа для определяемого сигнала;
- идентификатор `resolution_function_name` — имя функции, используемой для вычисляемого сигнала.

Функция разрешения коллизий, ассоциированная с разделяемым сигналом, его описанием, вызывается всякий раз, когда какой-либо из источников выполняет оператор присваивания нового значения сигналу. Сигнал существует непрерывно в модельном времени, и то, что другие источники в данный момент не изменяли значения на своих выходных портах, связанных с данным сигналом, не означает, что там "ничего нет". Такая ситуация означает, что на этих выходах сохраняются прежние значения. Соответственно, функция разрешения коллизий анализирует текущие значения сигнала от всех разделяющих его источников и вычисляет результирующее значение, которое и становится значением разделяемого сигнала.

Функция разрешения коллизий должна быть написана так, чтобы порядок, в котором в ее теле анализируются значения сигнала от множества источников, не влиял бы на результат работы функции.

Отметим, что функция разрешения коллизий в описании вычисляемого сигнала указана только своим именем. Сама функция описывается отдельно. Одна функция разрешения коллизий может использоваться для многих вычисляемых сигналов.

В листинге 3.24 приведен простой пример описания вычисляемого сигнала.

Листинг 3.24

```
-- используемый в примере базовый тип сигнала
type tri_state_logic is ('0','1','Z');
type tri_state_logic_arr is array (integer range <>) of tri_state_logic;
-- описание вычисляемого сигнала s1
signal s1: resolve_tri_state_logic   tri_state_logic;
-- описание функция формирования значения для вычисляемого сигнала
function resolve_tri_state_logic (values: in tri_state_logic_arr)
         return tri_state_logic is
  variable result: tri_state_logic:='Z';
  begin
    for index in values'range loop
      if values(index)/='Z' then result := values(index);
      end if;
    end loop;
  return result;
end function resolve_tri_state_logic;
```

В данном примере декларирован сигнал `s1`. Описана ситуация, когда результирующий выходной сигнал определяется на базе массива выходных сигналов, каждый из которых может принимать значение логического 0, логической 1 или находиться в высокоимпедансном ("третьем") состоянии. В массив входных сигналов входят все сигналы, от которых зависит значение результирующего выходного сигнала. Когда в модели изменяется один из этих сигналов, для определения значения результирующего сигнала вызывается функция `resolve_tri_state_logic`, указанная в описании сигнала `s1`. Возвращаемое значение определяет новое значение результирующего сигнала сигнал `s1`. Эта же функция используется и при инициализации сигнала.

Функция, описанная таким образом, как в приведенном ранее примере, не разрешает конфликтов, которые могут возникать, если сигналы в массиве приняли разные значения. Для описания поведения результирующего сигнала может потребоваться расширенный набор состояний. Тогда определяется таблица значений результирующего выходного сигнала в зависимости от значений сигналов в массиве; на базе этой таблицы и пишется функция.

Вычисляемый сигнал может иметь не только простой, но и составной тип. Пример приведен в листинге 3.25.

Листинг 3.25

```
package words is
  type xo1z is ('X','0','1','Z');
  type uword is array (0 to 31) of x01z;
 type uword_vector is array (natural range <>) of uword;
 function resolve_word (contribution : uword_vector) return uword;
 subtype word is resolve_word uword;
end package words;
package body words is
  type table is array (x01z, x01z) of x01z;
 constant resolution_table: table:=
--  'x'   '0'   '1'   'z'
 ((  'x',   'x',   'x',   'x'),   --'x'
 (  'x',   '0',   'x',   '0'),   --'0'
 (  'x',   'x',   '1',   '1'),   --'1'
 (  'x',   '0',   '1',   'z'))   --'z'
 function resolve_word (cont: uword_vector) return uword is
  variable result: uword:=(others=>'z');
 begin
  for index in cont'range loop
     for element in uword'range loop
        result(element):= resolution_table(result(element),
                                           contr(index,element));
     end loop;
  end loop;
  return result;
 end function resolve_word;
end package body words;
```

Таким образом нередко описываются входные/выходные сигналы портов.

Порты объекта моделирования (порты — обязательно сигналы, а не константы или переменные) также могут иметь тип, описанный как вычислимый. Если эти порты связаны с вычислимыми сигналами внутри модели, то при определении значения последних сначала выполняется функция определения значения порта, а потом — функция вычисления внутреннего сигнала (порт и связанный с ним внутренний сигнал могут иметь разные функции).

Если моделируемый объект имеет несколько уровней иерархии, то значение вычисляемых сигналов определяется сначала на самом нижнем уровне, за-

тем передается на уровень выше и т. д. Значение сигнала на верхнем уровне иерархии называется *эффективным значением.*

Вычисляемые сигналы могут использоваться в процедурах. Если вычисляемый сигнал является выходным, то после того, как в процедуре ему назначается новое значение, для него вызывается функция разрешения, которая и определяет его новое значение для модели. Если вычисляемый сигнал является входным, то в процедуре используется его значение, определяемое функцией вычисления.

Существует определенное ограничение при использовании вычисляемых сигналов: новое значение может присваиваться только *всему сигналу целиком.* Присваивание значения для части сигнала недопустимо.

Так, если объект моделирования имеет входной/выходной сигнал

```
data: inout word;
```

являющийся вычисляемым, то недопустим, например, такой оператор:

```
data[24 to 30]:=b'111111';
```

Практическое решение этой проблемы состоит в описании сигнала, который необходимо использовать по частям, как массива этих частей.

Примечание

В системе OrCAD Express нет возможности создавать новые вычисляемые типы сигналов. Однако в этой среде моделирования имеется встроенный пакет `std_logic_1164`, в котором описаны базовые вычисляемые типы для моделирования физических сигналов. Содержимое пакета и правила его использования приведены в *приложении 1.*

В Foundation Express также нет возможности создавать новые вычисляемые типы. Для моделирования вычисляемых сигналов используется специальный механизм комментариев, являющихся директивами компилятору.

Охраняемые сигналы

Охраняемые сигналы используются для организации автоматического отсоединения источников сигналов. Это является актуальным, например, при моделировании на высоких уровнях абстракции, когда для описания сигналов используются абстрактные типы. В этом случае отсоединение сигнала невозможно промоделировать, например, высокоимпедансным состоянием. В таких случаях используются так называемые *охраняемые сигналы.*

Охраняемые сигналы являются разновидностью вычисляемых сигналов. Их описание имеет следующий синтаксис:

```
signal_identifier {,...}: subtype_indication [register|bus]
[:=expression]
```

Сторожевой сигнал является вычисляемым по сути, поэтому его тип должен быть вычисляемым. Сторожевые сигналы могут быть двух типов: **bus** и **register**. Сигнал **bus** использует разрешающую функцию для определения значения сигнала, передавая ей пустой массив, если все источники сигнала отсоединены. Сигнал **register** отличается тем, что, в случае отсоединения всех источников сигнала, разрешающая функция не вызывается, а сигнал сохраняет свое последнее значение, что может использоваться, например, при моделировании динамических устройств памяти.

Отсоединение источников сигнала. Процесс может отсоединять свой источник (драйвер) сигнала, выполняя нулевую транзакцию. Она имеет следующий синтаксис:

```
[label:] name<=[delay_mechanism] waveform;
```

`waveform` — последовательность транзакций — новых значений, которые должен принимать сигнал после указанных задержек. Ее синтаксическое описание имеет следующий вид:

```
(value_expression [after time_expression] | null [after time_expression])
{,...}
```

Ключевое слово **null** используется для индикации того, что источник должен быть отсоединен после указанной задержки. Значение сигнала определяется с помощью решающей функции после выполнения нулевой транзакции и отсоединения источника.

Размер массива значений, передаваемых функции, определяется количеством отсоединенных источников. После того как выполнится ненулевая транзакция, источник вновь присоединяется к сигналу.

Пример приведен в листинге 3.26.

Листинг 3.26

```
architecture top_level of computer_system is
  function resolve_bits (bits: bit_vector) return bit is
    variable result: bit:='0';
  begin
  for index in bits'range loop
    result:=result or bits(index);
    exit when result='1';
  end loop;
  return result;
  end function resolve_bits;
signal write_en: resolve_bits bit bus;
```

```
begin
  CPU: process is
  begin
    write_en<='0' after Tpd;
    loop
      wait until clock:='1';
      if hold_req='1' then
        write_en<=null after Tpd;
        wait on clock until clock='1' and hold_req='0';
        write_en<='0' after Tpd;
      end if;
    end loop;
  end process CPU;
end architecture top_level;
```

В примере описывается система, включающая в себя процессор, память и устройство прямого доступа в память. Охраняемый сигнал `write_en` используется для контроля присоединения к памяти. Решающая функция выполняет логическую операцию "ИЛИ" для всех источников сигнала и возвращает 0, если нет присоединенных источников. Значение 0 указывает на то, что в этот момент память не активна, т. е. к ней никто не обращается. Когда процесс `CPU` инициализируется, сигналу `write_en` присваивается значение 0. Когда контроллер прямого доступа к памяти запрашивает доступ к памяти, он устанавливает сигнал `hold_req`. При этом, процесс `CPU` планирует нулевую транзакцию для сигнала `write_en`. В результате выполнения транзакции источник сигнала `write_en`, соответствующий процессу `CPU`, отсоединяется от него, т. е. удаляется из списка источников, на базе которых определяется значение этого сигнала. Когда контроллер прямого доступа к памяти сбрасывает `hold_req`, `CPU` присоединяет источник сигнала `write_en` посредством выполнения транзакции, которая присваивает ему значение 0.

Другой пример. Модель процессора приведена в листинге 3.27.

Листинг 3.27

```
architecture rtl of processor is
  subtype word is bit_vector (0 to 31);
  type word_vector is array (natural range <>) of word;
  function resolve_unique (drivers: word_vector) return word is
  begin
    return drivers(drivers'left);
  end function resolve_unique;
```

```
  signal source1, source2:resolve_unique word register;
begin
  source1_reg: process (phase1,source1_reg_out_en,...) is
    variable stored_value:word;
  begin
    if source1_reg_out_en='1' and phace1='1' then
      source1<=stored_value;
    else
      source1<=null;
    end if;
  end process source1_reg;
  alu: process (alu_opcode, source1, source2,...);
end architecture rtl;
```

Это модель процессора на уровне регистровых передач. Процессор содержит два регистра — источника операндов, присоединенных к АЛУ. В модели только один процесс должен управлять значением каждого из сигналов, соответствующих регистрам, в каждый момент времени. Процесс `source1_reg` представляет собой управление работой первого регистра (сигнал `source_1`). Когда сигнал разрешения выходов этого регистра и сигнал тактирования имеют значение 1, процесс передает сигналу его прежнее значение. Решающая функция получает массив, содержащий один элемент — это значение сигнала, которое присваивается ему и используется при работе АЛУ. В конце фазы тактирования процесс отсоединяется от `source_1` путем выполнения **null**-транзакции. Поскольку source_1 является сигналом типа **register**, а все его источники в этом случае оказываются отсоединенными, решающая функция не вызывается, и значение сигнала остается прежним до тех пор, пока его источники снова не будут отсоединены. Это позволяет промоделировать реальную ситуацию, когда значение операнда сохраняется на выходах регистров.

Охраняемые сигналы составных типов. При работе с охраняемыми сигналами составного типа (например, массива) важно учитывать, что внутри каждого источника сигнала все элементы должны быть или присоединены, или отсоединены. *Недопустимо использовать нулевую транзакцию отсоединения части элементов.* Причина в том, что решающей функции всегда передаются полные значения составных типов, содержащиеся в источниках.

Если структура модели такова, что возникает необходимость в отсоединении сигнала по частям, то такой сигнал должен описываться не как единое целое, а как группа сигналов.

Использование механизма инерционных задержек. При установлении новых значений охраняемых сигналов может использоваться механизм *инерционных*

задержек. При этом нулевая транзакция выполняется по тем же правилам, что и остальные. Ее значение считается отличным от любого другого значения. Например, пусть список транзакций для сторожевого сигнала s в начале имеет следующий вид:

```
11 ns- 1  16 ns – 0  18 ns- 1  19 ns- null  25 ns- 0
```

В момент модельного времени 10 выполняется оператор следующего вида:

```
s<=reject 3 ns inertial null after 10 ns;
```

В результате список транзакций будет иметь следующий вид:

```
11 ns- 1  16 ns- 0  19 ns- null    20 ns- null
```

Первые две транзакции остаются без изменений, поскольку они попадают в интервал импульса (17—20 нс). Транзакция, запланированная на 18 нс., стирается, поскольку ее значение отличается от нового (нулевого); транзакция, запланированная на 19 нс., сохраняется, поскольку она непосредственно предшествует новой транзакции с тем же значением. Транзакция, запланированная на 25 нс., стирается, поскольку была запланирована до выполнения полученного оператора.

Атрибуты охраняемых сигналов. Для охраняемых сигналов могут использоваться атрибуты 'driving_value — значение драйвера и 'driving — имеет булевский тип и принимает значение true, если драйвер связан с сигналом.

Охраняемые порты. Порты объекта моделирования могут быть описаны как охраняемые сигналы. В этом случае их называют *охраняемыми портами*. На охраняемые порты накладывается ограничение: они могут быть только типа **bus**. При описании сигнала порта как сторожевого, **bus** указывается через пробел после указания типа сигнала. Пример приведен в листинге 3.28.

Листинг 3.28

```
entity tri_state_reg is
  port (d: in resolved_byte,
    q: out resolved_byte bus, ...);
```

Процессы, находящиеся в теле этого объекта, могут осуществлять нулевые транзакции с q. Следует учитывать, что значение q вычисляется независимо от значения внешнего сигнала, ассоциированного с q. Даже когда все процессы внутри tri_state_reg отсоединяют свои источники от q, решающая функция вычисляется и передает значение внешнему, ассоциированному с q, сигналу. VHDL не предоставляет механизма отсоединения портов от ассоциированных с ними сигналов, что создает определенные трудности на высоких уровнях абстракции. На более низких уровнях абстракции отсоединение может моделироваться как z-состояние.

Охраняемые сигналы в качестве параметров подпрограмм. VHDL не предоставляет возможности указания того, что формируемый параметр-сигнал является охраняемым, однако фактический сигнал, связываемый в момент вызова процедуры с формальным параметром типа сигнал, может являться охраняемым.

В подпрограмме к любому параметру типа сигнал может быть применена нулевая транзакция, но если на его место будет подставлен фактический сигнал, не являющийся охраняемым, это приведет к ошибке. Подпрограммы работают с драйверами сигналов тех процессов, которые их вызвали.

Например, модель устройства-регистратора, просматривающего и отображающего состояния двух портов, может иметь вид, представленный листингом 3.29.

Листинг 3.29

```
architecture hidht_level of data_logger is
  subtype byte is bit_vector(7 downto 0);
  type byte_array is array (integer range <>)of byte;
  function resolver (butes: byte_array) return byte is
  begin
    if butes'lehgth>0 then
      return bytes(butes'laft);
    else
      return X"00";
    end if;
  end function resolver;
  subtype resolved_byte is resolver byte;
  procedure reg (signal clock, out_enable :in bit;
      signal d: in byte; signal q: out resolved byte) is
    variable stored_byte : byte;
  begin
    loop
      if clock='1' then
        stored+byte:=d;
      end if;
      if out_enable='1' then
        q<=stored_byte;
      else
        q<=null;
      end if;
```

```
      wait on clock, out_enable, d;
    end loop;
  end procedure reg;
  signal data_bus:resolved_byte bus;
begin
a_reg: reg(a_reg_clk,a_reg_read,port_a,data_bus);
b_reg: reg(b_reg_clk,b_reg_read,port_b,data_bus);
end architecture hight_lavel;
```

Для каждого из портов предусмотрен входной регистр. В один момент времени устройство-регистратор выдает на входную шину состояние одного из регистров. Процесс `reg` активизируется с использованием параллельного вызова процедур.

Настраиваемые параметры модели(Generics)

Проектирование современных цифровых систем обработки информации на СБИС предполагает широкое повторное использование разработанных узлов, блоков, устройств. Однако для того чтобы узел мог применяться в достаточно большом числе проектов, он должен быть адаптируемым, настраиваемым к условиям конкретного применения (и по функциональным свойствам, и по условиям реализации). Одним из основных механизмов языка VHDL для этих целей является механизм настраиваемых параметров **`generic`**.

Предположим, имеется готовый проект узла на VHDL, реализованного и успешно работающего при технологии, обеспечивающей задержку на вентиль в 3 нс. Для этого узла есть весь комплекс поведенческих и структурных моделей, тестовых векторов. Естественно, в описании проекта узла на языке VHDL прописаны значения задержек компонентов схемы. Предположим, что они все нормированы относительно базовой временной характеристики — задержки на вентиль, прописанной как константа в моделях объектов. Эти задержки используются, например, при временном моделировании.

Теперь у нас появляются новые технологические возможности реализации СБИС, имеющей задержку на вентиль в 2 нс. Для использования готового проекта узла в новых проектах, ориентированных на такую технологию, придется исправлять значения базовых констант во всех текстах программ. При достаточно сложных, многокомпонентных проектах с развитой иерархической декомпозицией моделей, с использованием библиотек и пакетов, проделать такую коррекцию крайне сложно.

Использование настраиваемых параметров, декларированных как **`generic`**, позволяет создавать на VHDL модели проектируемых устройств, моделируемые и синтезируемые таким образом, что их функционирование зависит

от заданных настроек этих обобщенных параметров. Это стандартизованный языком VHDL механизм передачи параметров настройки по иерархии структурного описания объектов, составляющих модель проектируемого устройства. Ключевое слово **generic** (обобщенные, общие для всех) отражает суть этого понятия в языке VHDL.

Настраиваемые параметры **generic** описываются в декларативной части описания объекта моделирования, **entity** и в секции декларации компонентов. Пример приведен в листинге 3.30.

Листинг 3.30

```
library ieee;
use ieee.std_logic_1164.all;

entity or3 is
generic (gate_delay: time);
port (in1, in2, in3: in std_logic;
  out_pin: out std_logic);
end entity or3;

architecture behov_or3 of or3 is
begin
out_pin <= (in1 or in2 or in3) after gate_delay;
end architecture behov_or3;
```

Настраиваемые параметры **generic** — это константные значения. Их значения могут использоваться в выражениях, но не могут изменяться в ходе выполнения программы на VHDL.

Настраиваемые параметры широко применяются в проектах цифровых устройств на VHDL не только для параметризации задержек, но и для параметризации разрядности шин, числа портов объектов и др.

Определение настроек, задание в проекте значений для настраиваемых параметров **generic** выполняется конструкцией **generic map**, входящей в оператор назначения компонента, который включает экземпляр объекта-компонента в структурное описание объекта более высокого уровня. Синтаксис этого определения:

```
generic map (generic_association_list)
```

Примечание

Обратите внимание, что после конструкции **generic map** (...) символ ; не ставится.

Список generic_association_list состоит из пар,

generic_parameter => expression

перечисленных через запятую. Здесь generic_parameter — имя настраиваемого параметра; выражение generic_expression задает значение параметра. Еще есть возможность задавать так называемое позиционное соответствие, когда присваиваемое значение и какому настраиваемому параметру оно присваивается, определяются их положением в списках в **generic** и **generic map**. Однако мы бы не рекомендовали его применять, как чреватое сложно выявляемыми ошибками.

Выражение generic_expression может включать константы и настраиваемые параметры, декларированные как **generic**. Выражения, задающие значения параметров **generic** при настройке, должны быть вычислимы на этапе компиляции программы на VHDL.

Важным преимуществом настраиваемых параметров объектов проекта, определенных как **generic**, является то, что устанавливаемые **generic map** значения настраиваемых параметров распространяются вглубь иерархии объектов, составляющих модель.

При описании настраиваемых параметров **generic** в модели объекта, им может задаваться конкретное исходное значение. Например, в листинге 3.30 мы можем поставить

```
generic (gate_delay: time:=3 ns);
```

Описанная таким образом модель объекта становится исполняемой, т. е. ее можно автономно моделировать, можно автономно синтезировать по ней реализацию специфицированного на VHDL устройства.

Однако когда мы включим такой объект в качестве компонента в структурное описание другого объекта (листинг 3.31), а в спецификации объекта более высокого уровня, используя **generic map**, зададим иное значение для этого настраиваемого параметра, например,

```
generic map (gate_delay => 2 ns)
```

то значение параметра gate_delay в компоненте-экземпляре рассматриваемого объекта будет перегружено с 3 нс. на 2 нс. В этом наглядно проявляется "сквозной" характер настроек значений параметров **generic**, задаваемых **generic map**.

Листинг 3.31

```
architecture structure_examp of example is
   component three_input_or is
     generic (gate_delay: time);
     port (x,y,z: in std_logic;
```

```
    o out std_logic);
  end component three_input_or;
... ... ...
  for three_input_or use entity or3(behov_or3);
    begin
... ... ...
     unit10: three_input_or
     generic map (gate_delay=> 2 ns)
     port map (x=>r1, y=>r2, z=>r3, 0=>req);
... ... ...
end architecture structure_examp;
```

Здесь наглядно видны также отличия правил учета вложенности описаний и определений значений настраиваемых параметров **generic** от правил видимости описания констант во вложенных программных конструкциях. Локальное описание констант как бы закрывает их от влияния глобально определенных значений констант с теми же именами, в то время как параметры **generic** действуют с верхнего уровня задания значений на всю глубину вложенности содержащих их конструкций.

Блоки

Назначение блоков в модели

В общем случае механизм блоков используется для иерархической декомпозиции программы на VHDL, для управления областью действия описаний переменных, процедур, аналогично применению блоков в традиционных языках программирования.

Простейший синтаксис описания блоков имеет следующий вид:

```
block_label:
  block [(guarded_expression)] [is]
  {declarative_item}
  begin {concurrent_statement}
  end block [block_label];
```

Здесь:

- `block_label` — идентификатор, имя блока (метка в описании блока обязательна);
- `declarative_item` — раздел декларации локальных переменных, констант, подпрограмм и др. составляющих блока;
- `guarded_expression` — сторожевое выражение.

Тело блока, обрамленное **begin ... end block**, включает исполняемые операторы.

Сторожевое выражение — выражение булевского типа, которое служит для определения значения описанного внутри блока сигнала, называемого *сторожем (guard)*. Этот сигнал виден только внутри блока. Обращение к нему в явном виде, по имени, для модификации его значения или присваивания его какому-либо другому объекту невозможно, можно только использовать его значение в выражениях и проверять. Если в ходе моделирования изменяет свое значение какой-либо из сигналов, входящих в состав сторожевого выражения, то значение сторожевого сигнала **guard** немедленно пересчитывается. Механизм сторожевых выражений родственен механизму охраняемых присваиваний значений сигналам. Использование механизма сторожевых выражений позволяет сгруппировать внутри блока операторы присваивания, для которых условие выполнения включает в себя общий компонент. Роль сигнала **guard** подобна роли списка чувствительности для процесса.

Например (листинг 3.32).

Листинг 3.32

```
reg_read_selector: block (reg_sel='1' and read='1') is
begin dbus <= reg0 when guard and reg_addr='0' else
    reg1 when guard and reg_addr='1' else
    "Z...Z";
end block reg_read_selector;
```

В приведенном примере значение `dbus` будет пересчитываться каждый раз при изменении значения сигнала `reg_sel` или `read`, входящих в состав сторожа.

Охраняемые присваивания новых значений сигналам

Основная функция сторожевых выражений в блоке — управление операциями присваивания новых значений охраняемым сигналам. Если выполняется параллельный оператор присваивания нового значения сигналу, являющемуся охраняемым, то должен использоваться охраняемый оператор присваивания значения сигналу.

Оператор присваивания сторожевого сигнала может быть двух типов:

- условный оператор охраняемого присваивания значения сигналу;

```
name <=[guarded] [delay_mechanism]
  {waveform when boolean_expression else}
  waveform [when boolean_expression];
```

- селективный оператор охраняемого присваивания значения сигналу.

```
with expression select
  name <= [guarded] {delay_mechanism]
  {waveform when choices,}
  waveform when choices;
```

Внешне данные операторы отличаются от других наличием ключевого слова **guarded**. Это означает, что такой оператор выполняется, когда сигнал-сторож меняет свое значение. Если с помощью такого оператора определяется значение сторожевого сигнала (т. е. меняется значение какого-либо сигнала, на базе которого и происходит это определение), то, если сигнал-сторож меняет свое значение с `true` на `false`, драйвер сторожевого сигнала автоматически отсоединяется с использованием нулевой транзакции. При изменении значения сигнала-сторожа с `false` на `true` драйвер вновь присоединяется.

В качестве примера рассмотрим работу триггера-защелки. Когда сигнал `enable='1'`, то значение (на входе) защелкивается и передается на выход; в противном случае состояние выходного сигнала `q` остается неизменным вне зависимости от изменения входного сигнала `d`. Листинг 3.33 иллюстрирует подобную ситуацию.

Листинг 3.33

```
entity latch is
  generic (width: positive);
  port (enable: in bit;
    d: in bit_vector (0 to width-1);
    q: out bit_vector(0 to width-1);
end entity latch;
architecture behavior of latch is
begin
 transfer_control: block (enable='1') is
  begin
    q <= guarded d;
  end block transfer_control;
end architecture behavioral;
```

Другой пример приведен в листинге 3.34. Рассмотрим архитектурное тело для процессорного узла мультипроцессорного компьютера. Пусть `address_bus` — сторожевой сигнал типа `bit_vector`. Блок, помеченный `cashe_to_address_buffer`, имеет выражение-сторож, которое принимает зна-

чение true, когда содержимое блока cashe_to_address_buffer становится неверным и блок требует замены (dirty='1'). Как только происходят какие-либо изменения в значениях сигналов, на базе которых определяется значение сигнала address_bus, сразу вычисляется его новое значение, которое помещается в источник. Однако источник является подключенным только при значении сторожа true, при этом значение источника сигнала видно в модели. Аналогично функционирует блок snoop_to_address_buffer. Логические выражения-сторожа построены таким образом, что только одно из них в конкретный момент принимает значение true. Таким образом, в каждый момент времени address_bus оказывается связанным только с одним из своих драйверов.

Листинг 3.34

```
architecture dataflow of processor_node is
  signal address_bus: resolve_unique word bus;
begin
   cache_to_address_buffer: block(cache_miss='1' and dirty='1') is
  begin
    address_bus<=guarded
    tag_section0&set_index&B"0000" when replace_section='0'
      else tag_section1&set_index&B"0000";
  end block cache_to_address_buffer;

   snoop_to_address_buff: block (snoop_hit='1' and flag_update='1') is
  begin
    address_bus<=guarded snoop_address(31 downto 4) & B"0000";
   end block snoop_to_address_buff;
end architecture dataflow;
```

Если операторы присваивания значения сторожевого сигнала используются для определения значения сигнала, не являющегося сторожевым, это не вызовет сообщения об ошибке, но и действия по отключению и подключению драйвера выполняться не будут.

Явно определенные сигналы-сторожа

Описанный выше механизм неявного определения сигналов-сторожей на базе сторожевых выражений, задаваемых в заголовке описания блока, оказывается недостаточным для многих моделей. VHDL предоставляет и более общий механизм для определения сигнала-сторожа — механизм *явного определения сигнала-сторожа.*

При явном определении сигнал-сторож также имеет булевский тип. При изменении его `true > false` драйверы сторожевых охраняемых сигналов отключаются, а при изменении `false > true` — подключаются.

Явно определенный сигнал-сторож обязательно должен носить имя **guard**. Его действие распространяется на всю область его определения и, соответственно, в этой области не может быть другого сигнала-сторожа.

Пример приведен в листинге 3.35.

Листинг 3.35

```
architecture abstract of computer_system is
  signal address_bus: resolve_word word bus;
  signal hold_req:bit;
begin
  cpu: block is
    signal guard: boolean:=false;
    signal cpu_internal_address: word;
  begin
    cpu_address_driver:
      address_bus<=guarded cpu_internal_address;
          -- остальные источники
  controller: process is
    begin  -- отпределяет когда cpu_bus_drivers=disable
      guard<=false;
      wait on clk until hold_req='0' and clk='1';
      guard<=true;      -- разрешение источников Cpu
    end process controller;
  end block cpu;
    -- далее могут следовать описания других блоков, входящих в состав
системы
end architecture abstract;
```

Модель компьютерной системы включает в себя CPU и контроллер прямого доступа к памяти. Контроллер прямого доступа к памяти выставляет сигнал hold_req, когда ему нужно работать с памятью. В этом случае CPU завершает текущую операцию и отсоединяет свои драйверы, затем он выдает подтверждение на использование шины. Адресная шина описывается сигналом cpu_address_driver. Сигнал guard контролирует этот сигнал. Процесс controller описывает управляющую секцию сигнала CPU, он управляет сигналом guard.

Спецификация отсоединения (disconnect)

Иногда в моделях бывает необходимо включить механизм задержек в оператор определения значения охраняемого сигнала (например, когда с этим сигналом может выполняться неявная нулевая транзакция в результате изменения значения сигнала **guard**).

Для включения механизма задержек в работу с охраняемым сигналом используется *спецификация отсоединения* с определением механизма задержек. Она имеет следующий синтаксис:

```
disconnect (signal_name {,...} | others | all): type_mark
  after time_expression;
```

Этот механизм позволяет для каждого сигнала из списка определить задержку, которая будет ему соответствовать при неявном выполнении для него нулевой транзакции. Такое определение никак не воздействует на выполнение явно описанных транзакций с сигналом.

Спецификация отсоединения должна быть включена в тот же список описаний, в котором описаны использованные в ней сигналы (листинг 3.36).

Листинг 3.36

```
signal mem_data_bus: resolved_word bus;
disconnect mem_data_bus: resolved_word after 3 ns;
mem_wr_buff: block (mem_sel and mem_write) is
 begin
     mem_data_bus<=guarded reject 2 ns internal cash_d_bus after 4 ns;
 end block mem_wr_buff;
```

Пока сторож имеет значение `true`, значение сигнала `cash_d_bus` будет копироваться в сигнал `mem_data_bus` с задержкой 4 нс., при минимальной длительности импульса 2 нс. После того как сторож примет значение `false`, источник отсоединится с задержкой 3 нс.; однако значение самого источника будет изменяться в соответствии с изменением `cash_d_bus`. После того как охраняемый сигнал вновь примет значение `true`, `mem_data_bus` получит значение из буфера с задержкой 4 нс.

Для того чтобы действие декларации отсоединения распространялось на все охраняемые сигналы данного типа, в декларации указывается ключевое слово **all**. В таком случае декларация должна стоять *после* описания всех сигналов этого типа.

Для того чтобы действие декларации отсоединения распространялось на все сигналы данного типа, которые еще не декларированы, необходимо использовать ключевое слово **others**. Декларация отсоединения, включающая в себя это слово, должна стоять *последней* в списке описаний (деклараций) блока.

Вложенность блоков и организация иерархической структуры модели

Полный синтаксис описания блока имеет следующий вид:

```
block_label: block[(duarded_expression)] [is]
  [generic (generic_interface_list);
  [generic map (generic_association_list);] ]
  [port (port_interface_list);
  [port map (port_association_list);] ]
  {block_declarative_item}
begin
  {concurrent_statements}
end block [block_label];
```

Декларативная часть позволяет описать объекты, которые будут использоваться только внутри блока, которому она принадлежит. В декларативной части блока описываются любые типы объектов, которые могут присутствовать в декларативной части архитектурного тела. Все объекты, видимые внутри того архитектурного тела, к которому принадлежит блок, видимы внутри блока. Сами описания блоков располагаются внутри архитектурного тела.

Блок может содержать внутри себя другие блоки. Правила видимости для вложенных блоков такие же, как и для вложенных процедур.

Вложение блоков друг в друга позволяет создавать иерархические структуры моделей, однако на практике вложенность блоков используется редко. Как правило, для декомпозиции проекта и описания иерархической структуры объекта используется механизм *компонентов*.

Для организации обобщенного описания блоков используются конструкции **generic** и **generic map**. Первая из них, **generic**, служит для описания объектов, значения которых задаются во второй конструкции, в **generic map**. Аналогично, конструкции **port** и **port map** служат для декларации портов и связывания входных и выходных сигналов блока с внешним миром.

Пример приведен в листинге 3.37.

Листинг 3.37

```
architecture contrived of example_entity is
  constant sig_width: positive:=16;
  signal s1,s2,s3: bit_vector (0 to sig_width-1);
  signal sel: bit;
```

```
begin
  mux: block
    generic (width: positive);
    generic map (width=>sig_width);
    port (
      d0,d1: in bit_vector(0 to width-1);
      y: out bit_vector (0 to width-1);
      sel: in  bit);
    port map(d0=>s1, d1=>s2, y=>s3,sel=>sel);
    constant zero: bit_vector(0 to width-1):=(others=>'0');
    signal gated_d0,gated_d1: bit_vector (0 to width-1);

    begin
      gated_d0<=d0 when sel='0' else zero;
      gated_d1<=d1 when sel='1' else zero;
      y<=gated_d0 or gated_d1;
    end block mux;
end architecture contrived;
```

Конфигурирование объектов, содержащих блоки

Конфигурирование блоков подобно конфигурированию компонентов. Когда пишется конфигурационная декларация для архитектурного тела, содержащего блоки, она должна отражать структуру блоков.

Декларация архитектурного тела в этом случае имеет следующий синтаксис:

```
configuration configuration_name of entity_name is
  block_configuration
end [configuration] [configuration_name];
```

Секция `block_configuration` имеет здесь следующий синтаксис:

```
for (architecture_name | block_label)
 {block_configuration |
  for component_specification [binding_indication;] [block_configuration]
  end for;}
end for
```

Из синтаксиса конструкции видно, что секции `block_configuration` могут вкладываться друг в друга.

В листинге 3.38 приведена в качестве примера модель интегральной схемы, которая определяет задержки на входах и выходах.

Листинг 3.38

```
-- Декларация объекта моделирования circuit
entity circuit is
  generic (inpad_delay, outpad_delay: delay_length);
  port (in1,in2,in3: in bit; out1,out2: out bit);
end entity circuit;

-- описание архитектуры with_pad_delays для объекта circuit
architecture with_pad_delays of circuit is
  component subcircuit is
    port (a, b: in bit; y1,y2: out bit);
  end component subcircuit;

  signal del_in1, del_in2, del_in3: bit;
  signal undel_out1,undel_out2: bit;

begin
input_delays: block is
  begin
       del_in1<=in1 after inpad_dalay;
  del_in2<=in2 after inpad_dalay;
       del_in3<=in3 after inpad_dalay;
  end block input_delays;

functionality: block is
  signal intermediate: bit;
       begin
  cell1: component subcircuit
    port map (del_in1, del_in2, undel_out1,intermadiate);
  cell2: component subcircuit
    port map (intermediate, del_in3, undel_out2,open);
       end block functionality;
output_delays: block is
       begin
  out1<=undelayed_out1 after outpad_delay;
  out2<=undelayed_out2 after outpad_delay;
       end block output_delays;

end architecture with_pad_delays;
```

```
-- Декларация конфигурации full для объекта circuit
configuration full of circuit is
-- для архитектурного описания "with_pad_delays"
  for with_pad_delays
-- для блока "functionality"
    for functionality
-- задаем конфигурацию, область действия которой – данный блок
      for all: subcircuit
        use entity work.real_subcircuit(basic);
      end for;
    end for;
  end for;
end configuration full;
```

В этом примере архитектурное тело `with_pad_delays` объекта `circuit` включает в себя блок input_delays, моделирующий входную задержку, блок output_delays, моделирующий выходную задержку, блок functionality, моделирующий основную функцию объекта. Конфигурационная декларация связывает компоненты `subcircuit` блока `functionality` (все компоненты, **all**, их там два — `cell1` и `cell2`) с объектами `real_subcircuit`, с архитектурой `basic` из библиотеки `work`. Конструкция **for** `with_pad_delays` указывает имя архитектурного тела, конфигурирование которого выполняется; конструкция **for** `functionality` определяет, в рамках какого блока будет действительна эта конфигурация.

Декларируемая конфигурация `full` не содержит конфигурационных спецификаций для двух других блоков модели, поскольку в их состав компоненты не входят.

Подпрограммы

В полном соответствии с классической терминологией языков программирования, в языке VHDL процедуры и функции объединяются общим термином — подпрограмма.

Процедуры

Описание процедуры.

Описание процедуры имеет следующий синтаксис:

```
procedure name [(parameter_interface_list)] is
  {subprogram_declarative_part}
```

```
begin
  {sequential_statement}
end [procedure] [name];
```

Декларативная часть процедуры `subprogram_declarative_part` может включать в себя описание типов, подтипов, констант и подпрограмм, которые в дальнейшем могут использоваться только внутри этой процедуры. Эти описания являются локальными.

Сигналы не могут быть описаны как локальные данные подпрограмм.

Например, описание процедуры может быть таким, как представлено в листинге 3.39.

Листинг 3.39

```
procedure average_samples is
  variable total: real:=0.0;
begin
  assert samples'length > 0 severity failure;
  for index in samples'range loop
    total:=total+samples(index);
end loop;
aveage:=total/real(samples'length);
end procedure average_samples;
```

Обратите внимание на условие в операторе **assert**. В условии указано значение величины `samples'length`. Если это значение положительно — оно рассматривается как правильное, в противном случае возникает ошибка уровня `failure`.

Тело процедуры может содержать вызовы других подпрограмм.

Список параметров. Если в процедуре имеется список параметров, то его описание имеет следующий синтаксис:

```
([ constant | variable | signal ] identifier {,...} : [in | out | inout]
    subtype_indication [:=static_expression]){,...}
```

Для каждого параметра может задаваться: класс данного (константа, переменная, сигнал); имя, под которым он будет использоваться в процедуре; вид (**in** | **out** | **inout**), тип и, возможно, начальное значение.

Пример приведен в листинге 3.40.

Листинг 3.40

```
procedure do_arith_op (op: in func_code) is
  variable result: integer;
```

```
begin
  case op is
    when add => result :=op1+op2;
    when subtract => result :=op1-op2;
  end case;
  dest<=result after Tpd;
end procedure do_arith_op;
```

Параметрам процедуры могут присваиваться значения по умолчанию. Значения по умолчанию могут присваиваться только параметрам, константам или переменным вида **in** (но не сигналам!).

В качестве формальных параметров процедур могут использоваться неограниченные массивы. Это позволяет расширить область применения процедур за счет того, что фактические параметры при вызове смогут принимать более широкий диапазон значений. Соответствующие фактические параметры должны быть массивами с заданным диапазоном, тип их элементов должен совпадать с типом элементов неограниченного массива — формального параметра.

Суммарная информация о параметрах процедур. Итак, описание каждого формального параметра может иметь пять аспектов:

- класс объекта (константа, переменная, сигнал);
- имя формального параметра;
- вид (**in**, **out**, **inout**), указывающий направление обмена между процедурой и вызвавшим ее объектом, а также возможность модификации значения этого параметра внутри процедуры;
- тип или подтип формального параметра;
- значение по умолчанию.

Оператор **return**. Оператор используется для завершения процедуры, до того как она будет полностью выполнена. Действие этого оператора по отношению к процедуре сходно с действием оператора **exit** по отношению к циклу.

Оператор **return** имеет следующий синтаксис:

```
[label:] return;
```

Например:

```
if reset;='1' then return; end if;
```

Вызов процедуры

Вызов процедуры имеет следующий синтаксис:

```
[label:]name[(parameter_list)];
```

Например, для описанной ранее (в листинге 3.39) процедуры:

```
average_samples;
```

При вызове процедуры, содержащей список параметров, необходимо указать их фактические значения, которые должны совпадать по типу с формальными, описанными в заголовке процедуры.

Описание списка формальных параметров имеет следующий синтаксис:

```
([parameter_name=>], expression | signal_name | variable_name | open,
{...})
```

Например, вызов вышеописанной процедуры может иметь следующий вид:

```
do_arith_op(add);  --(1)
do_arith_op(func);  --(2)
```

Параметру процедуры может присваиваться и константное значение, как в первом примере, и значение сигнала того же самого типа, как во втором примере.

Если для формального параметра указан вид **in**, он используется именно как *входной*, т. е. его значение не может быть модифицировано внутри процедуры. Аналогичный результат будет, если перед именем параметра поставить ключевое слово **constant**. Для облегчения понимания программы эти дублирующиеся описания могут использоваться одновременно.

Если значение объекта модифицируется в ходе выполнения процедуры, то его вид определяется как **out**. Аналогичного результата можно достигнуть, поставив перед именем параметра ключевое слово **variable**. Фактические значения таких параметров не могут быть ни константами, ни выражениями, а должны быть переменными.

Пример приведен в листинге 3.41.

Листинг 3.41

```
Procedure addu(a, b: in word32;
      result: out word32) is
variable sum: word32;
variable carry: bit :='0';
begin
  for index in sum'reverse_range loop
    sum(index) := a(index) xor b(index) xor carry;
    carry := (a(index) and b(index)) or (carry and (a(index) xor
b(index)));
  end loop;
result := sum;
end procedure addu;
```

Если параметр является и входным, и выходным, то для него следует указывать вид **inout**. Фактические значения параметров **inout** не могут быть константами.

При вызове процедуры, соответствующему формальному параметру вида **inout** присваивается значение параметра фактического. В ходе выполнения этой процедуры значение формального параметра может модифицироваться. По окончании выполнения процедуры параметру присваивается последнее значение, присвоенное ему в ходе выполнения тела процедуры.

Пример приведен в листинге 3.42.

Листинг 3.42

```
procedure negative (a: inout word32) is
  variable carry_in: bit :='1';
  variable carry_out: bit;
begin
a := not a;
for index in a'reverse_range loop
  carry_out := a(index) and carry_in;
  a(index) := a(index) xor carry_in;
  carry_in := carry_out;
end loop;
end procedure negative;
```

При вызове процедуры, параметры которой имеют значения по умолчанию, для использования этих значений, в списке фактических параметров указывается ключевое слово **open**. Если такой параметр последний в списке формальных параметров, то в списке фактических параметров он может быть опущен.

Например, пусть имеется процедура, имеющая следующий заголовок:

```
procedure increment (a: inout word32; by: in word32 :=X"0000_0001") is ,
```

тогда вызов процедуры может иметь следующий вид:

```
increment(count);
increment(count, by=>open);
```

Эти два вызова эквивалентны. В обоих случаях значение `count` будет увеличиваться на 1.

```
increment (count, X"0000_0004");
```

В этом случае значение `count` будет увеличиваться на 4.

Сигналы в качестве параметров процедуры.

Формальный параметр процедуры может быть сигналом. В этом случае перед его именем должно обязательно присутствовать ключевое слово `signal`. Такой параметр также может использоваться в видах **in**, **out** и **inout**.

Для параметров — констант и переменных — передача параметров в/из процедуры осуществляется по значению. В отличие от них, при вызове процедуры с параметром-сигналом происходит передача параметра не по значению, а по ссылке (иначе говоря — по имени). Иными словами, значение параметра-сигнала не копируется в тело процедуры, а соответствующее ему в теле процедуры обозначение формального параметра ассоциируется с сигналом, указанным фактическим параметром, и находящимся вне тела процедуры. Поэтому в ходе выполнения процедуры значение сигнала может меняться не только в результате выполняемых в процедуре действий, но и в результате действий, выполняемых параллельно с ней (другими процессами, параллельными операторами).

В процедуре каждый раз при обращении к параметру-сигналу происходит обращение к вызвавшему ее процессу (для получения текущего состояния сигнала, указанного фактическим параметром). В результате, в отличие от параметра-переменной, используется не значение параметра-сигнала на момент вызова процедуры, а текущее значение сигнала в каждый момент его использования в теле процедуры.

Пример приведен в листинге 3.43.

Листинг 3.43

```
architecture behavioural of receiver is
signal recovered_data: bit;
signal recovered_clock: bit;

procedure receive_packet (signal rx_data: in bit;
        signal rx_clock: in bit;
        data_buffer: out packet_array) is
begin
  for index in packet_array'range loop
    wait until rx_clock='1';
    data_buffer(index):= rx_data;
  end loop;
end procedure receive_packet;
begin
packet_assembler: process is
```

```
  variable packet: packet_array;
begin
receive_packet (recovered_data, recovered_clock, packet);
end process packet_assembler;
end architecture behavioural;
```

В ходе выполнения процесса `packet_assembler` происходит обращение к процедуре `receive_packet`. После вызова процедуры управление от процесса переходит ней. Однако когда выполнение процедуры доходит до оператора **wait**, управление вновь передается отложенному процессу. Процесс продолжает оставаться отложенным, но при этом он становится чувствителен к изменению сигнала, который указан в операторе **wait**. Как только значение этого сигнала станет отличным от 1 (что задано оператором **wait until** `rx_clock='1'`), выполнение процедуры продолжится. Следующий оператор процедуры, в котором происходит присваивание объекту значения сигнала, также производит обращение к процессу, вызвавшему процедуру. В результате, в операторе используется *текущее* значение сигнала, которое может отличаться от того, которое было на момент вызова процедуры.

В операторе присваивания `data_buffer(index):= rx_data;` происходит присваивание переменной (элементу массива) значения сигнала. Обратите внимание на форму оператора присваивания (:=, а не <=. Форма используемого оператора присваивания определяется объектом, который стоит в левой части выражения.

Если используется формальный параметр-сигнал вида **out**, то изменение значения этого параметра в процедуре влечет за собой изменение значения соответствующего ему фактического параметра-сигнала в вызвавшем процедуру процессе (так, как если бы изменение этого сигнала происходило непосредственно в процессе).

Каждый процесс, который работает с конкретным сигналом, должен иметь единственный источник, драйвер для этого сигнала. Для каждого сигнала процесс имеет один и только один драйвер. Однако каждый процесс, который работает с определенным сигналом, имеет свой собственный драйвер для него. Транзакции работают с сигналами посредством этих драйверов. Транзакция, сгенерированная в результате действия процедуры, работает с драйвером сигнала того процесса, который ее вызвал [3].

Формальные параметры-сигналы также могут иметь вид **inout**. В этом случае в качестве фактического параметра должен выступать сигнал, внешний к объекту моделирования, а не описанный внутри него. Это связано с тем, что в противном случае возможна неоднозначность при определении текущего значения сигнала, описанного таким образом (например, если процедура была неоднократно вызвана несколькими процессами внутри одного объекта мо-

делирования, в результате чего драйверы сигнала у разных процессов получили различные значения).

Операторы параллельного вызова процедур

Операторы параллельного вызова процедур имеют тот же синтаксис, что и операторы последовательного вызова. Однако они имеют статус процессов и, следовательно, отличаются от последовательных вызовов процедур тем, что размещаются не внутри процессов, а наряду с ними, в архитектурном теле объекта моделирования.

Параллельный вызов процедуры осуществляется в начале моделирования и при каждом изменении ее входных параметров. Таким образом, если какой-либо процесс должен выполняться только один раз в начале моделирования, то его удобно реализовать в виде вызываемой параллельно процедуры, не имеющей входных параметров. Кроме того, если в системе имеется несколько процессов, реализующих одни и те же действия (например, контрольные), то эти процессы целесообразно реализовать в виде процедуры, параллельно вызываемой в соответствующих местах.

Функции

Другим видом подпрограмм в языке VHDL являются функции.

Функции открывают широкие возможности по определению новых операторов.

Описание функции имеет много общего с описанием процедуры. Оно имеет следующий синтаксис:

```
[pure | impure]
function identifier [(parameter_interface_list)] return type_mark  is
  {subprogram_declarative_item}
begin
  {sequential_statement}
end [function] [identifier];
```

В отличие от процедуры, функция должна обязательно возвращать результат, тип которого определяется в описании после ключевого слова **return**.

Список формальных параметров функции имеет тот же самый синтаксис, что и список формальных параметров процедуры. На формальные параметры функции накладываются следующие ограничения:

- формальные параметры не могут принадлежать к классу переменных;
- если класс не определен в явном виде, то предполагается, что это константа;
- формальные параметры должны иметь вид **in**.

В ходе выполнения функции модификация ее формальных параметров невозможна. Модификация формальных параметров функции могла бы привести к серьезным проблемам при вычислении значения выражения, в котором эта функция используется (например, если в вычисляемое выражение входят также и объекты, используемые в качестве фактических параметров этой функции).

Пример функции, конвертирующей значение битового вектора в натуральное число, приведен в листинге 3.44.

Листинг 3.44

```
function bv_to_natural (bv: in bit_vector) return natural is
  variable result: natural :=0;
begin
  for index in bv'range loop
    result:=result*2+bit'pos(bv(index));
  end loop;
  return result;
end function bv_to_natural;
```

Пример использования функции может иметь также и следующий вид:

```
type rom_array (natural range 0 to rom_size-1) of bit_vector (0 to
word_size-1);
variable rom_data: rom_array;
data<=rom_data(bv_to_natural(address)) after Taccess;
```

Вызов функции обязательно является частью выражения. В остальном вызов функции имеет синтаксис, аналогичный вызову процедуры.

Функция обязательно должна возвращать какое-то значение. Следовательно, она не может завершаться просто по выполнению оператора `end` тела функции, поскольку в этом случае значение функции не будет определено.

Тело функции должно содержать как минимум один оператор `return`, а возможно, и больше. В этом операторе определяется значение, возвращаемое функцией.

Оператор **`return`** имеет следующий синтаксис:

```
[label:] return expression ;
```

Функции вида ***Pure*** *и* ***Impure***. Функцию можно рассматривать как обобщенную форму оператора. Если мы в различных выражениях используем какой-либо оператор с одними и теми же параметрами, то оператор будет возвращать одно и то же значение. Но в случае использования функции

это не всегда так, поскольку в ходе своего выполнения функция может работать не только со своими формальными параметрами, но и с глобально описанными переменными и сигналами объекта, внутри которого описана данная функция. Иными словами, функция может иметь побочный эффект. Кроме того, учитывая, что язык VHDL — язык параллельный, то переменные и сигналы, внешние относительно тела функции, могут изменять свои значения между вызовами функции, что может повлиять на значение самой функции.

Если такого влияния не должно быть, то перед ключевым словом **function** в описании функции должно стоять ключевое слово **pure** (буквально — чистая), что является гарантией того, что в результате всех вызовов функции с одними и теми же фактическими параметрами будет получен один и тот же результат. Если же это условие не обязательно, то используется ключевое слово **impure**, которое подразумевает, что при вызове функции с одними и теми же фактическими параметрами в разные моменты времени могут быть получены различные результаты. По умолчанию, если в описании функции не указано ни **pure**, ни **impure**, считается, что функция считается **impure**.

Функция ***now***. Примером функции **impure** является имеющаяся в языке VHDL встроенная функция **now**. Функция **now** возвращает текущее время моделирования. Этой функции соответствует следующее определение:

```
impure function now return delay_length
```

В процессе моделирования функция **now** возвращает различные значения при вызове ее в различные моменты времени в процессе моделирования. Тип `delay_length` является подтипом **time** и используется для задания неотрицательных отрезков времени.

Функция **now** нередко используется для проверки соответствия входов модели задаваемым ограничениям. Так, с ее помощью можно отслеживать ситуации, когда значение информационного сигнала на входе триггера изменяется после прихода активного фронта тактового импульса слишком быстро, в результате чего это значение не может быть передано на выход. Например, эта проверка может быть реализована в виде отдельного процесса (листинг 3.45).

Листинг 3.45

```
process (clk, d) is
  variable last_clk_edge_time:time := 0 fs;
  constant Thold_d_clk:time :=5 fs
```

```
begin
  if clk'event and clk='1' then
    last_clk_edge_time := now;
  end if;
  if d'event then
    assert now - last_clk_edge_time >= Thold_d_clk
      report "hold time violation";
  end if;
end process;
```

В начале моделирования переменной `last_clk_edge_time` присваивается значение 0. При каждом переднем фронте тактового сигнала ей присваивается значение момента времени, в который оно произошло. При каждом изменении информационного сигнала определяется разница между текущим временем и временем прихода последнего, на данный момент, активного фронта, значение которого хранится в переменной `last_clk_edge_time`. Если разница мала, то в отчете формируется сообщение об ошибке, поскольку в реальной системе такая ситуация приведет к тому, что на выходе триггера значение входного сигнала не появится.

Функции являются популярным механизмом в функциональном моделировании. В функции оформляют описание выполняемых преобразований над данными, а вызовы функций используются в параллельно выполняемых операторах присваивания новых значений сигналам [3].

Перегрузка процедур и функций

При написании подпрограмм им обычно присваиваются имена, соответствующие выполняемым действиям, что значительно облегчает понимание программы. Однако нередко оказывается, что одни и те же действия необходимо выполнять над разными наборами данных. Эти наборы данных могут отличаться типами отдельных элементов и их количеством. VHDL предоставляет так называемый *механизм перегрузки имен*, который позволяет нескольким подпрограммам иметь одно и то же имя.

Списки формальных параметров в описаниях перегруженных процедур или функций обязательно должны отличаться друг от друга количеством или типом элементов.

Когда в программе осуществляется вызов перегруженной подпрограммы, то *тип и количество фактических параметров* определяют, какая именно подпрограмма, по какому из описаний с одним и тем же именем, будет вызвана.

Например:

```
-- определение функции 1
procedure increment (a: inout integer; n: in integer :=1) is...
-- определение функции --2
procedure increment (a: inout bit_vector; n: in bit vector :=B"1") is...
variable count_int: integer:=2;
variable count_bv: bit_vector(15 downto 0) :=X"0002";
```

Тогда при использовании

```
increment (count_int)
```

будет вызвана процедура, помеченная номером 1, а при использовании

```
increment (count_bv)
```

будет вызвана процедура, помеченная номером 2.

Перегрузка операторных символов

Каждый оператор языка VHDL фактически является специальным образом определенной функцией. Каждый из них может работать с операндами определенных типов. Нередко при определении пользователем новых типов данных возникает необходимость расширить или изменить область применения операторов. Это можно сделать, описав перегруженную функцию в соответствии с обычными правилами описания. При этом в определении функции, в качестве имени функции, в кавычках указывается обозначение перегружаемого оператора.

Например:

```
function "+"(left, right: in bit_vector) return bit_vector is
...
end function "+";
```

Вызов этой функции ничем не отличается от использования оператора сложения:

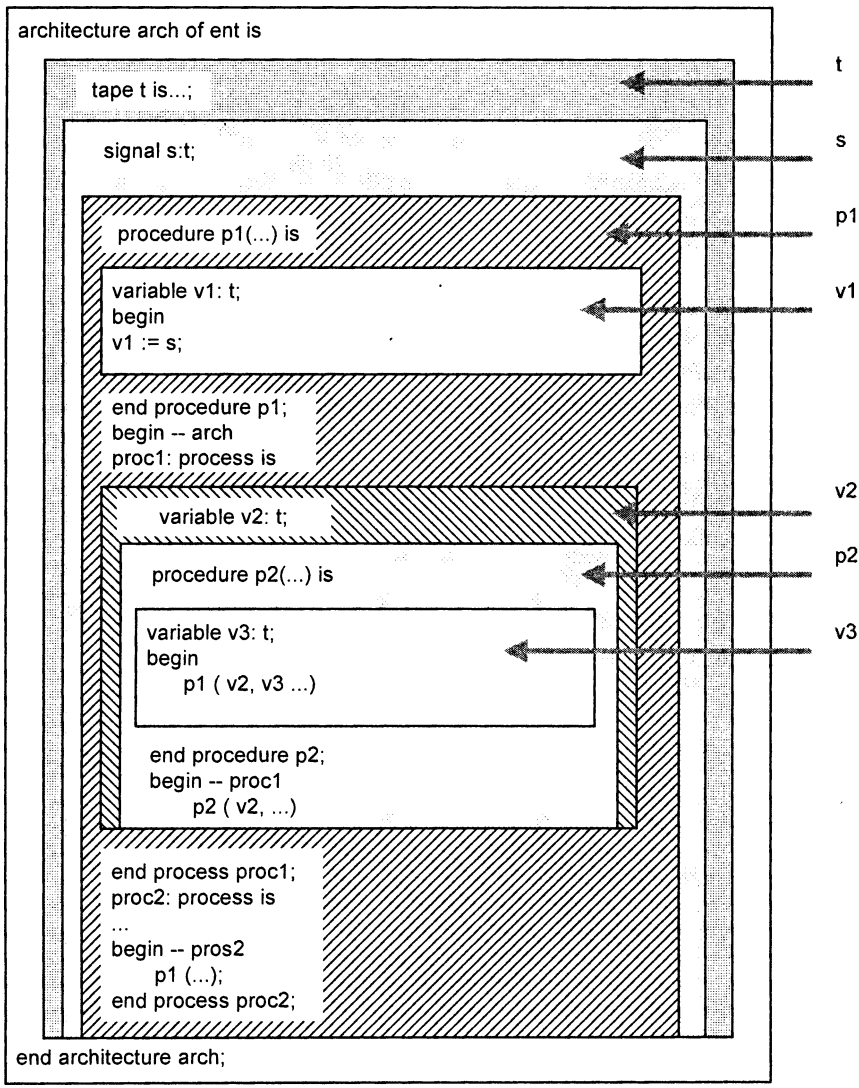
```
variable a1,a2: bit_vector (15 downto 0);
a2:=a1+X"0002";
```

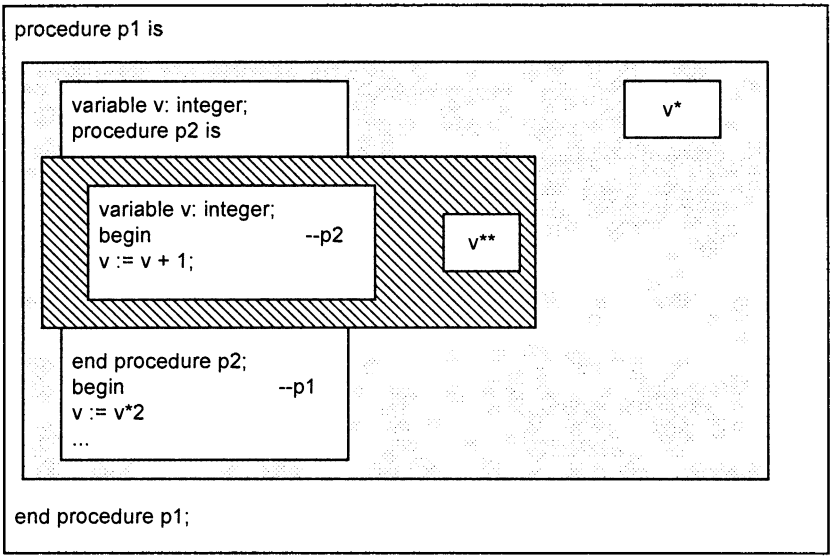
Видимость описаний

Правила видимости описаний в VHDL достаточно традиционны для языков программирования. Архитектурные объекты, процессы и подпрограммы состоят из двух частей: декларативной части и тела. Все описания, сделанные в декларативной части объекта, видны во всем этом объекте, но не за его пределами.

Рассмотрим пример (рис. 3.12, *а*). Области видимости объектов на рисунке ограничены линиями. Если на разных уровнях описания объекты получают одно и то же имя, то они рассматриваются как различные объекты, причем в рамках каждого уровня описания может использоваться только свой объект (см. пример на рис. 3.12, *б*).

Переменная `v` процедуры `p1` не видна в процедуре `p2`, поскольку там она перекрыта другой переменной с тем же именем.





б

Рис. 3.12. Области видимости объектов в описании

Описание подпрограмм в пакетах

Если в декларативной части пакета были сделаны описания процедур или сокращенные описания констант, то обязательно должно быть приведено описание тела пакета, содержащее полные описания этих объектов.

Полное описание подпрограммы состоит из заголовка и описания тела. Заголовок полного описания в точности соответствует заголовку в декларативной части пакета (имя подпрограммы, имена, типы параметров и др.). По умолчанию, в полном описании вид и значения формальных параметров должны полностью совпадать с параметрами, заданными в декларативной части пакета. Допускается только два отличия:

- формы записей числовых значений могут быть разными, но при этом их значения должны совпадать;
- простые имена могут быть заменены именами с указанием ссылки на библиотеку и, возможно, пакет (при условии, что эти имена соответствуют одному и тому же объекту).

Файлы

Файлы в проектах на VHDL находят широкое применение. Без файлов практически не обойтись при разработке и исследовании моделей проектируемых устройств. Файлы могут использоваться для хранения данных, не-

однократно участвующих в моделировании, могут формироваться текстовые файлы для входных и выходных данных, и др.

В языке VHDL для работы с файлами введен специальный тип объектов — файловые типы **file**. То, что в VHDL понятие файла включено в сам язык, а не оставлено на усмотрение среды проектирования и моделирования, является важным преимуществом, существенно улучшая переносимость проектов, выполненных на VHDL.

Описание файла

Описание *файлового типа* имеет следующий синтаксис:

```
type file_type_name is file of subtype_indication;
```

Описание *объекта файлового типа* имеет следующий синтаксис:

```
file file_name is file_type_name [[open file_open_kind_expression] is
string_expression];
```

Здесь:

- идентификатор `file_name` — имя логического файла в модели;
- идентификатор `file_type_name` — имя файлового типа;
- секция **open** `file_open_kind_expression` позволяет определить режим открытия файла;
- секция **string_expression** позволяет связать логический файл с конкретным физическим файлом на диске.

Такое описание позволяет создать один или более файловых объектов заданного типа.

При описании объекта файлового типа после слова **is** может следовать не только имя типа, но и непосредственное описание этого типа.

Например:

```
file fill1 is file of integer text.dat;
```

В результате будет создан логический файл `fill1`, связанный с физическим файлом `text.dat`.

После ключевого слова **open** может быть указан режим открытия файла: `read_mode` (открыт для чтения), `write_mode` (открыт для записи), `append_mode` (открыт для продолжения записи, для присоединения). Эти значения включены во встроенный тип `file_open_kind`.

В *стандарте VHDL 87* формат описания файла другой и имеет следующий синтаксис:

```
file identifier: subtype_indication is [in|out] string_expression;
```

Режим чтения определяется по умолчанию.

Работа с файлами

При открытии файла работа с ним выполняется последовательно, начиная с первого элемента.

Чтение из файла. По прочтении очередного элемента указатель позиции автоматически перемещается на следующий элемент. Чтение выполняется с помощью процедуры `read`. Для определения окончания файла используется функция `endfile`.

Если файл описан следующим образом:

```
type file_type is file of element_type;
procedure read (file f: file_type; value: out element_type);
function endfile (file f: file_type) return boolean;
```

то возможен, например, фрагмент, приведенный в листинге 3.46.

Листинг 3.46

```
type load_file_type is file of word;
file load_file: load_file_type open read_mode is 'tt.dat';
variable abload: word;
variable index: integer;
index:=0;
while not endfile (load_file) loop
  read (load_file, abload);
end loop;
```

Элементы файла могут иметь не строго определенную длину:

```
type bit_vector_file is file of bit_vector;
```

Для таких файлов процедура чтения определена следующим образом:

```
procedure read (file f: file_type; value: out element_type;
                length: out natural);
```

Например, чтение из файла может осуществляться так, как показано в листинге 3.47.

Листинг 3.47

```
file vectors: bit_vector_file open read_mode is 'vector.dat';
variable next_vector: bit_vector (63 downto 0);
variable actual_len: natural;
read(vectors, next_vector, actual_len);
```

Такое описание позволяет читать из файла векторы длиной до 64 элементов. Если действительная длина очередного вектора в файле не больше 64, то читаемый вектор будет размещен в левой части `next_vector`, в противном случае в этой переменной будут размещены первые 64 элемента вектора, а остальные утеряны. После выполнения процедуры чтения переменная `actual_len` будет содержать фактическую длину читаемого вектора.

Запись в файл. Если файл открыт в режиме для записи (`write_mode`), то запись в него осуществляется с помощью процедуры

```
write(file f: file_type, value: in element_type);
```

Если непустой файл открывается в режиме `write_mode`, то он автоматически очищается от содержащейся в нем информации.

Если необходимо выполнить дозапись в файл, не теряя имеющююся в нем информацию, то его необходимо открыть в режиме `append_mode`. В этом случае процедура **`write`** будет добавлять данные в конец файла.

Соответствие между физическими и логическими файлами

С одним физическим файлом одновременно должно быть связано не более одного логического файла. Если один физический файл связан с несколькими логическими, то при моделировании могут возникнуть непредсказуемые ситуации.

Необходимо проявлять особенную внимательность, если данное типа файл описано в архитектурном теле объекта моделирования, который может неоднократно включаться в качестве компонента (экземпляров компонента) в некоторый другой объект моделирования. В этом случае для каждого из блоков, использующего одно и то же данное типа файл, будет порожден свой логический файл. Все эти логические файлы будут связаны с одним и тем же физическим.

Для исключения таких ситуаций возможно два подхода [3]:

- если все экземпляры такого объекта должны выполнять запись в один и тот же файл, то этот файл необходимо описать в пакете. В этом случае обращения от различных экземпляров объекта к этому файлу будут автоматически скоординированы средой проектирования, т. е. во всех экземплярах объекта будет происходить обращение к одному логическому файлу;
- если каждый экземпляр объекта с таким архитектурным телом должен работать с отдельным файлом, то имя физического файла не должно быть константой.

Видимость описаний и автоматическое открытие/закрытие файлов в модели

Файлы, при описании которых указывается режим их открытия (секция **open** в описании файла), автоматически открываются и закрываются по следующим правилам:

- если объект типа файл описывается в архитектурном теле объекта моделирования или в процессе, файл открывается при начале моделирования и закрывается в конце моделирования автоматически. Также автоматически открываются и закрываются файлы, описанные в пакетах;
- если файл описывается в подпрограмме, он автоматически открывается при вызове этой подпрограммы и закрывается по окончании ее работы.

В подпрограммах файлы могут использоваться для определения значений констант. Например, может быть организована функция, которая читает из файла значения, передаваемые затем массиву констант как его начальные значения [3].

Открытие и закрытие файлов без использования автоматических режимов

Если при описании логического файла не указывать режим его открытия, то он не будет автоматически открываться и закрываться во время моделирования. Для открытия файла используется процедура **file_open**. Например, обращение к процедуре может выглядеть так:

```
file_open(vectors_w,"load5.dat",write_mode);
```

В этом случае логический файл `vectors_w` связывается с физическим файлом `load5.dat`, открытие файла производится для записи, на что указывает режим `write_mode`.

Использование этой процедуры предоставляет большую свободу программисту. Можно определить, насколько успешно прошло открытие файла, просмотрев значение `status`:

- `status_ok` — успешное открытие;
- `status_error` — этот файловый объект уже открыт и связан с другим физическим объектом;
- `name_error` — при чтении: файла с таким именем не существует; при записи или добавлении: файл с таким именем не существует и не может быть создан;
- `mode_error` — попытка открыть файл для записи, когда он открыт только для чтения.

Например,

```
If file_open(vectors_w,"load5.dat",write_mode)=status_ok then ...
    Else ...
End if;
```

Использование такой конструкции позволяет проверить, успешно ли он открылся.

Для закрытия файла может быть использована процедура

```
file_close (file f: file_type);
```

Использование комбинации таких процедур позволяет связывать один логический файл с различными физическими файлами в различные моменты времени [3].

Файлы как параметры подпрограмм

Файлы могут использоваться в качестве параметров подпрограмм.

Описание формального параметра файлового типа имеет следующий синтаксис:

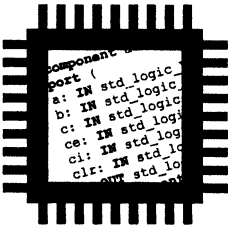
```
file identifier {,...} subtype indication;
```

Например:

```
type transform_file is file of real;
procedure read_transform (file f: transform_file);
```

Над объектом файлового типа внутри процедуры могут производиться все вышеописанные действия. Когда производится вызов процедуры, в нее, в качестве параметра, передается фактический объект файлового типа. Автоматическое открытие в начале работы процедуры и автоматическое закрытие в конце работы процедуры для него не выполняется.

В *стандарте VHDL 87*, в описании формального параметра файлового типа перед файлом не ставится ключевое слово **file**, он рассматривается как переменная файлового типа.



Глава 4

Проектирование на VHDL

В этой главе будут рассмотрены особенности использования конструкций языка VHDL для моделирования поведения и синтеза.

Использование конструкций VHDL для моделирования

Особенности использования временных задержек в операторе присваивания значения сигналу при поведенческом моделировании

Рассмотрим некоторый объект, имеющий входной и выходной сигнал типа **integer**. Функцией этого объекта является задержка выходного сигнала по отношению к входному на 5 нс. Для моделирования объекта воспользуемся механизмом задержек в операторе присваивания значения сигналу. Модель может выглядеть следующим образом (листинг 4.1).

Листинг 4.1

```
entity del is
  port (inn: in integer;
        ouu: out integer);
end entity del;

architecture behavior of del is
begin
  process (inn)
    begin
      ouu< = inn after 5 ns;
    end process;
end architecture behavior;
```

```
begin
end architecture behavior;
```

В данном примере использование конструкции **process** не обязательно. Можно использовать параллельный оператор присваивания.

Пример функционирования модели приведен на рис. 4.1.

Описанная в операторе присваивания задержка является (по умолчанию) задержкой *инерционной*. На состоянии выходного сигнала не сказываются входные импульсы, длительность которых (иными словами — длительность сохранения неизменным значения входного сигнала) меньше задержки, заданной в операторе. Они отфильтровываются схемой. Здесь секция **after** 5 ns задает задержку в 5 нс., так что значения '4' и '6', которые держатся на входе inn меньше 5 нс., будут отфильтрованы. Выходной сигнал принимает значение '0' через 5 нс. после начала моделирования. Значение '7' на выходе появляется через 5 нс. после его появления на входе, однако значения '4' и '6' с входа на выход не поступают.

В терминах транзакций ситуацию можно описать следующим образом. Когда в момент модельного времени 5 происходит событие изменения состояния inn, и входной сигнал принимает значение '4', то на момент времени 10 планируется транзакция присваивания выходному сигналу значения '4'. Однако в момент времени 8 входной сигнал принимает значение '6', в результате чего планируется новая транзакция присваивания выходному сигналу значения '6' в момент времени 13. Между моментом модельного времени 5 и моментом времени 8 прошло меньше 5 нс. Появление новой транзакции приводит к уничтожению еще находящейся в буфере, связанной с этим сигналом транзакции, запланированной на время 10. Когда в момент времени 12 входной сигнал снова меняет свое значение, транзакцию, запланированную на время 13, постигает та же участь.

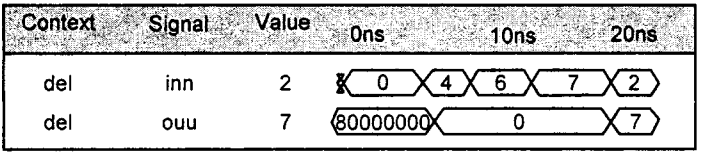


Рис. 4.1. Временная диаграмма, соответствующая функционированию модели устройства с листинга 4.1

При моделировании большинства устройств этот механизм отражает реальную ситуацию и оказывается довольно удобным. Однако иногда необходимо, чтобы на выходе наблюдалась реакция на входной сигнал независимо от его длительности. В таких случаях можно воспользоваться механизмом **transport**.

Тогда оператор присваивания будет выглядеть следующим образом (листинг 4.2):

Листинг 4.2

```
process (inn)
begin
  ouu < = transport inn after 5 ns;
end process;
```

С помощью ключевого слова **transport** мы включили механизм транспортных, а не инерционных задержек. Временная диаграмма работы такого устройства представлена на рис. 4.2, который демонстрирует ситуацию, когда, несмотря на то, что значение '4' входного сигнала inn сохранялось в течение времени 3 нс., а значение '6' — в течение времени 4 нс., эти значения сигналов появились на выходе.

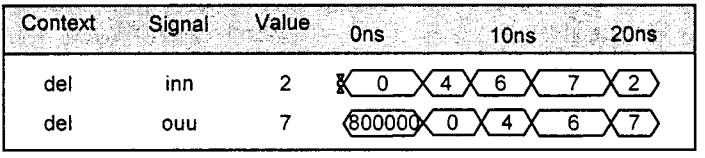


Рис. 4.2. Временная диаграмма, соответствующая функционированию модели с листинга 4.2

Если все же ограничение на минимальную длительность нахождения входного сигнала в некотором состоянии есть, но оно не равно задержке влияния на сигнал выходной (например, равно 4 нс.), то мы возвращаемся к инерционным задержкам, но задаем инерционность в явном виде, не приравнивая ее задержке (листинг 4.3).

Листинг 4.3

```
process (inn)
begin
  ouu < = reject 4 inertial inn after 5 ns;
end process;
```

В этом случае значение '4' и '6' на входе будет отфильтровано (расстояние между событиями перехода сигнала в это состояние и событием перехода из него в состояние '6' равно 3 нс., аналогично — для '6' составляет 4 нс., что не превосходит значения времени, указанного в секции reject). Значение же '7' пройдет на выход. Иллюстрация данного примера приведена на

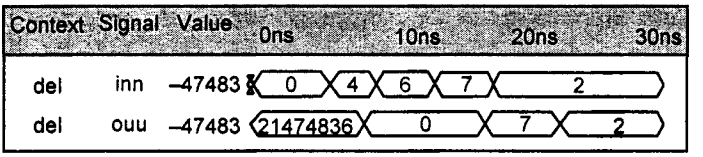


Рис. 4.3. Временная диаграмма, соответствующая функционированию модели с листинга 4.3

Использование процессов, сигналов и переменных в поведенческом моделировании

Процессы и присваивание значений сигналам

Часто встает вопрос, сколько процессов необходимо использовать, чтобы промоделировать одно устройство. Если устройство выполняет несколько функций (действий), то каждое из них реализуется в виде отдельного параллельного оператора или процесса для удобства понимания работы модели. Если устройство состоит из нескольких узлов, то каждый из них можно промоделировать в виде отдельного процесса. Однако это не обязательно. Во многих случаях в одном процессе можно выполнять несколько функций.

Рассмотрим некоторые проблемы, возникающие при совмещении нескольких действий по сути моделируемого устройства — действий параллельных, в одном процессе.

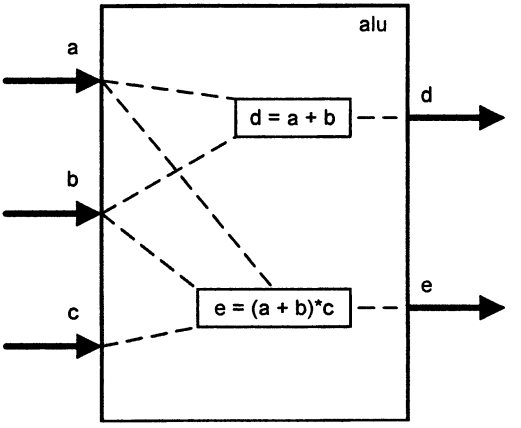


Рис. 4.4. Моделируемое устройство alu

Промоделируем устройство, которое имеет три входа (`a, b, c`) и два выхода (`d, e`). `d = a + b, e = (a + b)×c` (рис. 4.4). Как обычно, в реальном

устройстве все входы и выходы устройства функционируют параллельно в физическом времени. Значения задержек нас в данном случае не волнует, и в описании модели устройства мы их опускаем (модель будет работать по дельта-задержкам).

Попробуем промоделировать работу этого устройства с использованием одного процесса. Пусть текст модели выглядит так, как представлено в листинге 4.4.

Листинг 4.4

```
entity alu is
  port (a,b,c: in integer;
        d: inout integer;
        e: out integer);
end entity alu;

architecture behaviour of alu is
begin
pr_d_e: process(a,b,c)
  begin
    d < = a + b;
    e < = d*c;
  end process pr_d_e;
end architecture behaviour;
```

На первый взгляд модель должна замечательно работать. Однако можно наблюдать результат ее работы, который представлен на рис. 4.5.

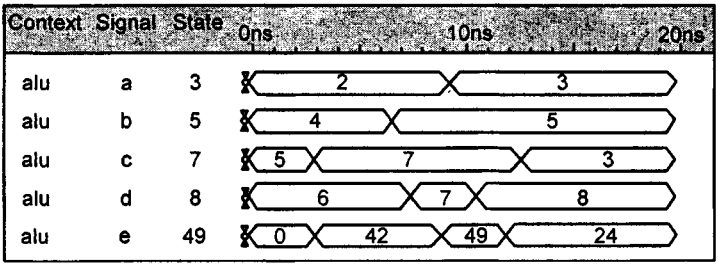


Рис. 4.5. Временная диаграмма работы устройства, описанного в листинге 4.4

На выходе d наблюдается желаемый результат, а выход e ведет себя странно: ожидаемый после момента времени 7 результат '49' появляется только в момент времени 10, когда на этом выходе уже должно быть значение '56', которому там так и не суждено появиться. Такое поведение выхода e связа-

но с тем, что его значение определяется на базе сигнала d, декларированного в **entity** alu как сигнал выходной (**out**), изменение значения которого осуществляется в теле того же процесса.

Процесс срабатывает каждый раз при изменении состояния хотя бы одного сигнала, входящего в его список чувствительности. При каждом срабатывании, для формирования выходных сигналов берутся те значения входных сигналов, которые они имеют в момент модельного времени, соответствующий текущему срабатыванию процесса.

Обратите внимание! Поскольку сигнал d, являющийся для объекта выходным, используется в левой части оператора присваивания, его тип должен быть определен как inout. Если это по каким-либо причинам нежелательно, необходимо использовать сигнал, описанный внутри архитектурного тела. Это может выглядеть следующим образом (листинг 4.5).

Листинг 4.5

```
architecture behaviour_a of alu is
signal dd:integer;
begin
pr_d_e: process(a,b,c)
  begin
    dd < = a + b;
    e < = dd*c;
    d< = dd;
  end process pr_d_e;
end architecture behaviour_a;
```

Такое описание будет функционировать, так же как и описание в листинге 4.4. Забегая вперед, можно сказать, что для того, чтобы схема работала желаемым образом, в данном случае нужно использовать не сигнал, а переменную.

Таким образом, когда процесс pr_d_e выполняется очередной раз, для определения нового значения сигнала e используется то значение сигнала d, которое сигнал d имел при запуске процесса pr_d_e на выполнение, т. е. *прежнее*, а не новое значение сигнала d, сформированное оператором d< = a + b (рис. 4.6). Несмотря на то, что оператор присваивания значения сигналу e написан текстуально, после операции присваивания нового значения сигналу d, и будет в теле процесса выполняться после него, в системе моделирования значение сигнала d сменится только после завершения акта выполнения процесса (единицы параллельных действий в модели на VHDL).

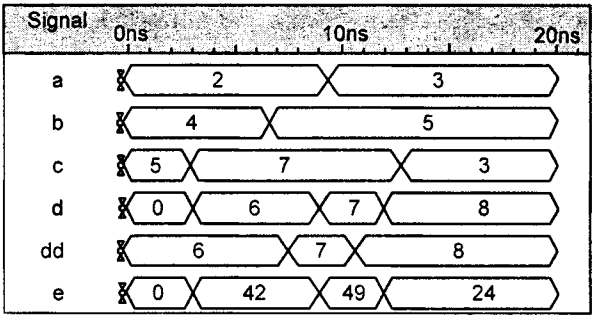


Рис. 4.6. Срабатывание процесса для момента модельного времени $T_{мод}$ = 7 нс

Механизм дельта-задержек здесь не включается, т. к. операторы присваивания значения сигналу, расположенные в теле процесса, рассматриваются как операторы *последовательные*, операторы в теле процесса `pr_d_e`. Единицей же планирования срабатывания в модельном времени выступает процесс в целом.

Возникшую проблему можно решить несколькими способами. Во-первых, в теле процесса можно переписать оператор присваивания значения для `e` следующим образом: `e< = (a + b)×c;` В данной модели это не вызовет проблем, но это возможно не всегда.

Более корректным решением будет разнесение операторов присваивания в *разные процессы*, как представлено в листинге 4.6.

Листинг 4.6

```
architecture behavior1 of alu is
begin
  pr_d: process(a,b,c)
  begin
    d< = a + b;
  end process pr_d;

  pr_e: process (c,d)
    begin
      e< = d*c;
    end process pr_e;
end architecture behavior1;
```

Кроме того, можно оставить оба оператора присваивания значений сигналам внутри тела одного процесса, но использовать промежуточную локальную переменную (представлено в листинге 4.7).

Листинг 4.7

```
architecture behaviour1a of alu is
begin
  pr_d_e1: process(a,b,c)
  variable var: integer;
  begin
    var: = a + b;
    d< = var;
    e< = var*c;
  end process pr_d_e1;
end architecture behaviour1a;
```

В отличие от сигнала, переменная немедленно меняет свое значение после выполнения операции присваивания, и это новое значение становится сразу же видимым для последующих операторов в теле процесса.

В обоих случаях модель будет функционировать правильно. Результат работы представлен на рис. 4.7.

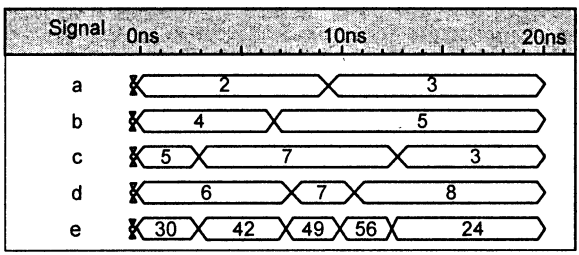


Рис. 4.7. Временная диаграмма работы устройств, описанных в листингах 4.6 и 4.7

Примечание

Переменная `var` — *локальная* переменная процесса pr_d_e1, следовательно, она не является разделяемой, т. е. не может быть использована никаким другим процессом.

Конечно, для этого простого примера мы можем выйти из положения, вообще не используя процесса в архитектурном теле, а поставив два простых параллельных оператора присваивания значений сигналам (листинг 4.8).

Листинг 4.8

```
architecture behaviour2 of alu is
begin
  d < = a + b;
  e < = d*c;
end architecture behaviour2;
```

Здесь, с учетом механизма дельта-задержек, изменение сигнала d, выполненное первым оператором, приведет к повторному срабатыванию второго оператора, задающего значение e в тот же момент модельного времени, но уже с новым значением d.

Отметим, что варианты, представленные листингами 4.6 и 4.8, являются и более гибкими. В случае изменения только одного сигнала c будет запущен только оператор e< = d×c; а оператор d< = a + b; срабатывать не будет. Если такие же операторы мы сделали бы последовательными операторами в теле процесса (листинги 4.4 и 4.7), то мы заставили бы выполняться *оба* оператора при каждом изменении любого из входных сигналов. Функциональной ошибки в этом нет, но на времени моделирования это, конечно, скажется не лучшим образом.

В заключение отметим, что при заданных в нашем примере простых преобразованиях, целесообразнее предпочесть более прямой вариант реализации (листинг 4.9).

Листинг 4.9

```
architecture behaviour3 of alu is
begin
  d < = a + b;
  e < = (a + b)*c;
end architecture behaviour3;
```

Здесь дважды вычисляется (a + b), но зато не включается механизм дельта-задержек и не происходит повторного выполнения оператора e< = d×c; в ответ на одно изменение входных сигналов, как это будет происходить в примере, приведенном в листинге 4.8.

Результаты работы программ (листинги 4.8 и 4.9) также соответствуют представленным на рис. 4.7 временным диаграммам.

Присваивание значений сигналу разными процессами, параллельными операторами

Рассмотрим, что будет происходить, если одному и тому же сигналу будут присваиваться значения в различных процессах. Пусть в одном процессе

выходному сигналу присваивается значение суммы входных сигналов с задержкой на 1 нс., а в другом — значение разности с задержкой на 2 нс. Программа представлена в листинге 4.10.

Листинг 4.10

```
entity alu2 is
port (a,b:in integer; d:out integer);
end entity alu2;

architecture behavior of alu2 is
begin
plus: process(a,b)
  begin
    d< = a + b after 1 ns;
  end process plus;

minus: process(a,b)
  begin
    d< = a-b after 2 ns;
  end process minus;
end architecture behavior;
```

Результат работы модели представлен на рис. 4.8.

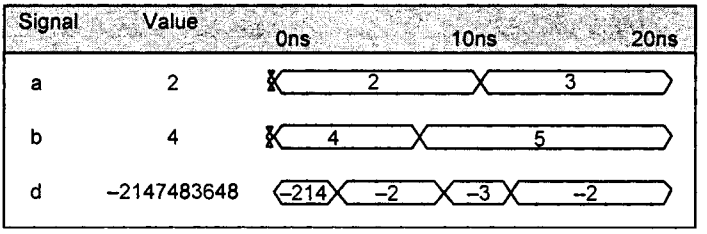


Рис. 4.8. Временная диаграмма работы устройства, описанного в листинге 4.10

В процессе `plus` устанавливается задержка присваивания значения сигналу в 1 нс., в то время как в процессе `minus`, запускаемом при событиях изменения тех же сигналов, — задержка в 2 нс. Сигнал имеет два источника, поскольку присваивание `d` выполняется в обоих процессах. Однако на выходе мы видим только результат работы второго процесса, т. к. в нем событие планируется на более позднее время (т. е. второй процесс стирает результат, полученный в первом процессе).

Использование списка чувствительности процесса и атрибутов сигналов.

Промоделируем D-триггер, срабатывающий по переднему фронту тактового импульса. Пусть триггер имеет асинхронный сигнал сброса — cl.

Рассмотрим альтернативные варианты поведенческого описания такого триггера. Первый вариант описания представлен в листинге 4.11.

Листинг 4.11

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity dtr is
port (d: in std_ulogic;
      clk: in std_ulogic;
      cl: in std_ulogic;
      q: inout std_ulogic;
      nq: out std_ulogic);
end entity dtr;

architecture behavior of dtr is
begin
reset:  process(cl)
 begin
   if cl = '1' then q< = '0'; nq< = '1'; end if;
 end process reset;

output1:  process (clk,d)
 begin
   if clk'event and clk = '1' then q< = d; else q< = q; end if;
 end process output1;

output2:  process(q)
 begin
   nq< = not q;
 end process output2;

end architecture behavior;
```

Это описание содержит три процесса.

Первый процесс reset предназначен для сброса триггера. В списке чувствительности процесса присутствует сигнал cl, — процесс reset активизируется при изменении сигнала cl. Тело процесса reset составляет оператор **if** с единственной секцией **then**, которая выполняется, если cl = '1'. Таким образом, сброс триггера происходит по восходящему фронту сигнала cl.

Второй процесс — процесс output1, используется для определения значения прямого выхода триггера. В списке чувствительности этого процесса присутствуют сигналы clk и d, — их изменение приводит к активизации процесса.

Тело процесса output1 составляет единственный оператор **if**. В условном выражении этого оператора присутствует атрибут сигнала clk — атрибут event. Конструкция clk'event and clk = '1' принимает значение **true** при приходе восходящего фронта тактового импульса. В этот момент времени в триггер записывается значение его входа.

Процесс output1 также активизируется при изменении сигнала d. Однако, если это изменение происходит в момент времени, не соответствующий переднему фронту тактового импульса, то выполняется ветвь **else** оператора if – q< = q (триггер сохраняет свое прежнее значение). Обратите внимание, в этой конструкции происходит чтение выходного порта q. Для того чтобы эта конструкция могла быть выполнена, сигнал q должен использоваться в режиме **inout**. Вообще, в данной конструкции можно и не использовать ветвь **else**, работать она будет точно также. Однако использование подобных конструкций может облегчать понимание логики работы модели.

Поскольку изменение только сигнала d не влияет на значение выходного сигнала, исключение его из списка чувствительности процесса output1 не повлияет на логику работы прооцесса. Включение таких сигналов в список чувствительности используется только для облегчения понимания логики работы процесса, поскольку указывается, что на их базе определяются значения выходных сигналов.

Если сигнал d исключить из списка чувствительности процесса output1, то условие в операторе **if** может быть упрощено, что представлено листингом 4.12.

Листинг 4.12

```
output1: process (clk)
begin
  if clk = '1' then q< = d; end if;
end process output1;
```

Подобно тому, как это было рассмотрено для процесса output1 (листинг 4.11), триггеру будет присваиваться новое значение только по восходящему фронту тактового импульса. Здесь восходящий фронт выявляется тем, что, с одной стороны — если сработал процесс output1, то было событие изменения значения сигнала (это только сигнал clk) из списка чувствительности (листинг 4.12); с другой стороны, — if clk = '1' дает условие, что сигнал clk равен '1'. Таким образом, clk изменился и стал равен '1', что и является передним фронтом тактового импульса.

Третий процесс output2 предназначен для определения значения инверсного выхода триггера. Этот процесс активизируется при изменении прямого выхода триггера. Конструкция nq< = not q; также может быть выполнена только в том случае, если порт q используется в режиме **inout**.

Действия, выполняемые в трех процессах, можно совместить в один (листинг 4.13).

Листинг 4.13

```
architecture behavior2 of dtr is
alloutputs: process(cl, clk)
 begin
   if cl = '1' and cl'event then q< = '0'; nq< = '1'; end if;
   if clk = '1' and clk'event then q< = d; nq< = not d end if;
 end process alloutputs;
end architecture behavior2;
```

При таком описании сигнал q не используется в правой части оператора присваивания, поэтому, в отличие от предыдущего варианта, этот порт d в декларации **entity** dtr мог бы иметь режим **out**.

Вместо списка чувствительности может быть использован оператор **wait**, что продемонстрировано листингом 4.14.

Листинг 4.14

```
architecture behavior3 of dtr is
begin
reset: process(cl)
 begin if cl = '1' then q< = '0'; nq< = '1'; end if;
 end process reset;

outputs: process
 begin
   q< = d;
```

```
    nq< = not d;
    wait until clk'event and clk = '1' ;
  end process outputs;

end architecture behavior3;
```

Список чувствительности в описании процесса `outputs` не указан. Значит, по умолчанию, процесс срабатывает при изменении любого из его входных сигналов.

Конструкция **`wait until`** приостанавливает выполнение процесса до выполнения условия, задающего передний фронт импульса. К этому моменту процессом `outputs` уже выполнены последовательные операторы присваивания значений сигналам `q< = d;` и `nq< = not d;`. Однако, как мы пояснили в предыдущем разделе, новые значению фактически присваиваются сигналу только в момент завершения процесса, а оно отложено до момента прихода переднего (восходящего) фронта импульса сигнала `clk`. Если событие уже произошло в момент срабатывания процесса, то условие в **`wait until`** истинно, и процесс завершится без приостановки.

Временная диаграмма работы триггера (для всех вариантов) представлена на рис. 4.9.

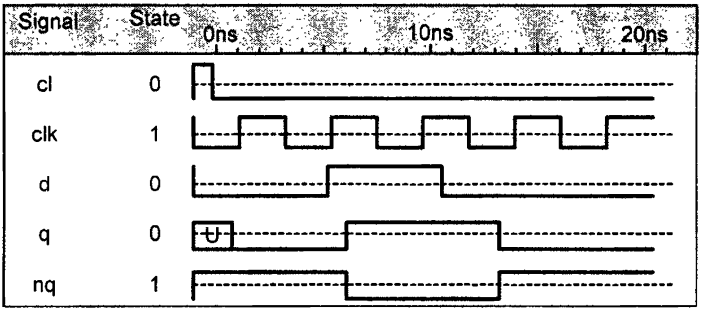


Рис. 4.9. Временная диаграмма работы триггера, варианты описания которого, приведены в листингах 4.11 — 4.14

Использование пакетов

При моделировании устройств на высоких уровнях абстракции нередко возникает необходимость в создании новых типов. Например, они могут использоваться для описания входных и выходных сигналов моделируемых объектов. Ясно, что такое описание необходимо выполнять до декларативной части **`entity`** объекта моделирования. Однако в структуре модели места для него не предусмотрено. Такие описания необходимо включать в пакеты.

Кроме того, в пакетах удобно размещать процедуры и функции, часто используемые различными объектами моделирования.

Промоделируем, например, элемент памяти конечного автомата, который может находиться в одном из следующих состояний: `idle`, `work`, `hard_work`, `error`.

Опишем новый тип `auto_state` и поместим это описание в пакете `auto_pack`. Пусть файл, в котором находится этот пакет, называется pack1.vhd. Текст приведен в листинге 4.15.

Листинг 4.15

```
package auto_pack is
  type auto_state is (idle, work, hard_work, error);
end auto_pack;
```

Текст модели, использующей введенный тип, может иметь следующий вид (листинг 4.6).

Листинг 4.16

```
use pack1.auto_pack.all;

entity el_mem is
   port (instate: in auto_state;
         outstate: inout auto_state;
         clk: in bit);
end entity el_mem;

architecture behavior of el_mem is
begin
  process(clk)
  begin
    if clk'event and clk = '1' then outstate< = instate;
    else outstate< = outstate; end if;
  end process;
end architecture behavior;
```

Использование секции **else** в данном примере не обязательно.

В начале текста модели содержится указание на используемый пакет. Оно состоит из имени файла, в котором расположен пакет, имени пакета и ключевого слова **all**, которое указывает на то, что используется все содержимое

пакета. Подключив пакет, мы получаем возможность сразу, в декларации **entity**, использовать нестандартный тип `auto_state`.

Результаты работы модели представлены на рис. 4.10.

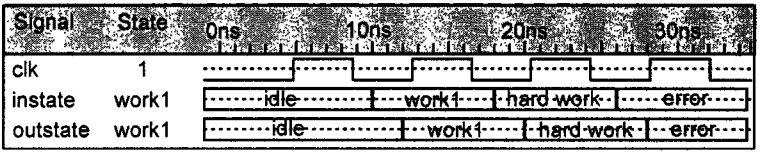


Рис. 4.10. Временная диаграмма работы устройства, описанного в листинге 4.16

Использование в моделях механизма wait

Ранее в одном из примеров мы использовали механизм **wait** (листинг 4.14), как альтернативу списку чувствительности (листинг 4.11). Однако список чувствительности и механизм **wait** в модели не всегда взаимозаменяемы.

Например, может возникнуть необходимость в написании процесса, который при наступлении определенного события прекратит выполнение до конца моделирования (например, процесс инициализации входных значений сигналов). Или же нужно создать процесс, который бы генерировал значения для входного сигнала. Например, это может быть полезно, когда предполагается использование модели широким кругом пользователей, не все из которых используют среду OrCAD Express для моделирования и, соответственно, не могут пользоваться файлами входных значений, сгенерированными в этой среде.

В таких случаях используется механизм **wait**. Этот механизм несовместим с механизмом списка чувствительности.

Рассмотрим модель простейшего триггера, где триггеру в начальный момент времени присваивается начальное значение 0 (листинг 4.17).

Листинг 4.17

```
entity tr is
port (d: in integer;
      clk:in bit;
      q:inout integer);
end entity tr;

architecture behavior of tr is
begin
  process(clk)
```

```
  begin
    if clk = '1' and clk'event then q< = d;  else q< = q; end if;
  end process;
initialise:  process
  begin
    q< = 0;
    wait for 10000;
  end process initialise;

end architecture behavior;
```

Для присвоения начального значения используется процесс initialise. Он выполняется 1 раз в 100 000 нс. и устанавливает триггер в начальное состояние.

Рассмотрим модель простейшего триггера, где производится автоматическая генерация тактового сигнала (листинг 4.18).

Листинг 4.18

```
entity tr is
port (d:in integer;
        q:inout integer);
end entity tr;

architecture tr1_behavior of tr is
 signal clk: bit;
 constant t: time: = 2 ns;
begin

tr_proc:  process(clk)
  begin
    if clk = '1' and clk'event then q< = d;  else q< = q; end if;
  end process tr_proc;

inner_clk:  process
  begin
    clk< = '1' after t, '0' after 2*t;
    wait for 2*t + 1 ns;
  end process inner_clk;

end architecture tr1_behavior;
```

Первый процесс модели (tr_proc) описывает работу триггера, а второй (inner_clk) служит для задания тактового сигнала. Обратите внимание на соотношение времен. Время, указанное в **wait**, определяет длительность такта, оно должно быть не менее суммарного времени в операторе присваивания. При $t = 2$ нс. диаграмма работы моделируемого устройства представлена на рис. 4.11.

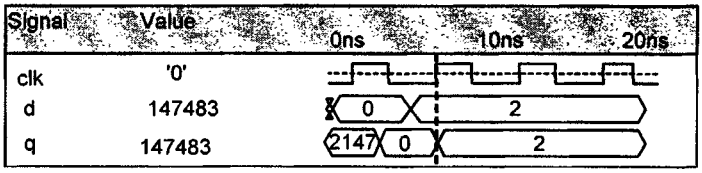


Рис. 4.11. Временная диаграмма работы триггера, описанного в листинге 4.18

Когда процесс inner_clk выполняется первый раз, в момент модельного времени $T_{мод} = 0$ происходит следующее: тактовый сигнал clk из начального значения (для типа **bit** это 0, если не задано иное) через время t переходит в состояние '1', а через время $2 \times t$ от начала выполнения процесса переходит в '0'. В результате формируется положительный импульс длительностью t, начало которого сдвинуто на t нс. от начала моделирования. После этого процесс приостанавливает свое выполнение на время, указанное в **wait** — на $2 \times t + 1$ нс. от момента начала текущего выполнения процесса в модельном времени. Сработал процесс в момент $T_{мод} = 0$, так что задержан он будет до момента $T_{мод} = 0 + (2 \times t + 1)$ нс. В примере (рис. 4.11) при $t = 2$ нс. это будет $T_{мод} = 0 + (2 \times 2 + 1) = 5$, т. е. до 5-й нс. модельного времени. По истечении заданной в **wait** задержки процесс вновь сработает (уже в момент модельного времени $T_{мод} = 5$ нс.). Теперь от этого момента будут отсчитываться задержки в операторе присваивания значения сигналу clk: в момент модельного времени $T_{мод} = 5 + 2 = 7$ будет сформирован передний фронт очередного импульса сигнала, а в момент $T_{мод} = 5 + (2 \times 2) = 9$ — задний фронт, что даст импульс, длительностью 2 нс. Затем процесс будет задержан в модельном времени оператором **wait** (на $2 \times t + 1$ нс) от момента текущего выполнения до момента $T_{мод} = 5 + (2 \times 2 + 1) = 10$, помеченного штрих-пунктирной линией на рис. 4.11. Затем процесс вновь начнет выполняться в момент модельного времени $T_{мод} = 10$, и т. д.

Для дополнительной иллюстрации работы механизма возьмем модификацию программного текста (листинг 4.7), вставив в него оператор ожидания **wait**. Новый вариант представлен листингом 4.19.

Листинг 4.19

```
architecture behav_wait of alu is
begin
pr_d_e2: process
variable var: integer;
begin
     wait on a,b,c;
     var: = a + b;
    d< = var;
    wait for 4 ns;
    e< = var*c;
end process pr_d_e2;
end architecture behav_wait;
```

Этот вариант модели поведения устройства будет формировать выходной сигнал e через 4 нс. после формирования выходного сигнала d.

Первый из операторов **wait** в теле процесса (оператор **wait on** a,b,c;) заменяет работу списка чувствительности (как мы помним, в одном процессе нельзя использовать и список чувствительности, и оператор **wait**). Он задержит работу процесса до наступления события изменения хотя бы одного сигнала из списка a,b,c. Второй оператор **wait** задаст приостановку выполнения действий процесса в модельном времени на 4 нс.

Оператор **wait** задает приостановку процесса, а также изменяет привязку порождаемых им событий к моментам модельного времени. Рис. 4.12 иллюстрирует соотнесение модельного времени и времени моделирования при выполнении процесса pr_d_e2 (листинг 4.19).

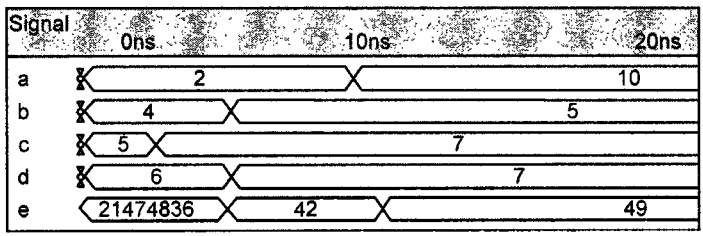


Рис. 4.12. Время моделирования и модельное время при выполнении процесса, использующего оператор **wait** ($T_{мод}1$ — момент модельного времени при запуске процесса; $T_{мод}2$ — момент события по сигналам a, b, c)

Использование типа запись

Тип запись может использоваться для группировки сигналов, имеющих в модели общее назначение. Например, в запись можно сгруппировать множество сигналов управления, что значительно облегчает работу с моделью при большом количестве сигналов. Записи могут использоваться как для внутренних сигналов архитектурного тела, так и для входных и выходных сигналов.

Использование типа запись для внутренних сигналов

Рассмотрим устройство, которое имеет один вход данных `data1`, один вход кода операции `codef` и один выход данных `datao`. Внутри устройства имеется 4 флага, в которые заносится информация из различных разрядов `data1`, в зависимости от значения на входе `codef`. На выход `datao` поступают значения флагов, в зависимости от `codef(1)`, в прямом или обратном порядке

`(datao(0) = fl1...datao(3) = fl4` или `datao(0) = fl4...datao(3) = fl1)`.

Для хранения значения каждого флага можно использовать отдельный внутренний сигнал, а можно объединить их в одну запись. Такой подход особенно удобен, когда внутренних сигналов достаточно много и их можно разделить на несколько групп по какому-либо признаку, например, по назначению или по условиям изменения значений.

Построим модель подобного устройства. Рассмотрим сначала часть объекта, в которой выполняется заполнение флагов значениями (листинг 4.20).

Листинг 4.20

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity my_ent is
port (data1: in std_ulogic_vector(3 downto 0);
      codef: in std_ulogic_vector(1 downto 0);
      datao: out std_ulogic_vector(3 downto 0)
     );
end entity my_ent;

architecture rlt of my_ent is
```

```
type my_flags is record
  fl1: std_ulogic;
  fl2: std_ulogic;
  fl3: std_ulogic;
  fl4: std_ulogic;
 end record;
signal flags1: my_flags;

begin
set_flags: process (data1, codef)
 begin
    case  codef is
        when "00" = > flags1.fl1< = data1(0); flags1.fl2< = data1(1);
                    flags1.fl3< = 'Z';
        when "01" = > flags1.fl3< = data1(2);flags1.fl1< = 'Z';
                    flags1.fl2< = 'Z'; flags1.fl4< = 'Z';
        when "10" = > flags1.fl3< = data1(0); flags1.fl4< = data1(3);
                    flags1.fl1< = 'Z'; flags1.fl2< = 'Z';
        when "11" = > flags1.fl2< = data1(2); flags1.fl4< = data1(1);
                    flags1.fl1< = 'Z'; flags1.fl3< = 'Z';
end case;
 end process set_flags;

out_sig:  process(flags1)
 begin
   if codef(1) = '0' then
         datao(0)< = flags1.fl1;
         datao(1)< = flags1.fl2;
         datao(2)< = flags1.fl3;
         datao(3)< = flags1.fl4;
     else
         datao(0)< = flags1.fl4;
         datao(1)< = flags1.fl3;
         datao(2)< = flags1.fl2;
         datao(3)< = flags1.fl1;
   send if;
 end process out_sig;

end architecture rlt;
```

Здесь сигнал с типом значения **record** декларируется внутри архитектурн го тела rlt.

Заполнение флагов значениями выполняется в первом процессе, set_flag Во втором процессе, out_sig, в зависимости от значения codef(1) форми руется выходной сигнал. Временная диаграмма работы устройства предста лена на рис. 4.13.

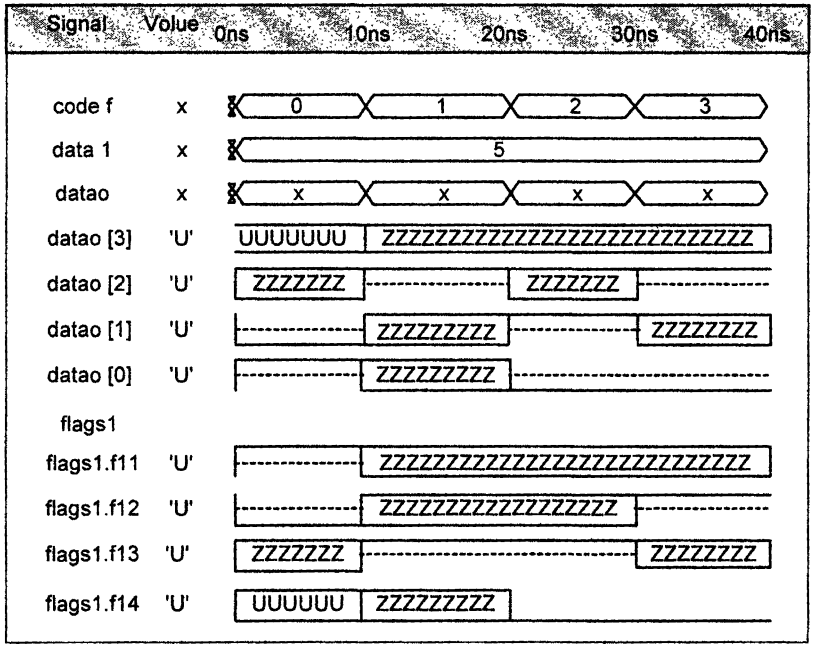


Рис. 4.13. Временная диаграмма работы устройства, описанного в листинге 4.20

В качестве полей типа запись, для описания внутренних сигналов можно использовать не только скалярные типы, но и массивы. Пусть, например первый из флагов в модели будет вектором (листинг 4.21).

Листинг 4.21

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity my_ent is
port (data1: in std_ulogic_vector(3 downto 0);
      codef: in std_ulogic_vector(1 downto 0);
```

```
     datao: out std_ulogic_vector(3 downto 0)
    );
end entity my_ent;

architecture rlt of my_ent is
type my_flags is record
   fl1: std_ulogic_vector(1 downto 0);
   fl2: std_ulogic;
   fl3: std_ulogic;
   fl4: std_ulogic;
  end record;
signal flags1: my_flags;

begin
  process (data1, codef)
  begin
case  codef is
    when "00" = > flags1.fl1(0)< = data1(0); flags1.fl1(1)< = data1(3);
flags1.fl2< = data1(1); flags1.fl3< = 'Z';
    when "01" = > flags1.fl3< = data1(2);flags1.fl1< = "ZZ";
flags1.fl2< = 'Z'; flags1.fl4< = 'Z';
    when "10" = > flags1.fl3< = data1(0); flags1.fl4< = data1(3);
flags1.fl1< = "ZZ"; flags1.fl2< = 'Z';
    when "11" = > flags1.fl2< = data1(2); flags1.fl4< = data1(1);
flags1.fl1< = "ZZ"; flags1.fl3< = 'Z';
end case;
  end process;

  process(flags1)
  begin
  if codef(1) = '0' then
        datao(0)< = flags1.fl1(0);
        datao(1)< = flags1.fl2;
        datao(2)< = flags1.fl3;
        datao(3)< = flags1.fl4;
  else
        datao(0)< = flags1.fl4;
        datao(1)< = flags1.fl3;
        datao(2)< = flags1.fl2;
```

```
        datao(3)< = flags1.fl1(0);
end if;
end process;

end architecture rlt;
```

Временная диаграмма работы устройства приведена на рис. 4.14.

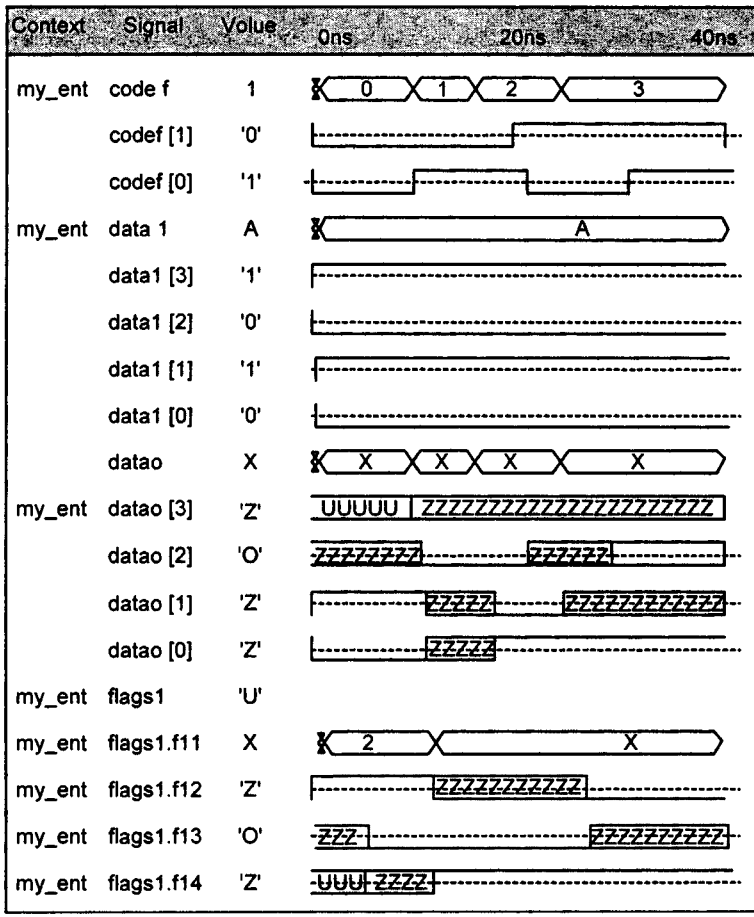


Рис. 4.14. Временная диаграмма работы устройства, описанного в листинге 4.21

Обратите внимание на следующую особенность OrCAD Express 9.1 (version 9.10.89, возможно, и другие версии). В списке чувствительности процесса может указываться только *весь* сигнал-запись. Указание отдельных полей приводит к неработоспособности модели: при выполнении моделирования возникает сообщение о зацикливании модели, после которого, для продолжения нормальной работы, необходимо перезагрузить OrCAD Express.

Использование типа запись для входных и выходных сигналов

Рассмотрим работу с типом запись на примере простого арифметико-логического устройства. Пусть это устройство будет иметь два входа данных (первый и второй операнд), один вход кода операции и один выход данных. Если код операции — 0, на выход поступает значение первого операнда; если 1 — на выход поступает инверсное значение первого операнда; если 2 — значение второго операнда; если 3 — инверсное значение второго операнда.

Описание устройства приведено в листинге 4.23, описание типа записи — в листинге 4.22. Для описания всей группы входных сигналов можно использовать запись. Этот подход, для облегчения читаемости, удобно применять в схемах с большим количеством сигналов.

Листинг 4.22

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

package my_types is
  constant opsize:integer: = 8;
  type in_rec is record
          data1: std_ulogic_vector((opsize-1) downto 0) ;
          data2: std_ulogic_vector((opsize-1) downto 0) ;
          opcode: std_ulogic_vector((opsize-1) downto 0) ;
  end record;
end package my_types;
```

Листинг 4.23

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use my_types.my_types.all;
entity alu is
 port(din: in  in_rec;
      dout: out std_ulogic_vector((opsize-1) downto 0));
 end entity alu;
```

```
architecture rtl of alu is
begin
  process
  begin
    wait until din'event ;
    if din.opcode = x"00" then dout< = din.data1;
     else
      if din.opcode = x"01" then dout< = not din.data1;
       else
         if din.opcode = x"02" then dout< = din.data2;
            else dout< = not din.data2;
         end if;
       end if;
      end if;
  end process;
end architecture rtl;
```

В данном случае использованы конструкции **if**, хотя грамотнее было бы использовать конструкцию case. Это связано с тем, что некоторые реализации OrCAD в этом случае некорректно работают с записью.

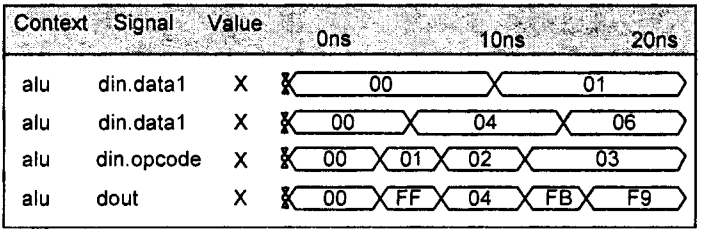


Рис. 4.15. Временные диаграммы для листинга 4.23

В данном случае в качестве полей записи используются не скалярные типы, а векторы. Можно было, вместо использования оператора **wait**, в списке чувствительности указать din — запись в целом, что не влияет на работоспособность модели. Однако, как отмечалось ранее, указание отдельных *полей* записи в списке чувствительности может привести к неработоспособности модели.

Внимание

В OrCAD Express 9.1 значения полей записи в случае, когда в процессе моделирования они задаются с помощью вкладки **Stimulis** и сравниваются с явно задаваемыми константами, должны выглядеть точно так же, как и те, которые были заданы на вкладке.

В приведенном примере значения кода операции в тексте программы на VHDL задавались в шестнадцатеричном формате. Если отредактировать модель так, как это представлено в листинге 4.24, можно получить результат моделирования, представленный на рис. 4.16.

Листинг 4.24

```
if din.opcode = "00" then dout< = din.data1;
 else
 if din.opcode = "01" then dout< = not din.data1;
  else
    if din.opcode = "10" then dout< = din.data2;
      else dout< = not din.data2;
   end if;
 end if;
end if;
```

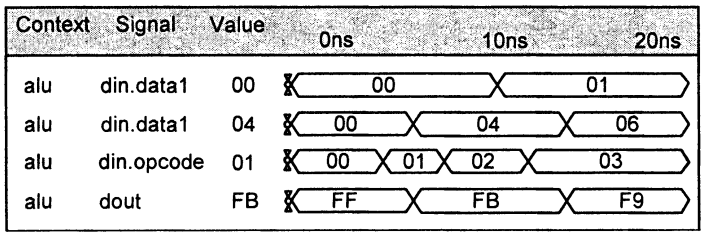


Рис 4.16. Временные диаграммы для листинга 4.24

Поведенческое моделирование элементов памяти

Рассмотрим пример описания элементов памяти для выполнения поведенческого моделирования в OrCAD Express 9.1.

В примере описана система памяти, включающая в себя два одинаковых блока. Структура системы представлена на рис. 4.17. Сигналы тактирования, адреса, входных и выходных данных для блоков памяти общие. Каждый блок имеет свой сигнал разрешения записи и разрешения чтения.

В программе на VHDL (листинг 4.25) каждый блок памяти в модели описывается как компонент.

Каждый блок памяти имеет следующие порты:

- `clk` — тактирование;
- `w` — разрешение записи;

- `r` — разрешение чтения;
- `addr` — адрес;
- `datai` — данные для записи в память;
- `datao` — данные, читаемые из памяти.

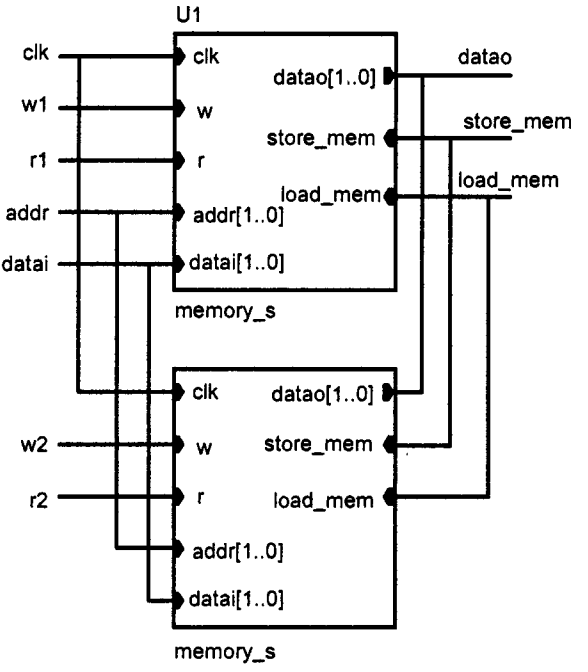


Рис 4.17. Структура системы памяти, включающая в себя два одинаковых блока

Для выполнения проверки правильности функционирования модели, включающей в себя систему памяти, можно сохранять в файлах образы памяти в определенные моменты времени. Начальные состояния памяти также могут быть загружены из файлов. В примере будет рассмотрено, как несколько компонентов могут работать с одним файлом.

Входные сигналы `load_mem` и `store_mem` не имеют эквивалентов на физическом уровне. Здесь `load_mem` — сигнал, по которому происходит загрузка данных из файла в блок памяти; `store_mem` — сигнал, по которому происходит запись данных из блока памяти в файл.

Описание интерфейса включает в себя также секцию **`generic`**, в которой описан параметр — `un_name`, предназначенный для хранения уникального идентификатора блока памяти.

Структурное описание схемы имеет следующий вид (листинг 4.25).

Листинг 4.25

```
library ieee;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use IEEE.std_logic_arith.all;
use textio.all;

-- подключение пакетов interf147 и memory_s1
use interf147.interf147.all;
use memory_s1.memory_s1.all;

-- декларация моделируемого объекта
entity schematics1 is
end entity schematics1;

-- описание архитектуры (структурное) для schematics1
architecture structure of schematics1 is
-- декларация компонентов
component memory_s is
generic (un_name: string(1 to 5));
port(clk: in std_logic;
     w: in std_logic;
     r: in std_logic;
     addr: in std_ulogic_vector((addr_length-1) downto 0);
     datai: in std_logic_vector((reg_size-1) downto 0);
     datao: out std_logic_vector((reg_size-1) downto 0) bus;
     load_mem: in std_logic;
     store_mem: in std_logic    );
end component;

-- декларация сигналов
signal clk: std_logic;
signal w1: std_logic;
signal rr1: std_logic;
signal w2: std_logic;
signal rr2: std_logic;
signal addr: std_ulogic_vector((addr_length-1) downto 0);
```

```
signal datai: std_logic_vector((reg_size-1) downto 0);
signal datao: std_logic_vector((reg_size-1) downto 0);
signal load_mem: std_logic;
signal store_mem: std_logic;

-- структурное описание архитектурного тела structure
-- для объекта schematics1;
-- включение компонентов в структуру устройства
begin
-- компонент-экземпляр u1
U1 : memory_s
generic map (un_name = > "name1")
port map (clk = > clk,
          w = > w1, r = > rr1,
          addr = > addr, datai = > datai, datao = > datao,
          load_mem = > load_mem,
          store_mem = > store_mem);

-- компонент-экземпляр u2
U2 : memory_s
generic map (un_name = > "name2")
port map (clk = > clk,
          w = > w2, r = > rr2,
          addr = > addr, datai = > datai, datao = > datao,
          load_mem = > load_mem,
          store_mem = > store_mem);

end architecture structure;
```

Пакет `interf147`, содержащий описание типов, констант и файлов, предназначенных для хранения образов памяти, представлен листингом 4.26. Пакет подключен и используется программой, представленной листингом 4.25. Описание файлов вынесено в пакет в связи с тем, что в нашем примере оба объекта — `U1` и `U2` (блоки памяти 1 и 2) работают с одним и тем же набором файлов.

Листинг 4.26

```
library IEEE;
use IEEE.std_logic_1164.all;
use std.textio.textio;
```

```
package interf147 is

constant reg_size:integer: = 8;
constant addr_length:integer: = 8;
constant mem_size:integer: = 4;

type mem_array is
  array(0 to (mem_size-1)) of std_logic_vector((reg_size-1) downto 0);

file vectors_w: text;
file vectors_r: text open read_mode is "load6.dat";
end package interf147;
```

В пакете описаны константы:

- `reg_size` — длина слова памяти в битах;
- `addr_length` — разрядность адреса в битах;
- `mem_size` — размер памяти в словах.

В этом пакете описан также новый тип `mem_array` — массив, предназначенный для хранения содержимого памяти. Массив описан как одномерный, но каждый его элемент является вектором `(std_logic_vector((reg_size-1)downto 0))`, соответствующим одному слову памяти.

Для работы с файлами, в этом пакете описаны две файловые переменные `vectors_w` и `vectors_r`. Оба эти логические файла имеют базовый текстовый тип. Процедуры и функции для работы с этим типом описаны в стандартном пакете `textio`. При описании файла `vectors_r` сразу же определяется режим его открытия — `read_mode` (для чтения) и имя физического файла, с которым он будет связан — файл `load6.dat`.

В файлах образы памяти располагаются последовательно: сначала образ первого блока памяти, затем образ второго блока памяти. Образ каждого из этих блоков памяти начинается с уникального идентификатора блока, после чего следуют `mem_size` слов памяти, начиная с первого.

Пакет `memory_s1` содержит в себе описание компонента `memory_s`. Пакет `memory_s1` подключен и используется программой на листинге 4.27.

Описание архитектуры `memory_s` на поведенческом уровне представлено листингом 4.27.

Листинг 4.27

```
library ieee;
use IEEE.STD_LOGIC_1164.all;
```

```
use IEEE.NUMERIC_STD.all;
use IEEE.std_logic_arith.all;
use std.textio.textio;
use interf147.interf147.all;

package memory_s1 is
component memory_s is
generic (un_name:string(1 to 5));
port(clk: in std_logic;
     w: in std_logic;
     r: in std_logic;
     addr: in std_ulogic_vector((addr_length-1) downto 0);
     datai: in std_logic_vector((reg_size-1) downto 0);
     datao: out std_logic_vector((reg_size-1) downto 0) bus;
     load_mem: in std_logic;
     store_mem: in std_logic
    );
end component memory_s;
end package memory_s1;

library ieee;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
use IEEE.std_logic_arith.all;
use std.textio.textio;
use interf147.interf147.all;
-- если в пакете package memory_s1 мы декларировали компонент memory_s,
-- то здесь декларируем сам объект memory_s;
-- объекты этого типа могут моделироваться отдельно,
-- а могут использоваться как компоненты в структурных описаниях
-- других объектов
entity memory_s is
generic (un_name:string(1 to 5));
port(clk: in std_logic;
     w: in std_logic;
     r: in std_logic;
     addr: in std_ulogic_vector((addr_length-1) downto 0);
     datai: in std_logic_vector((reg_size-1) downto 0);
     datao: out std_logic_vector((reg_size-1) downto 0) bus;
```

```
    load_mem: in std_logic;
    store_mem: in std_logic
   );
end entity memory_s;

architecture rtl of memory_s is
signal mem_arr: mem_array;

begin
  mem_from_file_or_bus: process(load_mem, clk)
variable vector: integer;
variable cou_0:natural;
variable c_name:string(1 to 5);
variable cn:integer;
begin
if load_mem'event and load_mem = '1' then  cn: = 0;
while not(endfile(vectors_r)) and cn = 0 loop
  read(vectors_r,c_name,5);
  if c_name = un_name then cn: = 1; end if;
  cou_0: = 0;
  while not(endfile(vectors_r)) and cou_0<mem_size loop
        read(vectors_r,vector,1);
        if c_name = un_name then
           mem_arr(cou_0)< = std_logic_vector(to_unsigned(vector,
           reg_size),reg_size);
        end if;
        cou_0: = cou_0 + 1;
  end loop;
end loop;
else
  if clk'event and clk = '1' then
        if w = '1' then
           mem_arr(to_integer(unsigned(addr)))< = datai;
        end if;
  end if;
end if;
end process mem_from_file_or_bus;

  mem_image_to_file: process(store_mem)
```

```
variable vector:integer;
variable strr:string(1 to 5);
variable cou_0:natural;
begin
if store_mem'event and store_mem = '1' then
     file_open(vectors_w,"load5.dat",append_mode);
     write(vectors_w,un_name);
     for cou_0 in 0 to (mem_size-1) loop
         vector: = to_integer(unsigned(mem_arr(cou_0)));
         write(vectors_w,vector);
     end loop;
     file_close(vectors_w);
end if;
end process mem_image_to_file;

  mem_word_to_bus: process (r, addr)
begin
if r = '1' then
      datao< = mem_arr(to_integer(unsigned(addr)));
 else datao< = "ZZZZZZZZ";
end if;
end process mem_word_to_bus;

end architecture rtl;
```

Рассмотрим описание объекта `memory_s`, соответствующего блоку памяти. Для хранения содержимого ячеек памяти используется сигнал (не переменная, поскольку сигналы удобнее просматривать на временных диаграммах) `mem_arr`, который имеет тип `mem_array`, описанный в пакете `interf147`. Описание поведения объекта `memory_s` включает в себя 3 процесса.

Первый процесс (`mem_from_file_or_bus`) предназначен для загрузки содержимого файла в память (т. е. установки начальных значений элементов памяти) и для записи в память данных с входа `datai`. Процесс активизируется при изменении сигнала `load_mem` или сигнала тактирования.

Чтение данных из файла и загрузка их в память происходит по восходящему фронту `load_mem`. Чтение файла осуществляется следующим образом. В процессе `mem_from_file_or_bus` описана переменная `vector`, имеющая тип `integer`. В нее считываются строки файла, соответствующие словам памяти. Переменная `cou_0` используется как счетчик количества слов. Переменная `c_name` служит для хранения идентификатора образа памяти, который чита-

ется из файла в текущий момент времени. Переменная cn используется как флаг, она принимает значение 1, если при чтении дошли до места, где расположено начало образа памяти, соответствующего данному блоку памяти.

В теле процесса выполняются следующие действия: переменной cn присваивается начальное значение 0, затем организуется цикл, где происходит считывание образов памяти. Считывание выполняется до тех пор, пока не будет достигнут образ памяти, соответствующий данному блоку, или не будет достигнут конец файла (для этих целей используется конструкция while...loop). Для проверки достижения конца файла используется функция endfile пакета textio.

Первое слово образа памяти содержит идентификатор блока памяти, который проверяется на совпадение с идентификатором текущего блока. Если совпадение произошло, переменой cn присваивается значение 1.

Обнуляется счетчик элементов памяти. Далее организуется цикл, в котором происходит поэлементное считывание образа блока памяти от первого до последнего слова. Если идентификатор текущего блока совпадает с идентификатором образа памяти, очередная запись файла записывается в очередное слово памяти. Операция чтения из файла осуществляется с помощью стандартной функции read, параметрами которой являются имя логического файла, переменная, в которую прочитывается значение элемента и размер читаемого элемента, если он является вектором. Номер слова памяти определяется значением счетчика.

Данные, читаемые из файла, имеют тип integer. Для того чтобы записать их в элемент массива памяти, имеющий тип std_logic, используется функция преобразования to_unsigned (преобразует тип integer в тип unsigned), размер результирующего вектора (reg_size), а также функция std_logic_vector (преобразует unsigned в logic_vector того же размера).

Если же сигнал load_mem имеет значение '0' или не менял своего значения (процесс инициирован событием изменения сигнала clk), то запись в память данных может быть выполнена с шины datai. Запись данных с входа datai в элемент памяти, номер которого определяется значением сигнала addr, выполняется по восходящему фронту сигнала тактирования, при условии, что сигнал разрешения записи w установлен в '1'. Сигнал addr имеет тип std_logic, поэтому он должен быть преобразован в тип integer, соответствующий типу индекса массива. Для этого используется функция unsigned, преобразующая значение типа std_logic в значение типа unsigned, а также функция to_integer, преобразующая значение типа unsigned в значение типа integer.

Второй процесс, mem_image_to_file, предназначен для сохранения образа памяти в файле. В нем описаны следующие переменные:

- `vector` — предназначена для хранения слова памяти, которое будет записано в файл;
- `strr` — строка — идентификатор блока памяти;
- `cou_0` — счетчик слов памяти.

Процесс записи активизируется по восходящему фронту сигнала `store_mem`. В начале выполняется открытие файла в режиме записи и связывание с физическим файлом `load5.dat`, для чего используется функция `file_open` с параметрами:

- `vectors_w` — имя логического файла;
- `load5.dat` — имя физического файла;
- `append_mode` — режим продолжения записи.

Использование этого режима позволяет выполнить дозапись образа блока памяти в конец файла, не разрушая предыдущую информацию, после чего в файл записывается идентификатор блока памяти при помощи функции `write` пакета `textio`. После этого в цикле выполняется запись содержимого массива памяти в файл (от первого до последнего слова памяти). В переменную `vector` записывается значение каждого элемента массива, предварительно преобразованного к типу `integer`. Значение переменной записывается в файл. После того, как содержимое памяти полностью записано в файл, он закрывается. Для этого используется процедура `file_close`. В первом процессе не выполняется открытие и закрытие файла, поскольку при описании логического файла производится связывание с физическим файлом, а дальнейшая работа с ним в этом случае осуществляется автоматически.

Третий процесс используется для выполнения операции чтения из памяти на шину `datao`. Чтение выполняется асинхронно, при наличии сигнала разрешения чтения. Активизация процесса выполняется при изменении значения сигнала `r` или `addr`. Если значение сигнала `r` равно `'1'` то на выход `datao` подаются данные элемента массива памяти, соответствующего текущему значению адреса. Пример временной диаграммы работы устройства приведен на рис. 4.18.

Эта диаграмма соответствует выполнению следующих действий. В момент времени `1` сигнал `load_mem` меняет значение с `'0'` на `'1'`, в результате чего в блоки памяти загружаются начальные значения. Далее, с момента времени `3` по момент времени `42`, сигнал разрешения записи в блок памяти `1` имеет значение `'1'`. На вход адреса подаются значения `0—3`, на вход данных — значения `1—4`, которые и записываются в первый блок памяти. Затем, с момента времени `42` по момент времени `100`, сигнал разрешения записи во второй блок памяти имеет значение `'1'`. На вход адреса подаются значения `0—3`, на вход данных — значения `5—8`; которые записываются во второй

блок памяти. С момента времени 38 по момент времени 50 установлен сигнал разрешения чтения из первого блока памяти. В результате на шину выходных данных с 38 по 40 выдается значение из третьей ячейки блока памяти, затем из нулевой ячейки в соответствии с текущим значением адреса. С момента времени 80 по момент 100 установлен сигнал разрешения чтения из второго блока памяти. В соответствии со значением адреса выполняется чтение третьей ячейки блока памяти. В момент времени 80 осуществляется запись содержимого блоков памяти в файл.

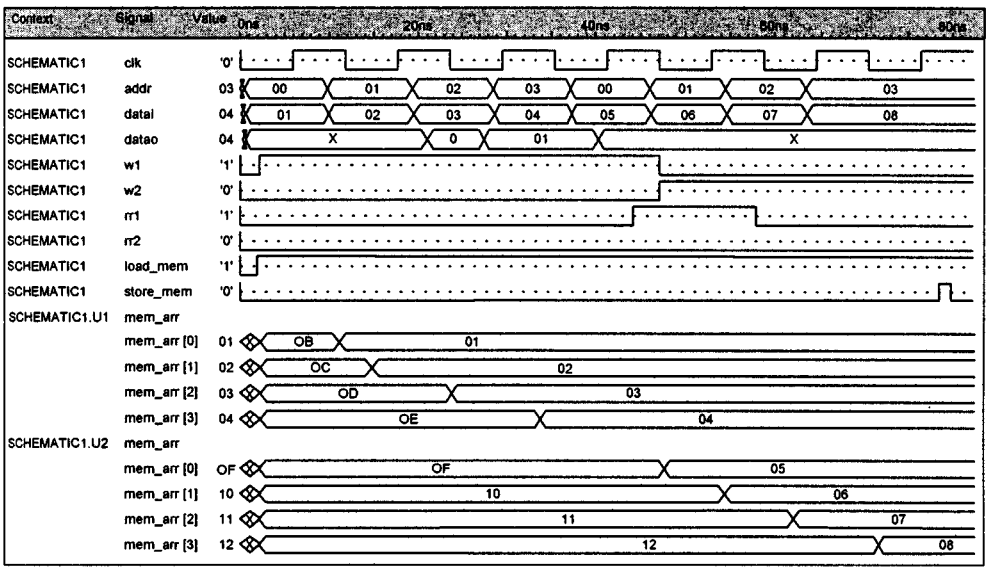


Рис. 4.18. Пример временной диаграммы работы устройства памяти `memory_s`

Пример использования подпрограмм в поведенческом моделировании

При выполнении поведенческого моделирования в проект могут включаться объекты, предназначенные для отладки и контроля функционирования модели. Такие объекты не используются при синтезе и не включаются в физическую реализацию. Например, это могут быть объекты — источники входных тестовых значений для основной модели.

Рассмотрим, как с использованием подпрограмм может быть создан источник входных значений, являющихся натуральными числами и равномерно распределенных, согласно случайному закону, на некотором интервале.

VHDL 93 не включает в себя функцию, возвращающую псевдослучайные числа. Рассмотрим один из вариантов написания такой функции.

В настоящее время существует множество алгоритмов генерации случайных чисел. Рассмотрим один из алгоритмов работы генератора псевдослучайной последовательности. Пусть целью алгоритма является генерация случайных трехзначных чисел, равномерно распределенных в интервале от 0 до 999 включительно. В алгоритме используется два положительных нечетных целых числа, каждое их которых может принимать значения в интервале от 0 до 999. Первое из этих чисел (в дальнейшем будем называть его ядром) не меняет своего значения на протяжении всего выполнения алгоритма, значение второго (в дальнейшем будем называть его множителем) меняется. Каждый раз, когда необходимо получить новое случайное число, значение множителя изменяется в соответствии с некоторой последовательностью. Генерация очередного псевдослучайного числа осуществляется следующим образом: ядро умножается на множитель, результатом чего является шестиразрядное целое, на основе которого определяется значение очередного псевдослучайного числа. Это произведение, в свою очередь, используется для получения нового значения множителя, которое будет применено для получения следующего псевдослучайного числа. Пусть, например, значение очередного псевдослучайного числа берется равным значению трех младших разрядов произведения, а значение множителя определяется равным значению 5 — 2 разрядов произведения.

Рассмотрим вариант реализации генератора псевдослучайных последовательностей на VHDL. Поместим его описание в пакете `random_1` (листинг 4.28). Собственно формирование случайного числа выполняется с использованием функции `rand_1`, которая имеет два параметра: `base` — ядро и `mn2` — множитель. Она возвращает значение типа `tl`. Этот тип так же описан в пакете `random_1` и представляет собой вектор размерности 2. Первый его элемент используется для хранения результата (очередного случайного числа), второй — для хранения множителя, используемого при следующей операции получения случайного числа. В пакете описана так же константа `rez_size` (разрядность генерируемого случайного числа).

Для получения очередного случайного числа переменной `pr` присваивается значение произведения ядра и множителя. Затем на базе `pr` в переменной `res` формируется значение очередного псевдослучайного числа, для чего используется цикл, который выполняется три раза. Каждый раз определяется значение одного разряда результата — сначала третьего, затем второго, а затем первого. Для этого используется вспомогательная функция `get_pos_n`, которая имеет два параметра: `n` — целое число и `pos` — номер позиции (разряда). Функция возвращает значение указанного разряда десятичного представления числа `n`. Следующее значение множителя определяется аналогичным образом.

Такие функции могут быть использованы для создания объекта, предназначенного для тестирования моделей. Пример такого объекта приведен в листинге 4.29.

Объект имеет вход тактирования clk и два информационных входа data и pos. Он имеет два выхода datao и datao1. На первый из них выдается значение функции get_pos, вызываемой с параметрами data и pos. На выход datao1 выдается значение функции rand_1, вызываемой с параметрами 433 (ядро) и dmn2. Сигнал dmn2 — внутренний сигнал, используемый для хранения значения множителя. (dmn2, dsum, ds декларированы как сигналы, поскольку так удобнее просматривать их на временных диаграммах). Эти действия выполняются в первом процессе, который выполняется при каждом изменении сигнала тактирования clk. Таким образом, каждый раз при изменении сигнала тактирования генерируется новое псевдослучайное число. Второй процесс предназначен для того, чтобы оценить качество псевдослучайной последовательности. В нем выполняются следующие действия: сигнал dsum используется для накопления суммы значений получаемых псевдослучайных величин, а сигнал ds — для подсчета количества полученных псевдослучайных величин. На базе этих двух сигналов может быть вычислено среднее значение случайной величины в любой момент времени.

Листинг 4.28

```
package random_1 is
type t1 is array (1 to 2) of natural;
constant rez_size:natural: = 3;

function get_pos_n(n:integer; pos:natural) return natural;
function rand_1(base: natural; mn2:natural) return t1;

end package random_1;

package body random_1  is
   function get_pos_n(n:integer; pos:natural) return natural is
   variable i:natural;
   variable n1,n2:natural;
   begin
       n2: = n;
       for i in 1 to pos loop
          n1: = n2 mod 10;
          n2: = n2 / 10;
      end loop;
```

```
  return n1;
end function get_pos_n;

   function rand_1(base: natural; mn2:natural) return t1 is
   variable pr,mn2_2,res,i1:natural;
   variable res_t:t1;
   begin
     pr: = base*mn2;
     res: = 0;
     for i1 in rez_size downto 1 loop
       res: = res*10 + get_pos_n(pr,i1);
     end loop;
    res_t(1): = res;
  n2_2: = 0;
  for i1 in rez_size + 2 downto 3 loop
    mn2_2: = mn2_2*10 + get_pos_n(pr,i1);
  end loop;
  res_t(2): = mn2_2;
  return (res_t);
end function rand_1;
end package body random_1;
```

Листинг 4.29

```
use random_1.random_1.all;

entity obj1 is
port (clk: in natural;
      data:in integer;
      pos:in natural;
      datao:out natural;
      datao1:out natural);
end entity obj1;
architecture rtl of obj1 is
signal dres:natural;
signal dmn2:natural: = 10;
signal dsum:natural: = 0;
signal ds:natural: = 0;
```

```
begin
        process(clk)
        begin
            datao< = get_pos_n(data,pos);
            datao1< = rand_1(433,dmn2)(1);
            dres< = rand_1(433,dmn2)(1);
            dmn2< = rand_1(433,dmn2)(2);
        end process;

        process(dres)
        begin
             dsum< = dsum + dres;
             ds< = ds + 1;
        end process;
end architecture rtl;
```

Результаты работы примера приведены на рис. 4.19 и 4.20.

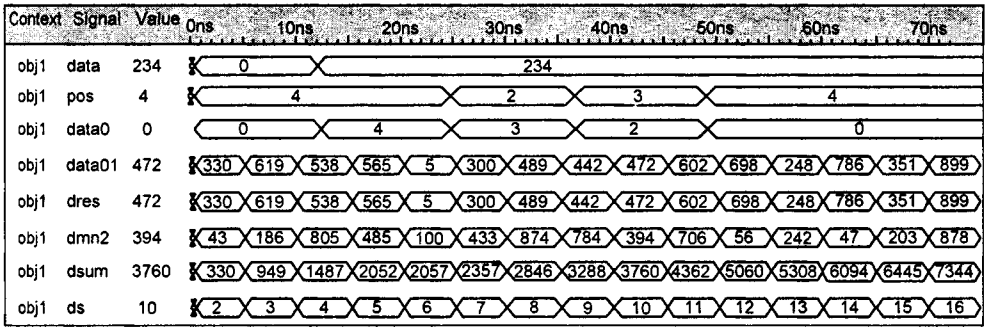


Рис. 4.19. Временные диаграммы работы устройства (начальный фрагмент листинга 4.28)

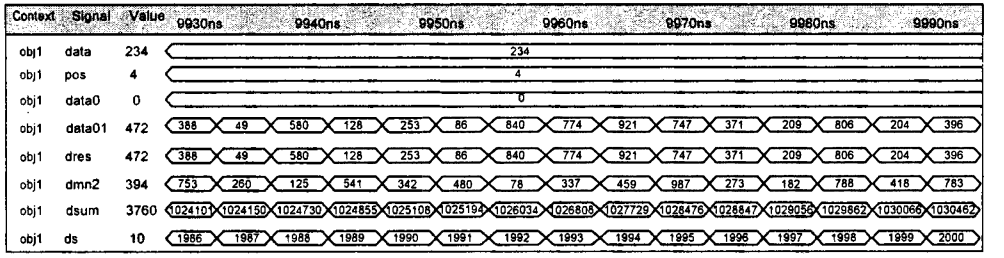


Рис. 4.20. Временные диаграммы работы устройства (другой фрагмент листинга 4.29)

Объединение сигналов на шине

Выбор типа сигнала для выходов на общую шину

Промоделируем устройство, структура которого изображена на рис. 4.21.

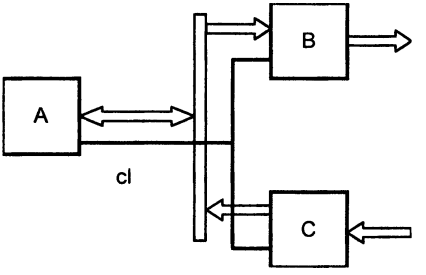


Рис. 4.21. Структурная схема устройства, представленного в листинге 4.30

В модели имеется шина, на которую подключаются выходные сигналы от блока A (сигнал является и входным и выходным) и блока C. Предполагается, что система должна работать так, чтобы в один момент времени на эту шину выдавал данные только один компонент. Для этого используется управляющий сигнал `cl`, который определяет моменты времени, в которые на шину выставляет данные тот или иной компонент. Если `cl = '1'`, то данные на шину выставляет компонент A (в модели ему соответствует сущность `aaa`), в противном случае на шину выставляет данные компонент C (в модели ему соответствует сущность `ccc`). Компонент A выставляет на шину константу, например, 23. Компонент C выставляет на шину данные, поступающие на его информационный вход. Компонент B (в модели ему соответствует сущность `bbb`) выставляет на свой информационный выход то, что в данный момент находится на его информационном входе, т. е. — то, что в данный момент находится на общей шине. Таким образом, при `cl = '1'` на выходе мы должны наблюдать значение константы 23, а в противном случае — значение, которое в данный момент находится на информационном входе компонента C.

Модель может быть реализована с использованием механизма компонентов (листинг 4.30). Она может быть целиком расположена в одном файле или каждое из устройств может быть расположено в отдельном файле. Если все эти файлы включены в один проект, то на работе это не отражается. Один из файлов, входящих в состав модели, может содержать описание каждого из устройств A, B, C, а другой файл — описание устройства в целом, в состав которого входят устройства A, B, C как блоки.

Предпочтительным является вариант расположения каждого устройства в отдельном файле, поскольку это облегчает процесс проектирования и позволяет нескольким разработчикам одновременно работать над одной моделью.

Листинг 4.30

```
entity aaa is
    port (aaainout: inout integer;
              cl:in bit);
end entity aaa;
architecture aaa_behaviour of aaa is
begin
    process(cl)
    begin
        if cl = '1' then aaainout< = 23; end if;
    end process;
end architecture aaa_behaviour;

entity bbb is
port (bbbout:out integer;
          bbbin: in integer;
          cl:in bit);
end entity bbb;
architecture bbb_behaviour of bbb is
begin
    process (bbbin)
    begin bbbout< = bbbin; end process;
end architecture bbb_behaviour;

entity ccc is
port (cccout:out integer;
          cccin: in integer;
          cl: in bit);
end entity ccc;
architecture ccc_behaviour of ccc is
begin
process (cccin)
       begin if cl = '0' then cccout< = cccin; end if;
end process;
end architecture ccc_behaviour;
library ieee;
use ieee.std_logic_1164.all
```

```
entity schematics1 is end schematics1;
architecture sch1_structure of schematics1 is
component aaa is
    port (aaainout: inout integer;
             c1: in bit);
  end component aaa;

component bbb is
port (bbbout:out integer;
        bbbin: in integer;
        c1: in bit);
  end component bbb;

component ccc is
port (cccout:out integer;
        cccin: in integer;
        c1: in bit);
   end component ccc;

signal orcad_unused:std_logic;
signal N00037: integer;
signal N00034: integer;
signal N00012: integer;
signal N00015: bit;

begin
-- включение компонента-экземпляра u1 типа aaa
u1: aaa
port map(aaainout = > N00037,
                c1 = > N00015);
-- включение компонента-экземпляра u2 типа bbb
u2: bbb
port map(bbbin = > N00037,
               bbbout = > N00034,
                c1 = > N00015);
-- включение компонента-экземпляра u3 типа ccc
u3: ccc
port map (cccout = > N00037,
                cccin = > N00012,
                 c1 = > N00015);
end sch1_structure;
```

Это структурное описание было синтезировано средствами OrCAD 7.0 на базе рисунка схемы. Названия сигналов (линий связей между компонентами N000XX) сгенерированы автоматически. Общей шине соответствует сигнал N00037. Сигнал N00012 соответствует информационному входу компонента, ccc — информационному входу системы в целом. Сигнал N00034 соответствует выходу компонента, bbb — информационному выходу системы в целом.

Результат работы этого устройства представлен на рис. 4.22.

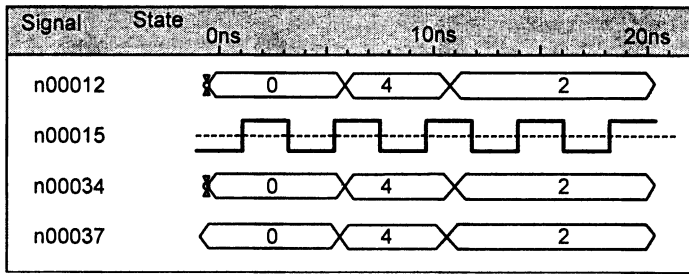


Рис. 4.22. Временная диаграмма работы устройства, описанного в листинге 4.30

Описание устройства, представленного листингом 4.30, выглядит вполне логично. Однако устройство не работает желаемым образом: на выходе все время присутствует то же самое, что и на входе, независимо от значения сигнала c1.

Это связано с выбором типа сигнала для выходов на общую шину. Выбран логический тип, который не позволяет промоделировать отсутствие сигнала. Для этого типа не определена вычисляемая функция, которая позволяла бы определить, какое значение будет иметь сигнал на шине, если от разных источников поступают различные значения.

Рассматриваемые инструменты проектирования не позволяют пользователю описывать новые вычисляемые функции. Заменим логический тип физическим, причем таким, который содержит третье состояние (например, на тип std_ulogic, для которого определена вычисляемая функция в пакете std_logic_1164). В листинге тип **integer** везде должен быть заменен типом std_ulogic. Константа 23 в компоненте A должна замениться константой '1'. Описание блоков A, B, C будет иметь вид, представленный в листинге 4.31.

Листинг 4.31

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity aaa is
   port (aaainout: inout std_ulogic;
```

```
            cl:in bit  );
    end entity aaa;
architecture aaa_behaviour2 of aaa is
begin
        process(cl)
          begin if cl = '1' then aaainout< = '1'; else aaainout< = 'Z';
end if;
       end process;
end architecture aaa_behaviour2;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity bbb is
port (bbbout:out std_ulogic;
         bbbin: in std_ulogic;
         cl:in bit);
end entity bbb;
architecture bbb_behaviour2 of bbb is
begin
  process (bbbin)
  begin bbbout< = bbbin; end process;
end architecture bbb_behaviour2;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ccc is
port (cccout:out std_ulogic;
         cccin: in std_ulogic;
         cl: in bit);
end entity ccc;
architecture ccc_behaviour2 of ccc is
begin
  process (cccin,cl)
  begin
          if cl = '0' then cccout< = cccin; else cccout< = 'Z'; end if;
  end process;
end architecture ccc_behaviour2;
```

В описании архитектуры устройств A, B и C, при изменении сигналов на входах xxxin, запускаются процессы, выставляющие на выходной порт значения сигнала с входного порта. Когда устройство A или C не должно выхо-

дить на шину, его выход переводится в третье состояние, а значение сигнала на шине определяется в соответствии с вычисляемой функцией для данного типа (текст решающей функции из пакета std_logic_1164 приводится в листинге 4.32): если один из выходящих на шину сигналов установлен в z, то значение сигнала на шине такое же, как и значение второго выходного сигнала на шину.

Листинг 4.32

```
CONSTANT resolution_table : stdlogic_table : = (
    --      ---------------------------------------------------------
    --      |  U    X    0    1    Z    W    L    H    -        |   |
    --      ---------------------------------------------------------
            ('U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'), -- | U |
            ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'), -- | X |
            ('U', 'X', '0', 'X', '0', '0', '0', '0', 'X'), -- | 0 |
            ('U', 'X', 'X', '1', '1', '1', '1', '1', 'X'), -- | 1 |
            ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X'), -- | Z |
            ('U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X'), -- | W |
            ('U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X'), -- | L |
            ('U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X'), -- | H |
            ('U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X')  -- | - |
    );

    FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic IS
        VARIABLE result : std_ulogic : = 'Z';  -- weakest state default
        ATTRIBUTE synthesis_return OF result:VARIABLE IS
"WIRED_THREE_STATE" ;
    BEGIN
        -- the test for a single driver is essential otherwise the
        -- loop would return 'X' for a single driver of '-' and that
        -- would conflict with the value of a single driver unresolved
        -- signal.
        IF     (s'LENGTH = 1) THEN     RETURN s(s'LOW);
        ELSE
            FOR i IN s'RANGE LOOP
                result : = resolution_table(result, s(i));
            END LOOP;
        END IF;
        RETURN result;
    END resolved;
```

Теперь устройство в целом работает правильно. Временная диаграмма его работы приведена на рис. 4.23.

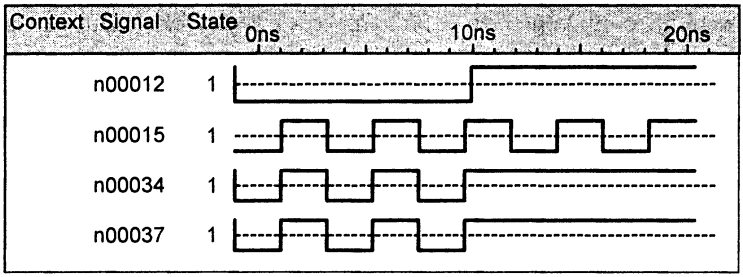


Рис. 4.23. Временная диаграмма, соответствующая устройству, описанному в листинге 4.31

Однако рассмотрим, что будет, если блоки A и C начнут одновременно выдавать информацию на общую шину. Такая ситуация могла бы возникнуть, если бы мы в описании блока C отредактировали тело описания процесса следующим образом (листинг 4.33).

Листинг 4.33

```
if c1 = '1' then cccout< = cccin;
   else cccout< = 'Z';
    end if;
```

Тогда и блок C, и блок A выдавали бы логический сигнал при одном и том же значении сигнала `c1 = '1'`. Диаграмма работы устройства в таком случае имела бы вид, приведенный на рис 4.24.

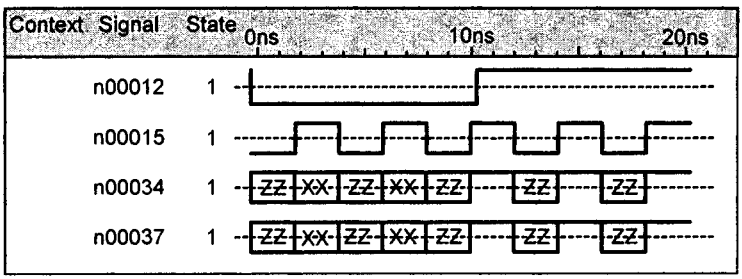


Рис. 4.24. Временная диаграмма работы устройства, соответствующего листингу 4.33

Когда два устройства одновременно выдают различную информацию на шину, на ней наступает неопределенное состояние, в соответствии с решающей функцией для данного типа. Когда и компонент A, и компонент C

выставляют на шину 'Z', то на шине присутствует сигнал 'Z'. Когда A выставляет '1', C выставляет '0', то на шине присутствует 'X' — неопределенное значение. Если и A и C выставляют на шину '1', то на шине присутствует значение '1'.

В данном примере рассматривалось поведение объединяемых на общую шину сигналов, источники которых находились в различных компонентах. Вычисляемая функция для типа std_ulogic использовалась по умолчанию. Такое поведение модели получено с использованием OrCAD 7.0. Для OrCAD 9.0, 9.1 характерно другое поведение сигналов, имеющих множество источников, если для них используются вычисляемые функции по умолчанию. Различия могут привести к несоответствию в поведении моделей при переносе из OrCAD 7 в OrCAD 9.

Особенности использования типов *std_logic* и *std_ulogic* для организации общих шин в OrCAD. Функции разрешения коллизий

В общем случае, тип std_logic_vector является подтипом std_ulogic_vector, но поведение сигналов этих типов в OrCAD имеет существенные отличия. Функции разрешения коллизий для них организованы по-разному.

Рассмотрим это на примере. Пусть имеется объект, описание которого имеет вид, представленный листингом 4.34.

Листинг 4.34

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
entity ent_1 is
port (i: in std_logic_vector(1 downto 0);
      o: out std_logic_vector(1 downto 0) ;
      oo2: out std_ulogic_vector(1 downto 0);
      oo01: out std_ulogic_vector(1 downto 0);
      oo02: out std_logic_vector(1 downto 0);
      r, rr: in std_logic;
      w: in std_logic);
end entity ent_1;

architecture rtl of ent_1 is
signal inv: std_logic_vector(1 downto 0);
begin
```

```
        process (w)
        begin
            if w = '1' and w'event then  inv< = i;  end if;
        end process;

        process (r)
        begin
           if r = '1' and r'event then o< = inv; oo2< = inv;
                                       oo01< = "11"; oo02< = "11";
             else o< = "ZZ"; oo2< = "ZZ";   end if;
        end process;

        process(rr)
        begin
             if rr = '1' and rr'event then oo01< = "00"; oo02< = "00";
end if;
        end process;
end architecture rtl;
```

Описанный объект `rtl` имеет один информационный вход `i`, три управляющих входа `r`, `rr`, `w` и четыре выхода `o`, `oo2`, `oo01`, `oo02`. Внутри объекта описан сигнал `inv`. Описание поведения этого объекта включает в себя три процесса. Первый из них срабатывает при изменении значения управляющего сигнала `w`. По восходящему фронту этого сигнала значение информационного входа объекта записывается в его внутренний сигнал `inv`. Второй процесс активизируется при изменении значения сигнала `r`. По восходящему фронту этого сигнала выходы объекта `o` и `oo2` устанавливаются равными значению внутреннего сигнала, а `oo01` и `oo02` устанавливаются в значение `"11"`. По нисходящему фронту `r` выходы `o` и `oo2` переводятся в высокоимпедансное состояние. Третий процесс активируется при изменении сигнала `rr`. По переднему фронту этого сигнала `oo01` и `oo02` устанавливаются в `"00"`.

Поведение сигналов `o` и `oo2` описано одинаково, поведение сигналов `oo01` и `oo02` также описано одинаково, но сигналы в этих парах имеют разные типы: `o` и `oo2` — тип `std_logic_vector`, а — тип `std_ulogic_vector`. Поэтому, как будет видно в дальнейшем, в различных ситуациях вести себя они будут по-разному. Обратите внимание! Значения сигналам `o` и `oo2` присваиваются в одном процессе, а сигналам `oo1` и `oo2` в двух процессах. Таким образом, последняя пара сигналов имеет множество источников — процессов.

Рассмотрим объект, описание которого имеет вид, представленный листингом 4.35.

Листинг 4.35

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity model2 is
port (i1: in std_logic_vector(1 downto 0);
      o1: out std_logic_vector(1 downto 0) ;
      o2: out std_ulogic_vector(1 downto 0);
      o3: out std_ulogic_vector(1 downto 0);
      o4: out std_ulogic_vector(1 downto 0);
      o5: out std_logic_vector(1 downto 0) ;
      o6: out std_logic_vector(1 downto 0) ;
      r1,r2,r3,r4: in std_logic;
      w1,w2: in std_logic);
end entity model2;

architecture struct of model2 is
component ent_1 is
port (i: in std_logic_vector(1 downto 0);
      o: out std_logic_vector(1 downto 0) ;
      oo2: out std_ulogic_vector(1 downto 0);
      oo01: out std_ulogic_vector(1 downto 0);
      oo02: out std_logic_vector(1 downto 0);
      r,rr: in std_logic;
      w: in std_logic);
end component ent_1;

signal si:  std_logic_vector(1 downto 0);
signal so:  std_logic_vector(1 downto 0) ;
signal so2: std_ulogic_vector(1 downto 0);
signal so3: std_ulogic_vector(1 downto 0);
signal so4: std_ulogic_vector(1 downto 0);
signal so5: std_logic_vector(1 downto 0);
signal so6: std_logic_vector(1 downto 0);
signal sr1,sr2,sr3,sr4:  std_logic;
```

```
signal sw1,sw2:  std_logic;

begin
U1:  ent_1
       port map(i = >si, o = >so, oo2 = >so2, oo01 = >so3,
                oo02 = >so5, r = >sr1, rr = >sr3,w = >sw1);
U2:  ent_1
       port map(i = >si, o = >so, oo2 = >so2, oo01 = >so4,
                oo02 = >so6, r = >sr2, rr = >sr4,w = >sw2);

  process(i1,so,so2, so3, so4,so5,so6,r1,r2,r3,r4,w1,w2)
  begin
          si< = i1; o1< = so;  o2< = so2; o3< = so3;
          o4< = so4; o5< = so5; o6< = so6;
          sr1< = r1; sw1< = w1; sr2< = r2; sw2< = w2; sr3< = r3;
sr4< = r4;
  end process;
end architecture struct;
```

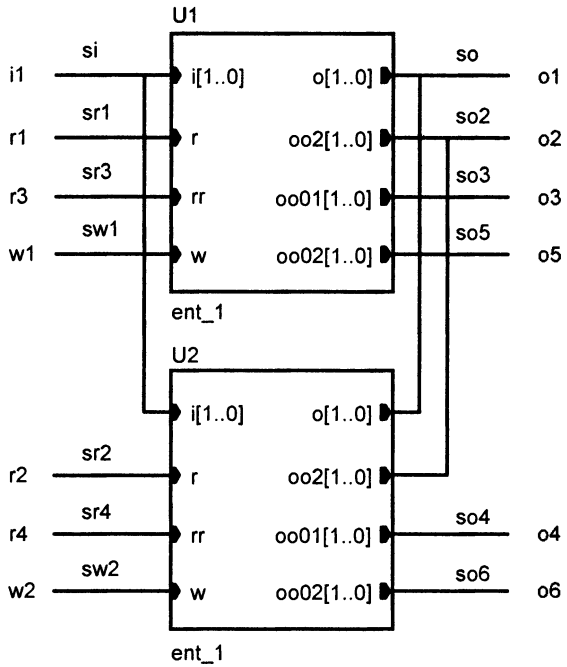


Рис. 4.25. Структура моделируемого объекта `model2`

Это описание архитектуры объекта model2 является комбинированным: оно включает в себя элементы структурного описания (компоненты-экземпляры U1 и U2 объектов типа ent_1), и поведенческого описания (процесс). Вообще говоря, в данном случае процесс можно было и не использовать — входные и выходные порты компонентов напрямую соединить с выходными и входными портами модели в целом, это не будет ошибкой. Однако особенности использования вычисляемых функций в OrCAD таковы, что в этом случае они не будут подключены. Поэтому для тех выходных портов объекта, для которых сигнал поступает от двух и более источников, должно использоваться подключение через сигналы, а не напрямую. Структурная схема объекта model2 приведена на рис. 4.25.

Как видно из рис. 4.25, выходы o и oo2 объектов ent_1 объединены на общие шины, выходы модели o1 и o2 имеют множество источников — компонентов.

Рассмотрим следующий пример поведения этого объекта, временная диаграмма которого приведена на рис. 4.26.

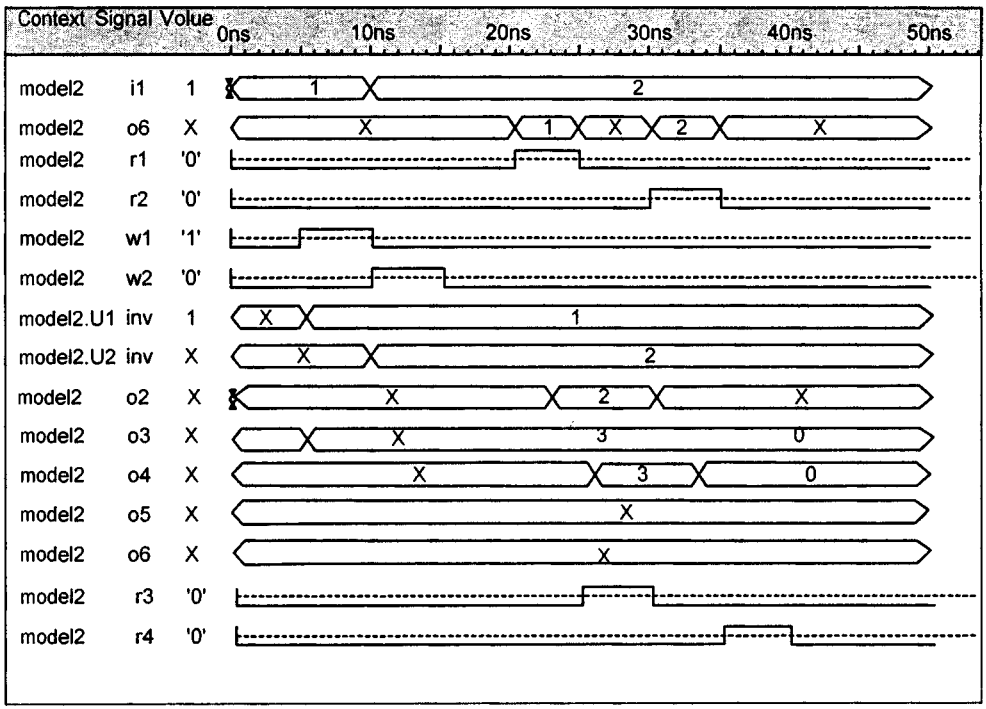


Рис. 4.26. Временные диаграммы объекта model2

С 0 нс. до 10 нс. на вход i1 подается значение 1. В момент времени 5 нс. w1 изменяет значение с 0 на 1, в результате чего значение с входа i1 записывается во внутренний сигнал inv блока U1. Аналогично производится запись во внутренний сигнал U2. Сигнал r1 имеет значение 1 в период времени с 20 нс. до 25 нс. В это время на выходы o1 и o2 должно передаваться значение внутреннего сигнала inv блока U1. На выходе o1 действительно появляется это значение, в то время как выход o2 остается в неопределенном состоянии. Это связано с декларацией типов для этих сигналов. На выход o1 передаются сигналы типа std_logic_vector. На выход o2 передаются сигналы типа std_ulogic_vector. Таким образом, решающая функция по умолчанию для сигналов, источники которых находятся в различных компонентах, используется для типа std_logic.

Теперь рассмотрим выходы o3 и o5, а также парные к ним выходы o4 и o6. Поведение сигналов на выходах o3 и o5 описано одинаково, но они имеют различный тип и, как видно из рис. 4.26, ведут себя по-разному. Сигнал на выходе o3 имеет тип std_ulogic, а на выходе o5 — std_logic. Сигнал на выходе o3 ведет себя в точности так, как того можно было ожидать по его описанию: когда сигнал на входе r1 устанавливается в 1, на o3 появляется значение 3, а когда на входе r3 появляется 1, на выходе o3 появляется значение 0. На выходе o5 такого не наблюдается. Сигнал все время находится в неопределенном состоянии.

Ничего не изменится, даже если в описании поведения ent_1 добавить секции **else**, в которых можно переводить эти сигналы, имеющие тип std_logic, в высокоимпедансное состояние. Это связано с тем, что значения сигналов oo01 и oo02 имеют множество источников на уровне процессов, а для этого уровня вычисляемые функции по умолчанию используются для типа std_ulogic.

Для того чтобы для выхода o5 получить такой же результат, как и для o3, необходимо, чтобы все изменения сигнала, передающегося на этот выход в рамках объекта ent_1, выполнялись *в одном и только одном* процессе (в описании архитектуры). Изменим описание его поведения (листинг 4.36).

Листинг 4.36

```
architecture rtl of ent_1 is
signal inv: std_logic_vector(1 downto 0);
begin
  process (w)
  begin
       if w = '1' and w'event then  inv< = i;  end if;
  end process;
```

```
  process (r, rr)
  begin
        if r = '1' and r'event then  o< = inv; oo2< = inv;
              oo01< = "11"; oo02< = "11";
        else o< = "ZZ"; oo2< = "ZZ"; -- было: oo01< = "ZZ";
oo02< = "ZZ";
        end if;
        if rr = '1' and rr'event then
         -- было: oo01< = "00";
         oo02< = "00";
         -- было:  else  oo01< = "ZZ";  oo02< = "ZZ";
        end if;
  end process;

  process(rr)
  begin
       if rr = '1' and rr'event then
       oo01< = "00";
        -- было:  oo02< = "00";
        -- было:  else  oo01< = "ZZ";  oo02< = "ZZ";
       end if;
  end process;
end architecture rtl;
```

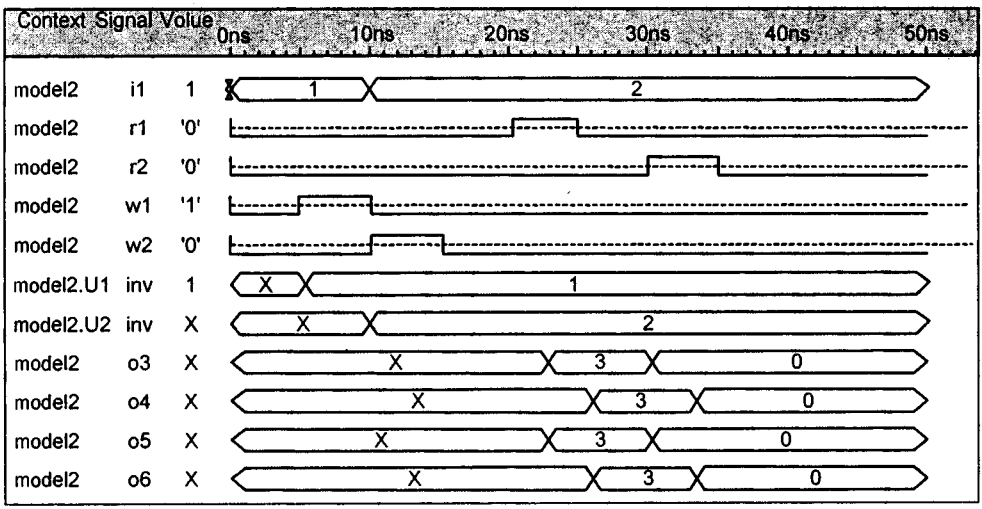


Рис. 4.27. Временные диаграммы объекта `model2` (при описании архитектуры `ent_1` листинга 4.36)

Здесь оператор изменения выхода `oo2` из тела третьего процесса перенесен в тело второго процесса, и список чувствительности второго процесса дополнен сигналом из списка чувствительности третьего процесса.

Результат представлен на рис. 4.27. Как видно из этого рисунка, теперь сигналы на выходах `o3` и `o5` ведут себя совершенно одинаково.

Таким образом, если в модели сигнал может иметь множество источников на уровне процессов, то он должен иметь тип `std_ulogic`, а если сигнал может иметь множество источников на уровне компонентов, то он должен иметь тип `std_logic`. Это связано с особенностями использования решающих функций по умолчанию для этих типов в OrCAD 9.0 и 9.1.

Особенности программирования на VHDL для синтеза

От программирования на VHDL для моделирования к программированию для синтеза

Моделирование и Синтез — взаимодополняющие деятельности в процессе проектирования систем на СБИС. И в Моделировании[1], и при Синтезе используются программы на языке VHDL. В современных технологиях автоматизированного проектирования систем на СБИС, и для Моделирования, и для Синтеза мы начинаем со спецификации структуры и поведения цифровой системы. Описание объекта на VHDL может быть использовано как для моделирования его поведения, так и для синтеза физической реализации.

В Моделировании программа на VHDL используется как модель, отражающая некоторое устройство (существующее в "железе" или пока только в нашем воображении). Система моделирования "исполняет" программу на VHDL, имитируя работу реального устройства. Поведение устройства представляется в модели через события изменения сигналов и формируемые временные диаграммы сигналов. Наблюдая это поведение, разработчик анализирует свой проект, делает выводы о его правильности и эффективности. Здесь программа на VHDL рассматривается как модель (simulation model). Чтобы избежать тавтологии в терминах ("модель для моделирования"), используем термин *VHDL программа-модель.*

Синтез является как бы обратным процессом. При Синтезе программа на VHDL рассматривается как спецификация, детальное описание, исходные данные и требования, по которым должна быть сгенерирована реализация

---

[1] Рассматривая Моделирование и Синтез как имена собственные соответствующих видов деятельности, будем писать их в тексте с заглавной буквы.

физического устройства, системы на СБИС. Синтезирующие программы системы автоматизации проектирования (иначе называемые *синтезирующими компиляторами*) по программе на VHDL генерируют реализацию проектируемого устройства. Здесь программа на VHDL выступает в другом качестве, как *программа-спецификация* для Синтеза (synthesis model).

Эта разница во взглядах на программу при Моделировании и при Синтезе определяет двойственность программирования на VHDL — программирование для Моделирования и программирование для Синтеза.

Может ли одна и та же программа на VHDL быть одновременно программой-моделью для Моделирования и программой-спецификацией для Синтеза? В общем случае, ответ будет "Нет". Не всякая программа-модель может быть использована как программа-спецификация для Синтеза. Не для всякой программы на VHDL ее поведение при Моделировании будет соответствовать поведению устройства, синтезированного по этой же программе.

Имея в виду невозможность адекватной отработки тех или иных программных конструкций языка VHDL при Синтезе, часто используют понятие "синтезируемого подмножества" языка VHDL. Имеется в виду та часть конструкций и понятий языка VHDL, которые могут использоваться в программе-спецификации для Синтеза, исключая запрещенные при Синтезе конструкции. Однако такого стандартизованного подмножества языка VHDL, как "синтезируемое подмножество VHDL", не существует. Разные синтезирующие компиляторы принимают, запрещают или игнорируют разные программные конструкции языка VHDL. Проблемы расхождения программ-моделей для Моделирования и программ-спецификаций для Синтеза не исчерпываются сообщениями о синтаксических ошибках, выдаваемых синтезирующим компилятором по программе, успешно исполняемой как программа-модель. Одни и те же допустимые и при моделировании, и при синтезе программные конструкции могут давать разные результаты. Стиль программирования, разница применения тех или иных программных конструкций, несущественная при Моделировании, может оказывать значительное влияние на эффективность результатов синтеза устройства по программе-спецификации на языке VHDL.

Поэтому, переходя от программирования на VHDL для Моделирования к программированию для Синтеза, мы должны заново познакомиться с языком, посмотреть на него с других позиций. Использование программы на VHDL как программы-спецификации для Синтеза меняет взгляд на объекты, сигналы, программные конструкции языка VHDL.

При переходе от составления программ-моделей к программам-спецификациям для Синтеза перед нами встанут две проблемы.

- *Проблема корректности.* Возможное изменение смысла, изменение толкования программных конструкций VHDL при их использовании для Синтеза, может вести к изменению функционирования синтезированной

схемы по сравнению с поведением ее модели по той же программе на VHDL. Возможно другое, неверное функционирование синтезированной реализации устройства по сравнению с его отлаженной моделью.

- *Проблема эффективности.* Программные конструкции VHDL и стиль программирования, дающие одинаковый результат при Моделировании, могут иметь существенные отличия по эффективности реализации при Синтезе (быстродействие, объем схемы).

Так же, как для формирования представления о языке VHDL, для понимания семантики конструкций языка VHDL при Моделировании пользуются понятиями дискретного событийного моделирования, так для осмысления конструкций языка VHDL как языка программ-спецификаций для Синтеза, используют понятие *процедуры вывода схемы* (inference). Синтезирующие компиляторы с языка VHDL "выводят" структуру реализации проектируемого устройства — состав базовых элементов и связи между ними, основываясь на программе-спецификации проектируемого устройства на языке VHDL. Язык VHDL — язык "программирования" аппаратуры, аппаратно реализуемых алгоритмов, параллельных по своей природе конструкций. Программируя на VHDL, мы должны "видеть" за программными конструкциями языка параллельно работающие компоненты, множество параллельно существующих и меняющихся сигналов в схеме.

Синтез оперирует с тремя видами информации:

- VHDL программа-спецификация проектируемого устройства;
- базис реализации — набор элементов, которые могут использоваться для конструирования реализации;
- набор ограничений для синтезируемой реализации устройства.

Можно провести аналогию Синтеза с компиляцией с традиционного языка программирования высокого уровня. Компилятор, синтаксически разобрав текст программы на исходном языке, транслирует программу на язык машинных кодов объектной ЭВМ, генерирует объектную программу из предопределенных элементов — машинных команд процессора, на котором программа будет исполняться [25].

Так и Синтез устройства — иными словами, компиляция устройства по VHDL программе-спецификации, может использовать разные наборы элементов ("элементный базис" реализации [27]) для построения синтезируемых устройств. Базис реализации может состоять из элементов разного уровня — от вентилей, от элементарных логических элементов классических базисов комбинационных схем в СДНФ (совершенной дизъюнктивной нормальной форме), от логических блоков табличного типа (LUT) до макроячеек сложных функциональных блоков — мультиплексоров, тактовых генераторов, регистровых блоков, готовых блоков памяти, АЛУ, умножителей, вплоть до "процессорных ядер" (например, реализованных на уровне

топологии СБИС аппаратных блоков PPC405 — процессоров с архитектурой PowerPC в СБИС Virtex-II Pro, [18]).

Как и в компиляции программ с традиционных языков программирования, вслед за "выводом" откомпилированной схемы реализации устройства производится оптимизация схемы с целью увеличения быстродействия или сокращения аппаратных затрат, уменьшения размеров на кристалле. Оптимизация схемы приводит к ее трансформациям, иногда весьма существенно меняющим структуру, которая просматривается в исходном тексте программы на VHDL. Результатом может быть расхождение в поведении модели устройства и поведении реализации устройства, синтезированного по той же программе на языке VHDL. Грамотный разработчик должен учитывать возможность таких трансформаций, когда пишет на VHDL программу-спецификацию для Синтеза проектируемого устройства.

Стратегия оптимизации схемы различна в разных синтезирующих программах. Однако имеются и общие подходы, характерные для основных распространенных пакетов автоматизации проектирования. Обычно в стратегиях оптимизации полагается, что комбинационные схемы предпочтительнее схем последовательностных (схем с памятью). Считается, что они дают меньшие задержки и меньшие затраты на реализацию. Поэтому синтезирующие компиляторы, как правило, стремятся при генерации фрагментов схемы, соответствующих определенным конструкциям программы VHDL, свести их к комбинационным схемам.

Те или иные программные конструкции языка VHDL синтезирующий компилятор может принимать или игнорировать. Когда мы говорим "игнорируется", это означает, что данная конструкция, ее секция или атрибут не будут влиять на результат синтеза схемы устройства по VHDL программе. Практическая реакция конкретных синтезирующих программ, наблюдаемая пользователем, может быть разной. Некоторые конструкции в исходном тексте на VHDL программа синтеза может просто игнорировать, другие будут вызывать сообщения о синтаксических ошибках (с точки зрения Синтеза). Эти аспекты работы синтезирующих программ не стандартизированы и определяются конкретным пакетом автоматизации проектирования.

Далее в этом разделе мы рассмотрим некоторые общие вопросы перехода к программированию на VHDL для Синтеза, не привязанные, в основном, к какому-либо инструменту и базису реализации.

Сигналы, переменные, константы

При описании модели на VHDL могут быть использованы сигналы, переменные, константы.

В ходе синтеза сигналы отображаются в линии связей, защелки, триггеры. Если описывается комбинационная схема, сигналы отображаются в линии

связи, по которым от одних элементов к другим передаются значения этих сигналов. Если описывается конечный автомат, то для хранения значений сигналов используются триггеры и защелки. Отображение сигнала в линию связи, триггер или защелку в этом случае зависит от того, как он используется в модели.

Разрядность сигналов

Разрядность линии связи, триггера или защелки зависит от типа сигнала. Количество битов, необходимое для реализации, может быть определено в явном виде, например, при указании размера вектора.

```
Signal data1: std_logic_vector (7 downto 0)
```

Для реализации сигнала, описанного таким образом, требуется 8 разрядов, 8 линий связи.

Для каждого предопределенного типа в языке VHDL определена разрядность, которая используется по умолчанию, если не задана явная спецификация разрядности декларируемого сигнала.

```
Signal count: integer;
```

В данном случае определение разрядности выполнено неявно (тип **integer** для своей реализации требует 32 бита).

Однако нередко уже на этапе проектирования известно, что сигнал может принимать значения только в определенном диапазоне (например, от 0 до 140). В этом случае, при описании сигнала целесообразно указать этот диапазон в явном виде:

```
Signal count: integer range 0 to 140;
```

Для реализации сигнала, описанного таким образом, будет использовано только 8 бит, что может существенно уменьшить размер схемы.

Рассмотрим, например, реализацию сумматора (листинг 4.37).

Листинг 4.37

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_adder is
port(data1, data2:in integer;
     datao:out integer);
end entity my_adder;
architecture adder_rtl of my_adder is
begin
       process (data1,data2)
```

```
    begin
        datao< = data1 + data2;
    end process;
end architecture adder_rtl;
```

Этот сумматор имеет два входа и один выход, все типа **integer** (разрядность 32). Рассмотрим характеристики его реализации на FPGA V400BG432. При синтезе по приведенной программе для его реализации было использовано 16 слоев, 32 четырехвходовых LUT, что соответствует 381 вентилю. При этом максимальная задержка передачи сигнала от входа к выходу составляет 19,760 нс., а максимальная задержка в линиях связи — 5,562 нс.

Если же известно, например, что слагаемые и результат лежат в диапазоне от 0 до 16, то их тип может быть описан как **integer range** 0 **to** 16 (разрядность линий связи 5 бит). В этом случае и комбинационная схема, выполняющая функции сумматора, будет существенно меньше. При синтезе для ее реализации потребовалось 5 слоев металлизации, 8 четырехвходовых LUT или 48 эквивалентных вентилей. Это почти в 8 раз меньше, чем в предыдущем случае. Максимальная задержка передачи сигнала от входа к выходу составляет 12,391 нс., а максимальная задержка в линиях связи — 1,747 нс., что также значительно меньше, чем в предыдущем случае.

Некоторые инструменты синтеза способны при анализе кода программы на VHDL определить диапазоны возможных значений для сигналов и использовать эту информацию для оптимизации их разрядности. Например, если синтезирующий компилятор VHDL обнаруживает, что описанный сумматор `my_adder` используется для сложения значений четырехразрядных сигналов, то при проведении оптимизации он может сам сократить разрядность реализации данного экземпляра сумматора с 32 до 5 разрядов. Однако синтезирующие программы вынуждены быть достаточно консервативными в своих стратегиях оптимизации, стараясь не нарушить логику функционирования проектируемого устройства, заданную программой-спецификацией на VHDL. Поэтому использование в программах явных спецификаций диапазонов значений декларируемых данных и сигналов является хорошим стилем, часто помогающим синтезирующим программам оптимизировать генерируемую реализацию устройства.

Перечислимые типы и кодировка значений

Рассмотрим, как при синтезе обрабатываются перечислимые типы, определенные пользователем на базе явно указанного множества возможных значений. Пусть, например, тип имеет следующее определение:

```
Type my_state is (Idle, Work1, Work2, Error);
```

В этом случае инструментарий синтеза определяет количество возможных значений. Каждому из них присваивается порядковый номер, на базе кото-

рого формируется двоичная кодировка, разрядность определяется количеством возможных значений. По данному примеру обычно будет синтезирована следующая кодировка:

```
Idle  - '00';
Work1 - '01';
Work2 - '10';
Error - '11'.
```

Многие инструменты синтеза предоставляют пользователю возможность влиять на кодировку значений. Эта возможность реализуется в виде настроек инструментария. Например, количество значений может определять разрядность при позиционном кодировании значений перечислимого типа. Тогда возможна следующая кодировка:

```
Idle -  '0001';
Work1 - '0010';
Work2 - '0100';
Error - '1000'.
```

Кодировки, выбираемые синтезирующей программой, зависят от схемотехнического базиса реализации проектируемого устройства. В ряде ситуаций, например, при синтезе реализаций конечных автоматов, выбранные кодировки состояний автомата могут оказывать существенное влияние на качество синтезированной схемы. Существует целый ряд вариантов кодировок, предназначенных для минимизации аппаратных затрат, увеличения быстродействия, обеспечения помехоустойчивости.

Использование перечислимых типов вместо цифровых кодировок соответствующих программных понятий является хорошим стилем в современном программировании. В отношении программирования на VHDL это справедливо еще в большей степени, поскольку не только способствует лучшему пониманию программы и сокращению числа ошибок программиста, но и облегчает синтезирующим компиляторам с VHDL оптимизацию генерируемой реализации устройства.

Поэтому эффективный, с точки зрения Синтеза, стиль программирования на VHDL диктует широкое использование в программе перечислимых типов, оставляя их числовые кодировки синтезирующему компилятору, не вмешиваясь в кодировки значений без особой необходимости. И, разумеется, не следует использовать в операторах числовые коды для значений перечислимых типов данных, даже когда мы их знаем. Так, в программе следует писать:

```
if sig1 = Idle then ...
```

а не

```
if sig1 = '00' then ...
```

Начальные значения

Синтаксис описания сигналов в VHDL позволяет задавать начальные значения декларируемых сигналов. Этой возможностью часто пользуются при написании VHDL программы для Моделирования. Однако для Синтеза такой конструкцией пользоваться не стоит. Большинство программ синтеза *будут игнорировать секцию задания начального значения в декларации сигнала.*

Если какие-либо сигналы в проектируемом устройстве должны устанавливаться в некоторые начальные значения, то в VHDL программе-спецификации рекомендуется явно программировать установку сигналов в начальные состояния по сигналам начального сброса устройства (сигналам типа `reset`).

Для уменьшения расхождений между VHDL программой-моделью и VHDL программой-спецификацией такого стиля надо придерживаться и при программировании для Моделирования. Однако оборотной стороной такого стиля при Моделировании будет увеличение времени моделирования.

При описании констант в декларации должны быть явно заданы их значения, они не будут игнорироваться при Синтезе.

Переменные

Сигналы в языке VHDL предназначены для представления состояний элементов аппаратуры. При Синтезе реализации проектируемого устройства по VHDL программе-спецификации сигналам сопоставляются линии связи, защелки, триггеры. В отличие от сигналов, переменные в VHDL рассматриваются как вспомогательные программные объекты, используемые, в основном, для программирования вычисления значений сигналов. Во многих случаях переменным не сопоставляются какие-либо элементы генерируемой при Синтезе реализации проектируемого устройства. Так, переменная `var` в программе (см. листинг 4.7) носит вспомогательный характер и при Синтезе не найдет прямого отражения в каких-либо элементах сгенерированной схемы.

Однако существуют ситуации, когда при Синтезе переменная отражается в элементах памяти — триггерах, триггерах-защелках, в генерируемой схеме устройства. Учитывая сказанное выше о значительной "цене" последовательностных элементов в реализации, возможности таких ситуаций полезно иметь в виду при программировании на VHDL.

Такая ситуация может складываться при использовании переменных в процессах. В соответствии с семантикой VHDL, переменная сохраняет свое значение между двумя срабатываниями процесса. В нашем примере (см. листинг 4.7) это свойство переменной не используется: при каждом срабатывании процесса `pr_d_e1` переменная `var` получает новое значение, прежде чем значение `var` будет использовано в выражениях в левых частях

операторов присваивания. Но если взять пример листинга 4.38, то здесь, при каждом срабатывании процесса, значение переменной acc используется в выражении прежде, чем переменной будет присвоено значение. Таким образом, переменная acc действительно должна хранить значение между срабатываниями процесса, и при Синтезе по данной программе, переменной acc будет сопоставлен триггер.

Листинг 4.38

```
library ieee;
use ieee.std_logic_1164.all
entity accumulator is
port (clk,x,y: in std_logic;
       z: out std_logic);
end entity accumulator;
architecture accum_behav of accumulator is
begin
    process
    variable acc: std_logic;
    wait until(rising_edge(clk));
          acc: = acc or x and y;
          z< = acc;
    end process;
end architecture accum_behav;
```

Данный пример иллюстрирует общее правило, реализуемое синтезирующими компиляторами с языка VHDL. Если значение переменной в теле процесса используется прежде, чем присваивается в текущем срабатывании процесса, то такой переменной при синтезе сопоставляется элемент памяти — триггер, триггер-защелка, регистр, и др. Если это входит в наши намерения — прекрасно; если нет, — стоит внимательно посмотреть, как модифицировать программу.

Операторы присваивания

Задержки в операторах присваивания игнорируются!

Обработка задержек, специфицированных в операторах присваивания — это одна из неожиданностей, которая нас поджидает, когда в работе с языком VHDL мы переходим от Моделирования к Синтезу. То, что мы так старательно программировали на VHDL в программе-модели, что детально про-

сматривали и анализировали при Моделировании, при Синтезе, оказывается, не имеет никакого значения и просто игнорируется!

Это можно понимать так. При Моделировании нельзя описать функционирование моделируемого устройства во времени, не задав задержки при формировании значений одних сигналов на основе значений других. Задержки, указанные в операторах присваивания значений сигналам, отражают наши представления о временных характеристиках работы элементов схем физических устройств. Можно сказать, что задержки, указываемые в операторах присваивания, являются как бы вынужденными. В семантике языка VHDL предполагается, что они указываются в операторах присваивания не потому, что разработчик хочет задержать тот или иной сигнал, а потому, что реальные элементы схемы работают не мгновенно, а вносят задержку, которую мы и стараемся аккуратно отразить в модели устройства.

При Синтезе эти вынужденные задержки не надо описывать, они будут автоматически определяться используемыми базовыми элементами схемы, — физически существующими и работающими с конечными задержками, диктуемыми схемотехническим базисом реализации. В реальной, синтезированной схеме эти задержки изменения сигналов будут определяться не тем, что написано в операторе присваиванияв программе на VHDL, а схемотехническим базисом реализации и физическим размещением схемы на кристалле. Выполняемое после синтеза моделирование, которое идет уже не по VHDL-модели, а по структуре синтезированной схемы устройства, покажет фактические задержки аппаратных элементов схемы.

Соответственно, трактуемые таким образом *указания задержек в операторах присваивания значений сигналам игнорируются при Синтезе.*

В операторах присваивания значения сигналу, кроме секций указания значений задержек **after**, может использоваться также атрибут **transport**, указывающий на использование в этой конструкции транспортных, а не инерционных задержек. При Синтезе этот атрибут в операторе присваивания игнорируется. Аналогично игнорируются атрибуты **inertial** и **reject**. В Foundation Express (по крайней мере, вплоть до версии 3.1) игнорируется также атрибут **unaffected**.

Оптимизация выражений в операторах присваивания

Одно и то же выражение может иметь несколько алгебраически эквивалентных форм записи.

Например, `a and b and c and d = (a and b) and (c and d)`

или `(a and b) or c = a or c and b or c`.

Большинство ранее разработанных инструментов синтеза были чувствительны к форме написания арифметических и логических выражений в операторах присваивания. В частности, выражение `a and b and c and d` синтезирова-

лось как ((a **and** b) **and** c) **and** d. В результате длина линий связи получалась больше, чем при синтезе эквивалентного ему выражения (a **and** b) **and** (c **and** b).

Инструментарий синтеза в Foundation Express снабжен средствами оптимизации выражений, используемых в операторах присваивания. Выражения можно записывать в любой форме, это не влияет на синтезируемую схему.

Тем не менее, для улучшения переносимости программ на VHDL в разные пакеты автоматизации проектирования и для уменьшения зависимости результатов синтеза от используемого синтезирующего компилятора сохраняется старая рекомендация: использовать в выражениях скобки для явного указания последовательности вычисления выражения и, таким образом, управлять параллельностью его воплощения в синтезируемой схеме.

Использование переменных и констант в операторах присваивания

Результаты использования переменных и констант при синтезе могут отличаться от результатов, получаемых при моделировании. Это происходит в случаях, когда внутри тела процесса расположен оператор присваивания, в правой части которого присутствует объект, значение которого определяется ранее в теле этого же процесса. Например:

```
S1< = a and b;
S2< = S1 or c
```

Значение объекта S2 определяется на базе значения S1. При моделировании, если используются сигналы, то S2 получит значение S1 от предыдущего выполнения процесса, а если переменные — то сразу. Если в описании сигналы были использованы для того, чтобы синтезированная схема вела себя в соответствии с поведенческой моделью, потребовалось бы использовать для сохранения предыдущего значения сигнала S1 защелку. Этого, однако, не делается. В результате в обоих случаях получается совершенно одинаковая схема.

Синтез управляющих конструкций, содержащих операторы присваивания

Общие подходы к синтезу комбинацио нных и последовательностных схем по программе на VHDL

Один из ключевых вопросов при Синтезе в отображении программы на VHDL в схему реализации проектируемого устройства — это выбор между комбинационными схемами и последовательностными. Различные программные пакеты синтеза, *синтезирующие компиляторы* языка VHDL, ис-

пользуют некоторые общие принципы трансляции программных конструкций языка в синтезируемые схемы реализации проектируемого устройства. Общий подход к выбору между комбинационными и последовательностными схемами базируется на сопоставлении свойств сигналов в программе на VHDL с естественными свойствами этих классов схем.

В любой комбинационной схеме (без обратных связей) в каждый момент физического времени формируется значение для каждого выхода схемы. Выходные сигналы определены для любой комбинации значений входных сигналов. Схема, после формирования выходного сигнала в какой-то момент времени, в следующий момент формирует новое значение на основании входных сигналов. Значения выходных сигналов в каждый момент времени не зависят от значений этих сигналов в предыдущие моменты. Комбинационная схема не должна хранить и не хранит значения своих внутренних и выходных сигналов.

Природным свойством последовательностных схем является зависимость формируемых значений выходных сигналов как от значений входных сигналов, так и от значений, сформированных в предыдущие моменты времени. Для хранения этих, сформированных ранее, значений последовательностные схемы содержат элементы памяти разных видов.

Программа на VHDL описывает, как и когда сигналы получают свои значения. Опираясь на качественные свойства комбинационных и последовательностных схем, синтезирующий компилятор анализирует исходную программу на VHDL. Если компилятор определяет, что в архитектурном теле всем сигналам, помеченным как выходные, присваиваются значения при всех возможных комбинациях входных сигналов, то из этого "выводится" комбинационная схема. Если в программе выходным сигналам присваиваются значения не при всех возможных комбинациях значений входных сигналов, то подразумевается, что такие сигналы сохраняют свои прежние значения. Для сохранения значений приходится включать элементы памяти, и тогда "выводится" последовательностная схема. Последовательностная схема выводится также и в случае, если для формирования выходного сигнала используются не только текущие значения входных сигналов, но и значения сигналов и переменных, сформированные в предыдущие моменты времени (напомним, что при Синтезе задержки в операторах присваивания сигналам игнорируются).

Это соответствует тому, что происходит в реальной аппаратуре. Внутренний сигнал комбинационной схемы (сигнал, не являющийся для нее ни входным, ни выходным) получает новое значение (с выхода некоторого логического элемента) *прежде*, чем он может быть использован (как вход другого элемента схемы) для вычисления значения другого сигнала. Таким образом, внутренние сигналы схемы всегда получают новые значения, прежде чем

эти значения используются для формирования значений каких-либо других сигналов. Соответственно, нет необходимости специально хранить их в элементах памяти.

Условные операторы

Синтез конструкций **if then elsif else** и **case** осуществляется примерно по одинаковой схеме. Для каждой ветви генерируется своя комбинационная схема. Выходы схем объединяются через мультиплексор, управление которым осуществляется на базе сигналов, входящих в условное выражение оператора **if** или **case**. При очередном выполнении процесса выполняется только одна ветвь оператора **if** или **case**.

С точки зрения Синтеза имеет существенное значение, каким сигналам присваиваются значения в альтернативных ветвях условных конструкций. Во всех ветвях могут определяться значения для одного и того же набора сигналов или же наборы сигналов в различных ветвях могут быть различными. В последнем случае предполагается, что сигналы, которым не присваиваются новые значения в ходе выполнения конкретной ветви, сохраняют свои прежние значения. Если некоторый сигнал при определенных условиях не меняется, то его значение должно быть зафиксировано и держаться на соответствующей линии связи в синтезируемой схеме. Комбинационная схема сделать этого не может, и мы попадаем в класс схем последовательностных, схем с памятью. Поэтому при Синтезе, для каждого сигнала, значение которого переопределяется не во всех ветвях условной конструкции, будет сгенерирован триггер-защелка, фиксирующий и хранящий прежнее значение сигнала, если при текущем срабатывании условного оператора его значение не меняется. Это приводит к дополнительным аппаратным затратам и снижению быстродействия схемы. Поэтому во всех случаях, где такая ситуация допускается логикой работы, во всех ветвях должны переопределяться одинаковые наборы сигналов. В примере, приведенном в листинге 4.39, в обоих ветвях оператора `if` присваиваются значения одному и тому же набору сигналов.

Листинг 4.39

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_if is
port(in1,in2:in std_logic;
```

```
    c1: in std_logic_vector(1 downto 0);
    out1, out2 : out std_logic);
end entity my_if;
architecture rtl of my_if is
begin
      process(in1,in2,c1)
      begin
          if c1 = "00" then
          out1< = in1; out2< = in2;
          else out1< = in2; out2< = in1;
          end if;
      end process;
end architecture rtl;
```

Сравним характеристики схемы, синтезированной на базе листинга 4.39, и схемы, в которой тело процесса имеет вид, представленный в листинге 4.40.

Листинг 4.40

```
if c1 = "00" then
     out1< = in1;
   else out1< = in2; out2< = in1;
end if;
```

В первом случае в обеих секциях оператора **if** определяются значения для одинакового набора сигналов `out1` и `out2` (разные значения, но для одного и того же набора сигналов). Во втором случае (листинг 4.40) в секции **then** определяется только значение сигнала `out1`; предполагается, что `out2` должен сохранить свое прежнее значение. Поэтому во второй схеме, в отличие от первой, для фиксации значения сигнала `out2` генерируется триггер-защелка.

Реализация варианта схемы по листингу 4.39 занимает один слой, для нее необходимо два четырехвходовых LUT (12 эквивалентных вентилей). Максимальная задержка передачи сигнала от входа к выходу составляет 9,345 нс., максимальная задержка сигнала в линии связи составляет 0,888 нс.

Реализация второго варианта схемы (листинг 4.40) занимает два слоя, для нее необходимо два четырехвходовых LUT и один триггер-защелка (17 эквивалентных вентилей). Максимальная задержка передачи сигнала от входа к выходу составляет 10,679 нс., максимальная задержка сигнала в линии связи составляет 1,149 нс. Эта схема содержит элемент памяти, поэтому для

нее определяется минимальный период изменения сигнала — 3,325 нс., что соответствует максимальной частоте 300,752 МГц.

Как можно видеть из этого простого примера, даже при появлении одного сигнала, значение которого определяется не во всех ветвях, увеличиваются аппаратные затраты и несколько ухудшаются временные характеристики.

В некоторых случаях этого, однако, можно избежать. Если в данном примере режим сигнала `out2` изменить с `out` на `inout` (или `buffer`), то в ветви **then** ему можно будет присваивать его собственное значение. Такое описание приведено в листинге 4.41.

Листинг 4.41

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_if2 is
port(in1,in2:in std_logic;
     c1: in std_logic_vector(1 downto 0);
     out1 : out std_logic;
     out2: inout std_logic);
end entity my_if2;
architecture rtl of my_if2 is
begin
   process(in1,in2,c1)
   begin
   if c1 = "00" then
         out1< = in1; out2< = out2;
      else out1< = in2; out2< = in1;
   end if;
   end process;
end architecture rtl;
```

Схема, описанная таким образом, будет иметь поведение, аналогичное поведению схемы листинга 4.40, а аппаратные затраты на ее реализацию и временные характеристики будут такие же, как для схемы листинга 4.39.

Оператор **case** может содержать только одно условное выражение, в то время как в операторе **if** каждая ветвь **elsif** может содержать свое условное выражение. В соответствии с этим, при синтезе для оператора **case** используется только один мультиплексор, в то время как для каждой секции **elsif**

используется отдельный мультиплексор (как и для вложенных операторов **if**). Это приводит к увеличению аппаратных затрат и ухудшению временных характеристик схем. Поэтому во всех случаях, где может быть использован оператор **case**, предпочтительнее использовать его.

Сравним характеристики схем, синтезированных на базе листингов 4.42 и 4.43. Эти схемы выполняют одну и ту же функцию, но в первом случае для ее описания использован оператор **case**, а во втором — набор операторов **if**.

Листинг 4.42

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_case is
port(in1,in2,in3,in4:in std_logic;
     c1: in std_logic_vector(1 downto 0);
     out1 : out std_logic   );
end entity my_case;
architecture rtl of my_case is
begin
      process(in1,in2,in3,in4,c1)
      begin
           case c1 is
              when "00" = > out1< = in1;
              when "01" = > out1< = in2;
              when "10" = > out1< = in3;
              when "11" = > out1< = in4;
              when others = > out1< = in1;
           end case;
      end process;
end architecture rtl;
```

Листинг 4.43

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_case1 is
```

```
port(in1,in2,in3,in4:in std_logic;
     c1: in std_logic_vector(1 downto 0);
     out1 : out std_logic );
end entity my_case1;
architecture rtl of my_case1 is
begin
       process(in1,in2,in3,in4,c1)
       begin
         if c1 = "00" then out1< = in1;
           else if c1 = "01" then out1< = in2;
                 else if c1 = "10" then out1< = in3;
                       else out1< = in4;
         end if; end if; end if;
      end process;
end architecture rtl;
```

Для реализации VHDL программы листинга 4.42 используется один слой и два четырехвходовых LUT, или 15 эквивалентных вентилей. Максимальная задержка передачи сигнала от входа к выходу составляет 9,694 нс., максимальная задержка сигнала в линии связи составляет 1,048 нс.

Для реализации описания листинга 4.43 используется один слой и три четырехвходовых LUT или 18 эквивалентных вентилей. Максимальная задержка передачи сигнала от входа к выходу составляет 10,296 нс., максимальная задержка сигнала в линии связи составляет 1,084 нс.

Несколько изменив стиль программирования, можно улучшить характеристики и при использовании условного оператора **if**. В данной программе можно использовать не вложенные, а последовательные операторы **if** (листинг 4.44).

Листинг 4.44

```
if c1 = "00" then out1< = in1; end if;
if c1 = "01" then out1< = in2; end if;
if c1 = "10" then out1< = in3; end if;
if c1 = "11" then out1< = in4; end if;
```

В этом случае аппаратные затраты будут такие же, как и для листинга 4.43, а временные характеристики чуть хуже.

Схема, выполняющая ту же функцию, может быть описана и с использованием параллельного селективного оператора присваивания. Такое описание приведено в листинге 4.45.

Листинг 4.45

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_sel is
port(in1,in2,in3,in4:in std_logic;
     c1: in std_logic_vector(1 downto 0);
     out1 : out std_logic    );
end entity my_sel;
architecture rtl of my_sel is
begin
with c1 select
out1 < = in1 when "00",
        in2 when "01",
        in3 when "10",
        in4 when "11",
        in1 when others;
end architecture rtl;
```

В данном примере аппаратные затраты на реализацию такого описания и временные характеристики полностью совпадают с характеристиками схемы листинга 4.42, в которой был использован оператор **case**.

При использовании оператора селективного присваивания следует помнить, что его семантика фиксирует последовательный порядок проверки условий, перечисленных в его ветвях, причем первое же выполнение условия прекращает проверку остальных. При Синтезе это проявится как генерация реализации с включением приоритетной схемы объединения комбинационных схем, соответствующих каждой из ветвей. Приоритетная схема может увеличивать аппаратные затраты и задержки схемы.

При оптимизации синтезирующие программы стараются распознать ситуации, когда все условия, указанные в ветвях оператора селективного присваивания, являются взаимоисключающими. Если это удается обнаружить, приоритетная схема убирается, что мы и наблюдаем по результатам синтеза (фрагмент программы листинга 4.45). В других же случаях, замена **case** на **with...select** может быть не столь безобидна.

Работа с типами сигналов из стандартных пакетов и библиотек (на примере std_logic)

Целый ряд перечислимых типов задается в стандартных библиотеках, широко применяемых при программировании на VHDL как для Моделирования,

так и для Синтеза. Эти типы хорошо продуманы, отражают важные свойства моделируемых и синтезируемых схем. Их применение в программах на VHDL способствует лучшей мобильности программ и расширяет возможности применения в проектах широкого спектра библиотек компонентов.

Однако при работе с ними нельзя забывать, что эти типы являются перечислимыми, надо помнить соответствующий им набор значений. Тип `std_logic` из пакета `std_logic_1164` является распространенным примером такого рода типа, рекомендуемым для моделирования логических сигналов *(см. приложение 1)*. Этому типу соответствуют 9 возможных значений. При работе с сигналами типа `std_logic` важно помнить об этом, не путать их с логическими сигналами двоичного типа. В противном случае при Моделировании мы рискуем получить неверное поведение моделируемого устройства, а при Синтезе — генерацию неожиданно громоздкой и медленной реализации устройства. Возьмем простой пример (листинг 4.46).

Листинг 4.46

```
library ieee;
use ieee.std_logic_1164.all
entity alu is
port (code,x,y: in std_logic;
      z: out std_logic);
end entity alu;
architecture alu_behav of alu is
begin
process
      if (code = '0') then
           z< = x or y;
      end if;
      if (code = '1') then
           z< = x and y;
      end if;
end process;
end architecture alu_behav;
```

Здесь проверяется состояние входного сигнала `code` (сравнение его со значениями '0' и '1'). В обеих ветвях присваивается значение одному и тому же сигналу — выходному сигналу `z`. Казалось бы, при Синтезе результатом будет чисто комбинационная схема, компактная и быстрая. Однако вспомним, что тип `std_logic` имеет девять значений, из которых мы проверили только два — '0' и '1'. При Моделировании по такой программе, если сигнал

не равен ни '0', ни '1', сигнал z не должен изменить своего значения. При Синтезе же это означает, что синтезированная схема должна сохранять текущее значение сигнала z. Тогда на выходе схемы будет поставлен элемент памяти — триггер-защелка, который либо пропускает на выход сформированное значение z (в случае если сигнал code равен 0 или 1), либо хранит предыдущее значение z (если сигнал code имеет какое-нибудь другое значение). Но существуют инструменты, которые при синтезе рассматривают тип std_logic как содержащий только два значения: '1' и '0' (Например, Foundation 2.1 i). В таком случае будет синтезирована чисто комбинационная схема.

Операторы цикла

Большинство существующих в настоящее время инструментов синтеза, в том числе и Foundation Express, поддерживают синтез только тех операторов цикла, для которых количество повторений можно определить на этапе синтеза. Для каждого повторения цикла синтезируется своя комбинационная схема. Синтезирующий компилятор, транслирующий программу на VHDL в аппаратуру, должен определять количество и состав синтезируемой аппаратуры прямо на этапе компиляции-синтеза.

Из операторов цикла такими свойствами однозначно обладает цикл вида **for...loop**. Именно этот вид цикла поддерживается в режиме Синтеза большинством синтезирующих компиляторов языка VHDL.

Например, используя циклическую конструкцию, мы можем компактно описать 8-разрядный сдвиговый регистр (листинг 4.47).

Листинг 4.47

```
library IEEE;
use IEEE.std_logic_1164.all;
entity shift_reg8 is
port (clk, indata: in std_logic;
      outdata: out std_logic_vector(7 downto 0)
end entity shift_reg8;
architecture for_synth of shift_reg8 is
signal register_bits: std_logic_vector(7 downto 0)
begin
process (clk, indata) is
begin
        if (rising_edge(clk)) then
                for i in 7 downto 1 loop
```

```
                register_bits(i) < = register_bits(i-1);
        end loop;
        register_bits(0) < = indata;
        outdata < = register_bits;
        end if;
end process;
end architecture for_synth;
```

В этом примере число итераций цикла известно компилятору, поэтому такая конструкция будет синтезируема. По своему результату она окажется эквивалента развернутой записи соответствующего числа итераций тела цикла (листинг 4.48).

Листинг 4.48

```
library IEEE;
use IEEE.std_logic_1164.all;
entity shift_reg8 is
port (clk, indata: in std_logic;
      outdata: out std_logic_vector(7 downto 0)
end entity shift_reg8;
architecture for_synth2 of shift_reg8 is
signal register_bits: std_logic_vector(7 downto 0)
begin
process (clk, indata) is
begin
      if (rising_edge(clk)) then
          register_bits(7) < = register_bits(6);
          register_bits(6) < = register_bits(5);
          register_bits(5) < = register_bits(4);
          register_bits(4) < = register_bits(3);
          register_bits(3) < = register_bits(2);
          register_bits(2) < = register_bits(1);
          register_bits(1) < = register_bits(0);
          register_bits(0) < = indata;
          outdata < = register_bits;
      end if;
end process;
end architecture for_synth2;
```

Наглядно видно удобство и компактность записи листинга 4.47 по сравнению с листингом 4.48. Легко представить себе разницу, если, допустим, надо описать 64-разрядный регистр.

Кроме того, циклическая конструкция удобна для задания параметризованных описаний устройств. Например, используя механизм настраиваемых параметров **generic**, можно описать сдвиговый регистр с параметризованным числом разрядов (листинг 4.49).

Листинг 4.49

```
library IEEE;
use IEEE.std_logic_1164.all;

entity shift_regN is
generic (N: positive);
port (clk, indata: in std_logic;
      outdata: out std_logic_vector(N-1 downto 0)
end entity shift_regN;

architecture for_synth3 of shift_regN is
signal register_bits: std_logic_vector(N-1 downto 0)
begin
process (clk, indata) is
begin
        if (rising_edge(clk)) then
              for i in N-1 downto 1 loop
                    register_bits(i) < = register_bits(i-1);
              end loop;
              register_bits(0) < = indata;
              outdata < = register_bits;
        end if;
end process;
end architecture for_synth3;
```

Настраиваемые параметры **generic** – это константные значения, известные синтезирующему компилятору в момент компиляции программы-спецификации на VHDL.

Если количество повторений невозможно определить на этапе синтеза, как, например, для листинга 4.50, синтез схемы не выполняется.

Листинг 4.50

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_while is
port(in1,in2:in std_logic;
     c1: in integer range 0 to 15;
     out1: out std_logic   );
end entity my_while;
architecture rtl of my_while is
begin
      process(in1,in2,c1)
      variable vc:integer range 0 to 15;
      begin
        vc: = 0;
        while (vc<c1) loop
            out1< = in2;
            vc: = vc + 1;
        end loop;
        out1< = in1;
      end process;
end architecture rtl;
```

Конечно, с точки зрения программирования, можно написать цикл с числом итераций, определяемым на этапе компиляции, используя и другие виды циклов, например, цикл с условием **while**. Программу (см. листинг 4.47) можно переписать так, как представлено в листинге 4.51.

Листинг 4.51

```
library IEEE;
use IEEE.std_logic_1164.all;
entity shift_reg8 is
port (clk, indata: in std_logic;
      outdata: out std_logic_vector(7 downto 0)
end entity shift_reg8;
architecture for_synth4 of shift_reg8 is
signal register_bits: std_logic_vector(7 downto 0)
```

```
begin
process (clk, indata) is
variable i: natural;
begin
      if (rising_edge(clk)) then
         i: = 7;
         while (i>0) loop
              register_bits(i) < = register_bits(i-1);
              i: = i-1;
         end loop;
         register_bits(0) < = indata;
         outdata < = register_bits;
      end if;
end process;
end architecture for_synth4;
```

Однако не все синтезирующие компиляторы поддерживают конструкцию **while**. При программировании на VHDL для Синтеза более надежным будет использовать циклы вида **for...loop**.

Стоит напомнить, что в проектах систем на VHDL для одного типа объекта, одного **entity**, мы можем иметь множество различных описаний архитектуры. Выбираем то или иное описание архитектуры при задании конфигурации компонентов, включаемых в структуру описываемого устройства. На разных стадиях работы с проектом устройства мы можем выбирать разные архитектурные тела для компонентов его структуры, при Моделировании — одни, при Синтезе — другие.

Процессы и компоненты

Список чувствительности процесса

Список чувствительности процесса имеет большое значение при программировании на VHDL для Моделирования. Он определяет, когда процесс срабатывает в ходе моделирования. Если при описании процесса в список чувствительности включены не все входные сигналы (т. е. сигналы, значения которых используются в выражениях в операторах процесса), то процесс будет вызываться только при изменении тех сигналов, которые явно перечислены в его списке чувствительности. При изменении других сигналов процесс вызваться не будет, и операторы, заданные в его теле, не будут выполняться.

При Синтезе иная ситуация. Когда по программе на VHDL выполняется синтез, для процесса синтезируется фрагмент схемы, использующий все сигналы, указанные в теле процесса. Представим себе, что процессу сопоставляется комбинационная схема. Она по своей природе реагирует на изменения *любых* входных сигналов. Соответственно такому "аппаратному" взгляду на сигналы, синтезирующие компиляторы игнорируют список чувствительности. При синтезе фрагмента схемы, сопоставляемого процессу, компиляторы полагают, что в список чувствительности включены *все* сигналы (если в процессе не использован оператор **wait**).

В результате, преобразования сигналов, заданные процессом, в синтезированной схеме выполняются всегда, при изменениях любых входных сигналов, в то время как в программе-модели на VHDL эти преобразования не выполняются, если не было изменений сигналов из списка чувствительности. Это является одной из типовых причин расхождения результатов Моделирования устройства *до* Синтеза (т. е. моделирования по программе на VHDL) и результатов моделирования устройства *после* Синтеза (т. е. выполняемого не по программе на VHDL, а по синтезированной схеме). В таком случае мы не сможем использовать сравнение результатов моделирования до и после Синтеза для проверки корректности синтезированной схемы проектируемого устройства.

Этой довольно неприятной ситуации можно избежать за счет соответствующего стиля программирования на VHDL.

Если мы хотим иметь программу на VHDL, которая будет использоваться и как программа-модель, и как программа-спецификация для синтеза реализации проектируемого устройства, то список чувствительности процессов стоит рассматривать чисто как средство сокращения времени моделирования (за счет исключения запуска процесса при изменении некоторых сигналов). Не стоит использовать список чувствительности как средство управления логикой работы процесса в зависимости от событий изменения сигналов. Для этого лучше применять оператор **wait**. Часто дают и более радикальную рекомендацию — всегда включать в список чувствительности процесса все его входные сигналы [19].

Опять же, стоит напомнить про возможность наличия в проекте на VHDL множества архитектурных описаний для одного объекта. На этапах предварительного моделирования можно конфигурировать проект на использование архитектурных тел компонентов, которые написаны только для Моделирования и имеют сокращенные списки чувствительности процессов где только можно. При моделировании перед Синтезом, для получения временных диаграмм сигналов, которые надо сопоставлять с получаемыми в синтезированной схеме, необходимо конфигурировать проект на использование архитектурных тел, в которых списки чувствительности процессов содержат все входные сигналы.

Использование процессов и компонентов для описания объектов

Нередко описание одного и того же объекта может быть произведено либо с использованием одного или нескольких процессов, либо с использованием механизма компонентов.

Рассмотрим, как влияет выбор способа описания на реализацию на аппаратном уровне. Возьмем в качестве примера объект, имеющий три входа `i1`, `i2`, `r1` и один выход `out1`. Этот объект выполняет следующую функцию: `out1 = (i1 + i2) or r1`. Архитектурное описание данного объекта может включать в себя один процесс, в котором выполняется это действие (листинг 4.52). Действие может быть разделено и на два процесса, в первом из которых выполняется сложение, а во втором — логическая операция **or** (листинг 4.53). Наконец, описание объекта может быть выполнено на базе компонентов, в одном из которых выполняется сложение, а в другом — логическая операция **or** (листинг 4.54).

Листинг 4.52

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ent_1 is
port(i1: in std_logic_vector (2 downto 0);
     i2: in std_logic_vector (2 downto 0);
     r1: in std_logic_vector(2 downto 0);
     out1: out std_logic_vector (2 downto 0));
 end entity ent_1;

architecture rtl_1 of ent_1 is
 begin
 process(i1,i2,r1)
 begin
 out1< = (i1 + i2) or r1;
 end process;
end architecture rtl_1;
```

Листинг 4.53

```
architecture rtl_2 of ent_1 is
 signal o1: std_logic_vector (2 downto 0);
```

```
 begin
 p_1: process(i1,i2)
       begin
            o1< = (i1 + i2);
       end process;
 p_2: process(o1,r1)
       begin
           out1< = o1 or r1;
       end process;
end architecture rtl_2;
```

Листинг 4.54

```
architecture rtl_3 of ent_1 is
component ent_3_1 is
     port (i1: in std_logic_vector (2 downto 0);
          i2: in std_logic_vector (2 downto 0);
          ou1: out std_logic_vector (2 downto 0));
end component ent_3_1;
component ent_3_2 is
     port (ii1: in std_logic_vector (2 downto 0);
          r1: in std_logic_vector (2 downto 0);
          ou2: out std_logic_vector (2 downto 0));
end component ent_3_2;
signal o1: std_logic_vector (2 downto 0);

begin
u1: ent_3_1
    port map (i1 = >i1, i2 = >i2, ou1 = >o1);
u2: ent_3_2
    port map (ii1 = > o1, r1 = >r1, ou2 = >out1);
end architecture rtl_3;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ent_3_1 is
port (i1: in std_logic_vector (2 downto 0);
     i2: in std_logic_vector (2 downto 0);
```

```
    ou1: out std_logic_vector (2 downto 0));
end entity ent_3_1;
architecture rtl of ent_3_1 is
begin
process(i1,i2)
begin
ou1< = i1 + i2;
end process;
end architecture rtl;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity ent_3_2 is
port (ii1: in std_logic_vector (2 downto 0);
    r1: in std_logic_vector (2 downto 0);
    ou2: out std_logic_vector (2 downto 0));
end ent_3_2;
architecture rtl of ent_3_2 is
begin
process(ii1,r1)
begin
ou2< = ii1 or r1;
end process;
end architecture rtl;
```

По результатам синтеза, все эти три варианта описания будут иметь совершенно одинаковую физическую реализацию. Это связано с тем, что при использовании нескольких процессов вместо одного, *если значение каждого сигнала определяется только в одном из процессов*, инструментарий синтеза Foundation Express генерирует такую же физическую реализацию, как если бы в описании использовался только один процесс.

При использовании механизма компонентов, когда в модель включаются описания компонентов на VHDL (компоненты описаны на уровне `Soft`), в ходе синтеза, при генерации списка связей, происходит переход от иерархического представления описания к одноуровневому представлению. В результате для приведенного примера получается та же самая физическая реализация. Если же описания компонентов в модель включаются на уровне списков связей, например, в формате EDIF (описание на уровне `firm` или `hard`), то сохранение иерархической структуры или переход к одноуровневому представлению зависит от настроек инструментария синтеза.

Рассмотрим ситуацию, когда при переходе от описания с использованием одного процесса к описанию с использованием нескольких процессов (или нескольких компонентов) в модели появляются сигналы, которым присваиваются значения в этих процессах или в нескольких компонентах. В таких случаях считается, что сигнал имеет несколько источников. Пример будет рассмотрен далее, в этой главе, при описании модели конечного автомата. Результирующее значение такого сигнала определяется на базе решающей функции.

В Foundation Express 2.1i механизм решающих функций в явном виде не поддерживается. Для организации решающей функции могут быть применены специальные директивы компилятору, которые, однако, не предусмотрены стандартом языка VHDL. Использование их в модели может привести к тому, что синтез ее другими средствами автоматизированного проектирования будет приводить к неверному результату.

Применение механизма компонентов в этом случае допускает добавление в модель дополнительных блоков, которые в явном виде реализуют решающую функцию. Однако такой подход приводит к дополнительным аппаратным затратам, поэтому необходимо стремиться разрабатывать модели устройств таким образом, чтобы они не содержали сигналов, имеющих несколько источников.

Описание модели конечного автомата

Рассмотрим вариант реализации на VHDL конечного автомата, диаграмма состояний которого представлена на рис. 4.28.

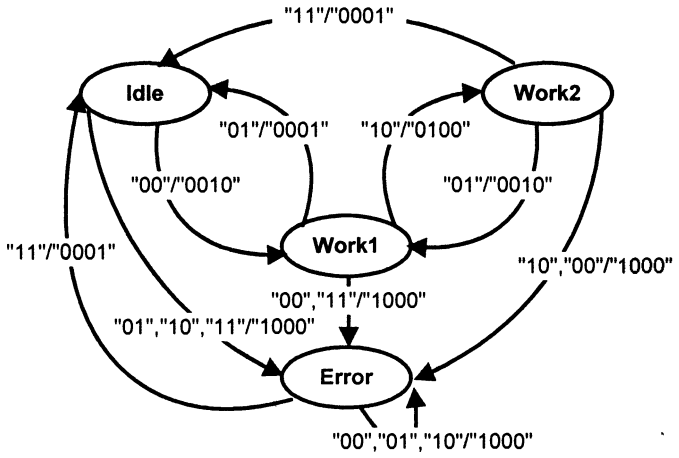


Рис. 4.28. Диаграмма состояний конечного автомата, описание которого приведено в листинге 4.56

Этот автомат имеет четыре состояния: `Idle`, `Work1`, `Work2` и `Error`. Описание типа, соответствующего множеству состояний автомата, можно вынести в пакет (листинг 4.55).

Листинг 4.55

```
package p1 is
type my_state is (idle, work1, work2, err);
end package p1;
```

Вариант описания объекта, соответствующего такому конечному автомату, приведен в листинге 4.56.

Листинг 4.56

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use work.p1.all;

entity my_automat is
port(in1:in std_logic_vector(1 downto 0);
     out1: out std_logic_vector(3 downto 0);
     clk: in std_logic;
     rst: in std_logic
   );
end entity my_automat;

architecture rtl of my_automat is
signal cur_state: my_state;
signal next_state: my_state;
begin
states: process (in1, cur_state)
begin
case cur_state is
  when idle = > if in1 = "00" then
                  next_state< = work1; out1< = "0010";
                else next_state< = err; out1< = "1000"; end if;
  when work1 = > case in1 is
                   when "01" = > next_state< = idle; out1< = "0001";
                   when "10" = > next_state< = work2; out1< = "0100";
```

```
                    when others = > next_state< = err; out1< = "1000";
                  end case;
  when work2 = > case in1 is
                    when "01" = >next_state< = work1; out1< = "0010";
                    when "11" = >next_state< = idle; out1< = "0001";
                    when others = >next_state< = err; out1< = "1000";
                  end case;
  when err = > if in1 = "11" then
                    next_state< = idle; out1< = "0001";
                else next_state< = err; out1< = "1000";
                end if;
end case;
end process states;

transition: process (clk)
begin
if rst = '1' then cur_state< = idle;
else
if clk = '1' and clk'event then cur_state< = next_state; end if;
end if;
end process transition;
end architecture rtl;
```

Архитектурное описание этого конечного автомата содержит два внутренних сигнала: `curstate` и `nextstate`. Сигнал `curstate` предназначен для хранения текущего состояния автомата. Физически он реализуется в форме триггеров (регистра соответствующей разрядности). Сигнал `nextstate` используется для определения следующего состояния — состояния, которое станет текущим в следующем такте. Значение этого сигнала определяется на базе значений входных сигналов и текущего состояния автомата. Физически этот сигнал реализуется в форме линии связи.

Типичное описание конечного автомата включает в себя два процесса. В одном из них (в данной модели — в процессе `states`) определяются значения выходных сигналов и следующего состояния. Физическая реализация такого процесса, как правило, является комбинационной схемой. Однако если в теле процесса присутствуют операторы условного перехода, в ветвях которых определяются значения для различных наборов сигналов, в схеме могут появиться защелки.

В другом процессе (в данной модели — в процессе `transition`) выполняется собственно переход из состояния в состояние по соответствующему фронту сигнала тактирования `clk`. В данном случае переход осуществляется по восходящему фронту тактового импульса.

Обратите внимание, что здесь сброс в начальное состояние по сигналу `rst` также описан в том же процессе `transition`. При написании модели для синтеза в Foundation 2.1i это действие нежелательно выделять в отдельный процесс. Это связано с тем, что если сигналу может присваиваться значение в различных процессах, считается, что сигнал имеет множество источников. В таком случае значение сигнала определяется на базе решающей функции. Однако в Foundation 2.1i механизм решающих функций в явном виде не реализован, поэтому придется использовать специальные директивы компилятору, которые не являются стандартными для VHDL, что может привести к неправильному функционированию модели, когда синтез ее производится с использованием других средств проектирования.

В программе (см. листинг 4.56) мы этого счастливо избежали, вставив оператор `if rst = '1' then cur_state< = idle;` в тело процесса `transition`. Таким образом, для сигнала `cur_state` есть только один источник, представленный процессом `transition`.

На уровне физической реализации процессу `transition` будет соответствовать регистр.

Синтез устройств, описания которых включают в себя подпрограммы

Рассмотрим, как выполняется синтез устройств, описание которых включает в себя подпрограммы.

Для каждой подпрограммы синтезируется фрагмент схемы, соответствующий выполняемым ею действиям. Если подпрограмма является функцией, то ее реализация, как правило, является комбинационной схемой. Это связано с тем, что большинство инструментов синтеза не позволяют в функциях использовать конструкции, приводящие к появлению в схеме триггеров и защелок. В процедурах же такие конструкции, как `if clk = '1' and clk'event` допустимы. Реализация процедур может включать в себя также и элементы памяти.

При синтезе устройства, в каждую точку схемы, соответствующую вызову подпрограммы, добавляется соответствующий ей фрагмент схемы. Его входы и выходы соединяются с линиями сигналов, соответствующим фактическим параметрам подпрограммы. Таким образом, с точки зрения физической реализации, при Синтезе использование механизма подпрограмм оказывается эквивалентным использованию механизма компонентов, описанных на уровне `Soft`.

Рассмотрим это на следующем примере. Пусть моделируемый объект имеет четыре входа: `in1`, `in2`, `in3`, `in4` и два выхода `out1`, `out2`; при этом `out1 = in1 or in3; out2 = in2 or in4`. Описание моделируемого объекта может иметь вид, представленный листингом 4.57.

Листинг 4.57

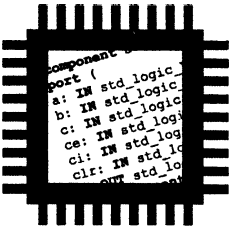
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
entity my_f1 is
port(in1,in2,in3,in4: in std_logic;
     out1,out2: out std_logic
   );
end entity my_f1;
architecture rtl_my_f1 of my_f1 is
begin
process(in1,in2,in3,in4)
begin
out1< = in1 or in3;
out2< = in2 or in4;
end process;
end architecture rtl_my_f1;
```

Значения выходных сигналов определяются на базе однотипных действий. Для их реализации может быть использована функция. В этом случае модель будет иметь вид, приведенный в листинге 4.58.

Листинг 4.58

```
architecture rtl of my_f1 is
function func_1 (i1,i2:std_logic) return std_logic is
  begin
      return i1 or i2;
  end;
begin
process(in1,in2,in3,in4)
begin
out1< = func_1(in1,in3);
out2< = func_1(in2,in4);
end process;
end architecture;
```

Формируемая при синтезе физическая реализация описаний листингов 4.57 и 4.58 будет одинакова.



Глава 5

Практика применения VHDL

Язык VHDL как средство формализованного представления спецификаций интерфейсов и протоколов

При проектировании систем на СБИС большое значение имеет освоение и корректное использование спецификаций и стандартов, определяющих правила и технические детали, организация взаимодействия проектируемого компонента — узла, блока, модуля, СБИС, и пр., с другими компонентами проектируемой системы. Спроектировав свой блок, модуль, СБИС, мы даем ее техническое описание, важную часть которого составляет описание его интерфейса, а в современных функционально сложных блоках — и протокола, взаимодействия с другими компонентами в системах.

Эта информация дается в технической документации, в стандартах на интерфейсы и протоколы. Она имеет вид словесных технических описаний, сопровождаемыми схемами, алгоритмами, рисунками и временными диаграммами. Качество этой документации бывает разное, однако по своей природе такая документация неформальна, не может исключать неоднозначности и пропуска деталей, оказывающихся важными в конкретных ситуациях. Даже в самых выверенных текстах стандартов еще долго после их опубликования практики находят неточности, неясности и противоречия. Что уж говорить о технических описаниях на изделия, поступающих от коллег-разработчиков.

Имея дело с неполной информацией об описываемом объекте, разработчик сталкивается с множеством вопросов. Все, что у него есть — это техническая документация, которой он и должен "задавать" эти вопросы и стараться угадать ответы, полагаясь на свои знания предмета и интуицию. Однозначности понимания ситуации разными разработчиками здесь добиться сложно.

В этих условиях роль формализованного средства описания может взять на себя язык VHDL. Как мы указывали в главе 1, одним из важных направлений использования языка VHDL является создание моделей-спецификаций, дополняющих словесное описание систем. Такая модель дает формализованное описание объекта. Разбирающий описание объекта пользователь, прочитав тестовую часть, может исследовать модель объекта на VHDL, чтобы получить ответы на неясные вопросы описания, получить достоверный ответ на каждую из конкретных ситуаций, перечислить которые в тексте описания для сложного объекта просто невозможно.

Это относится и к описаниям, и к стандартам на современные интерфейсы и протоколы, являющиеся сложными и многоуровневыми. Дополнение текстового описания моделями на VHDL даст их формализованное описание, создаст модель-спецификацию, исследуя которую, пользователь сможет получить однозначный ответ на любой вопрос по интересующему интерфейсу или протоколу. Практика показывает, что создание моделей-спецификаций такого рода заставляет разработчиков описаний интерфейсов и протоколов обдумывать их более детально и тщательно. Бумага все стерпит, а программа-модель на VHDL часто просто не заработает, пока не будут тщательно исследованы и запрограммированы все комбинации возможных ситуаций в описываемом интерфейсе.

Возьмем в качестве примера временные диаграммы, используемые для пояснения логики и временной последовательности обмена сигналами взаимодействующих устройств. Мы видим множество таких временных диаграмм в документации на любые СБИС, во всех стандартах на шины, на стандартизированные интерфейсы взаимодействия с внешними устройствами.

Возьмем для примера временную диаграмму, поясняющую состав и ход выполнения транзакции на шине PCI [22] (рис. 5.1).

Описывать формирование временных диаграмм при выполнении транзакции на шине можно по-разному. Можно описывать поведение ведущих и ведомых устройств за счет их функционирования и взаимодействия. В тексте стандартов, описания такого рода в виде структурных схем устройств и конечного автомата, задающего их функционирование, часто приводятся как "справочная реализация" (`reference implementation`) специфицируемого интерфейса или протокола. Однако в тексте стандарта такой материал обычно относят к разделам информативным (informative), поясняющим текст, а не нормативным (normative), задающим содержательную сторону стандарта, обязательную для выполнения. Пока еще не сложилось культуры формализованного описания на языке VHDL в стандартах, скажем, правил функционирования ведущего и ведомого устройств шины, которое было бы *нормативной* частью стандарта. Однако тенденции такого рода развития нормативных технических документов уже налицо, например, в области систем-на-кристалле.

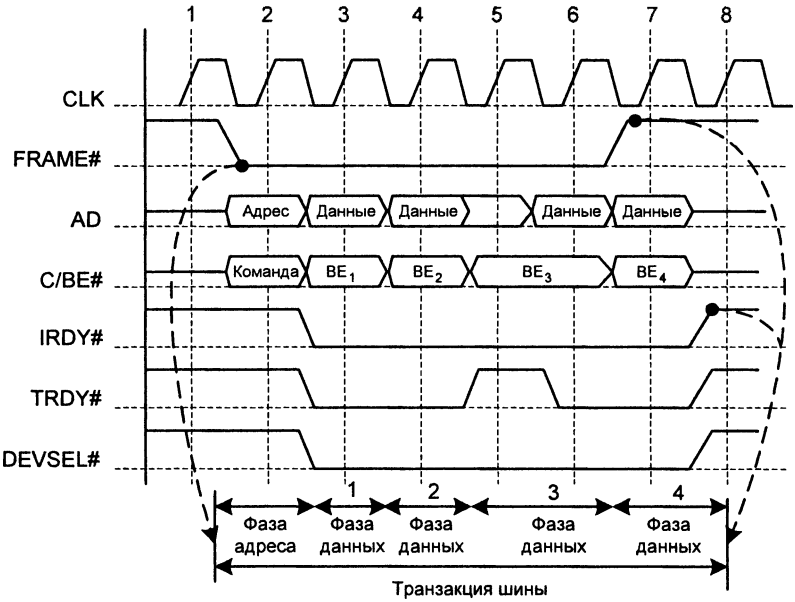


Рис. 5.1. Транзакция шины PCI

Такого рода формализованное описание будет, по стилю программирования на VHDL, достаточно близко к модели систем с шиной PCI, рассматриваемой в следующем разделе. Некоторая опасность здесь состоит в том, чтобы в модели такого рода не отойти от общности стандарта, не внести в формализованное описание такую детализацию, которая фиксирует конкретные варианты решения тех или иных вопросов, самим стандартом оставляемых на усмотрение разработчиков. Это вполне допустимо для информативного материала, с соответствующими оговорками в тексте, но не годится для нормативной части стандарта.

В данном разделе мы проиллюстрируем другой стиль формализованного описания — прямое моделирование временных диаграмм, безотносительно к тому, кто, какое устройство в структуре системы с заданной шиной их формирует.

Спецификация транзакции шины PCI (рис. 5.1) в данном примере программируется в несколько упрощенном виде. Как и на рисунке, модель строится для 32-разрядной шины, при использовании 32-разрядных адресов, для команд шины "чтение". Считается, что ведущее устройство — быстрое, и не вносит задержек, дополнительных тактов в фазах данных при выполнении транзакции. Программа-спецификация на VHDL приведена в листинге 5.1.

Листинг 5.1

```
-- описание пакета (должно быть выполнено в отдельном файле)
package pci_t is
          type acc_time is array (positive range <>) of integer;
end package;

library ieee;
use ieee.std_logic_1164.all;
use pci_t.pci_t.all;
entity PCI_transactions is
generic (F: integer;  -- тактовая частота шины PCI
         delay: time; -- типовая задержка изменения сигнала
                      -- после фронта сигнала тактирования
         taddr_hold, tdata_hold: time; -- времена выдержки сигнала
                      --  адреса и сигнала данных (после тактирования)
         bytes_code: std_logic_vector (3 downto 0);
     -- позиционный код байтов на шине данных AD:
           -- bytes_code(i)='1' - байт используется для передачи
           -- bytes_code(i)='0' - байт не используется для передачи
        words : positive; -- число слов в транзакции
        target_delays: acc_time(1 to words); --задержка обращений
                                             -- к ведомому устр-ву, в нс
        PCIcommand: std_logic_vector (3 downto 0) - код команды шины PCI
       );
end PCI_transactions;

architecture read_transactions of PCI_transactions is
 type bus_data is ('A','D','X');
 constant tau: time:=32 ns; --1/F ;
 constant taur: real:=32; --1/F;
 signal clk: std_logic := '0';
 signal FRAME, IRDY, TRDY, IRDY, DEVSEL: std_logic := '1';
 signal AD: bus_data:='X';
 signal C_BE: std_logic_vector (3 downto 0):="XXXX";
-- контрольные сигналы для отладки
 signal n : integer;
 signal i: natural:=1;
 signal lastdata: Boolean:=true;
```

```
begin
PCI_Clk:  process (clk)      -- временная диаграмма сигнала Clk
begin
    if clk='1' then clk<='0' after tau/2;
      else clk<='1' after tau/2;
    end if;
end process PCI_Clk;

PCI_Frame:  process      -- временная диаграмма сигнала FRAME#
begin
-- начинаем транзакцию после первого же момента тактирования
wait until rising_edge (Clk); -- clk0
FRAME<='0' after delay;
wait until rising_edge (Clk); -- пропускаем такт Фазы адреса
-- держим сигнал FRAME='0' до начала последней Фазы данных;
-- процесс PCI_Frame знает число передаваемых в транзакции слов, words
if words>1 then
  for j in 1 to (words-1) loop                    -- цикл по фазам данных
    wait until (rising_edge (Clk) and TRDY='0'); -- последний такт
                                                  -- текущей фазы данных
  end loop;
end if;
-- перед первым тактом последней Фазы данных
FRAME<='1' after delay;
-- ждем конца последней фазы данных
wait until (rising_edge (Clk) and TRDY='0'); end process PCI_Frame;
end process PCI_Frame;

PCI_AD:  process     -- временная диаграмма сигналов по шине AD
  begin
    wait until (FRAME='0');    -- обнаружено начало транзакции
      AD <='A';                -- Фаза адреса
      wait until rising_edge(Clk);
      wait for taddr_hold;
      -- Фазы данных
      lastdata:=false;
data_phases: loop
clocks_in_dataphase: loop
  if (TRDY='0') then
    AD<='D';
```

```
    wait until rising_edge(Clk);
    if (FRAME='1') then lastdata<=true; -- последняя Фаза данных
    end if;
    wait for tdata_hold;
    exit clocks_in_dataphase;           -- конец текущей Фазы данных
  else
    AD<='X';
    wait on TRDY;
  end if;
   end loop clocks_in_dataphase;
   exit data_phases when lastdata;      -- конец текущей транзакции
  end loop data_phases;
  AD<='X';
end process PCI_AD;

PCI_C_BE:  process     -- временная диаграмма сигналов по шине C/BE#
  variable lastdata: boolean;
  begin
    wait until falling_edge(FRAME);
    C_BE <=PCIcommand;      -- в Фазе адреса – команда
    wait until rising_edge(Clk);
    wait for taddr_hold;
    lastdata:=false;
data_phases: loop
    C_BE<=bytes_code;       -- в Фазе данных – код задействованных
                            -- байтов шины AD
clocks_in_dataphase: loop
  if (TRDY='0') then
    wait until rising_edge(Clk);
    if (FRAME='1') then lastdata:=true; --последняя Фаза данных
    end if;
    wait for tdata_hold;
    exit clocks_in_dataphase; -- конец текущей Фазы данных
  else
    wait on TRDY;
  end if;
end loop clocks_in_dataphase;
exit data_phases when lastdata; -- конец текущей транзакции
end loop data_phases;
```

```
C_BE<="XXXX";
end process PCI_C_BE;
PCI_IRDY:  process      -- временная диаграмма сигнала IRDY#
 begin
wait until FRAME'event;
if falling_edge(FRAME) then
wait until rising_edge(Clk);
IRDY <='0' after delay;
 else
wait until (rising_edge(Clk) and TRDY='0');
IRDY <='1' after delay;
   end if;
end process PCI_IRDY;

PCI_TRDY:  process      -- временная диаграмма сигнала TRDY#
variable i1 integer;
begin
wait until falling_edge(FRAME); -- ждем начала транзакции
-- после момента тактирования Фазы адреса
wait until rising_edge(Clk);
-- входим в цикл формирования сигнала TRDY готовности
-- ведомого устройства для фаз данных
i1:=1;
i<=i1;
data_phases: loop   -- цикл, пока FRAME не покажет конец транзакции
if i1>0 then
 if target_delays(i1)>taur then -- тогда Фаза данных – несколько тактов
TRDY<='1' after delay;  -- устанавливаем признак неготовности
  -- ведомого устройства
n<=target_delays(i1)/tau;
several_clocks: for k in 1 to n+1 loop
wait until rising_edge(Clk);
end loop several_clocks;
 else n<=0;
 end if;
else
 if target_delays(1)>taur then
TRDY<='1' after delay;
n<=target_delays(1)/tau;
```

```
  several_clocks2: for k in 1 to n+1 loop
wait until rising_edge(Clk);
end loop several_clocks2;
  else n<=0;
  end if;
end if;

TRDY<='0' after delay; -- устанавливаем готовность на следующий такт
wait until rising_edge(Clk); -- держим готовность до момента тактирования
--  проверяем, не была ли текущая Фаза данных последней
if (FRAME='1') then -- конец транзакции
      i1:=0; i<=i1;
      TRDY<='1' after delay;
      exit loop data_phases;
  else
      if i1=words then i1:=1; i<=i1; else i1:=i1+1; i<=i1; end if;
end if;
-- переходим к следующей Фазе данных
end loop data_phases;
-- транзакция завершена
end process PCI_TRDY;

PCI_DEVSEL:  process      -- временная диаграмма сигнала DEVSEL#
begin
    wait until frame'event;
    if falling_edge(FRAME) then
        wait until rising_edge(Clk);
        DEVSEL <='0' after delay;
      else
        wait until (rising_edge(Clk) and TRDY='0');
        DEVSEL <='1' after delay;
    End if;
  end process PCI_DEVSEL;
end architecture read_transactions;

entity test is
end entity test;
architecture test1 of test is
  component PCI_transactions is
    generic (F: integer;
```

```
            delay: time;
            taddr_hold, tdata_hold: time;
            bytes_code: bit_vector (3 downto 0);
            words: positive;
            target_delays: acc_time(1 to words);
            PCIcommand: std_logic_vector (3 downto 0));
  end component PCI_transactions;
begin
my_test: PCI_transactions
  generic map (F=>33,
               delay=> 5 ns,
               taddr_hold=>15 ns, tdata_hold=> 20 ns,
               bytes_code=>"1111",
               words=>4,
               target_delays=>(20,20,40,20),
               PCIcommand=> 0110", -- команда шины PCI "чтение памяти"
);
end architecture test1;
```

В этой программе мы строим отдельный процесс для формирования каждой из заданных на рисунке временных диаграмм. Взаимодействие процессов использует только те сигналы, которые определены стандартом шины PCI и присутствуют на рисунке.

Описанные в архитектурном теле сигналы `n`, `i`, `lastdata`, с точки зрения логики работы программы, вполне могли бы быть описаны и как переменные. Однако определение их как сигналов удобно для анализа и отладки программы, поскольку изменения сигналов мы можем легко отслеживать на временных диаграммах, а переменные на временные диаграммы не выводятся.

Процесс `PCI_Clk` формирует последовательность тактовых импульсов шины PCI.

Процесс `PCI_AD` соответствует временной диаграмме сигналов по шине AD. Полное выполнение процесса соответствует одной транзакции, формированию временной диаграммы сигналов по шине AD в ходе выполнения одной транзакции.

Запускается процесс по событию изменения сигналов `FRAME`, `TRDY`, `Clk`.

Запустившись, процесс `PCI_AD` останавливается в ожидании перехода сигнала `FRAME` из '1' в '0'. При обнаружении ниспадающего фронта сигнала `FRAME`, процесс продолжает работу. Начинается Фаза адреса — на шину AD выставляется адрес ведомого устройства (на диаграмме адрес на шине AD обозначается буквой 'A'). Адрес держится на шине до очередного восходящего

фронта сигнала тактирования Clk и далее — на время выдержки сигнала адреса (параметр настройки taddr_hold). На этом Фаза адреса заканчивается, и начинается Фаза данных, возможно, первая из нескольких.

В Фазе данных на шине могут находиться значения пересылаемых данных или неопределенные значения, что указывается управляющим сигналом TRDY шины PCI: если TRDY=0, то на шине — данные (на диаграмме обозначаем "D"), если TRDY≠0, то на шине неопределенные значения (на диаграмме обозначаем "X"). Фаза данных может занимать 1 такт или может быть растянута на несколько тактов (для того чтобы дождаться поступления данных на шину). Как это принято на синхронных шинах, проверка значений всех сигналов производится в моменты тактирования. На шине PCI — это момент восходящего фронта тактового импульса на линии Clk.

Признаком последнего такта текущей Фазы данных является значение '0' сигнала TRDY в момент тактирования. Если в этот момент сигнал TRDY не равен нулю, значит, Фаза данных продолжается, не завершается на этом такте.

Таким образом, в транзакции шины PCI, вслед за Фазой адреса, может быть несколько Фаз данных, каждая из которых может занимать несколько тактов. Соответственно, для их представления в программе процесса, ставим 2 цикла: внешний цикл data_phases — по Фазам данных, внутренний цикл cloks_in_dataphase — по тактам шины PCI внутри Фазы данных.

В синхронных шинах сигналы проверяются в моменты тактирования, а меняют свои значения — *между* моментами тактирования. Чтобы состояние сигнала могло надежно определяться в момент тактирования, сигнал должен начать меняться раньше, и к моменту тактирования иметь установившееся значение. Из диаграмм в документах на шину PCI мы видим, что значения данных на шине AD появляются в тот момент, когда сигнал TRDY переходит в состояние '0', как и на нашем рисунке (рис. 5.1). Здесь мы не ставим себе задачу уточнять или доопределять временные диаграммы из документов на PCI, а хотим их только строго воспроизвести моделью на языке VHDL. Соответственно, в соответствии с этими диаграммами, будем формировать значение "данные" (обозначаем 'D') сигнала AD тогда, когда сигнал TRDY принимает значение '0'.

В случае если TRDY=0, сигнал AD устанавливается в состояние 'D', держится в этом состоянии до ближайшего момента тактирования плюс время выдержки сигнала данных (параметр настройки taddr_hold) после него. На этом текущая фаза данных заканчивается.

При формировании на шине AD Фазы данных важно, является ли эта Фаза данных последней в транзакции. В соответствии со стандартом PCI, сигнал FRAME, который устанавливался в '0' в начале транзакции, переводится в '1' перед последней Фазой данных транзакции. Последняя Фаза данных выполняется при значении FRAME='1'.

Поэтому, при TRDY=0, по ветке **then** нам приходится разделять фиксацию значения сигнала AD='D' до ближайшего момента тактирования и выдержку значения сигнала после момента тактирования. В момент тактирования (здесь он соответствует последнему такту текущей Фазы данных) вставляем проверку значения сигнала FRAME для определения, — является ли вся эта Фаза данных последней в транзакции. Вообще-то, сигнал FRAME указывает на то, что эта Фаза данных будет последней в текущей транзакции, уже на ее *первом* такте. Однако в процессе PCI_AD, для формирования диаграммы сигнала AD, это неважно.

По ветке **else**, что соответствует TRDY≠0, в AD заносится 'X' (обозначает неопределенное значение сигнала), после чего ждем изменения сигнала TRDY, и, по циклу clocks_in_dataphase, вновь переходим на проверку TRDY.

Закончив формирование сигнала AD для текущей Фазы данных, по циклу data_phases переходим к формированию следующей Фазы данных, если текущая фаза не была последней.

С завершением последней Фазы данных завершается транзакция, а в нашей программе на VHDL завершается выполнение процесса PCI_AD. Вновь этот процесс будет запущен при начале новой транзакции.

Процесс PCI_C_BE, формирующий временную диаграмму шины C/BE#, во многом написан аналогично процессу PCI_AD. В нашем примере мы намеренно писали эти процессы как независимые друг от друга. Как видно из программы, мы могли бы и объединить их в один (читателю несложно будет сделать это самому, в качестве упражнения).

Процесс PCI_TRDY выдает сигнал TRDY, который оказывает управляющее воздействие на формирование временных диаграмм других сигналов, определяет длительность фаз данных и, таким образом, длительность транзакции. Запускается процесс PCI_TRDY также один раз на транзакцию. Запустившись, тут же приостанавливается в ожидании перехода сигнала FRAME с '1' на '0', отмечающего начало транзакции. При обнаружении на последнем такте очередной фазы данных, что FRAME=1 (означает, что это была последняя фаза данных транзакции), процесс устанавливает TRDY в '1', выходит из цикла, и завершается. Запустившись вновь, процесс PCI_TRDY остановится, ожидая начала следующей транзакции.

В нашем примере считаем, что инициатор обмена постоянно находится в состоянии готовности (как и представлено диаграммами на рис. 5.1).

Несколько по-другому построены процессы PCI_IRDY, PCI_DEVSEL. В отличие, например, от процесса PCI_AD, срабатывающего один раз за транзакцию, процесс PCI_IRDY срабатывает несколько раз. При изменении сигнала FRAME, если это был нисходящий фронт, процесс сначала приостанавливается до ближайшего момента тактирования и еще на некоторое время delay (настраиваемый

параметр модели), после чего переводит сигнал IRDY в состояние '0' и завершает текущее срабатывание. Если процесс PCI_IRDY сработал при изменении сигнала FRAME с '0' на '1', то он приостанавливается до того момента тактирования, в котором сигнал TRDY='0', что соответствует последнему такту последней фазы данных; поскольку сигнал FRAME переведен из '0' в '1'. Как мы определили при рассмотрении процесса PCI_AD, признаком последнего такта последней Фазы данных текущей транзакции будет равенство '0' сигнала TRDY в момент тактирования (после того, как сигнал FRAME перешел из '0' в '1').

Операнды в операции деления должны быть одного типа. При преобразовании действительного числа к целому производится округление до ближайшего целого, поэтому в выражении результат деления будет, при преобразовании в integer, округлен до ближайшего целого.

Обратите внимание, что для определения момента тактирования, т. е. момента восходящего фронта сигналов Clk, FRAME, в программе используется функция rising_edge(...) из пакета std_logic_1164, а, например, не конструкция (Clk'event and Clk='1'). Это связано с тем, что данные сигналы декларированы нами как std_ulogic и могут иметь не два, а девять состояний. Так, конструкция (Clk'event and Clk='1') окажется истинной и при переходе, например, Clk из 'X' в '1', что не соответствует восходящему фронту сигнала.

Для формирования временных диаграмм с желаемыми параметрами используем компонент-экземпляр my_test в архитектурном описании test1 тестового объекта test. Настройку параметров, определяющих характеристики формирования временных диаграмм, задаем с помощью оператора отображения **generic map** настраиваемых параметров модели.

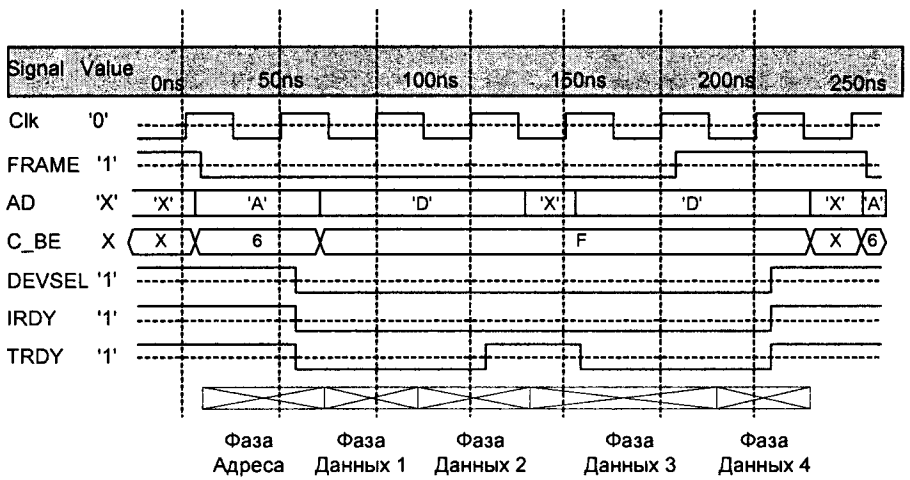


Рис. 5.2. Пример результатов моделирования сигналов шины PCI при транзакции

На временных диаграммах (рис. 5.2) приведен пример результатов моделирования для чтения четырех слов в транзакции (времена доступа в память – 20 нс., 20 нс., 40 нс. и 20 нс., при последовательном обращении к первому, второму, третьему и четвертому слову).

Моделирование на VHDL обмена данными между устройствами по шине PCI

В организации современных вычислительных систем — модульных, развиваемых, с открытой архитектурой, важное место занимают средства взаимодействия компонентов системы, средства обеспечения информационных потоков между ними, координации и управления их согласованным функционированием Решение этих проблем во многих ВС возлагается на основную системную магистраль. Унификация и стандартизация системной магистрали является необходимой предпосылкой создания открытых систем, развития независимо разработанных функциональных блоков, плат, модулей, способных к комплексированию и взаимодействию в разных комбинациях для построения законченных вычислительных систем.

Среди стандартизованных шин современных высокопроизводительных систем важное место занимает шина PCI, которая получила широкое распространение в вычислительных системах различных классов — от персональных ЭВМ до рабочих станций, высокопроизводительных серверов, компьютеров промышленного применения и встраиваемых микропроцессорных систем.

Шина PCI является локальной, процессорно-независимой, предназначенной для организации взаимодействия процессорных модулей и контроллеров в рамках одной платы.

Структура модели

Промоделируем процесс обмена данными между несколькими ведущими устройствами и памятью по шине, с поддержанием упрощенного протокола шины PCI [21]. В рассматриваемой модели не обеспечивается возможность блокировки фрагмента памяти, ведомое устройство не может запросить остановку текущего цикла обмена; не поддерживаются прерывания и обращения в пространство конфигурации.

Пусть моделируемая система имеет структуру, представленную на рис. 5.3.

В этой модели два ведущих устройства независимо друг от друга выполняют простые действия, связанные с чтением и записью в память. Друг с другом абоненты — ведущие устройства — не взаимодействуют.

Существует несколько способов описания такой модели на VHDL. Каждому из устройств, входящих в ее состав, может сопоставляться один или несколько процессов, или же устройство может быть описано как блок. Наиболее

простой способ, с точки зрения написания и отладки, — реализовать эту модель в виде системы с двухуровневой иерархией. На нижнем уровне иерархии каждое из устройств на шине реализуется в виде самостоятельного объекта, а на верхнем — они входят в состав модели в качестве компонентов.

Модель написана для OrCAD 7.0.

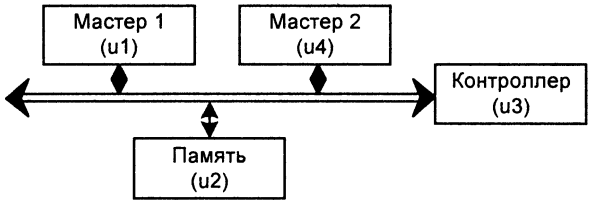


Рис. 5.3. Структура моделируемой системы

Модель ведущего устройства

Рассмотрим объект `master` — ведущее устройство в цикле обмена по шине PCI. Оба объекта типа `master` в модели имеют сходную структуру. Объекты организованы как конечные автоматы, функционирование которых описывается следующимобразом: ведущее устройство прочитывает одно слово из памяти, обрабатывает данные и записывает одно слово в память. Таким образом, устройство последовательно обрабатывет некоторый массив ячеек памяти. Если шина оказывается занятой в момент обращения к ней, то ведущее устройство ожидает ее освобождения.

Рассматриваемое ведущее устройство может находиться в одном из следующих состояний: `MIdle`, `MAdr`, `MData`, `Mwork`. Граф автомата, соответствующего ведущему устройству, представлен на рис. 5.4. В состоянии `Midle` устройство не выполняет никаких действий, в ожидании освобождения шины, необходимой ему для обмена данными с памятью. В состоянии `Madr` ведущее устройство выставляет на шину адрес памяти, по которому нужно прочитать или записать данные. В состоянии `Mdata` оно прочитывает или записывает данные из памяти. В состоянии `Mwork` оно обрабатывает данные (что в данной реализации модели будем представлять тактом простоя).

Описание этих состояний оформим в пакет `master_pack`, листинг 5.2.

Листинг 5.2

```
library ieee;
use ieee.std_logic_1164.all;
package master_pack is
type master_state is (MIdle, MAdr, MData, MWork);
end master_pack;
```

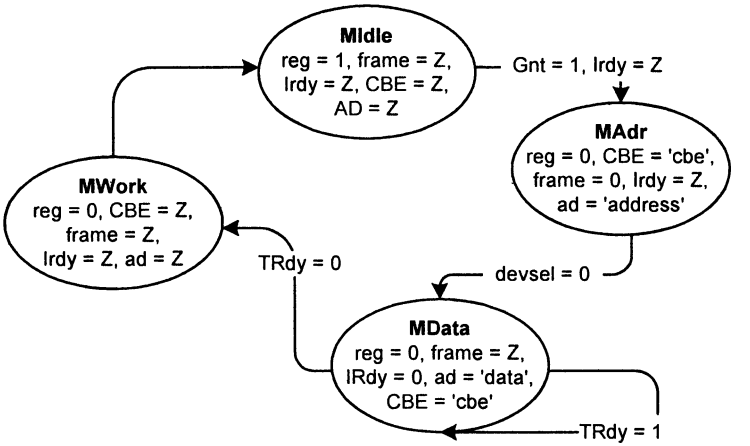


Рис. 5.4. Граф состояний автомата ведущего устройства шины PCI

Модель поведения ведущего устройства рассмотрим на примере `master1`, листинг 5.3.

Листинг 5.3

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use master_pack.master_pack.all;
entity master1 is
generic (towrite: std_ulogic_vector (31 downto 0);
port     (clk: in bit;

          FRAME: out std_ulogic;
          AD: inout std_ulogic_vector (31 downto 0);
          CBE: out std_ulogic_vector(3 downto 0);
          IRDY: out std_ulogic;
          TRDY: in std_ulogic;
          DEVSEL: in std_ulogic;
          Req: out std_ulogic;
          Gnt: in std_ulogic;);

end entity master1;
architecture behavior of master1 is
signal state: master_state:=MIdle;
signal ttrdy: std_ulogic;
signal adr: std_ulogic_vector(31 downto 0);
```

```
signal ddata: std_ulogic_vector(31 downto 0);
signal coua0: natural:=15;
signal coua1: natural:=0;
signal couc0: natural:=1;
signal couc1: natural:=0;
signal cbe1:std_ulogic_vector(3 downto 0);
function evec2int(l:std_ulogic_vector) return natural is
 variable result: natural:=0;
 variable t1: natural;
begin
   for t1 in l'range loop
       result:=result*2;
     if l(t1)='1' or l(t1)='H' then result:=result+1; end if;
    end loop;
    return result;
  end evec2int;
begin

p_1: process(state)
begin
if state=MIdle  then req<='1'; frame<='Z'; IRDY<='Z';
else req<='0';end if;
end process p_1;

p_2: process(clk)
begin
if clk='1' and clk'event then ttrdy<=trdy; end if;
end process p_2;

p_3: process(clk)
begin
if clk='0' and clk'event then
case state is
when MIdle =>
if Gnt='1' and Irdy='Z'  then state<=MAdr; Req<='0';
      if couc1=0 then CBE<="0110"; CBE1<="0110";
           else CBE<="0111";CBE1<="0111";end if;
      FRAME<='0';
      irdy<='0' after 3 ns;
```

```
        AD<=adr;
      else  CBE<="ZZZZ"; cbe1<="ZZZZ"; FRAME<='Z';
      AD<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
      IRDY<='Z';
end if;
when MData =>
  FRAME<='Z';
if TTRDY='0' then state<=MWork;
   CBE<="ZZZZ";
   AD<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
   IRDY<='Z'; coua1<=coua1+1; couc1<=couc1+1;
  else state<=MData; irdy<='0'; end if;
when MWork =>
     state<=MIdle;
     Ddata<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
     cbe1<="ZZZZ"; req<='1';
 end case;
end if;
end process p_3;

p_4: process (Devsel,state,clk)
begin
if DevSel='0' and state=MAdr and clk'event and clk='0' then
    state<=MData; IRDY<='0'; Frame<='Z';
    if cbe="0110" then
       AD<="ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";end if;
     end if;
end process p_4;

p_5: process(state,clk)
begin
if state=mData and CBE1="0110" and trdy='0' and clk='1' and clk'event
then  ddata<=ad; end if;
end process;
p_6: process (coua1)
begin if coua1>coua0 then coua1<=0; end if; end process;
p_7: process (couc1)
begin if couc1>couc0 then couc1<=0; end if; end process;
p_8: process (coua1)
```

```
begin
case coua1 is
when 0 => Adr<="00000000000000000000000000000000";
when 1 => Adr<="00000000000000000000000000000001";
when 2 => Adr<="00000000000000000000000000000010";
when 3 => Adr<="00000000000000000000000000000011";
-- для 4 — 13 заполняется аналогично
when 14=>  Adr<="00000000000000000000000000001110";
when 15 => Adr<="00000000000000000000000000001111";
end case ;
end process;
p_9: process (coua1,state)
begin
if state'event and state=Mdata and CBE1="0111" then
--writing to memory
case coua1 is
when 15 => ddata<= towrite or "00000000000000000000000000000000";
ad<="00000000000000000000000000000000";
when 14 => ddata<= towrite or "00000000000000000000000000000001";
ad<="00000000000000000000000000000001";
when 13 => ddata<= towrite or "00000000000000000000000000000010";
ad<="00000000000000000000000000000010";
when 12 => ddata<= towrite or "00000000000000000000000000000011";
ad<="00000000000000000000000000000011";
-- для 11 — 3 заполняется аналогично
when 2 => ddata<= towrite or "00000000000000000000000000001101";
ad<="00000000000000000000000000001101";
when 1=>  ddata<= towrite or "00000000000000000000000000001110";
ad<="00000000000000000000000000001110";
when 0 => ddata<= towrite or "00000000000000000000000000001111";
ad<="00000000000000000000000000001111";
end case ;
end if;
end process p_9;
end architecture behavior;
```

Рассмотрим функционирование модели ведущего устройства `master`. Назначение портов `master` соответствует описанию одноименных сигналов для шины PCI.

В модели описано несколько внутренних сигналов. Первый из них — state, предназначен для отслеживания текущего состояния ведущего устройства.

Сигналы adr и ddata предназначены для хранения адреса и данных. Они, в соответствующие моменты времени, поочередно оказываются связанными с сигналом ad. Значение adr выдается на ad в фазе адреса, а значение ddata передается на ad (или прочитывается с ad) в фазе данных.

Ведущее устройство master организовано таким образом, что оно может последовательно обрабатывать N ячеек памяти. В данной реализации N=16. Значение N хранится в сигнале coua0. Это связано с тем, что значения сигналов в OrCAD 9.1 при моделировании отслеживать удобнее, чем значения переменных. Можно было использовать и константу, но реализация с использованием переменного значения позволяет легко модифицировать модель так, чтобы ведущее устройство могло работать с массивами переменной длины.

Для хранения текущего номера обрабатываемой ячейки также служит сигнал — coua1. Сигнал, а не переменная, в этом случае служит не только для удобства наблюдения за его значением, но и для того, чтобы его можно было использовать в списке чувствительности процесса. После каждого обращения к памяти, значение сигнала coua1 увеличивается на 1, если же оно достигает значения, максимально возможного в модели (coua0), то обнуляется. Сигналы couc0 и couc1 используются по аналогичной схеме. С их помощью в каждом такте определяется направление обмена данными. Значение couc0 определяет длину периода последовательности направлений обменов. Значение couc1 определяет номер обмена в периоде. После каждого обмена значение couc1 увеличивается на 1, когда его значение достигает couc0, оно обнуляется. На базе значения couc1 определяется направление обмена данными. В этой реализации couc1=0 является чтением данных из памяти, couc1=1 — записью данных в память.

В модели приведено несколько функций, однако, их можно использовать только в OrCAD Express 9.1 и старше. Функция evec2int(l: std_ulogic_vector) return natural предназначена для преобразования адреса, передаваемого по шине, в индекс элемента массива памяти, который является натуральным числом. Функция NCBE(couc1:natural) return std_ulogic_vector(3 downto 0) служит для генерации направления обмена при очередном цикле обмена с памятью. В этой версии процесс генерации очередного адреса и очередных данных для записи в память также можно реализовать в виде функции.

Собственно, поведение ведущего устройства описывается с помощью набора процессов. Первый процесс (p_1) в наборе обеспечивает неактивное состояние выходных сигналов ведущего устройства, если оно находится в состоянии midle (ожидание предоставления шины). Второй процесс (p_2)

служит для запоминания состояния сигнала готовности от ведомого устройства по переднему фронту тактового импульса. Запоминание производится во внутренний сигнал `ttrdy`, который используется в остальных процессах. Необходимость этого возникает вследствие того, что сигнал `trdy` может быть изменен ведомым устройством для следующего такта в момент прихода переднего фронта тактового импульса (для тех же целей используется и внутренний сигнал `cbe1`). В результате, в процессе моделирования по переднему фронту тактового импульса, было бы использовано уже новое значение сигнала, которое на самом деле будет действительным только для следующего такта.

В третьем процессе (`p_3`) реализуется собственно логика работы ведущего устройства как конечного автомата. Переход автомата в новое состояние осуществляется на основе состояний сигналов в момент прихода обратного фронта тактового импульса. Если ведущее устройство находится в состоянии `Midle`, то при приходе от контроллера шины сигнала разрешения на использование этой шины, ведущее устройство переходит в состояние адреса. Это сопровождается выдачей на шину очередного адреса, направления обмена, сигнала `frame`, а также сигнала готовности ведущего устройства. Очередной адрес определяется в конце предыдущего обмена данными, на базе значения сигнала `coua1`. Аналогично направление обмена определяется на базе `couc1`, очередное значение которого определяется в конце предыдущей фазы обмена. Если ведущее устройство находится в состоянии чтения или записи данных, а ведомое устройство поддерживает сигнал готовности, то ведущее устройство заканчивает текущий обмен данными, освобождая шину, и переходит в состояние обработки данных. Кроме того, в конце фазы данных, значения сигналов `coua1` и `couc1` увеличиваются на 1, что обеспечивает выполнение другого действия в очередной фазе работы ведущего устройства.

В данной модели считается, что состояние обработки данных продолжается в течение одного такта, в ходе которого ведущее устройство ничего явного, на шине, не делает и поддерживает свои выходы на нее в неактивном состоянии. Затем оно переходит в состояние `Midle` и запрашивает шину для нового цикла работы с памятью.

Однако в рассматриваемом процессе не реализован переход из состояния адреса в состояние данных. Это связано с тем, что он имеет дополнительное условие, — ведомое устройство должно распознать свой адрес и выдать сигнал `Devsel`, уведомляющий об этом. Вообще этот переход можно было не выделять в отдельный процесс (4-й процесс), но это выделение облегчает чтение модели.

Пятый процесс (`p_5`) обеспечивает чтение с шины в фазе данных при выполнении операции чтения.

Шестой (p_6) и седьмой (p_7) процессы служат для обнуления соответствующих счетчиков при достижении ими максимальных значений. Они выполняются при каждом изменении значений сигналов-счетчиков (т. е. в конце каждой фазы данных).

Восьмой процесс (p_8) определяет адрес для очередной фазы обмена на основе значения счетчика coua1 (он также выполняется в конце каждой фазы данных). В каждой следующей фазе обмена выдается следующий адрес. Адрес, сформированный в конце *i*-й фазы данных, будет использован для фазы *i*+1.

Последний процесс (p_9) служит для определения значения на шине данных при очередной операции записи в память. При использовании для моделирования OrCAD 9.1 эти процессы можно реализовать на базе функций.

Организация устройства памяти

Граф конечного автомата, соответствующего устройству памяти, представлен на рис. 5.5. Описание модели памяти приведено в листинге 5.4.

Листинг 5.4

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use pack_mem.pack_mem.all;
entity mem1 is
 port (AD: inout std_ulogic_vector (31 downto 0);
         IRDY: in std_ulogic;       TRDY: out std_ulogic;
         DEVSEL: out std_ulogic;      FRAME: in std_ulogic;
         C_BE: in std_ulogic_vector (3 downto 0); clk: in bit);
 end;
 architecture behavior_mem1 of mem1 is
 signal ADDRES: std_ulogic_vector (31 downto 0);
 signal tryadr:bit:='1';
 signal DATA: std_ulogic_vector (31 downto 0) := (others=>'1');
 signal MEM: mem_r:=ready;
 signal STATE: mem_state:=idle;
 signal Mem_time: integer:=0;
 signal T_mem:integer:=1;
 signal ind_tmem:integer:=1;
 signal mem_arr: std_ulogic_vector(511 downto 0):=(others=>'1');
 signal adn:natural;
 signal clk1:bit;
```

```
begin
p_1: process(clk)
begin clk1<=clk after 1 ns; end process;
p_2: process (state,frame,irdy,mem,clk1)
begin
if clk1='0' and clk1'event then
case state is
  when idle =>
if frame='0' then state <= adress; end if;
  when adress =>
if frame='0' and irdy='0' and mem=ready and tryadr='1' then
 state<=datam; end if;
if not (frame='0') and irdy='0' and mem=ready  and tryadr='1' then
 state<=data1; end if;
if not (frame='0') and irdy='0' and mem=nready and tryadr='1' then
 state<=a_wait1; end if;
if frame='0' and irdy='0' and mem=nready and tryadr='1' then
 state<=a_wait; end if;
if tryadr='0' then state<=idle; end if;
   when data1 =>
if frame='0' then state<=adress; else state<=idle; end if;
   when datam =>
if not(frame='0') and irdy='0' and mem=ready then
 state<=data1; end if;
if not (irdy='0' and mem=ready) and frame='0' then
 state<=waitt; end if;
if not (irdy='0' and mem=ready) and not (frame='0') then
 state<=wait1; end if;
    when a_wait1 =>
if irdy='0' and mem=ready then state<=data1;
 else state<=a_wait1; end if;
    when wait1 =>
if irdy='0' and mem=ready then state<=data1;
 else state<=wait1; end if;
    when a_wait =>
if irdy='0' and mem=ready then state<=datam;
 else state<=a_wait; end if;
     when waitt =>
```

```
if irdy='0' and mem=ready then state<=datam;
 else state<=waitt; end if;
end case;
end if;
end process p_2;

p_3: process(state,clk)
begin
if clk='1' and clk'event and state=adress then addres<=AD; end if;
end process;

p_4: process (addres)
begin
case addres is
when "00000000000000000000000000000000" => adn<=0;
when "00000000000000000000000000000001" => adn<=1;
when "00000000000000000000000000000010" => adn<=2;
when "00000000000000000000000000000011" => adn<=3;
when "00000000000000000000000000000100" => adn<=4;
when "00000000000000000000000000000101" => adn<=5;
when "00000000000000000000000000000110" => adn<=6;
when "00000000000000000000000000000111" => adn<=7;
when "00000000000000000000000000001000" => adn<=8;
when "00000000000000000000000000001001" => adn<=9;
when "00000000000000000000000000001010" => adn<=10;
when "00000000000000000000000000001011" => adn<=11;
when "00000000000000000000000000001100" => adn<=12;
when "00000000000000000000000000001101" => adn<=13;
when "00000000000000000000000000001110" => adn<=14;
when "00000000000000000000000000001111" => adn<=15;
end case;
end process p_4;

p_5: process (state, clk)
begin
if state=idle then AD<=(others=>'Z');
   trdy<='Z'; devsel<='Z'; tryadr<='1'; mem_time<=0; end if;
if state'event then
case state is
```

```
when adress =>
  mem_time<=1;  AD<=(others=>'Z');
  if AD(31 downto 30)="00" then
      tryadr<='1'; devsel<='0' after 1 ns; trdy<='1';
    else tryadr<='1'; end if;
when data1 =>
   trdy<='0';
  if c_be="0110" then  --reading from memory
  case adn is
when 0=> AD<=mem_arr(31 downto 0);data<=mem_arr(31 downto 0);
when 1=> AD<=mem_arr(63 downto 32);data<=mem_arr(63 downto 32);
when 2=> AD<=mem_arr(95 downto 64);data<=mem_arr(95 downto 64);
when 3=> AD<=mem_arr(127 downto 96);data<=mem_arr(127 downto 96);
when 4=> AD<=mem_arr(159 downto 128);data<=mem_arr(159 downto 128);
when 5=> AD<=mem_arr(191 downto 160);data<=mem_arr(191 downto 160);
when 6=> AD<=mem_arr(223 downto 192);data<=mem_arr(223 downto 192);
when 7=> AD<=mem_arr(255 downto 224);data<=mem_arr(255 downto 224);
when 8=> AD<=mem_arr(287 downto 256);data<=mem_arr(287 downto 256);
when 9=> AD<=mem_arr(319 downto 288);data<=mem_arr(319 downto 288);
when 10=> AD<=mem_arr(351 downto 320);data<=mem_arr(351 downto 320);
when 11=> AD<=mem_arr(383 downto 352);data<=mem_arr(383 downto 352);
when 12=> AD<=mem_arr(415 downto 384);data<=mem_arr(415 downto 384);
when 13=> AD<=mem_arr(447 downto 416);data<=mem_arr(447 downto 416);
when 14=> AD<=mem_arr(479 downto 448);data<=mem_arr(479 downto 448);
when 15=> AD<=mem_arr(511 downto 480);data<=mem_arr(511 downto 480);
  end case;
  end if;
 mem_time<=0;
when datam =>  -- when or case state...
 trdy<='0';
 case adn is
when 0=> AD<=mem_arr(31 downto 0);data<=mem_arr(31 downto 0);
when 1=> AD<=mem_arr(63 downto 32);data<=mem_arr(63 downto 32);
when 2=> AD<=mem_arr(95 downto 64);data<=mem_arr(95 downto 64);
when 3=> AD<=mem_arr(127 downto 96);data<=mem_arr(127 downto 96);
when 4=> AD<=mem_arr(159 downto 128);data<=mem_arr(159 downto 128);
when 5=> AD<=mem_arr(191 downto 160);data<=mem_arr(191 downto 160);
when 6=> AD<=mem_arr(223 downto 192);data<=mem_arr(223 downto 192);
```

```
when 7=> AD<=mem_arr(255 downto 224);data<=mem_arr(255 downto 224);
when 8=> AD<=mem_arr(287 downto 256);data<=mem_arr(287 downto 256);
when 9=> AD<=mem_arr(319 downto 288);data<=mem_arr(319 downto 288);
when 10=> AD<=mem_arr(351 downto 320);data<=mem_arr(351 downto 320);
when 11=> AD<=mem_arr(383 downto 352);data<=mem_arr(383 downto 352);
when 12=> AD<=mem_arr(415 downto 384);data<=mem_arr(415 downto 384);
when 13=> AD<=mem_arr(447 downto 416);data<=mem_arr(447 downto 416);
when 14=> AD<=mem_arr(479 downto 448);data<=mem_arr(479 downto 448);
when 15=> AD<=mem_arr(511 downto 480);data<=mem_arr(511 downto 480);
  end case;
 mem_time<=1;
end case; -- от case state...
end if;
end process p_5;

p_6: process (state, clk)
begin
if clk'event and clk='0' then
if state=a_wait or state=a_wait1 or state=waitt or state=wait1
then mem_time<=mem_time+1; end if;
end if;
end process;

p_7: process (state,clk)
begin
if clk='1' and clk'event then
  if state=data1 or state=datam then
if c_be="0111"  then data<=AD; -- пишем в память
case adn is
  when 0=> mem_arr(31 downto 0)<=AD;
  when 1=> mem_arr(63 downto 32)<=AD;
  when 2=> mem_arr(95 downto 64)<=AD;
  when 3=> mem_arr(127 downto 96)<=AD;
  when 4=> mem_arr(159 downto 128)<=AD;
  when 5=> mem_arr(191 downto 160)<=AD;
  when 6=> mem_arr(223 downto 192)<=AD;
  when 7=> mem_arr(255 downto 224)<=AD;
  when 8=> mem_arr(287 downto 256)<=AD;
  when 9=> mem_arr(319 downto 288)<=AD;
```

```
  when 10=> mem_arr(351 downto 320)<=AD;
  when 11=> mem_arr(383 downto 352)<=AD;
  when 12=> mem_arr(415 downto 384)<=AD;
  when 13=> mem_arr(447 downto 416)<=AD;
  when 14=> mem_arr(479 downto 448)<=AD;
  when 15=> mem_arr(511 downto 480)<=AD;
 end case;
end if;
  end if;
end if;
end process;
p_8: process(mem_time)
begin
if mem_time>=T_mem then mem<=ready; else mem<=nready; end if;
end process;
end architecture behavior_mem1;
```

Память в этом описании организована как одномерный массив; она может быть организована также как двумерный массив или как одномерный массив, элементами которого являются векторы. Текущее состояние памяти отражается сигналом `mem_arr: std_ulogic_vector(511 downto 0):=(others=>'1');` Слову памяти с адресом 0 соответствуют разряды 31—0 этого сигнала, с адресом 1—63— 32, и так далее. В начале работы все ячейки памяти устанавливаются в 1.

В модели используются внутренние сигнал `Address` и `Data`. В них, в соответствующие моменты времени, записывается значение шины `AD`.

Память может находиться в одном из следующих состояний: `idle`, `adress`, `data1`, `datam`, `a_wait`, `a_wait1`, `waitt`, `wait1`. Если обращений к памяти нет, она находится в состоянии `idle`. Состояние `address` соответствует адресной фазе обмена. Если память не готова начать обмен первым словом данных сразу после адресной фазы, то она переходит в состояние `d_wait1` или `d_wait` (`d_wait1` соответствует обмену одним словом данных, т. е. одиночному обмену; `d_wait` соответствует обмену несколькими словами данных, т. е. множественному обмену). Если память готова выполнять обмен, то переходит в состояние `data1` (при условии, что осуществляется одиночный обмен или последняя фаза данных во множественном обмене) или же память переходит в состояние `data_m`, соответствующее множественому обмену данными. Если память или ведущее устройство не готовы к обмену очередным словом данных, то память из состояния `data_m` переходит либо в состояние `waitt` (при условии, что это слово не последнее), либо в состояние `wait1` (в противном случае).

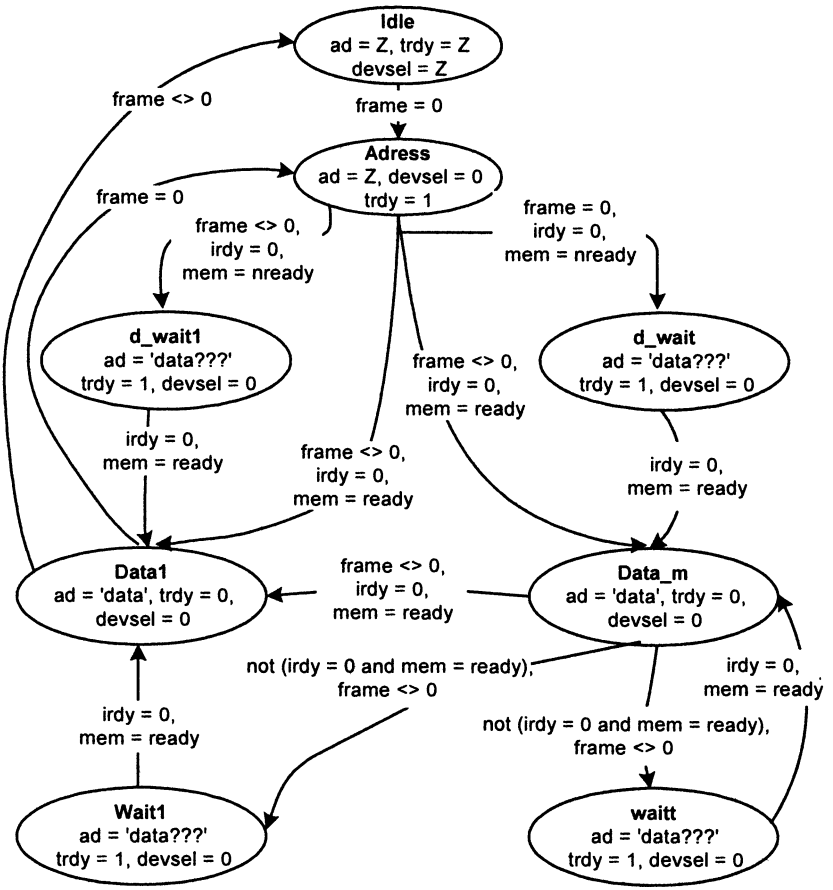


Рис. 5.5. Граф конечного автомата, соответствующего устройству памяти

Состояние памяти в текущий момент времени отражается внутренним сигналом `state`. Внутренний сигнал `mem` отражает готовность данных в памяти. Типы данных, использованные для этих сигналов, описаны в пакете `pack_mem`. Текст пакета приведен в листинге 5.5.

Листинг 5.5

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package pack_mem is
  type mem_state is (idle,adress,data1,datam,a_wait,a_wait1,waitt,wait1);
  type mem_r is (ready,nready);
end pack_mem;
```

Для моделирования используется сигнал clk1, который задержан на 1 нс. по отношению к сигналу тактирования. Сигнал tryadr используется для индикации правильности адреса в адресной фазе. Если его значение равно 1, то в фазе адреса память распознала обращение к себе и будет участвовать в дальнейшем обмене.

Сигналы mem_time и t_mem служат для управления процессом определения готовности памяти. В t_mem задается количество фаз ожидания, которое предшествует наступлению готовности памяти. В mem_time отражается текущее количество фаз ожидания. После того, как оно достигает значения t_mem, память переходит в состояние готовности.

Первый процесс (p_1) используется для управления вспомогательным сигналом тактирования. Второй процесс (p_2) используется для определения нового состояния памяти. Если память находится в состоянии Idle и сигнал Frame=0, то память переходит в состояние приема адреса (Address). Если память находится в состоянии Address, то, при условии готовности памяти к приему/передаче данных, она переходит в состояние данных (одиночная или множественная передача/прием в зависимости от frame). Если же память находится в состоянии неготовности, то она переходит в состояние ожидания. После состояния data1, при наличии frame=0, память может перейти в состояние адреса. После datam, если память готова, она может или вновь вернуться в datam (если продолжается множественный обмен данными), или перейти в data1 (если идет последняя фаза обмена). Если память не готова, то она переходит в одно из состояний ожидания (wait1 — если за ним будет следовать последняя фаза данных, или waitt — если за ним может следовать несколько фаз данных). Если память находится в одном из состояний ожидания, то, при наступлении сигнала готовности, она переходит в одно из состояний данных.

Третий процесс (p_3) служит для защелкивания адреса. Это действие выполняется в середине фазы адреса. Четвертый процесс (p_4) служит для преобразования адреса в номер ячейки памяти, что может быть сделано с помощью таблицы (как это приведено в листинге 5.4). Для выполнения этих действий могут быть использованы также функции преобразования. Процесс выполняется в каждой фазе адреса, при прочтении с AD в adress нового значения адреса.

Пятый процесс (p_5) служит для формирования значений на выходных линиях памяти. Если память в состоянии Idle, то выходные линии переводятся в пассивное состояние. Если память в состоянии фазы данных, то данные или выставляются на AD (при чтении из памяти), или считываются памятью с AD (при записи). Сигнал mem_time устанавливается в 1. Если память находиться в одном из состояний ожидания, то значение сигнала mem_time увеличивается на 1, что выполняется в шестом процессе (p_6).

Седьмой процесс (p_7) служит для записи данных в массив памяти. Это действие выделено в отдельный процесс, поскольку выполняется в середине такта, а не в начале.

Последний процесс (p_8) предназначен для отслеживания перехода памяти в состояние готовности. Данная модель организована таким образом, что память переходит в состояние готовности после заданного количества состояний ожидания.

Модель контроллера шины

Описание модели контроллера шины приведено в листинге 5.6.

Листинг 5.6

```
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity controller_PCI is
port(req1,req2:in std_ulogic;
        gnt1,gnt2: inout std_ulogic;
        frame: in std_ulogic;
        start:in bit);
end entity controller_PCI;
architecture bechavior  of controller_PCI is
signal flw: bit:='1';
signal fln: integer:=2;
begin
p_1: process(req1,req2,flw,start)
begin
if flw='1' then
   if Req1='1' then gnt1<='1'; gnt2<='0';
   elsif req2='1' then gnt2<='1'; gnt1<='0';
   else gnt1<='0'; gnt2<='0';
  end if;
-- вариант с динамическим приоритетом
--if (req1='1' and req2='0') or (req1='1' and fln=2) then
--  gnt1<='1';gnt2<='0'; fln<=1;
--elsif (req1='0' and req2='1') or (req2='1' and fln=1) then
--  gnt2<='1';gnt1<='0'; fln<=2;
--       end if;
```

```
end if;
end process p_1;

p_2: process (gnt1,gnt2)
begin
if (gnt1'event and gnt1='1') or (gnt2'event and gnt2='1') then flw<='0';
end if;
end process;
p_3: process (frame)
begin
if frame'event and not(frame='0') then gnt1<='0'; gnt2<='0';
  flw<='1'; end if;
 end process;
end;
```

Описываемый контроллер предназначен для работы с двумя ведущими устройствами. Его поведение описывается с помощью трех процессов.

Первый процесс (`p_1`) служит для определения ведущего устройства, которому будет предоставлена шина. В данной реализации первое ведущее устройство имеет больший приоритет.

Как вариант, в приведенном тексте модели закомментирована реализация, в которой ведущее устройство, только что использовавшее шину, имеет меньший приоритет, т. е. приоритет определяется динамически. Для этого применяется внутренний сигнал `fln`, в котором регистрируется номер ведущего устройства, использовавшее шину последним.

Первый процесс должен выполняться только перед фазой адреса (после завершения очередного обмена). Для отслеживания этого, используется внутренний сигнал `flw`, устанавливаемый в `'0'` вторым процессом после выбора очередного ведущего устройства, которое будет использовать шину. После того как ведущее устройство освобождает шину, этот сигнал в третьем процессе устанавливается в `'1'`.

Описание модели на верхнем уровне иерархии

Описание модели приведено в листинге 5.7.

Листинг 5.7

```
Library IEEE;
use IEEE.std_logic_1164.all;
entity model1 is end model1;
architecture structure of model1 is
```

```
-- описания компонентов
component master1 is
generic (towrite: std_ulogic_vector (31 downto 0);
port (clk: in bit;
        FRAME: out std_ulogic;
        AD: inout std_ulogic_vector (31 downto 0);
        CBE: out std_ulogic_vector(3 downto 0);
        IRDY: out std_ulogic;
        TRDY: in std_ulogic;
        DEVSEL: in std_ulogic;
        Req: out bit;
        Gnt: in bit);
end component master1;
component mem1 is
 port (AD: inout std_ulogic_vector (31 downto 0);
         IRDY: in std_ulogic;
         TRDY: out std_ulogic;
         DEVSEL: out std_ulogic;
         FRAME: in std_ulogic;
         C_BE: in std_ulogic_vector (3 downto 0);
         clk: in bit);
end component mem1;
component controller_PCI is
port(req1,req2:in std_ulogic;
       gnt1,gnt2: out std_ulogic;
       frame: in std_ulogic;
       start: in bit);
end component controller_PCI;
-- описание сигналов
signal sreq1: std_ulogic;
signal sreq2: std_ulogic:='0';
signal sgnt1,sgnt2:std_ulogic;
signal sframe: std_ulogic;
signal sC_BE: std_ulogic_vector (3 downto 0);
signal sDEVSEL:  std_ulogic;
signal sAD: std_ulogic_vector (31 downto 0);
signal sIRDY: std_ulogic;
signal sTRDY:  std_ulogic;
signal sCLK: bit;
```

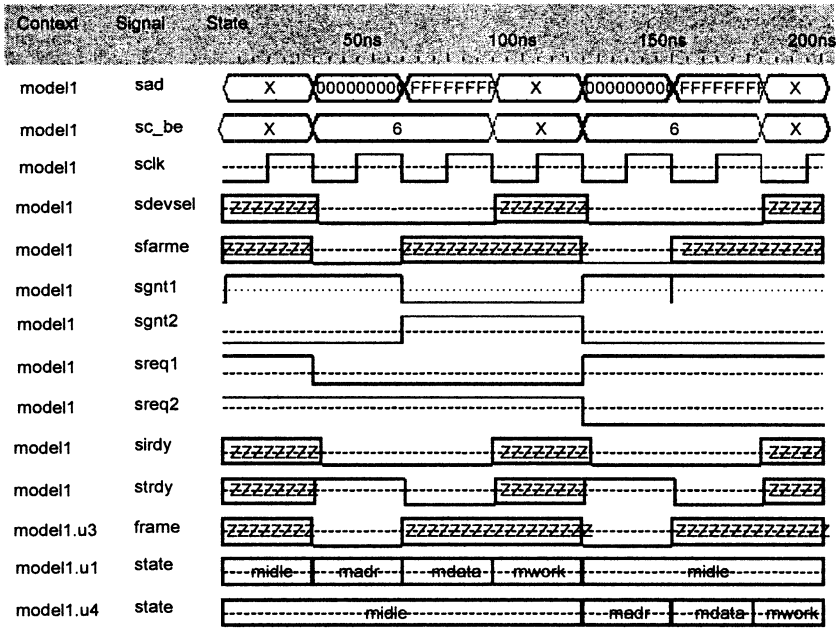
```
signal sstart: bit;
-- включение компонентов-экземпляров в структурное описание
-- архитектуры модели
begin
u1: master1
  generic map (towrite => "00000000000000000000000000000000")
port map (clk=>sclk, FRAME =>sframe, AD=>sAD, CBE=>sC_BE,
          IRDY=>sIRDY, TRDY=>strdy, DEVSEL=>sdevsel, Req=>sreq1,
         Gnt=>sgnt1);
u4: master1
  generic map (towrite => "10000000000000000000000000000000")
port map (clk=>sclk, FRAME =>sframe, AD=>sAD, CBE=>sC_BE,
          IRDY=>sIRDY, TRDY=>strdy, DEVSEL=>sdevsel, Req=>sreq2,
         Gnt=>sgnt2);
u2: mem1
 port map (AD=>sad, IRDY=>sirdy, TRDY=>strdy, DEVSEL=>sdevsel,
          FRAME=>sframe, C_BE=>sc_be, clk=>sclk);
u3: controller_PCI
port map(req1=>sreq1, req2=>sreq2, gnt1=>sgnt1, gnt2=>sgnt2,
                frame=>sframe, start=> sstart);
end architecture structure;
```

Рассмотрим теперь работу модели.

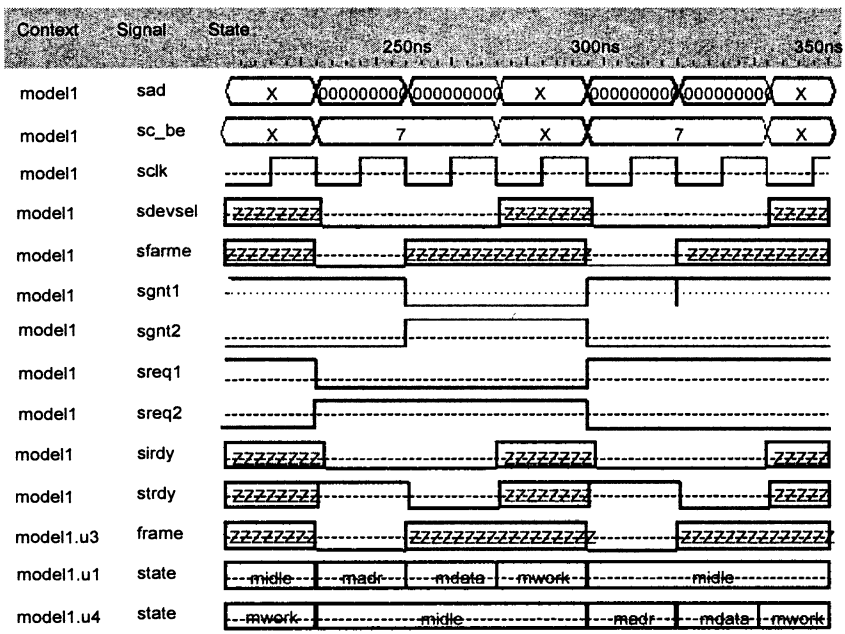
Диаграмма работы приведена на рис. 5.6 (*а*, *б*). Предполагается следующая последовательность моделируемых действий. Сначала к памяти обращается первое ведущее устройство, поскольку оно имеет наивысший приоритет. Оно выполняет операцию чтения из памяти. В фазе данных по шине прочитывается 11–1, поскольку таково начальное значение всех ячеек памяти. Далее первое ведущее устройство переходит в стадию обработки данных. В этот момент шину занимает второе ведущее устройство, которое также выполняет чтение данных из памяти. Затем первое ведущее устройство выполняет запись в память, после чего второе ведущее устройство выполняет запись в память.

Состояние памяти после моделирования представлено на рис. 5.7—5.9.

Второе ведущее устройство работает по тому же алгоритму, что и первое, но с той разницей, что в словах, которые второе устройство записывает в память, добавлена 1 в старшем разряде. Это позволяет отследить на временной диаграмме изменения, вносимые в память первым ведущим устройством, — первое изменение ячейки, и вторым мастером, — второе изменение ячейки. Для отражения разницы в значениях, записываемых в память двумя компонентами-экземплярами, — `u1` и `u4`, принадлежащих одному типу компонентов `master1`, использован механизм `generic`.



а



б

Рис. 5.6. Диаграмма работы модели (*а* – модельное время от 0 до 200 нс., *б* – модельное время от 200 нс. до 350 нс.).

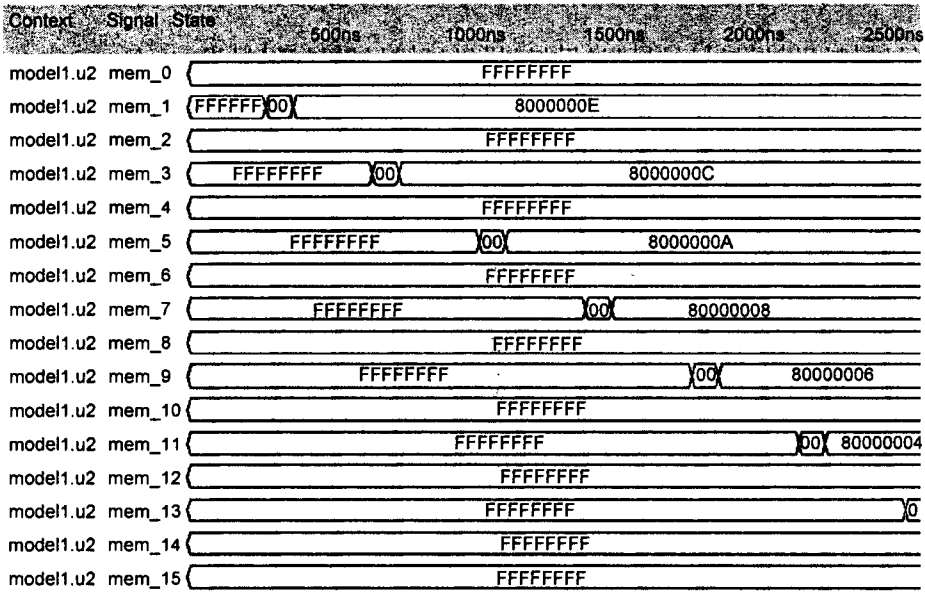


Рис. 5.7. Состояние памяти с 0 нс. по 2500 нс.

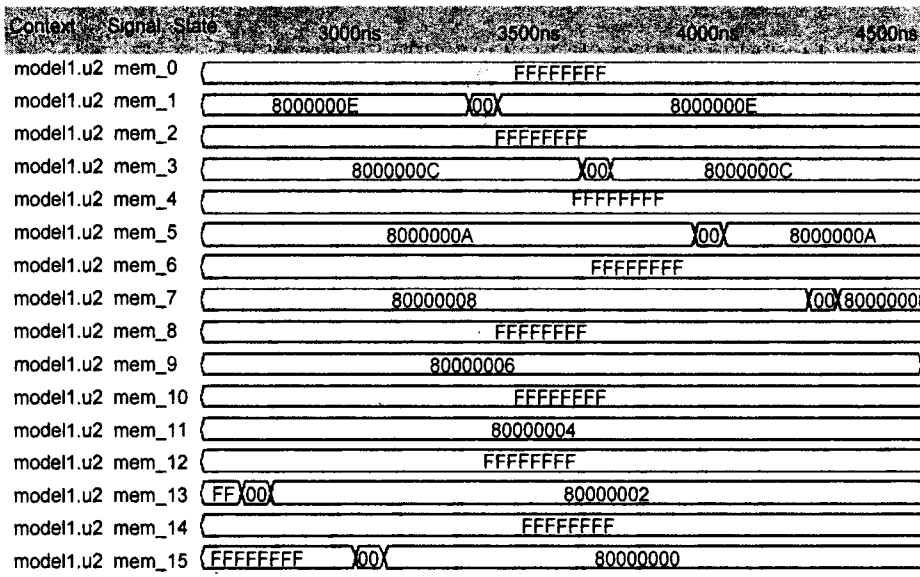


Рис. 5.8. Состояние памяти с 2500 нс. по 4500 нс.

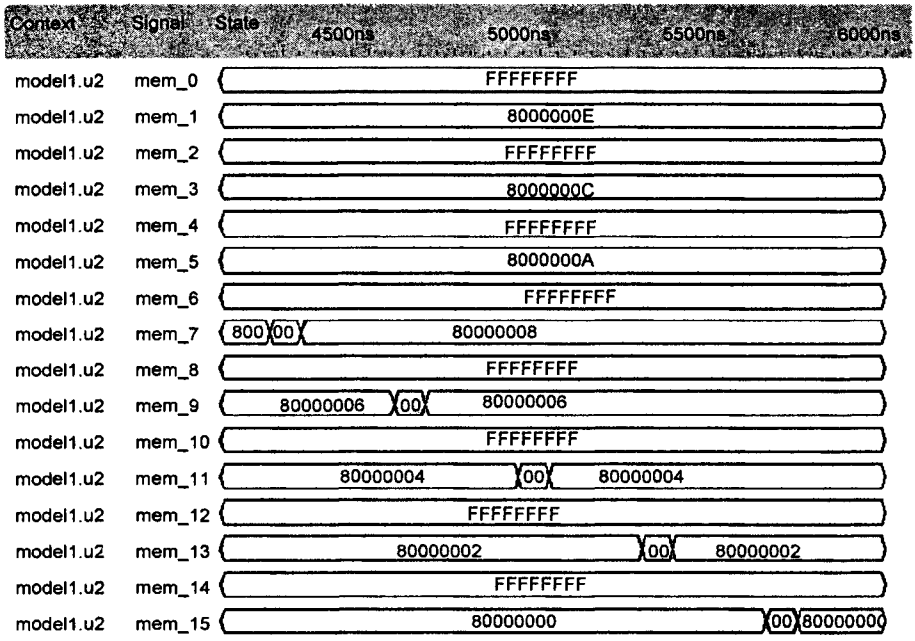


Рис. 5.9. Состояние памяти с 4500 нс. по 6000 нс.

Проектирование систем-на-кристалле на основе шины AMBA

Спецификация AMBA (Advanced Microcontroller Bus Architecture) [2], разработана как стандарт коммуникаций для высокопроизводительных систем-на-кристалле.

Шина AMBA — процессорнонезависимая шина, пригодная для использования с процессорными ядрами разных архитектур. Она разрабатывалась с учетом возможностей и специфики коммуникаций внутри кристалла, с ориентацией на минимизацию аппаратных затрат на кристалле для организации взаимодействия и пересылки информации между объединяемыми шиной модулями. Шина AMBA стимулирует разработку модульных систем, с возможностью использования процессорных ядер, блоков памяти и периферийных блоков различных архитектур. Объединение шиной AMBA готовых блоков дает прямой и относительно простой путь их интеграции в законченную систему-на-кристалле. Система шин AMBA разрабатывалась как инвариантная к технологиям реализации систем-на-кристалле, пригодная как для заказных, так и для полузаказных СБИС, для СБИС на вентильных матрицах и для ПЛИС (FPGA).

Стандарт шины AMBA определяет совокупность сигналов и протокол взаимодействия объединяемых шиной модулей. Будучи технологическинезависимой спецификацией, стандарт шины AMBA задает ее функционирование в терминах тактов сигналов синхронизации шины (AMBA — шина синхронная). Зависящие от технологии реализации СБИС характеристики, — электрические характеристики, временные характеристики, частота работы, не задаются стандартом на шину AMBA. Они определяются на этапе реализации проекта в СБИС. Задаваемые стандартом AMBA правила построения взаимодействия модулей на шине обеспечат корректную и эффективную работу проекта при различных технологиях его воплощения в кристалле.

Стандарт AMBA, протокол и организация шины хорошо согласуются с проектированием синтезируемых, параметризуемых модулей и систем-на-кристалле на их основе. Так, разрядности и число линий данных не фиксируются однозначно стандартом на шину AMBA. Для систем на плате, собираемых из готовых СБИС и блоков на их основе, такое умолчание стандарта о разрядности шин может служить препятствием для обеспечения совместимости независимо разработанных узлов. Для систем-на-кристалле наоборот, такая позиция разработчиков стандарта дает необходимую гибкость, адаптируемость применяемых готовых модулей, IP-блоков к требованиям конкретных проектируемых устройств на СБИС. Синтезируемые по описанию на языке VHDL, узлы легко параметризуются по параметрам такого рода.

Язык VHDL является необходимым средством для проектирования систем-на-кристалле, использующих такую идеологию проектирования гибких, параметризуемых к условиям применения модулей и компоновки из них Систем-на-кристалле.

Стандарт шины AMBA включает в себя спецификации трех шин.

- AHB — `Advanced High-performance Bus`. Эта шина предназначена для организации коммуникаций между быстродействующими модулями, обладающими сложной системой поведения, такими как процессорные ядра, контроллеры и блоки памяти.
- ASB — `Advanced System Bus`. Эта шина так же предназначена для организации системы коммуникаций между модулями, обладающими сложной системой поведения, но меньшим быстродействием.
- APB — `Advanced Peripheral Bus`. Эта шина предназначена для организации взаимодействия ядра системы-на-кристалле с периферийными блоками. Протокол обмена по этой шине упрощен, по сравнению с протоколами AHB и ASB.

В стандарте AMBA предусмотрено три варианта физической реализации подключения устройств: по схеме с тремя состояниями, по схеме с мультиплексированием выходов, по схеме с логикой OR. Для систем-на-кристалле, в соответствии с требованиями технологии, наиболее широкое применение получила схема с мультиплексированием выходов.

В настоящее время коммуникационные системы на базе стандарта AMBA находят широкое применение в системах-на-кристалле аэрокосмического назначения. Например, шины AMBA используются для организации системы коммуникаций в системах-на-кристалле на процессорном ядре LEON, организованном в соответствии с архитектурой SPARC V8 [7]. Шина AMBA AHB используется и в разрабатываемых отечественных системах-на-кристалле, например, в рамках проекта "Мультикор" [21].

Высокоскоростная шина AMBA AHB для систем-на-кристалле

Этот стандарт разработан для построения высокопроизводительных систем. Обмен данными в соответствии с этим стандартом осуществляется в синхронном режиме. В рамках стандарта предусмотрена поддержка пакетных передач и расщепленных транзакций (`split transactions`). В системе должно быть не более 16 ведущих устройств, число ведомых устройств не ограничено. Организация коммуникаций по шине осуществляется под управлением арбитра.

Поскольку в настоящее время для построения систем-на-кристалле используется, в основном, схема подключения устройств к шине с мультиплексированием выходов, в дальнейшем в этой главе будет рассматриваться именно она.

Каждое из устройств шины AHB может принадлежать к одному из следующих типов:

- *ведущее устройство*, способное начинать циклы обмена по шине;
- *ведомое устройство*, способное отвечать на запросы ведущего устройства;
- *арбитр* — устройство, управляющее предоставлением доступа ведущих и ведомых устройств к шине в соответствии с выбранной схемой арбитража.

Рассмотрим назначение сигналов, входящих в состав AHB. Можно выделить следующие основные группы сигналов:

- *адрес* — по этим линиям происходит передача адреса от ведущих устройств к ведомым;
- *управление* — по этим линиям происходит передача управляющей информации от ведущих *устройств к ведомым;*
- *данные* для записи — по этим линиям происходит передача данных от ведущих устройств для записи в ведомые;
- *данные для чтения* — по этим линиям происходит передача данных от ведомых устройств к ведущим;
- *сигналы подтверждения* — по этим линиям передается информация о состоянии от ведомых устройств к ведущим.

Шина AMBA AHB является шиной с раздельными линиями для адреса и для данных. Более того, линии данных разделены на две независимые группы: линии данных для передачи от ведущих устройств к ведомым и линии данных для передачи в обратном направлении — от ведомых устройств к ведущим.

Кроме дополнительных возможностей распараллеливания информационных потоков в системе, такая организация шин адреса и данных, основных по числу линий, определяет симплексный, однонаправленный характер большинства линий на шине AHB. Это является важным положительным фактором, и с точки зрения скоростных характеристик, и с точки зрения аппаратных затрат при их схемотехнической и топологической реализации в кристалле, а также для разработки модулей, soft- и firm-IP-блоков, инвариантных к схемотехнике и специфике технологий реализации в кристалле [1].

Для групп сигналов 1—3 источниками являются ведущие устройства. Их выходы подключаются на шины через мультиплексоры, управление которыми осуществляет арбитр. Подключение нескольких источников к шине через мультиплексор иллюстрирует рис. 5.10. Каждое ведущее устройство связано с арбитром выделенными линиями запроса шины и подтверждения получения шины. Арбитр, в соответствии с выбранной в системе схемой арбитрирования, определяет, какому ведущему устройству будет предоставлена шина.

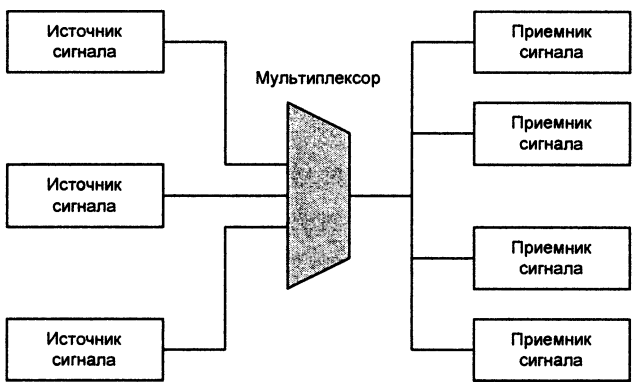


Рис 5.10. Схема подключения нескольких источников сигнала к шине через мультиплексор.

На эти шины подключаются входы всех ведомых устройств. Обработку сигналов осуществляет то ведомое устройство, которое выбирается арбитром в соответствии с адресом, который выставленным ведущим устройством, получившим доступ к шине. Каждое ведомое устройство связано с арбитром выделенной линией выбора ведомого устройства. Ведомые устройства, не получившие сигнал выбора, игнорируют значения сигналов на своих входах.

Для групп сигналов 4—5 источниками являются ведомые устройства. Их выходы также подключаются на шину через мультиплексоры, управление

которыми осуществляет арбитр. Входы всех ведущих устройств подключаются к этим шинам, однако обработку сигналов осуществляет только то ведущее устройство, которому арбитром предоставлена шина. Остальные ведущие устройства игнорируют эти сигналы.

Сигналы шины AMBA AHB

На рис. 5.11 представлены интерфейсы ведущего устройства, ведомого устройства и арбитра для шины AMBA AHB.

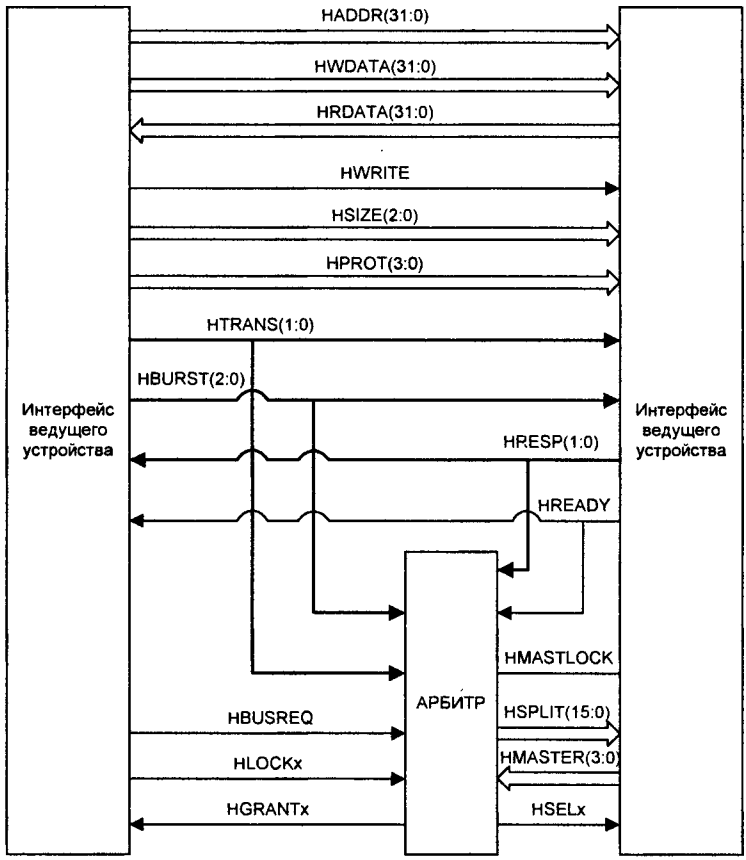


Рис. 5.11. Интерфейсы ведомого устройства, ведущего устройства и арбитра на шине AMBA AHB

Перечень и описание сигналов шины AHB приведены в табл. 5.1. В этой таблице в графе "Источник сигнала" использованы следующие обозначения:

- `Sys` — системный сигнал;
- `м` — источником сигнала является ведущее устройство;

- S — источником сигнала является ведомое устройство;
- A — источником сигнала является арбитр.

Таблица 5.1. *Сигналы шины AHB*

Название	Источник сигнала	Описание
HCLK	Sys	Системный сигнал тактирования. Значения всех сигналов на шинах анализируются по восходящему фронту сигнала тактирования
HRESETn	Sys	Системный сигнал сброса. Активным считается низкий уровень сигнала
HADDR[31..0]	M	Адресная шина
HTRANS[1..0]	M	Тип текущей передачи, см. табл. 5.2
HWRITE	M	Направление обмена. 1 — запись данных в ведомое устройство, 0 — чтение данных из ведомого устройства
HSIZE[2..0]	M	Разрядность передаваемого слова данных, см. табл. 5.3
HBURST[2..0]	M	Индикатор пакетной передачи, см. табл. 5.4
HPROT[2..0]	M	Указатель типа защиты. Предоставляет дополнительную информацию о типе выполняемого запроса, см. табл. 5.5.
HWDATA[hd..0]	M	Шина передачи данных от ведущего устройства к ведомому устройству
HSELx	A	Сигнал выборки ведомого устройства. Для каждого ведомого устройства используется выделенная линия для передачи этого сигнала
HRDATA[hd..0]	S	Шина передачи данных от ведомого устройства к ведущему
HREADY	S	Сигнал завершения передачи, 1 указывает на завершение ведомым устройством обработки текущей передачи
HRESP[1..0]	S	Информация о статусе выполнения передачи, см. табл. 5.6
HBUSREQx	M	Сигнал запроса шины ведущим устройством. Для каждого ведущего устройства используется выделенная линия
HLOCKx	M	Сигнал блокировки шины. Если значение этого сигнала установлено в 1, то шина должна быть предоставлена ведущему устройству до тех пор, пока значение этого сигнала не будет сброшено в 0. Для каждого ведущего устройства используется выделенная линия

***Таблица 5.1** (окончание)*

Название	Источник сигнала	Описание
HGRANTx	A	Установка 1 на этой линии указывает ведущему устройству, на то, что оно получило наивысший приоритет в использовании шины. Ведущее устройство может начать использовать шину, если его сигнал HGRANT установлен в 1 только после того, как сигнал HREADY также будет установлен в 1, что указывает на завершение выполнения предыдущего обмена данными
HMASTER[3..0]	A	Идентификатор (номер) ведущего устройства, которое в данный момент использует шину
HMASTLOCK	A	Если этот сигнал установлен в 1, этоозначает, что ведущее устройство, которое в данный момент использует шину, выполняет запрос с блокировкой
HSPLITx[15..0]	S	Каждая линия этой шины соответствует одному ведущему устройству с соответствующим номером. Если ведомое устройство устанавливает 1 на какой-либо линии, это указывает на его готовность к завершению расщепленной транзакции от соответствующего ведущего устройства

Шина данных в AMBA AHB разделена на две однонаправленные шины — HRDATA и HWDATA. Для ведущих устройств шина HRDATA является входной (по ней ведущее устройство читает, принимает данные из памяти), а шина HWDATA — выходной (по ней ведущее устройство пересылает данные для записи в ведомом устройстве), рис. 5.11. Для ведомых устройств, наоборот, шина HRDATA является выходной, а шина HWDATA — входной. Эта логика наименования шин данных понятна, если представить себе в качестве примера ведущего устройства процессор, а в качестве ведомого — память.

Стандартом AMBA AHB предусматривается возможность различной разрядности шин данных. Протоколом шины определен числовой ряд возможной разрядности шин данных: 8, 16, 32, 64, 128, 256, 512 и 1024 разряда. Так что в таблице 5.1 параметр *hd* может быть, соответственно равным 7, 15, 31, 63, 127, 255, 511, 1023. Типовой разрядностью шин данных считается 32 разряда, *hd* =31. Разрядность шин HRDATA и HWDATA — одинакова.

Рассмотрим более подробно назначение управляющих сигналов. Сигнал HTRANS предназначен для указания типа текущей передачи. Значения, которые может принимать этот сигнал, описаны в табл. 5.2.

Таблица 5.2. Коды типа передачи, HTRANS[1:0]

Значение	Название	Описание
00	IDLE	Ведущее устройство не требует обработки данных. Ведомое устройство должно в этом случае выставить HRESP=OKAY
01	BUSY	Позволяет ведущему устройству вставить IDLE-циклы (пустые циклы) в середине пакетного обмена. Когда ведущее устройство выдает BUSY, оно должно выдавать адрес и команду управления для следующей передачи в пакете. В ответ на это ведомое устройство должно выдать OKAY
10	NONSEQ	Указывает на первую передачу в пакете или одиночную передачу. Сигналы адреса и управления не связаны с предыдущими передачами
11	SEQ	Идет последовательная передача. Адрес связан с предыдущим. Управляющая информация такая же, как и при предыдущей передаче

Сигналы HSIZE позволяют указать разрядность передаваемого слова данных. Описание значений этих сигналов приведено в табл. 5.3.

Таблица 5.3. Код разрядности передаваемых данных, HSIZE

Значение	Название	Описание
000	8 бит	Передача одного байта
001	16 бит	Передача полуслова
010	32 бита	Передача слова
011	64 бита	Передача двойного слова
100	128 бит	Передача четырехсловной строки
101	256 бит	Передача восьмисловной строки
110	512 бит	Передача 16-словной строки
111	1024 бит	Передача 32-словной строки

Сигнал HBURST — позволяет определить тип пакета при выполнении пакетной передачи (burst transaction). Возможные типы приведены в табл. 5.4.

Таблица 5.4. *Код типа пакета, HBURST*

Значение	Название	Описание
000	SINGLE	Одиночная передача
001	INCR	Инкрементируемая пакетная передача нефиксированной длины. Адрес каждого последующего слова получается путем инкрементирования адреса предыдущего
010	WRAP4	4-словный пакет, перебор адресов с оборотом по границе области адресного пространства
011	INCR4	4-словный пакет, последовательный перебор адресов
100	WRAP8	8-словный пакет, перебор адресов с оборотом по границе области адресного пространства
101	INCR8	8-словный пакет, последовательный перебор адресов
110	WRAP16	16-словный пакет, перебор адресов с оборотом по границе области адресного пространства
111	INCR16	16-словный пакет, последовательный перебор адресов

В пакетной передаче пересылаются несколько слов. Код типа пакета `HBURST[2..0]` определяет, сколько слов передается в пакете, как перебираются адреса пересылаемых слов. На шине AHB предусмотрено два режима перебора адресов последовательно пересылаемых по шине слов.

Один режим — это режим `increment`, `INCRx`, обычный режим последовательного перебора адресов пересылаемых слов (разрядность слова определяется кодом `HSIZE`).

Второй режим — режим `wrapping`, `WRAPx` —режим последовательного перебора адресов пересылаемых слов, с оборотом по границе области адресного пространства. Размер области адресного пространства определяется числом слов в пакете, умноженным на число байтов в слове. Этот размер всегда есть некоторая степень двойки, что естественным образом определяет границы области в адресном пространстве как набор адресов — от XXX000 до XXX11..1. В этом режиме, если начальный адрес передачи не выровнен по границе области адресного пространства, соответствующей размеру запроса, то адреса пересылаемых слов перебираются последовательно от заданного адреса до границы области адресного пространства, а по достижении границы, происходит оборот перебора адресов, и адреса продолжают перебираться, начиная с начального адреса этой области. Например, пусть запрашива-

ются 4 слова по 4 байта; размер передаваемого пакета — 16 байт. Если заданный в начале обмена адрес является адресом первого передаваемого слова (0x34), то потом передаются слова с адресами 0x38, 0x3C (достигнута граница), 0x30.

Сигнал HPROT[3..0] позволяет указать режим защиты. Он определяет, какого рода данные обрабатываются: идет ли выборка кода операции, выполняется ли запрос в пользовательском или привилегированном режимах. Обработка этих сигналов определяется устройствами, участвующими в обмене. Описание возможных значений этого сигнала приведено в табл. 5.5.

Таблица 5.5. *Коды режима защиты, HPROT*

HPROT[3] cache-able	HPROT[2] Buffer-able	HPROT[1] Privileged	HPROT[0] Data/opcode	Описание
—	—	—	0	Выборка кода операции
—	—	—	1	Выборка данных
—	—	0	—	Запрос в пользовательском режиме
—	—	1	—	Запрос в привилегированном режиме
—	0	—	—	Небуферируемый запрос
—	1	—	—	Буферируемый запрос
0	—	—	—	Некэшируемый запрос
1	—	—	—	Кэшируемый запрос

Сигнал HREADY используется подчиненным устройством для указания его готовности завершить текущий обмен данными. Подчиненное устройство определяет завершение первой фазы адреса обмена по сигналу HREADY='1'. Первая фаза адреса текущего обмена может совпадать с последней фазой данных предыдущего обмена, в котором участвовало другое подчиненное устройство. По этой причине, если в системе имеется более одного подчиненного устройства, то каждое такое устройство должно иметь также и входной сигнал HREADY для того, чтобы отслеживать состояние выходного сигнала HREADY от подчиненного устройства, которое участвовало в предыдущем обмене (на рис. 5.10 это не отражено).

Сигнал HRESP используется ведомым устройством для указания статуса выполнения передачи. Его возможные значения приведены в табл. 5.6.

Таблица 5.6. Коды состояния, HRESP

Значение	Название	Описание
00	OKAY	Указывает, что выполнение передачи идет успешно. Если при этом HREADY='1' — передача успешно завершена, если HREADY='0' — необходимы дополнительные такты для завершения передачи
01	ERROR	Указывает на возникновение ошибки во время передачи. Этот сигнал должен быть выставлен в течение 2-х тактов
10	RETRY	Указывает ведущему устройству на необходимость повторной передачи (ведущее устройство должно выполнять повторные передачи до тех пор, пока этот сигнал не будет снят). Этот сигнал должен быть выставлен в течение 2-х тактов
11	SPLIT	Указывает ведущему устройству на невозможность выполнения запроса в данный момент времени. В этом случае арбитр может на время передать шину другим ведущим устройствам. Ведомое устройство указывает свою готовность завершить запрос с использованием линий HSPLIT. Этот сигнал должен быть выставлен в течение 2-х тактов

Организация обменов по шине AHB

Каждый запрос на шине AHB состоит из фазы адреса и фазы данных. В отличие, например, от шины PCI, шина AMBA предусматривает использование отдельных линий адреса и данных. Поэтому на шине AHB фаза адреса последующего запроса перекрывается фазой данных предыдущего запроса, что увеличивает пропускную способность шины. Таким образом, возможны ситуации, когда шины данных используются одним ведущим устройством, а шины адреса и управления — другим. Фаза адреса продолжается в течение одного такта и не может быть расширена. Продолжительность фаз данных может управляться ведомым и ведущим устройством, участвующими в обмене.

После прихода восходящего фронта тактового сигнала ведущее устройство, которому предоставлено право использования шины, выставляет действительный адрес и команду управления на соответствующие шины.

По следующему восходящему фронту тактового сигнала ведомое устройство запоминает адрес и команду управления. Собственно обмен данными выполняется, начиная со следующего такта, в соответствии с выставленной командой управления.

Ведущее устройство определяет количество слов данных, входящих в текущий запрос. Для этого оно использует сигнал HBURST. Могут быть запросы к одиночному слову или пакетные запросы. Пакетные запросы, в свою очередь, можно разделить на запросы с заранее определенным количеством слов — запросы определенной длины, и запросы неопределенной длины. Если выполнение пакетного запроса не должно прерываться (ведущее устройство выставляет сигнал HLOCK), то запрос является запросом с блокировкой.

Для указания характера выполняемых действий ведущее устройство использует линии HTRANS. Если ведущее устройство не требует от ведомого устройства выполнения каких-либо операций, оно выставляет HTRANS=IDLE. Как правило, это используется, если ведущее устройство получило шину как ведущее устройство по умолчанию, однако, в данный момент ему не нужно использовать ее. Если ведущему устройству необходимо вставить пустые такты в середине запроса, оно выставляет HTRANS=BUSY. Это может использоваться, например, в том случае, если ведущее устройство не успело обработать данные, принятые от ведомого устройства. Если ведущее устройство выполняет первый обмен в пакете или одиночный обмен, оно выставляет на эти линии HTRANS=NONSEQ. Это позволяет указать ведомому устройству, что полученный им адрес и управление не связаны с предыдущим обменом. Если ведущее устройство выполняет очередной обмен в рамках пакетного обмена, оно выставляет на линии HTRANS сигнал SEQ. Это указывает ведомому устройству, что идет последовательная передача, адрес связан с предыдущим, а управляющая информация та же, что и в предыдущем обмене.

Рассмотрим влияние ведомого устройства на выполнение запроса. Ведомое устройство может обозначить необходимость дополнительных тактов в фазе данных сигналом HREADY. Интерпретация значения этого сигнала ведущим устройством и арбитром выполняется в сочетании с интерпретацией сигнала HRESP. Для ведомого устройства возможны разные схемы действий. Ведомое устройство может обработать текущий обмен данными в течение одного такта или ему для этого может потребоваться несколько тактов — в этих случаях обмен данными может быть завершен успешно. Ведомое устройство может оказаться не в состоянии выполнить обмен по каким-либо причинам — в этом случае обмен завершается с ошибкой, и на линии HRESP выставляется сигнал ERROR. Если внутри ведомого устройства для выполнения указанного вида обмена может потребоваться много тактов работы, оно может отрабатывать необходимые действия, не блокируя шину AHB в течение этого промежутка времени. Это позволяет увеличить реальную пропускную способность шины. Для этого ведомое устройство может воспользоваться механизмом повторных передач или механизмом расщепления транзакции.

Механизм *повторных передач* используется, как правило, в тех случаях, когда на момент обращения к ведомому устройству оно занято выполнением непрерываемых внутренних функций и не может ответить на запрос от ведущего устройства немедленно, либо ему на обработку запроса требуется много времени. При этом, в течение данного времени, ведущее устройство также не сможет обрабатывать запросы от других ведущих устройств. Арбитр при получении от ведомого устройства подтверждения, указывающего на использование механизма повторной передачи (HRESP=RETRY), может предоставить шину другому ведущему устройству, если в системе имеются запросы на шину. В этом случае арбитр обрабатывает так называемую нормальную схему приоритетов, когда право использовать шину может получить только ведущее устройство, приоритет которого выше приоритета того устройства, в ответ на запрос которого было получено HRESP=RETRY. Ведущее устройство, к запросу которого был применен механизм повторных передач, продолжает запрашивать шину. Когда оно получает ее вновь, то может продолжить выполнение запроса с того места, на котором он был прерван.

Механизм *расщепленных транзакций* (split transaction) используется в том случае, если ведомому устройству для обработки данного запроса требуется много времени, но оно при этом может выполнять запросы от других ведомых устройств. Арбитр, при получении от ведомого устройства подтверждения, указывающего на использование этого механизма (HRESP=SPLIT), предоставляет шину другому ведущему устройству, которое в данный момент запрашивает шину, или ведущему устройству по умолчанию, если в данный момент никакое другое ведущее устройство ее не запрашивает. Ведущее устройство, транзакция которого была расщеплена, лишается возможности использовать шину до тех пор, пока само ведомое устройство не выставит сигнал готовности завершить расщепленную транзакцию. Ведомое устройство, которое инициировало выполнение расщепленной транзакции, может обрабатывать запросы от других ведущих устройств, не завершая начатой расщепленной транзакции.

При проектировании ведомых устройств и системы в целом необходимо учитывать, что SPLIT и RETRY могут привести к блокировке шины. Одиночная передача не может привести к блокировке, поскольку ведомые устройства должны проектироваться таким образом, чтобы завершить передачу за конечное количество циклов. При SPLIT и RETRY ведущее устройство практически блокируется, причем в общем случае на неопределенное время.

Обработка сигнала сброса HRESETn

Этот сигнал является сигналом сброса для всех элементов шины. Активным уровнем сигнала сброса является низкий уровень. Сигнал может выставляться асинхронно, но отрабатываться должен синхронно, по восходящему

фронту тактового импульса. По этому сигналу все ведущие устройства должны выставить на линии адреса и управления действительные значения. HTRANS[1:0] должен быть установлен в IDLE.

Управление доступом к шине. Арбитр шины AHB

Механизм арбитража обычно должен гарантировать, что в один момент времени только одно ведущее устройство будет работать с шиной. Функции арбитра на шине AHB несколько шире:

- выбор ведущего устройства, которому будет предоставлена шина;
- выбор ведомого устройства, которое будет участвовать в обмене;
- контроль обменов с блокировкой, завершения выполнения запросов со статусом RETRY и ERROR, расщепленных транзакций со статусом SPLIT.

Выбор ведущего устройства, которому будет предоставлена шина

Рассмотрим, как осуществляется выбор ведущего устройства, которому будет предоставлена шина. Как правило, арбитр предоставляет шину следующему ведущему устройству только по завершении предыдущего обмена.

Следующее ведущее устройство получает право использовать линии адреса и управления во время выполнения последней фазы данных предыдущего обмена. Возможность использовать линии данных оно получит после завершения последней фазы данных предыдущего обмена. Однако арбитр может завершить пакетный обмен и по своей инициативе, раньше, чем это сделало бы ведущее устройство,— в том случае, если арбитр получает запрос шины от ведущего устройства с более высоким приоритетом. Такое досрочное, принудительное завершение пакетного обмена недопустимо только в случае, если выполняется запрос, для которого выставлен сигнал HLOCK.

Таким образом, арбитр должен отслеживать состояние выполняемого запроса. Арбитр анализирует значение сигнала HBURST. Если выполняется одиночная передача, арбитр сразу же (в ходе выполнения фазы адреса текущего обмена) может определить ведущее устройство, которому будет предоставлена шина для следующего обмена.

Если выполняется запрос фиксированной длины, то арбитр считает количество выполненных передач. При выполнении предпоследней передачи запроса он определяет ведущее устройство, которому будет предоставлена шина для следующего обмена. Во время выполнения последней передачи, линии адреса и управления предоставляются следующему ведущему устройству.

Если ведущее устройство выполняет запрос неопределенной длины, то оно должно держать сигнал запроса шины до начала последней передачи. Снятие этого сигнала указывает арбитру, что он может приступить к определению ведущего устройства для следующего обмена данными.

Арбитр считает, что очередная передача окончена, если ведомое устройство устанавливает сигнал HREADY в '1'.

Рассмотрим последовательность действий, выполняемых при предоставлении шины очередному ведущему устройству.

Ведущее устройство, которому необходимо использовать шину, может выставить сигнал запроса HBUSREQx в любой момент времени. Если этот запрос должен быть блокируемым (его выполнение не должно прерываться), то, одновременно с этим сигналом, ведущим устройством должен выставляться сигнал HLOCKx. Арбитр, определив, что далее шина может быть предоставлена следующему ведущему устройству, по восходящему фронту тактового сигнала просматривает имеющиеся запросы. Арбитр использует некоторый алгоритм арбитража, не специфицируемый стандартом на шину AMBA AHB, для выбора ведущего устройства, которому будет предоставлена шина. Для этого ведущего устройства арбитр выставляет сигнал HGRANTx. При нормальной схеме арбитража (арбитр не прерывает обменов по своей инициативе) HGRANTx выставляется следующему ведущему устройству, когда текущее ведущее устройство выставляет адрес последнего слова в обмене. Если ни одно из ведущих устройств не запрашивает шину, арбитр предоставляет ее устройству, являющемуся ведущим по умолчанию. Поэтому важно, чтобы все ведущие устройства, которые не запрашивают шину, выставляли на HTRANS тип IDLE. Тогда, если ведущее устройство в данной системе является ведущим по умолчанию, получение им шины по умолчанию не спровоцирует обмена данными, которого оно не запрашивало.

Ведущее устройство может считать, что ему предоставлены линии адреса и управления, когда по переднему фронту сигнала тактирования HGRANTx=1 и HREADY=1, что указывает на окончание предпоследней фазы предыдущего запроса. В этот же момент арбитр должен выставить на HMASTER[3..0] идентификатор ведущего устройства, которое получило шину (линии HMASTER[3..0] могут использоваться для управления мультиплексорами шин адреса и управления). Если ведущее устройство, которому предоставляются шины адреса и управления, выставляло сигнал HLOCK, то, одновременно с установкой сигнала HMASTER, арбитр должен выставить сигнал HMASTLOCK, указывающий на то, что будет выполняться блокируемая последовательность передач.

Однако шины данных пока еще предоставлены предыдущему ведущему устройству. По ним идет передача последнего слова в обмене. Уже получив права на шину адреса и управления, прежде чем использовать еще и линии данных, ведущее устройство должно вновь дождаться появления сигнала HREADY=1, что указывает на окончание последней фазы предыдущего запроса.

Выбор ведомого устройства, которое будет участвовать в обмене

На шине AHB арбитр выполняет также декодирование адреса и определяет ведомое устройство, которое будет участвовать в обмене. Эти действия выполняются в начале каждого нового запроса на шине. Каждое ведомое устройство связано с арбитром отдельной, выделенной линией HSELx, по которой оно получает сигнал разрешения работы. В декодировании участвуют старшие разряды адреса. Минимальное адресное пространство, которое может соответствовать одному ведомому устройству — 1 Кбайт. Работа ведущих устройств при выполнении пакетных запросов должна быть организована таким образом, чтобы в ходе пакетного обмена они не переходили через границу в 1 Кбайт, чтобы не возникло ситуации, когда ведомое устройство меняется в середине пакетного запроса. Если система организована таким образом, что используется не все адресное пространство, она должна содержать некоторое ведомое устройство по умолчанию, которое бы выдавало ответы при обращении к несуществующим адресам. Функции этого ведомого устройства могут быть возложены на арбитра. Если по несуществующему адресу выполняется запрос типа NONSEQUENTIAL или SEQUENTIAL, ведомое устройство по умолчанию должно выдать подтверждение типа ERROR. Если запрос имел тип BUSY или IDLE, то должно быть выдано подтверждение OKAY при нулевом состоянии ожидания.

Ведомое устройство при получении сигнала HSELx должно защелкнуть адрес и управляющую информацию в том такте, в котором ведомое устройство, участвующее в предыдущем обмене, выставило '1' на линии HREADY, указывающую, что обмен завершен.

Контроль выполнения обменов с блокировкой

Если при запросе шины ведущее устройство выставило сигнал HLOCKx, это указывает арбитру на то, что выполнение запрошенного обмена не может быть прервано с момента начала до момента сброса этого сигнала. После того, как обмен с блокировкой завершен, арбитр должен сохранить шину за тем же ведущим устройством на дополнительный обмен. Это необходимо для того, чтобы гарантировать, что завершена обработка данных, соответствующих последней фазе данных обмена с блокировкой, и что не появится подтверждение типа SPLIT или RETRY. Поэтому рекомендуется (это не обязательное требование), чтобы ведущее устройство, после обмена данными с блокировкой, вставляло один обмен с типом IDLE. Арбитр также ответственен за формирование сигнала HMASTLOCK, который должен выставляться тогда же, когда и сигнал адреса и управления.

Рассмотрим, как арбитр обрабатывает завершения обменов, в которых ведомое устройство на линии HRESP выставило подтверждение, отличное от OKAY. Завершение обмена с подтверждением OKAY занимает один такт, во всех остальных случаях — два такта. В первом из этих тактов ведомое уст-

ройство должно выставить соответствующий сигнал подтверждения и HREADY=0. Во втором такте — выставить HREADY=1 для указания завершения обмена; HRESP[1..0] при этом должен оставаться без изменений.

Если HRESP=ERROR — возникла ошибка, для следующего запроса арбитр может предоставить шину любому ведущему устройству, в том числе и тому, запрос которого только что закончился с ошибкой (если оно продолжает выставлять сигнал запроса шины).

Если HRESP=RETRY, для следующего обмена шина может быть предоставлена только ведущему устройству, приоритет которого выше, чем приоритет устройства, которое получило подтверждение RETRY. Если ни одно из ведущих устройств с более высоким приоритетом не запрашивает шину, то она остается за прежним ведущим устройством (получившим подтверждение RETRY).

Расщепленные транзакции

Ведомое устройство, для выполнения расщепленной транзакции, выставляет на линии подтверждения HRESP сигнал SPLIT. После получения такого кода состояния арбитр должен заблокировать сигнал запроса шины ведущего устройства, которому оно предназначалось, до тех пор, пока ведомое устройство не выставит сигнал готовности завершить расщепленную транзакцию на соответствующий разряд сигнала HSPLIT. Как правило, для этих целей арбитр содержит специальный регистр маски, каждый бит которого соответствует одному ведущему устройству. При возникновении этой ситуации арбитр определяет ведущее устройство, которому будет предоставлена шина для следующего запроса. Это может быть любое ведущее устройство, для которого в настоящий момент нет расщепленных транзакций.

Поведение ведущего устройства в ходе обмена

Если ведущему устройству необходимо использовать шину, оно выставляет сигнал запроса шины HBUSREQx. Этот сигнал выставляется асинхронно и может быть выставлен в любой момент времени. Если должен быть выполнен обмен с блокировкой, то одновременно с HBUSREQx должен быть выставлен сигнал HLOCKx. После того как ведущее устройство получает сигнал HGRANTx, прежде чем начать фазу адреса, оно должно дождаться установки сигнала HREADY в '1', что указывает на завершение последней фазы адреса предыдущего запроса. После этого начинается первая фаза адреса нового обмена. Поскольку выходы ведущих устройств подключаются на шину через мультиплексор, управляемый арбитром, ведущее устройство может выставлять адрес и соответствующие ему сигналы управления одновременно с сигналом запроса шины HBUSREQx. Первая фаза адреса продолжается до тех пор, пока не будет завершена последняя фаза данных предыдущего запроса, т. е. до тех пор, пока сигнал HREADY в очередной раз не будет установлен в 1. Затем начинается первый обмен данными нового запроса. Теперь веду-

щее устройство может сбросить сигнал запроса шины HBUSREQx, если оно выполняет обмен одиночным словом данных или пакетный обмен фиксированной длины, т. е. HBURST≠INSR. В противном случае сигнал HBUSREQx должен оставаться активным до начала последней фазы обмена данными. Если ведущее устройство намерено выполнить еще один запрос после того, который выполняется в данный момент, оно должно вновь выставить сигнал запроса шины HBUSREQx в течение выполнения текущего запроса.

Рассмотрим, как выполняется один обмен данными в ходе выполнения запроса. Адрес и управляющие сигналы для этого обмена были выставлены ведущим устройством и защелкнуты ведомым в предыдущем такте (т. е. в последнем такте предыдущего обмена данными). В первом такте текущего обмена данными ведущее устройство выставляет адрес и управляющие сигналы для следующего обмена данными, если текущий обмен данными не последний в этом запросе. Если в ходе текущего обмена должна быть выполнена запись данных в ведомое устройство, ведущее устройство выставляет данные. Далее, по восходящему фронту тактового импульса, ведущее устройство анализирует значение сигналов HRESP и HREADY от ведомого устройства. Если HRESP=OKAY и HREADY='1' — текущий обмен считается успешно завершенным. Если это была запись данных в ведомое устройство, то данные успешно записаны; если это было чтение данных из ведомого устройства, то оно выставило действительные данные на шину, и по рассматриваемому переднему фронту тактового импульса они могут быть записаны ведущим устройством.

Если HRESP=OKAY и HREADY='0', — значит, ведомое устройство запрашивает дополнительные такты для завершения текущего обмена данными. В этом случае ведущее устройство должно оставить все выставленные им сигналы без изменений.

Если HRESP=ERROR, то ведущее устройство должно выполнить свой алгоритм обработки ошибки. При возникновении такой ситуации, в зависимости от используемой в системе схемы арбитража, арбитр может предоставить шину другому ведущему устройству, но может и оставить шину за прежним устройством, если оно продолжает выставлять сигнал запроса шины.

При HRESP=SPLIT или HRESP=RETRY ведущее устройство должно выполнять одинаковую последовательность действий, оно должно продолжать требовать шину до тех пор, пока обмен не будет завершен нормально (с подтверждением OKAY) или с ошибкой (с подтверждением ERROR). После того как это ведущее устройство вновь получит шину, оно должно продолжить выполнение запроса с того места, на котором был прерван. Должна быть повторена фаза адреса для того обмена данными, выполнение которого в предыдущий раз закончилось подтверждением SPLIT или RETRY. Однако значения управляющих сигналов при повторении могут быть иными. Проиллюстрируем это следующим примером. Пусть запрос фиксированной

длины из 8 слов был прерван при передаче четвертого слова (полностью обработаны передачи трех слов). В этом случае, когда ведущее устройство вновь получит шину, оно может выполнить или пакетную передачу неопределенной длины из пяти слов или пакетную передачу неопределенной длины из одного слова, и затем — пакетную передачу фиксированной длины из четырех слов. В любом из этих вариантов, значение HBURST будет отличаться от выставленного ведущим устройством первоначально.

Протокол шины AHB позволяет каждому ведущему устройству иметь по одной незаконченной транзакции. Если физическое устройство, работающее как ведущее на AHB, может иметь большее количество незаконченных транзакций, то такое устройство должно иметь соответствующее количество интерфейсов ведущего устройства на шине (одно физическое устройство может иметь несколько интерфейсов).

Рассмотрим теперь, как ведущее устройство использует сигнал HTRANS. Если ведущее устройство получило шину как ведущее устройство по умолчанию, но ему не нужно ее использовать, то HTRANS=IDLE. Это указывает ведомому устройству, что ему не нужно выполнять каких-либо действий. Поэтому, если ведущее устройство не выполняет запрос на шине и не запрашивает ее, то оно должно на HTRANS выставлять IDLE; при этом неважно, какие сигналы будут присутствовать на остальных линиях.

Если ведущему устройству необходимо вставить пустые такты в ходе выполнения запроса (например, в результате того, что оно не успевает подготовить данные для пересылки), то оно выставляет HTRANS=BUSY. При этом на линии адреса и на остальные линии управления ведущее устройство должно выставлять адрес и сигналы управления, соответствующие следующему обмену.

Коды сигналов HTRANS позволяют также ведущему устройству указать ведомому, является ли текущий обмен первым (HTRANS=NONSEQ) или он является очередным в пакете (HTRANS=SEQ), т. е. его адрес связан с адресом предыдущего обмена, а сигналы управления полностью совпадают. Это может быть полезным для таких ведомых устройств, как памяти.

Если ведущее устройство выполняло запрос с блокировкой, рекомендуется (это не обязательное требование), чтобы ведущее устройство после обмена данными с блокировкой вставляло один обмен с типом IDLE. Это необходимо для того, чтобы, если ведомое устройство при выполнении последнего обмена выставило подтверждение, отличное от OKAY, ведущее устройство не потеряло бы право на использование шины, что в данном случае может привести к прерыванию выполнения запроса с блокировкой.

Поведение ведомого устройства в ходе обмена

Если ведомое устройство получило от арбитра сигнал HSELx, значит, оно выбрано для участия в следующем обмене данными. В этом случае ведомое

устройство должно защелкнуть адрес и сигналы управления при HREADY=1. После этого оно должно начать обработку первого слова данных запроса – выполнение первого обмена данными.

Ведомое устройство может завершить обмен одним из следующих способов:

- завершить обмен немедленно — в первом же такте обмена;
- вставить одно или несколько состояний ожидания для получения дополнительного времени для завершения запроса;
- инициировать выполнение повторения или расщепления транзакции для того, чтобы освободить шину для других обменов;
- завершить обмен с ошибкой.

Если ведомое устройство готово завершить обмен немедленно, оно устанавливает HREADY='1' и HRESP=OKAY. Если выполняется операция записи, это означает, что ведомое устройство записало данные. Если выполняется операция чтения, это означает, что оно выставило действительные данные на шину.

Если ведомому устройству необходимо вставить один или несколько тактов ожидания, то в этих тактах оно выставляет HREADY='0' и HRESP=OKAY. Разработчикам необходимо учитывать, что все такты, когда HREADY='0' и HRESP=OKAY, шина AHB простаивает. Поэтому рекомендуется, чтобы ведомое устройство не вставляло более 16-ти тактов ожидания подряд.

Как уже отмечалось ранее, выставление подтверждения, отличного от OKAY, должно выполняться в течение двух тактов. Поэтому, если ведомое устройство требует выполнения повторения или расщепления транзакции или сообщает о завершении запроса с ошибкой, оно в первом такте выставляет HREADY='0' и HRESP=RETRY, HRESP=SPLIT или HRESP=ERROR соответственно. Во втором такте оно выставляет HREADY='1', а HRESP остается без изменений.

Если ведомое устройство выставило HRESP=ERROR, тип ошибки интерпретируется в зависимости от выполняемого типа обмена (например, попытка записи в привилегированную область памяти в пользовательском режиме). Никакими специальными средствами для указания типа ошибки ведомое устройство не располагает.

Если ведомое устройство выставило HRESP=RETRY, от него не требуется выполнения каких-либо дополнительных действий. Действия по завершению обработки расщепленной транзакции будут рассмотрены далее.

Рассмотрим поведение ведомого устройства, способного выполнять расщепленные транзакции. Такое ведомое устройство в начале каждого обмена должно защелкивать не только адрес, по которому идет обращение, но и идентификатор ведущего устройства (HMASTER), инициировавшего обмен. Если в ходе обмена с этим ведущим устройством ведомое устройство ини-

циировало расщепление транзакции, то для того, чтобы потом завершить эту транзакцию, оно должно выставить '1' на соответствующей линии HSPLITx. Каждому *i*-му ведущему устройству соответствует отдельная линия HSPLIT[i]. Сигнал на HSPLIT должен поддерживаться ведомым устройством в течение одного такта. Этот сигнал обрабатывается арбитром, после чего соответствующее ведущее устройство, в свою очередь, получает возможность использовать шину для завершения расщепленной транзакции. Если система содержит несколько ведомых устройств, поддерживающих расщепленные транзакции, то сигналы HSPLITx, идущие от этих устройств, могут объединяться по монтажному **или**.

Ведомое устройство может иметь несколько незаконченных транзакций от различных ведущих устройств. При этом ведомое устройство может не запоминать адрес отложенного обращения и команду управления, а только идентификатор ведущего устройства. Когда ведомое устройство будет готово к работе с конкретным ведущим устройством, оно сможет выставить соответствующий сигнал HSPLITx, и ведущее устройство в начале обмена, в любом случае, должно повторить и адрес обращения, и команду для очередного обмена в транзакции.

Если ведомое устройство выдало подтверждение SPLIT для нескольких ведущих устройств, в дальнейшем оно может завершать транзакции в любом порядке. Однако если среди них была транзакция с блокировкой, она должна быть завершена в первую очередь.

Ведомое устройство, которое может выставлять RETRY, должно в один момент времени запрашиваться только одним ведущим устройством. В большинстве случаев ведомые устройства, которые способны выставлять RETRY — это какие-либо приемо-передающие устройства, и то, что в один момент времени к ним обращается только одно ведущее устройство, отслеживается на более высоких уровнях протокола обмена. Предполагается, что когда ведомое устройство выставляет сигнал RETRY, оно запоминает номер ведущего устройства. Между этим моментом времени и временем, когда данный запрос будет успешно завершен, это ведомое устройство всем остальным ведущим устройствам может выдавать сообщение об ошибке.

Проектирование на VHDL блоков подключения на шину AMBA AHB

Реализация на VHDL компонентов интерфейса шины AMBA AHB

В этом разделе рассмотрены варианты реализации интерфейса ведущих устройств, ведомых устройств и арбитра шины AMBA AHB.

Во всех примерах данного раздела используется пакет AMBA_AHB_p.vhd. этом пакете описаны константы, соответствующие допустимым значения управляющих сигналов, константы, используемые для определения разряд ности шины адреса и шин данных, а также группа типов, используемых дл описания входных сигналов для модуля коммуникационной системы. На значение этих типов более подробно будет рассмотрено в дальнейшем Текст пакета AMBA_AHB_p приведен в листинге 5.8.

Листинг 5.8

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
package AMBA_AHB_p is
--HTRANS
constant HTRANS_IDLE: std_logic_vector (1 downto 0):="00";
constant HTRANS_BUSY: std_logic_vector (1 downto 0):="01";
constant HTRANS_NONSEQ: std_logic_vector (1 downto 0):="10";
constant HTRANS_SEQ: std_logic_vector (1 downto 0):="11";
--HBURST
constant HBURST_SINGLE: std_logic_vector (2 downto 0):="000";
constant HBURST_INCR: std_logic_vector (2 downto 0):="001";
constant HBURST_WRAP4:std_logic_vector (2 downto 0):="010";
constant HBURST_INCR4:std_logic_vector (2 downto 0):="011";
constant HBURST_WRAP8:std_logic_vector (2 downto 0):="100";
constant HBURST_INCR8:std_logic_vector (2 downto 0):="101";
constant HBURST_WRAP16:std_logic_vector (2 downto 0):="110";
constant HBURST_INCR16:std_logic_vector (2 downto 0):="111";
--HRESP
constant HRESP_OKAY: std_logic_vector (1 downto 0):="00";
constant HRESP_ERROR: std_logic_vector (1 downto 0):="01";
constant HRESP_RETRY: std_logic_vector (1 downto 0):="10";
constant HRESP_SPLIT: std_logic_vector (1 downto 0):="11";
--
constant H_ADDR: natural:=32;
constant H_DATA: natural:=32;
--
type H1 is array(natural range <>)of std_logic;
```

```
type H2 is array(natural range <>)of std_logic_vector(1 downto 0);
type H3 is array(natural range <>)of std_logic_vector(2 downto 0);
type H4 is array(natural range <>)of std_logic_vector(3 downto 0);
type HA is array(natural range <>)of std_logic_vector(H_ADDR-1 downto 0);
type HD is array(natural range <>)of std_logic_vector(H_DATA-1 downto 0);
--
end package AMBA_AHB_p;
```

В рассматриваемых далее примерах предполагается, что все устройства имеют тридцатидвухразрядные выходы на шины данных, и значение сигнала HSIZE всегда "010", что соответствует обмену четырехбайтовыми словами данных. Поэтому в моделях ведущих и ведомых устройств значение этого сигнала не анализируется. Не рассматриваются также возможности использования сигнала HPROT.

Интерфейс ведущего устройства на шину AMBA AHB

Интерфейс ведущего устройства при чтении одного слова

Рассмотрим модель простейшего интерфейса ведущего устройства. Этот интерфейс позволяет ведущему устройству записывать в ведомое одиночные слова данных. Граф конечного автомата, соответствующего функционированию этого ведущего устройства, приведен на рис. 5.12.

Этот автомат содержит четыре состояния. В состоянии WORK ведущее устройство выполняет действия, не связанные с использованием шины. В этом состоянии HBUSREQ='0'. Выход HTRANS=IDLE, что указывает ведомому устройству не выполнять каких-либо действий, если рассматриваемое ведущее устройство в системе окажется ведущим устройством по умолчанию. При этом на остальных выходах ведущего устройства на шине AHB могут быть любые значения. При необходимости использовать шину ведущее устройство переходит из состояния WORK в состояние ACKB. В этом состоянии HBUSREQ='1'. В нашем примере предполагается, что рассматриваемое устройство не выполняет запросов с блокировкой, поэтому HLOCK='0'. В этом состоянии ведущее устройство может установить на линиях адреса и управления значения, соответствующие фазе адреса первого обмена. После того как ведущее устройство получает сигнал предоставления шины (сигнал HGRANT), оно должно дождаться того момента, когда ведущее устройство, участвующее в предыдущем обмене, освободит линии адреса и управления. Этот момент определяется по HREADY='1'. После этого ведущее устройство переходит в состояние адреса — USEB_A. В этом состоянии оно должно выставить действительные значения на линии адреса и управления. При приходе очередного сигнала HREADY='1', рассматриваемое устройство из состояния USEB_A переходит в состояние передачи данных — USEB_D.

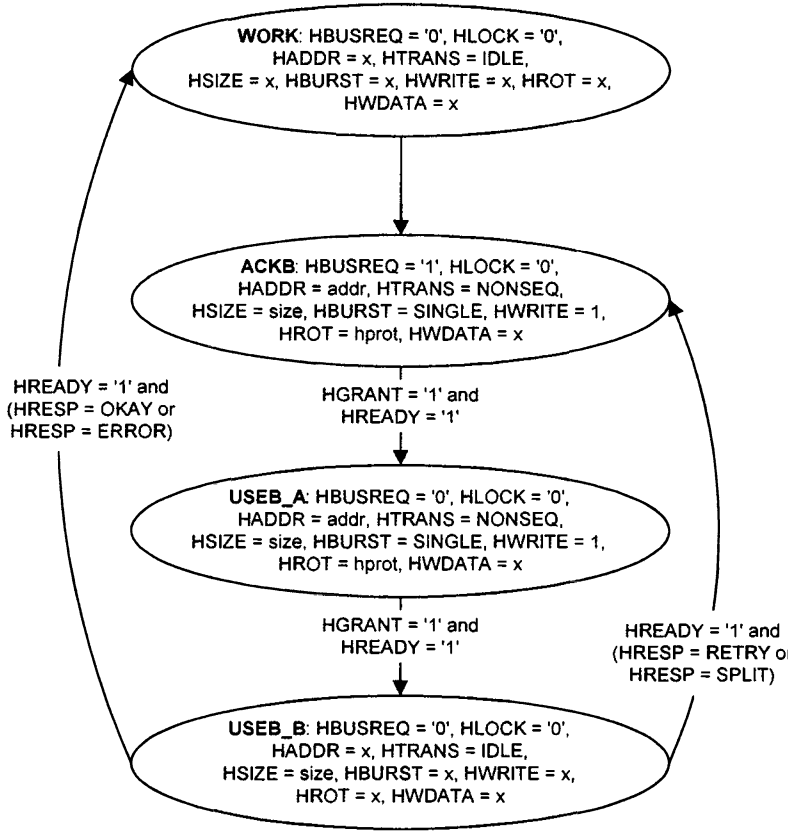


Рис. 5.12. Граф состояний конечного автомата, описание которого приведено в листинге 5.9

Такая модель поведения ведущего устройства является упрощенной. В ней не учитывается, что если предыдущий запрос завершается подтверждением, отличным от OKAY, то арбитр может изменить ведущее устройство, которому будет предоставлена шина для очередного обмена. Для учета этого необходимо было бы добавить переход из состояния USEB_A в состояние ACKB по HGRANT='0'.

В состоянии USEB_D устройство находится до тех пор, пока не получит от ведомого устройства сигнал завершения обмена HREADY='1'. В зависимости от значения сигнала HRESP, ведущее устройство переходит или в состояние WORK, или в состояние ACKB. При переходе в состояние WORK, HRESP=OKAY или HRESP=ERROR ошибка не обрабатывается специальным образом. При переходе в состояние ACKB, HRESP=RETRY или HRESP=SPLIT ведущее устройство в обоих случаях должно вновь запросить шину и, после того как она будет предоставлена, повторить запрос еще раз.

Текст модели интерфейса ведущего устройства приведен в листинге 5.9. Названия входных и выходных портов в тексте программы образованы следующим образом: название начинается с m1, далее следует название сигнала в соответствии со стандартом, но с опущенной первой буквой н (например, порт сигнала HADDR называется m1addr, HTRANS – m1trans).

Листинг 5.9

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
entity master1 is
port(m1reset:in std_logic;
     m1clk: in std_logic;
     m1busreq: out std_logic;
     m1lock: out std_logic;
     m1grant: in std_logic;
     m1addr: out std_logic_vector((H_ADDR-1) downto 0);
     m1trans: out std_logic_vector(1 downto 0);
     m1size: out std_logic_vector(2 downto 0);
     m1burst: out std_logic_vector(2 downto 0);
     m1write: out std_logic;
     m1prot: out std_logic_vector(3 downto 0);
     m1wdata: out std_logic_vector((H_DATA-1) downto 0);
     m1rdata: in std_logic_vector((H_DATA-1) downto 0);
     m1ready: in std_logic;
     m1resp: in std_logic_vector(1 downto 0)
    );
end entity master1;
architecture rtl of master1 is
type m1_state_type is (WORK, ACKB, USEB_A, USEB_D);
signal m1c_state,m1c_nextstate: m1_state_type;
signal cou: natural range 0 to 2;
begin
p_lockstate: process (m1reset,m1clk)
begin
if m1reset='0' then m1c_state<=WORK; cou<=2;
```

```
else
  if rising_edge(m1clk) then
        if cou>0 then cou<=cou-1;
        else if m1c_nextstate=WORK then cou<=2; end if;
        end if;
        if m1ready='1' or m1c_state=WORK then m1c_state<=m1c_nextstate;
        end if;
  end if;
end if;
end process p_lockstate;

p_outs: process (m1c_state,cou,m1grant,m1ready)
begin
case m1c_state is
when WORK => m1busreq<='0'; m1lock<='0';m1addr<=(others =>'0');
             m1trans<=HTRANS_IDLE; m1size<="010"; m1burst<=HBURST_SINGLE;
             m1write<='0'; m1prot<="0001"; m1wdata<=(others=>'0');
             if cou>0 then m1c_nextstate<=WORK;
                else m1c_nextstate<=ACKB; end if;
when ACKB =>m1busreq<='1'; m1lock<='0';
             m1addr <="00000000000000000000000000000001";
             m1trans<=HTRANS_NONSEQ; m1size<="010";
             m1burst<=HBURST_SINGLE; m1write<='1'; m1prot<="0001";
             m1wdata<=(others=>'0');
             if m1grant='1' and m1ready='1' then m1c_nextstate<=USEB_A;
               else m1c_nextstate<=ACKB; end if;
when USEB_A => m1busreq<='0'; m1lock<='0';
             m1addr <="00000000000000000000000000000001";
             m1trans<=HTRANS_NONSEQ; m1size<="010";
             m1burst<=HBURST_SINGLE; m1write<='1'; m1prot<="0001";
             m1wdata<=(others=>'0');
             if m1grant='1' and m1ready='1' then m1c_nextstate<=USEB_D;
               else m1c_nextstate<=USEB_A; end if;
when USEB_D => m1busreq<='0'; m1lock<='0'; m1addr <=(others=>'0');
             m1trans<=HTRANS_IDLE; m1size<="010"; m1burst<=HBURST_SINGLE;
             m1write<='0'; m1prot<="0001";
             m1wdata<="01010101010101010101010101010101";
             if m1ready='0' then m1c_nextstate<=USEB_D;
               else
```

```
                if (m1resp=HRESP_OKAY) or (m1resp=HRESP_ERROR) then
                    m1c_nextstate<=WORK;
                  else m1c_nextstate<=ACKB;
                end if;
              end if;
when others =>m1busreq<='0'; m1lock<='0';m1addr<=(others =>'0');
              m1trans<=HTRANS_IDLE;
              m1size<="010"; m1burst<=HBURST_SINGLE; m1write<='0';
              m1prot<="0001"; m1wdata<=(others=>'0');
              m1c_nextstate<=WORK;
end case;
end process p_outs;
end architecture rtl;
```

Рассмотрим архитектурное описание интерфейса ведущего устройства на шину AHB. Тип `m1_state_type` определяет множество возможных состояний для ведущего устройства. Сигнал `m1c_state` служит для хранения текущего состояния, `m1c_nextstate` — для следующего состояния. Применяем здесь сигналы, поскольку их можно использовать в списках чувствительности, а их значения можно наблюдать на временных диаграммах.

Внутри архитектурного описания используется также вспомогательный сигнал `cou`, предназначенный для определения количества тактов, в течение которых устройство будет пребывать в состоянии `WORK`. В нашем примере `cou` не превышает 2, что определяет максимальное время пребывания в этом состоянии, равное 3-м тактам.

Архитектурное описание включает в себя два процесса. Процесс `p_lockstate` предназначен для защелкивания очередного значения `m1c_state`. Если сигнал сброса установлен в активное состояние, `m1c_state=WORK`. Если сигнал сброса неактивен, то по восходящему фронту сигнала тактирования в `m1c_state` защелкивается текущее значение `m1c_nextstate` (при условии, что `m1c_state=WORK` или `m1ready='1'`). Время пребывания ведущего устройства в состоянии `WORK` не зависит от его взаимодействия с шиной и определяется внутренним алгоритмом работы ведущего устройства. Время пребывания в остальных состояниях зависит от состояния шины. Очередной обмен по шине завершается при `HREADY='1'`, поэтому переход из одного состояния в другое происходит, если `HREADY='1'`, по восходящему фронту сигнала тактирования.

Процесс `p_outs` предназначен для определения значений выходных сигналов и следующего состояния в соответствии с графом состояний автомата.

Примечание

Обратите внимание! Даже если из состояния существует единственный переход в следующее состояние и при этом значения выходного сигнала в обоих состояниях одинаковы, в следующем состоянии это значение выходного сигнала все равно определяется. Если предполагается, что модель предназначена только для поведенческого моделирования, этого можно не делать, однако, как было рассмотрено в главе 4, при синтезе это может привести к лишним аппаратным затратам.

Примеры временных диаграмм функционирования устройств приведены на рис. 5.13 и 5.14.

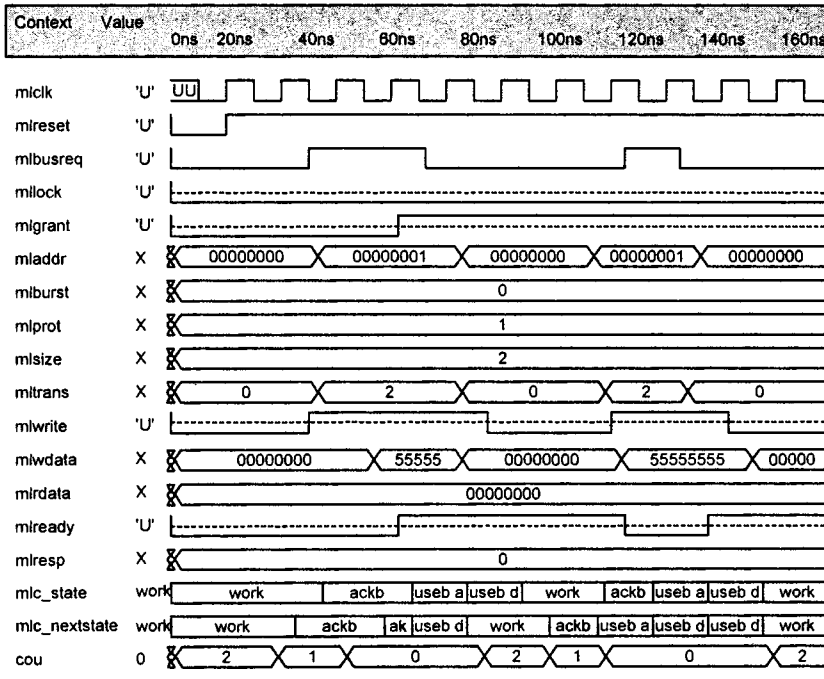


Рис. 5.13. Временная диаграмма запроса одного слова данных

Рассмотрим временную диаграмму запроса одного слова данных. В этом и последующих примерах начало и, соответственно, конец такта будем считать по восходящему фронту тактового импульса. Длительность такта – 10 нс. (это значение выбрано для удобства рассмотрения временных диаграмм, оно не имеет отношения к физической реализации). Первый такт начинается в момент времени 0 нс. и заканчивается в момент времени 10 нс. Второй такт начинается в момент времени 10 нс. и заканчивается в момент времени 20 нс., и так далее.

В первом такте сигнал сброса `mlreset` имеет активное значение, — выполняется сброс. Следующие три такта ведущее устройство находится в состоя-

нии WORK, в пятом такте оно переходит в состояние ACKB. Сигнал HGRANT для этого устройства устанавливается в '1' в начале 7-го такта, в этом же такте сигнал HREADY устанавливается в '1', поэтому в 8-м такте ведущее устройство переходит в USEB_A. Поскольку HREADY по-прежнему сохраняет значение '1' в 9-м такте устройство переходит в состояние USEB_D. Поскольку в конце этого такта HRESP=OKAY и HREADY='1', что указывает на успешную обработку запроса ведомым устройством, в 10-м такте ведущее устройство вновь переходит в состояние WORK и в 13-м такте вновь запрашивает шину. Первая фаза данных этого запроса выполняется в 15-м такте. В этом такте HREADY='0', поэтому обмен данными продливается на следующий такт, в котором HREADY='1' и обмен успешно завершается.

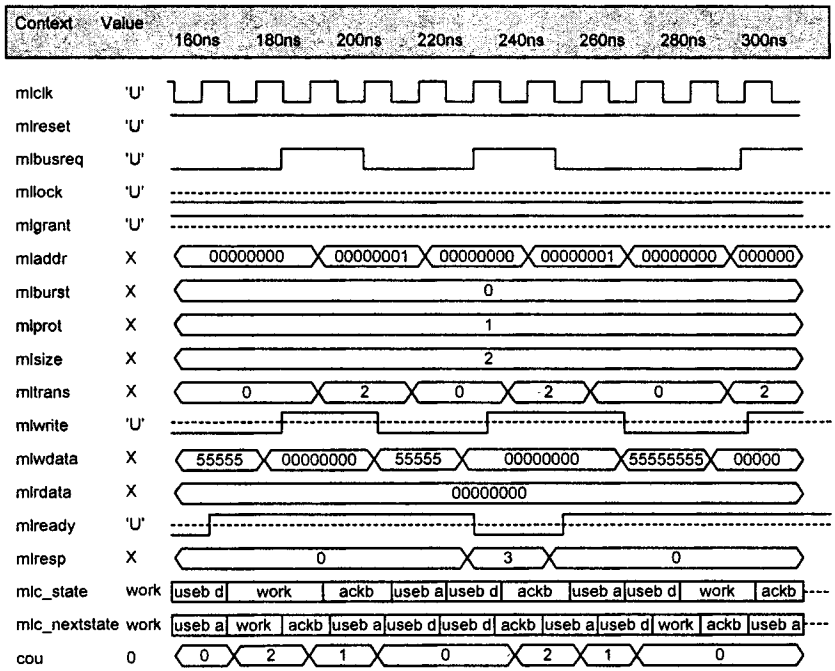


Рис. 5.14. Временная диаграмма запроса одного слова данных, в ходе выполнения которого ведомое устройство инициировало расщепленную транзакцию

Рассмотрим теперь обработку ведущим устройством расщепленной транзакции. Временная диаграмма представлена на рис. 5.14. В 20-м такте (момент модельного времени 190 нс.) ведущее устройство вновь запрашивает шину. В 22-м такте начинается фаза данных обмена. В этом такте ведомое устройство устанавливает HRESP=SPLIT и HREADY='0', в следующем такте HREADY='1' — ведомое устройство инициировало расщепление запроса. В этом случае ведущее устройство переходит в состояние запроса шины, а по-

сле того, как шина ему предоставлена, оно повторяет запись в ведомое устройство тех же данных по тому же адресу.

Интерфейс ведущего устройства при чтении последовательности слов

Рассмотрим модель интерфейса ведущего устройства, позволяющего ему читать из ведомого устройства последовательности слов. Граф состояний конечного автомата, соответствующий этому интерфейсу, приведен на рис. 5.15.

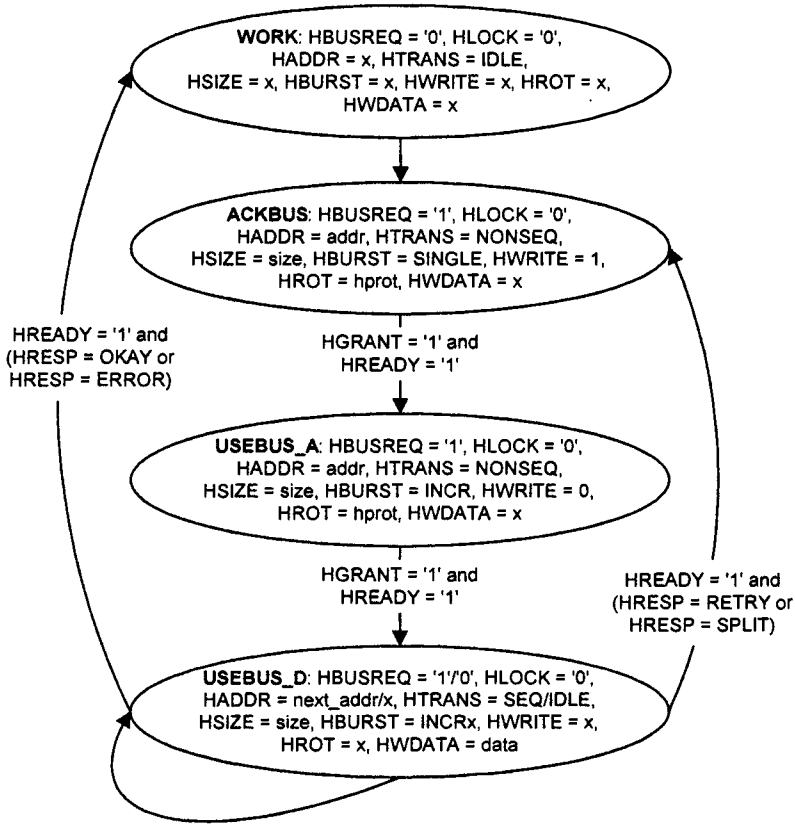


Рис. 5.15. Граф состояний конечного автомата, соответствующего ведущему устройству, описание которого приведено в листинге 5.10

Граф имеет сходную структуру с графом рис. 5.12, однако в состоянии `WORK` возможен дополнительный переход в само себя, если не все слова данных еще переданы.

Текст поведенческого описания модели приведен в листинге 5.10. Эта модель имеет такое же описание портов, как и модель ведущего устройства (листинг 5.9), но названия всех сигналов начинаются не с `m1`, а с `m2`.

Листинг 5.10

```
architecture rtl of master2 is
type m2_state_type is (WORK, ACKBUS, USEBUS_A, USEBUS_D);
signal m2c_state,m2c_nextstate: m2_state_type;
signal cou1:natural range 0 to 2;
signal cou2:natural range 0 to 3;
signal readed_data: std_logic_vector((H_DATA-1) downto 0);
signal n_addr: natural;
begin
p_lockstate: process(m2reset,m2clk,m2ready)
begin
if m2reset='0' then  m2c_state<=WORK; cou1<=1; cou2<=0; n_addr<=0;
else
  if rising_edge(m2clk) then
       if m2c_state=USEBUS_D and m2ready='1' and m2resp=HRESP_OKAY then
       readed_data<=m2rdata;  end if;
       if cou1>0 then cou1<=cou1-1;
         else if m2c_nextstate=WORK then cou1<=1; end if; end if;
       if cou2>0  then
          if m2c_state=USEBUS_D and m2ready='1' and m2resp=HRESP_OKAY
             then cou2<=cou2-1; n_addr<=n_addr+4;
             else if m2c_nextstate=WORK then cou2<=2; n_addr<=0; end if;
          end if;
       else
          if m2c_nextstate=WORK then cou2<=2; n_addr<=0; end if;
       end if;
       if m2ready='1' or m2c_state=WORK then
          m2c_state<=m2c_nextstate; end if;
  end if;
end if;
end process p_lockstate;

p_outs: process  (m2c_state,cou1,cou2,m2grant,m2ready)
begin
case m2c_state is
when WORK => m2busreq<='0'; m2lock<='0';m2addr<=(others =>'0');
             m2trans<=HTRANS_IDLE; m2size<="010"; m2burst<=HBURST_SINGLE;
             m2write<='0'; m2prot<="0001"; m2wdata<=(others=>'0');
```

```
            if cou1>0 then m2c_nextstate<=WORK;
              else m2c_nextstate<=ackbus; end if;
when ACKBUS =>m2busreq<='1'; m2lock<='0';
            m2addr<= conv_std_logic_vector(conv_unsigned(n_addr,32),32);
            m2trans<=HTRANS_NONSEQ; m2size<="010"; m2burst<=HBURST_INCR;
            m2write<='0'; m2prot<="0001"; m2wdata<=(others=>'0');
            if m2grant='1' and m2ready='1' then m2c_nextstate<=USEBUS_A;
              else m2c_nextstate<=ACKBUS; end if;
when USEBUS_A => m2busreq<='1'; m2lock<='0';
            m2addr<= conv_std_logic_vector(conv_unsigned(n_addr,32),32);
            m2trans<=HTRANS_NONSEQ; m2size<="010"; m2burst<=HBURST_INCR;
            m2write<='0'; m2prot<="0001"; m2wdata<=(others=>'0');
            if m2grant='1' and m2ready='1' then m2c_nextstate<=USEBUS_D;
              else m2c_nextstate<=USEBUS_A; end if;
when USEBUS_D =>
            if cou2>1 then m2busreq<='1'; else m2busreq<='0'; end if;
            m2lock<='0';
            m2addr<= conv_std_logic_vector(conv_unsigned(n_addr,32),32);
            if cou2>0 then m2trans<=HTRANS_SEQ;
              else m2trans<=HTRANS_IDLE; end if;
            m2size<="010"; m2burst<=HBURST_INCR; m2write<='0';
            m2prot<="0001"; m2wdata<=(others=>'0');
            if m2ready='0' then m2c_nextstate<=USEBUS_D;
              else
              if m2resp=HRESP_OKAY then
                   if cou2=0 then m2c_nextstate<=WORK;
                       else m2c_nextstate<=USEBUS_D; end if;
              else if m2resp=HRESP_ERROR then m2c_nextstate<=WORK;
                      else m2c_nextstate<=ACKBUS;
                   end if;
              end if;
            end if;
when others =>m2busreq<='0'; m2lock<='0';m2addr<=(others=>'0');
            m2trans<=HTRANS_IDLE; m2size<="010"; m2burst<=HBURST_SINGLE;
            m2write<='0'; m2prot<="0001"; m2wdata<=(others=>'0');
            m2c_nextstate<=WORK;
end case;
end process p_outs;
end architecture rtl;
```

Рассмотрим отличия этого поведенческого описания от описания, приведенного в листинге 5.9. Назначение сигналов m2c_state и m2c_nextstate аналогично назначению сигналов m1c_state и m1c_nextstate. Сигнал cou1 используется так же, как и cou в предыдущей модели, для подсчета количества тактов, в течение которых автомат пребывает в состоянии WORK. В данной модели автомат пребывает в этом состоянии два такта. Сигнал cou2 используется для определения количества передач данных, выполняемых в течение одного запроса.

В данной модели в рамках одного запроса выполняется чтение из ведомого устройства трех слов данных. В состоянии USEBUS_A это ведущее устройство выставляет те же значения управляющих сигналов, что и в предыдущем примере, но HBURST=INCR, что указывает на выполнение пакетного запроса. Из состояния USEBUS_A ведущее устройство переходит в состояние USEBUS_D, в течение которого должно быть прочитано три слова из ведомого устройства. Сигнал cou2 используется как счетчик непрочитанных слов. Его значение определяется в процессе p_lockstate. Когда автомат находится в состоянии WORK, сигналу cou2 присваивается значение 2. После того, как автомат переходит в состояние USEBUS_D, каждый раз, когда в конце такта HREADY='1' и HRESP=OKAY, значение этого счетчика уменьшается на 1, пока не дойдет до нуля, что указывает на то, что все три слова данных успешно прочитаны. После этого автомат переходит в состояние WORK.

Если во время пребывания автомата в состоянии USEBUS_D, он получает HRESP=ERROR, то автомат прерывает выполнение текущего запроса и переходит в состояние WORK. В этом случае, независимо от текущего значения cou2, ему присваивается значение 2. Если HRESP=RETRY или HRESP=SPLIT, то автомат переходит в состояние запроса шины, при этом в cou2 сохраняется количество непрочитанных из памяти слов, что позволяет автомату после того, как он вновь получит шину, продолжить чтение с того слова, на котором оно было прервано.

В состоянии USEBUS_D, если значение cou2>1, сигнал запроса шины установлен в '1'; в противном случае — в '0'. Сигнал cou2=1, когда на шину выставляется адрес последнего (третьего) слова данных. В следующем такте шина адреса и управления уже не будет нужна этому ведущему устройству. Сброс сигнала запроса шины указывает арбитру шины, что в следующем такте она может быть предоставлена другому ведущему устройству. В состоянии USEBUS_D устанавливается HTRANS=SEQ, что указывает ведомому устройству на то, что текущая управляющая информация связана с предыдущей, и HBURST=INCR, что указывает на пакетный запрос. В фазе данных последнего обмена сигналы имеют такие же значения, как и сигналы в USEB_D в предыдущей модели.

Примеры временных диаграмм работы устройства приведены на рис. 5.16 и 5.17.

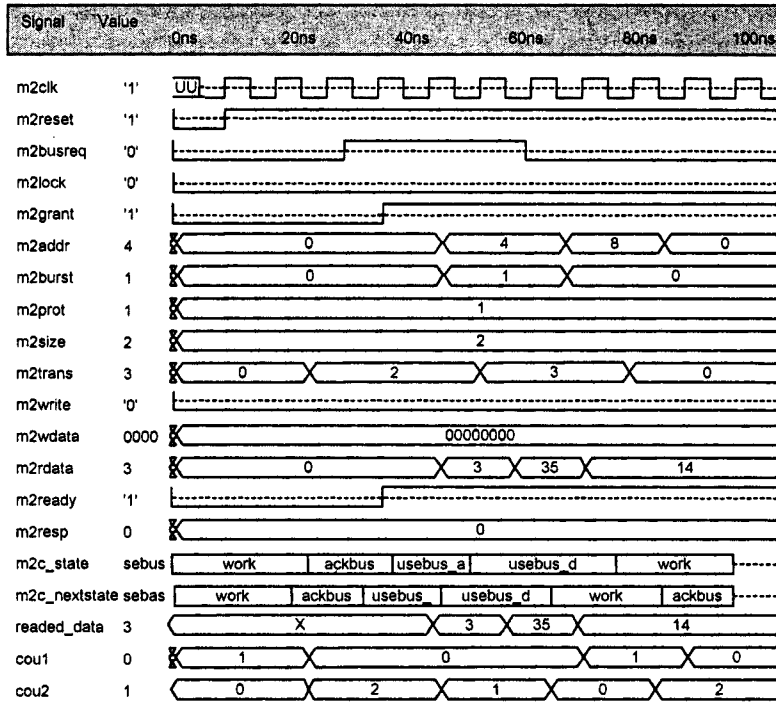


Рис. 5.16. Временная диаграмма последовательного чтения 3-х слов

На рис. 5.16 приведена временная диаграмма последовательного чтения 3-х слов из ведомого устройства. Чтение каждого слова сопровождается HRESP=OKAY и HREADY='1' в первом же такте, поэтому фаза данных продолжается в течение трех тактов.

На рис. 5.17 рассмотрено выполнение запроса, в ходе которого ведомое устройство выдало подтверждение RETRY (обработка подтверждения SPLIT выглядела бы аналогично). Это подтверждение выдается в 7-м такте в ходе чтения второго слова данных (cou2=1). После получения этого подтверждения ведущее устройство переходит в состояние ACKBUS. В 10-м такте устройство вновь получает шину, в 11-м такте начинается фаза данных — чтение второго слова, того слова, при чтении которого в предыдущий раз возникло подтверждение RETRY.

Интерфейс ведомого устройства на шину AMBA AHB

Рассмотрим организацию интерфейса ведомого устройства на примере простейшего блока памяти.

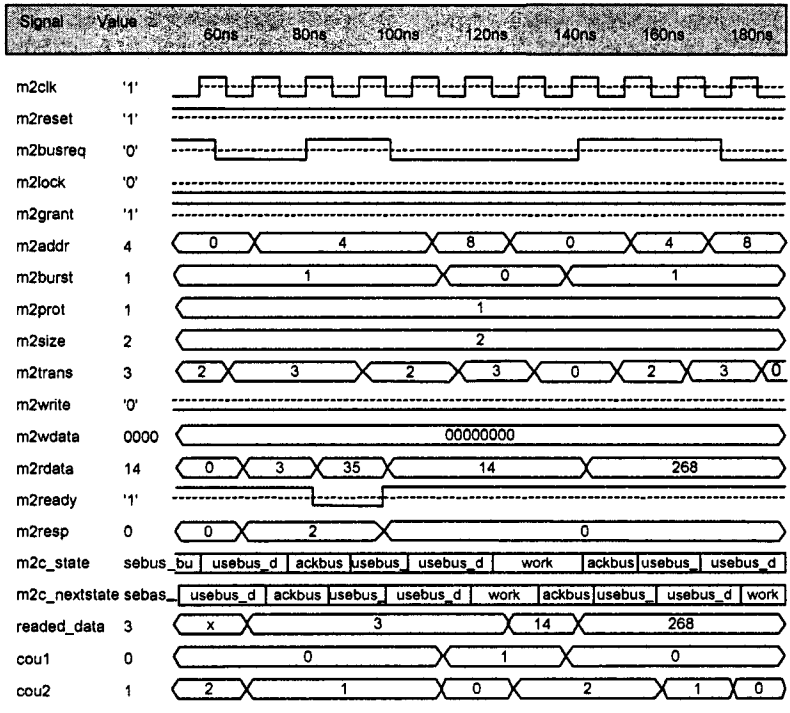


Рис. 5.17. Временная диаграмма при выдаче ведомым устройством кода подтверждения 'RETRY'

Если арбитр устанавливает сигнал выборки HSELx для ведомого устройства в '1', оно является выбранным для участия в текущем обмене. До тех пор, пока этот сигнал установлен, по каждому восходящему фронту тактового импульса ведомое устройство должно защелкивать значение адреса и управляющих сигналов, поступающих по шине от ведущего устройства. Вообще, поскольку в течение текущего обмена ведущее устройство выставляет одну и ту же управляющую информацию, что может продолжаться в течение нескольких тактов, ведомое устройство может защелкивать управляющую информацию только при HREADY='1', т. е. только в конце очередного обмена.

В соответствии с защелкнутыми сигналами адреса и управления в следующем такте, ведомое устройство должно выполнить обработку информации и выдать соответствующее подтверждение. Будем считать, для простоты, что рассматриваемое устройство памяти выполняет чтение или запись слова данных за один такт, не инициирует расщепленных транзакций и требований повторной передачи, при работе с ним не возникает ситуаций ошибки. Таким образом, в каждом такте обмена данными это устройство устанавливает сигнал готовности в '1' и сигнал подтверждения HRESP — в 'OKAY'. Эти сигналы могут иметь (и в нашей модели имеют) те же значения и в случае,

если устройство памяти не является выбранным для текущего обмена. Поскольку все ведомые устройства в системе подключаются к общим шинам через мультиплексоры, управляемые арбитром, для функционирования системы неважно, какие значения выставляют на выходы невыбранные ведомые устройства. Такие значения сигналов позволяют исключить необходимость специальной обработки ситуации, когда рассматриваемое устройство в системе является ведомым по умолчанию. Если в системе возникает ситуация, когда ни одному ведущему устройству не нужно использовать шину, арбитр предоставляет шину ведущему устройству по умолчанию. Это устройство должно выставить сигнал HTRANS=IDLE, что указывает ведомому устройству на то, что оно не должно выполнять каких-либо действий. В ответ на это ведомое устройство должно выставить HREADY='1' и HRESP=OKAY.

В модели интерфейса ведомого устройства, наряду с сигналом собственной готовности (HREADY), который является выходным, должен присутствовать второй сигнал HREADY, являющийся для ведомого устройства входным — это тот же сигнал готовности, который поступает к ведущему устройству, являющемуся в текущий момент времени собственником линий данных. Необходимость этого сигнала в модели связана с тем, что при выполнении последнего обмена в очередном запросе, шины адреса и управления уже используются новой парой (ведущее устройство — ведомое устройство), а шины данных еще используются прежней парой. Момент окончания последнего обмена и, соответственно, переход к первому обмену нового запроса определяется ведомым устройством, завершающим выполнение обмена установкой сигнала HREADY='1'. В следующем за этим такте, шины данных уже принадлежат новой паре: ведущее устройство — ведомое устройство.

Фаза данных последнего обмена предыдущего запроса и, соответственно, фаза адреса нового обмена может продолжаться в течение *нескольких* тактов. Рассмотрим, что произойдет, если ведомое устройство не сможет отслеживать действительный момент начала фазы данных первого обмена, а будет начинать ее в следующем такте, сразу после первого такта фазы адреса. Если происходит чтение данных из памяти, то данные, в соответствии с адресом, будут выставлены ведомым устройством на выходные линии данных. Однако, поскольку эти линии подключаются к шине через мультиплексор, управляемый арбитром, на шину они не попадут и появление их раньше времени не повлияет на функционирование системы. Поскольку фаза адреса будет продолжаться ведущим устройством, ведомое устройство просто будет выставлять данные в течение нескольких лишних тактов. Если же осуществляется операция записи в память, то ведомое устройство запишет данные, не дождавшись действительного момента начала фазы данных, т. е. запишет данные, ему не предназначенные. Далее, когда действительно начнется его фаза данных, поскольку ведущее устройство адрес не изменяло, поверх тех данных будут записаны действительные данные. Таким образом, отслеживание сигнала готовности от ведомого устройства, участвующего в

предыдущем обмене, критично только для тех ведомых устройств, в которых, по их внутренней организации, действительные данные должны находиться постоянно, а не только на момент завершения обмена. Это также существенно для ведомых устройств, которые в ходе обработки данных используют сигнал HBURST для выполнения предвыборки.

Текст модели интерфейса блока памяти с шиной, как ведущего устройства, приведен в листинге 5.11. Названия портов объекта образованы подобно тому, как это было сделано в листингах 5.9 и 5.10, но первая буква в названии — s. Входной сигнал smready — это сигнал подтверждения, тот же, что поступает к ведущим устройствам. Он используется для определения момента конца предыдущего запроса, в котором данное ведомое устройство не участвовало.

Листинг 5.11

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
entity c_mem1 is
port(sreset: in std_logic;
     sclk: in std_logic;
     ssel: in std_logic;
     saddr: in std_logic_vector((H_ADDR-1) downto 0);
     swrite: in std_logic;
     strans: in std_logic_vector(1 downto 0);
     ssize: in std_logic_vector(2 downto 0);
     sburst: in std_logic_vector(2 downto 0);
     swdata: in std_logic_vector((H_DATA-1) downto 0);
     smaster: in std_logic_vector(3 downto 0);
     smastlock: in std_logic;
     sready: out std_logic;
     smready: in std_logic;
     sresp: out std_logic_vector(1 downto 0);
     srdata: out std_logic_vector((H_DATA-1) downto 0);
     ssplit: out std_logic_vector(15 downto 0)
    );
end entity c_mem1;
architecture rtl of c_mem1 is
type mem_cont is array(1 to 32) of std_logic_vector((H_DATA-1) downto 0);
```

```
signal memory_contents: mem_cont;
signal addr: std_logic_vector((H_ADDR-1) downto 0);
signal write: std_logic;
signal trans: std_logic_vector(1 downto 0);
signal size: std_logic_vector(2 downto 0);
signal burst: std_logic_vector(2 downto 0);
signal sel: std_logic;
begin
p_ac_lock: process (sclk)
begin
if rising_edge(sclk) then
  if ssel='1' then
  addr<=saddr ;
  write<=swrite;
  trans<=strans;
  size<=ssize ;
  burst<=sburst;
  end if;
end if;
end process p_ac_lock;
p_sel: process  (sclk)
begin
if rising_edge(sclk) and smready='1' then sel<=ssel; end if;
end process p_sel;
p_resp: process  (sclk)
begin
if rising_edge(sclk) then sready<='1'; sresp<=HRESP_OKAY; end if;
end process p_resp;

p_write: process  (sclk,sel,addr)
begin
if sel='1' then
   if rising_edge(sclk) then
    if write='1' and trans/=HTRANS_BUSY and trans/=HTRANS_IDLE then
      memory_contents(conv_integer(addr,32))<=swdata;
     end if;
   end if;
end if;
end process p_write;
```

```
p_read: process (sel,write)
begin
if sel='1' then
  if write='0' then srdata<=memory_contents(conv_integer(addr)); end if;
end if;
end process p_read;

end architecture rtl;
```

Рассмотрим функционирование этого блока памяти. Внутри архитектурного описания определен тип mem_cont, с его использованием описан сигнал memory_contents, предназначенный для хранения значений ячеек памяти. Сигналы addr, write, trans, size, burst используются для хранения значений HADDR, HWRITE, HTRANS, HSIZE и HBURST с шины AHB. Сигнал sel используется для защелкивания значения сигнала HSEL.

Процесс p_ac_lock предназначен для защелкивания адреса и управляющей информации по восходящему фронту тактового импульса, если этот блок памяти выбран арбитром для участия в текущем обмене.

Процесс p_sel предназначен для защелкивания значения сигнала выбора ведомого устройства на момент прихода очередного переднего фронта тактового импульса и сигнала готовности от предыдущего ведомого устройства. Необходимость защелкивания сигнала ssel связана с тем, что в ходе передачи последнего слова данных текущего обмена, может выполняться фаза адреса нового обмена, в ходе которой выставляется сигнал выборки нового ведомого устройства, которое должно начать работу с линиями адреса и управления, а сигнал выборки прежнего ведомого — снимается. Однако линии адреса и подтверждения должны использоваться прежним ведомым устройством. Таким образом, если входной сигнал ssel (HSEL) указывает ведомому устройству на необходимость чтения значений сигналов адреса и управления, то внутренний сигнал sel указывает на необходимость использования линий данных и подтверждения.

Процесс p_resp предназначен для выдачи сигнала готовности и сигнала подтверждения. По рассмотренным выше причинам, для данного блока памяти эти сигналы постоянно имеют одно и то же значение.

Процесс p_write предназначен для записи нового значения в память. Запись осуществляется, если в предыдущем такте ведущее устройство выставило сигнал HWRITE в '1', и при этом значение сигнала HTRANS соответствовало NONSEQ или SEQ (т. е. указывало на необходимость выполнения действий), а сигнал sel='1' (что указывает на завершение предыдущего обмена). Процесс p_read предназначен для чтения данных из памяти.

На рис. 5.18 приведен пример временной диаграммы работы устройства памяти. В первом такте сигнал сброса установлен в активное значение. Начи-

ная со второго такта, сигналы `sready` и `sresp` устанавливаются в `'1'` и `"00"=OKAY` соответственно. Эти значения сохраняются до конца функционирования системы. В третьем такте одно из ведущих устройств получает шину и выставляет на нее адрес 1, который определяется арбитром как адрес обращения к памяти. Арбитр устанавливает `ssel` для устройства памяти в активное состояние. В результате, память защелкивает значение сигналов адреса и управления. В четвертом такте, в соответствии с этими значениями, выполняется операция записи в память (по адресу 1) слова данных, которое находилось на шине данных в течение четвертого такта (в этот период времени на линиях данных выставлено число 6). Поскольку в четвертом такте `ssel='0'`, то в пятом такте блок памяти не участвует в обмене данными. В шестом такте блок памяти вновь выбирается арбитром для обмена. В седьмом и восьмом тактах выполняется запись в ячейки памяти 2 и 3 соответственно. В восьмом такте память все еще выбрана, однако теперь `swrite='0'`, что соответствует команде чтения и `strans="11"` – `BUSY`. Таким образом, память выставляет данные в девятом такте, но ведущее устройство считывает их с шины только в десятом такте.

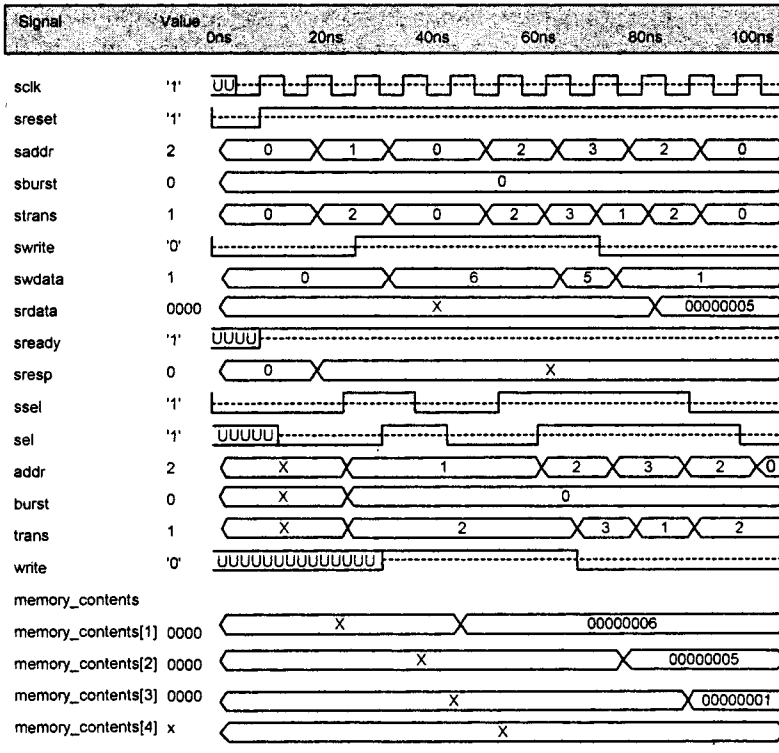


Рис. 5.18. Временная диаграмма работы модели устройства памяти

Устройства, выполняющие функции ведущего и ведомого на шине AMBA AHB

Наряду с устройствами, которые на шине AHB выполняют только одну функцию (например, память является ведомым устройством), возможно существование устройств, которые могут быть и ведомыми, и ведущими.

Рассмотрим структуру такого устройства на примере функционирования блока вычислителя специальной функции в составе системы-на-кристалле. Процессорное ядро системы-на-кристалле обращается к этому блоку по шине AHB как к ведомому устройству, передает ему тип функции, которую необходимо вычислить, и значение аргумента. Вычислитель выполняет задание; после чего он, уже как ведущее устройство шины AHB, обращается к процессорному ядру, как к ведомому устройству, и записывает в него полученный результат.

В нашем примере адрес блока спецвычислителя не определяется, модель реагирует на любой адрес. Блок процессорного ядра с идентификатором 1 имеет адрес "00000000000000000000000000000001"; с идентификатором, отличным от 1 — адрес "00000000000000000000000100000001".

Такой вычислитель может иметь следующую структуру:

- компонент вычисления специальной функции;
- компонент интерфейса ведущего устройства;
- компонент интерфейса ведомого устройства.

Если к компоненту интерфейса ведомого устройства происходит обращение по шине AHB, он анализирует правильность этого обращения — в данном примере в компонент можно только писать. Обращение на чтение рассматривается как ошибка. В компонент в ходе обращения должно быть передано два и только два слова данных, первое из которых является идентификатором функции. Попытка передачи другого количества слов данных или передача идентификатора несуществующей функции также должны рассматриваться как ошибка; однако в данном примере для того, чтобы не усложнять пример программы на VHDL, это игнорируется. Компонент интерфейса ведомого устройства передает идентификатор функции и аргумент в компонент вычисления специальной функции.

Компонент вычисления специальной функции

Компонент вычисления специальной функции, в соответствии с полученными данными, формирует результат. Сформированный результат поступает в компонент интерфейса ведущего устройства, который возвращает результат процессорному ядру. Вычисление результата может занимать несколько тактов. Если за это время процессорное ядро вновь обратиться к вычислителю для вычисления следующей функции, интерфейс ведомого устройства выдаст подтверждение `SPLIT` (или `RETRY`). Если ведомое устройство выдает

код подтверждения SPLIT, оно должно запоминать идентификатор ведущего устройства, чтобы потом послать сообщение о готовности завершить расщепленную транзакцию. Кроме того, запомненный идентификатор ведущего устройства, инициировавшего транзакцию, используется также компонентом интерфейса ведущего устройства спецвычислителя для того, чтобы возвратить результат именно тому устройству, которое его запрашивало. Структурная схема вычислителя приведена на рис. 5.19.

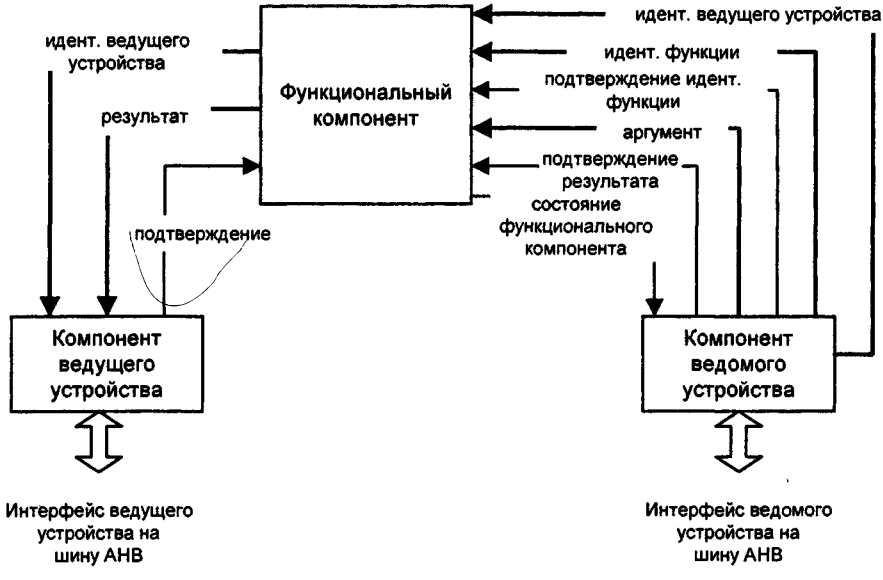


Рис. 5.19. Структурная схема вычислителя специальной функции

Каждый из компонентов может быть реализован как конечный автомат. Определим названия для сигналов, которыми обмениваются компоненты (табл. 5.7).

Таблица 5.7. *Обозначения сигналов*

Обозначение сигнала	Наименование сигнала
f_send	Подтверждение идентификатора функции
func_type	Идентификатор функции
Smaster_id	Идентификатор ведущего устройства, выдавшего задание, передается из компонента ведомого устройства в функциональный компонент

Таблица 5.7 (окончание)

Обозначение сигнала	Наименование сигнала
a_send	Подтверждение значения аргумента
arg_value	Значение аргумента
State	Состояние функционального компонента
Send	Подтверждение результата
Mmaster_id	Идентификатор ведущего устройства, выдавшего задание, передается из функционального компонента в компонент ведущего устройства
res_value	Значение результата

В названиях портов функционального компонента к именам соответствующих сигналов (табл. 5.7) добавляется префикс f. В названиях портов ведущего устройства к именам сигналов добавляется префикс m. В названиях портов ведомого устройства к именам сигналов добавляется префикс s, этот же префикс добавляется к именам портов, образующим интерфейс ведомого устройства.

С точки зрения компонента ведомого устройства, функциональный компонент может находиться или в состоянии ожидания очередного задания — IDLE, или в состоянии обработки очередного задания — WORK. Тип, соответствующий этому множеству состояний, описан в пакете ms_p:

```
type f_states is (IDLE, WORK);
```

Рассмотрим конечный автомат, соответствующий функциональному компоненту. Граф состояний, соответствующий этому конечному автомату, приведен на рис. 5.20.

Этот автомат включает в себя пять состояний. В состоянии IDLE функциональный компонент не выполняет каких-либо функций, он ждет команды на вычисление функции. Если компонент ведомого устройства устанавливает сигнал f_send='1', это указывает функциональному компоненту, что начался прием очередного задания. Компонент переходит в состояние F_LOCK, и защелкивает идентификатор функции и идентификатор ведущего устройства. По сигналу a_send='1' от компонента ведомого устройства он переходит в состояние A_LOCK и защелкивает значение аргумента. В следующем такте автомат переходит в состояние COUNT, в котором выполняет вычисление результата функции. После завершения вычислений автомат переходит в состояние W_RES, в котором передает результат и идентификатор ведущего устройства в компонент ведущего устройства. Текст описания этого компонента приведен в листинге 5.12.

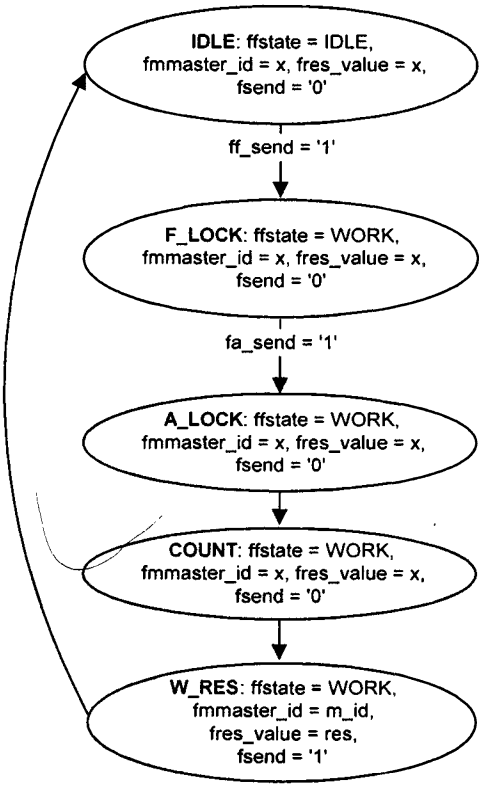


Рис. 5.20. Граф состояний конечного автомата функционального компонента

Листинг 5.12

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
use ms_p.ms_p.all;
entity f_unit is
port(freset: in std_logic;
     fclk: in std_logic;
     --
     fmmaster_id: out std_logic_vector(3 downto 0);
```

```
    fres_value: out std_logic_vector((H_DATA-1) downto 0);
    fsend: out std_logic;
    --
    fstate: out f_states;
    ffunc_type: in std_logic_vector((H_DATA-1) downto 0);
    ff_send:in std_logic;
    farg_value: in std_logic_vector((H_DATA-1) downto 0);
    fa_send: in std_logic;
    fsmaster_id: in std_logic_vector(3 downto 0)
  );
end entity f_unit;

architecture rtl of f_unit is
type instate is (IDLE,F_LOCK,A_LOCK,COUNT,W_RES);
signal cf_state, cf_nstate: instate;
signal f_ident: std_logic_vector((H_DATA-1) downto 0);
signal arg:std_logic_vector((H_DATA-1) downto 0);
signal res: integer;
signal mast_id:std_logic_vector(3 downto 0);
signal cou: natural;
begin
p_state: process (freset,fclk)
begin
if freset='0' then cf_state<=IDLE;
 else
     if rising_edge(fclk) then  cf_state<=cf_nstate ; end if;
end if;
end process p_state;
data_lock: process (fclk,cf_nstate)
begin
if rising_edge(fclk) then
  if cf_nstate=F_LOCK then f_ident<=ffunc_type; mast_id<=fsmaster_id;
  end if;
  if cf_nstate=A_LOCK then arg<=farg_value; end if;
end if;
end process data_lock;
p_outs: process (cf_state,ff_send,fa_send,cou,farg_value)
begin
case cf_state is
```

```
when IDLE => if ff_send='1' then cf_nstate<=F_LOCK;
                 else cf_nstate<=IDLE; end if;
             fsend<='0';fmmaster_id<=(others=>'0');
             fres_value <=(others=>'0'); fstate<=IDLE;
when F_LOCK => if fa_send='1' then cf_nstate<=A_LOCK;
                  else cf_nstate<=F_LOCK; end if;
               fsend<='0';fmmaster_id<=(others=>'0');
               fres_value<=(others=>'0'); fstate<=WORK;
when A_LOCK => cf_nstate<=COUNT;
               fsend<='0';fmmaster_id<=(others=>'0');
               fres_value<=(others=>'0'); fstate<=WORK;
when COUNT => if cou>0 then cf_nstate<=COUNT;
                 else cf_nstate<=W_RES; end if;
              fsend<='0';fmmaster_id<=(others=>'0');
              fres_value<=(others=>'0'); fstate<=WORK;
when W_RES => cf_nstate<=IDLE;
              fsend<='1'; fmmaster_id<=mast_id;
              fres_value<=conv_std_logic_vector(res,32); fstate<=WORK;
when others => cf_nstate<=IDLE; fsend<='0';fmmaster_id<=(others=>'0');
              fres_value<=(others=>'0');  fstate<=IDLE;
end case;
end process p_outs;
p_cou: process (freset,fclk)
begin
if freset='0' then cou<=0;
 else
  if fclk='1' and fclk'event then
      if cf_state=A_LOCK then cou<=5;
         else if cf_state=COUNT and cou>0 then cou<=cou-1; end if;
      end if;
  end if;
end if;
end process p_cou;
p_count: process (cou,arg)
variable varg:natural;
begin
if cou=5 then
  varg:=conv_integer(arg,32);
```

```
  case conv_integer(f_ident,32) is
when 1 => res<=70;--res<=1+varg+(1/2)*varg*varg;
when 2 => res<=varg-
(1/6)*varg*varg*varg+(1/120)*varg*varg*varg*varg*varg;
when 3 => res<=1-(1/2)*varg*varg+(1/24)*varg*varg*varg*varg;
when 4 => res<=varg-(1/2)*varg*varg+(1/3)*varg*varg*varg;
when others => res<=0;
  end case ;
end if;
end process p_count;
end architecture rtl;
```

Рассмотрим это описание. Внутри описания архитектуры определен новый перечислимый тип instate, он определяет множество состояний конечного автомата. Сигналы cf_state и cf_nstate, принадлежащие этому типу, используются для хранения текущего и следующего состояния автомата. Сигналы mast_id, f_ident и arg используются для защелкивания идентификатора ведущего устройства, а также идентификатора функции и аргумента, поступающих в функциональный модуль от компонента ведомого устройства. Сигнал cou используется как счетчик времени вычисления функции.

Процесс p_state используется для определения очередного состояния автомата. По сигналу freset автомат переходит в начальное состояние IDLE. Если этот сигнал находится в неактивном состоянии, то по восходящему фронту сигнала тактирования осуществляется переход в следующее состояние.

Процесс data_lock предназначен для защелкивания входных данных. Защелкивание данных осуществляется по восходящему фронту сигнала тактирования. Если следующее состояние F_LOCK, то выполняется защелкивание значения идентификатора функции и идентификатора ведущего устройства. Если следующее состояние A_LOCK, то выполняется защелкивание аргумента.

Процесс p_outs предназначен для определения значений выходных сигналов и следующего состояния автомата.

Процессы p_cou и p_count предназначены для поведенческого моделирования вычисления функции. В процессе p_cou определяется значение сигнала cou, на основе которого в процессе p_outs определяется переход из состояния COUNT в состояние W_RES, т. е. количество тактов, необходимое для вычисления значения функции. Процесс p_count используется для вычисления значения функции. В приведенном листинге идентификатору функции '1' соответствует экспоненциальная функция, функции '2' — sin(x), функции '3' — cos(x), функции '4' — логарифмическая функция. Для их вычисления применяются степенные ряды. Приведенное описание

может использоваться только для поведенческого моделирования. Синтез вычислителя по такому алгебраическому описанию не поддерживается Foundation Express. Для синтеза необходимо описать или использовать готовые компоненты, предназначенные для выполнения функции деления. Поскольку на этапе получения задания не происходит контроля поступающего идентификатора функции, то в этом процессе, при получении неправильного идентификатора функции, результату просто присваивается 0.

Компонент ведомого устройства

Рассмотрим конечный автомат, соответствующий компоненту ведомого устройства. Граф конечного автомата приведен на рис. 5.21.

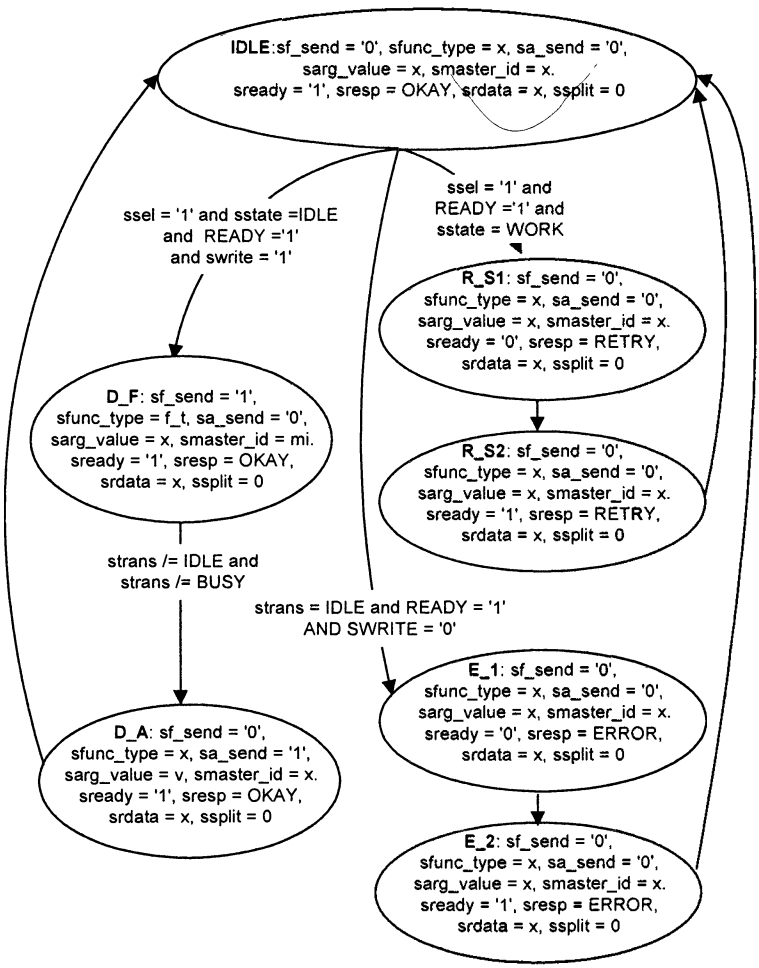


Рис. 5.21. Граф конечного автомата компонента ведомого устройства

Если ведомое устройство не выбрано для участия в очередном запросе, оно находится в состоянии IDLE. Если ведомое устройство получает от арбитра шины AHB сигнал выборки и при этом READY='1', то ведомое устройство анализирует управляющие сигналы от ведущего устройства, а также состояние функционального компонента. Здесь имеется в виду сигнал, выдаваемый ведомым устройством, участвующим в предыдущем обмене; READY='1' указывает на то, что началась фаза адреса нового обмена.

Если от ведущего устройства поступает команда чтения, это рассматривается как ошибка, ведомое устройство переходит в состояние ошибки E_1. Если от ведущего устройства поступает команда записи, но функциональный компонент находится в состоянии вычисления функции, то ведомое устройство переходит в состояние R_S1, в котором выдает подтверждение RETRY, указывающее ведущему устройству на то, что запрос должен быть отложен, поскольку устройство занято. Возможна также выдача подтверждения SPLIT. Если выдается подтверждение RETRY, ведомое устройство не должно выполнять каких-либо действий. Если выдается подтверждение SPLIT, то ведомое устройство должно отслеживать момент времени, когда функциональный модуль завершит вычисления, чтобы сообщить арбитру о готовности завершить выполнение запроса. Ведущее устройство, согласно стандарту на шину AHB, в обоих случаях должно запрашивать шину до тех пор, пока запрос не будет завершен. Поэтому в данном случае подтверждения SPLIT и RETRY практически одинаковы с точки зрения функционирования системы, но использование RETRY связано с меньшими аппаратными затратами.

Если от ведущего устройства поступает команда записи, а функциональный компонент находится в состоянии IDLE, то компонент ведомого устройства переходит в состояние D_F, в котором осуществляется запись в функциональный компонент идентификатора функции и идентификатора ведущего устройства. Ведомое устройство не содержит регистров для хранения данных, поступающих от ведущего устройства, данные записываются непосредственно в регистры функционального модуля, ведомое устройство только генерирует сигналы, по которым функциональный компонент защелкивает данные. В приведенном примере эти сигналы в состоянии D_F устанавливаются в активное состояние, безусловно, не учитывается значение сигнала HTRANS от ведущего устройства. Если этот сигнал имеет значение IDLE или BUSY, то на линиях данных могут быть недействительные значения. Отработка такой ситуации была рассмотрена в примере ведомого устройства — памяти. После состояния D_F компонент ведомого устройства переходит в состояние D_A, в котором выполняется запись значения аргумента в функциональный модуль, после чего компонент ведомого устройства переходит в состояние IDLE.

Автомат содержит два состояния ошибки — E_1 и E_2, а также два состояния выдачи подтверждения RETRY – R_S1 и R_S2. Это связано с тем, что подтверждения ERROR, RETRY и SPLIT, согласно стандарту на шину AHB, выдаются в течение двух тактов, в первом из которых HREADY='0', а во втором HREADY='1'.

Текст описания компонента ведомого устройства приведен в листинге 5.13.

Листинг 5.13

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
use ms_p.ms_p.all;
entity s_comp is
port(sreset: in std_logic;
     sclk: in std_logic;
     ssel: in std_logic;
     saddr: in std_logic_vector((H_ADDR-1) downto 0);
     swrite: in std_logic;
     strans: in std_logic_vector(1 downto 0);
     ssize: in std_logic_vector(2 downto 0);
     sburst: in std_logic_vector(2 downto 0);
     swdata: in std_logic_vector((H_DATA-1) downto 0);
     smaster: in std_logic_vector(3 downto 0);
     smastlock: in std_logic;
     smready: in std_logic;
     sready: out std_logic;
     sresp: out std_logic_vector(1 downto 0);
     srdata: out std_logic_vector((H_DATA-1) downto 0);
     ssplit: out std_logic_vector(15 downto 0);
     --
     sstate: in f_states;
     sfunc_type: out std_logic_vector((H_DATA-1) downto 0);
     sf_send: out std_logic;
     sarg_value: out std_logic_vector((H_DATA-1) downto 0);
     sa_send: out std_logic;
     smaster_id: out std_logic_vector(3 downto 0)
    );
end entity s_comp;
architecture rtl of s_comp is
type s_state is (IDLE, D_F, D_A, E_1, E_2, R_S1, R_S2);
```

```
signal cs_state,cs_nstate: s_state;
signal addr: std_logic_vector((H_ADDR-1) downto 0);
signal write: std_logic;
signal trans: std_logic_vector(1 downto 0);
signal size: std_logic_vector(2 downto 0);
signal burst:  std_logic_vector(2 downto 0);
signal master: std_logic_vector(3 downto 0);
signal sel: std_logic;
begin
p_lockstate: process(sreset,sclk)
begin
if sreset='0' then cs_state<=IDLE;
  else
   if rising_edge(sclk) then  cs_state<=cs_nstate; end if;
end if;
end process p_lockstate;
p_lockaddr: process (sclk)
begin
   if rising_edge(sclk) then
   if smready='1' then
      if cs_nstate=D_F or cs_state=D_F then
          addr<=saddr; write<=swrite; trans<=strans; size<=ssize;
          burst<=sburst; master<=smaster; end if;
   end if;
end if;
end process p_lockstate;
p_outs: process    (cs_state,ssel,swrite,smready,swdata)
begin
case cs_state is
when IDLE => sf_send<='0'; sfunc_type <=(others =>'0');
             smaster_id <= (others =>'0');
             sa_send<='0'; sarg_value <=(others =>'0');
             sready<='1'; sresp<=HRESP_OKAY;
             if ssel='1' and smready='1' then if sstate=IDLE then
                                 if swrite='0' then cs_nstate<=E_1;
                                    else cs_nstate<=D_F; end if;
                               else cs_nstate<=R_S1; end if;
            else cs_nstate<=IDLE; end if;
```

```
when E_1 =>  sf_send<='0'; sfunc_type <=(others =>'0');
             smaster_id <= (others =>'0');
             sa_send<='0'; sarg_value <=(others =>'0');
             sready<='0'; sresp<=HRESP_ERROR;
             cs_nstate<=E_2;
when E_2 =>  sf_send<='0'; sfunc_type <=(others =>'0');
             smaster_id <= (others =>'0');
             sa_send<='0'; sarg_value <=(others =>'0');
             sready<='1'; sresp<=HRESP_ERROR;
             cs_nstate<=IDLE;
when R_S1 =>  sf_send<='0'; sfunc_type <=(others =>'0');
             smaster_id <= (others =>'0');
             sa_send<='0'; sarg_value <=(others =>'0');
             sready<='0'; sresp<=HRESP_RETRY;
             cs_nstate<=R_S2;
when R_S2 =>  sf_send<='0'; sfunc_type <=(others =>'0');
             smaster_id <= (others =>'0');
             sa_send<='0'; sarg_value <=(others =>'0');
             sready<='1'; sresp<=HRESP_RETRY;
             cs_nstate<=IDLE;
when D_F => if smready='1' then sf_send<='1'; else sf_send<='0'; end if;
            sfunc_type<=swdata; smaster_id <=master;
            sa_send<='0'; sarg_value <=(others =>'0');
            sready<='1'; sresp<=HRESP_OKAY;
            if smready='1' then cs_nstate<=D_A;
               else cs_nstate<=D_F; end if;
when D_A => sf_send<='0'; sfunc_type <=(others =>'0');
            smaster_id <= (others =>'0');
            if smready='1' then sa_send<='1' ; else sa_send<='0'; end if;
            sarg_value <=swdata;
            sready<='1'; sresp<=HRESP_OKAY;
            if smready='1' then cs_nstate<=IDLE;
                 else cs_nstate<=D_A; end if;
end case;
end process p_outs;
end architecture rtl;
```

Внутри архитектурного описания определен тип `s_state`, который задает множество состояний конечного автомата. Сигналы `cs_state` и `cs_nstate`

используются для хранения значений текущего и следующего состояний автомата. Сигналы addr, write, trans, size, burst используются для защелкивания управляющей информации от ведущего устройства в конце очередной адресной фазы. Сигнал master используется для защелкивания идентификатора ведущего устройства, а сигнал sel — для защелкивания сигнала выбора ведомого устройства.

Процесс p_lockstate используется для перехода из состояния в состояние.

Процесс p_lockaddr используется для защелкивания управления от ведущего устройства в конце очередной адресной фазы. Первая фаза адреса начинается, когда ведомое устройство находится в состоянии IDLE. Оно получает сигнал выбора ведомого устройства от арбитра. В соответствии с этим сигналом, его следующее состояние определяется как D_F. По восходящему фронту тактового импульса, соответствующего окончанию фазы адреса (smready='1'), оно должно защелкнуть управляющую информацию. Вторая фаза адреса соответствует состоянию автомата D_F.

Процесс p_outs используется для определения значений выходных сигналов и определения следующего состояния.

Компонент ведущего устройства

Рассмотрим конечный автомат, соответствующий компоненту ведущего устройства. Граф конечного автомата аналогичен графу, приведенному на рис. 5.11, за исключением того, что переход из состояния WORK (в котором этот компонент, впрочем, ничего не делает) осуществляется при получении сигнала fres='1', что указывает на завершение процесса вычисления аргумента.

Текст описания компонента ведущего устройства приведен в листинге 5.14.

Листинг 5.14

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
use ms_p.ms_p.all;
entity m_comp is
port(mreset: in std_logic;
     mclk: in std_logic;
     mbusreq: out std_logic;
```

```
    mlock: out std_logic;
    mgrant: in std_logic;
    maddr: out std_logic_vector((H_ADDR-1) downto 0);
    mtrans: out std_logic_vector(1 downto 0);
    msize: out std_logic_vector(2 downto 0);
    mburst: out std_logic_vector(2 downto 0);
    mwrite: out std_logic;
    mprot: out std_logic_vector(3 downto 0);
    mwdata: out std_logic_vector((H_DATA-1) downto 0);
    mrdata: in std_logic_vector((H_DATA-1) downto 0);
    mready: in std_logic;
    mresp: in std_logic_vector(1 downto 0);
    --
    mmaster_id: in std_logic_vector(3 downto 0);
    mres_value: in std_logic_vector((H_DATA-1) downto 0);
    msend: in std_logic
   );
end entity m_comp;
architecture rtl of m_comp is
type m_state_type is (work, ackb,useb_a, useb_d);
signal mc_state,mc_nextstate: m_state_type;
signal res_value: std_logic_vector((H_DATA-1) downto 0);
signal master_id: std_logic_vector(3 downto 0);
begin
p_lockstate: process (mreset,mclk)
begin
if mreset='0' then mc_state<=WORK;
else
   if rising_edge(mclk) then  mc_state<=mc_nextstate; end if;
end if;
end process p_lockstate;
p_lockdata: process(mclk,mc_nextstate,mmaster_id,mres_value)
begin
if rising_edge(mclk) then
   if mc_state=WORK then
     res_value<=mres_value; master_id<=mmaster_id; end if;
end if;
end process p_lockdata;
```

```
p_outs: process(mc_state,mgrant,mready,msend)
begin
case mc_state is
when WORK => mbusreq<='0'; mlock<='0';maddr<=(others =>'0');
             mtrans<=HTRANS_IDLE; msize<="010"; mburst<=HBURST_SINGLE;
             mwrite<='0'; mprot<="0001"; mwdata<=(others=>'0');
             if msend='0' then mc_nextstate<=WORK;
                else mc_nextstate<=ACKB; end if;
when ACKB =>mbusreq<='1'; mlock<='0';
            maddr <="00000000000000000000000000000001";
            mtrans<=HTRANS_NONSEQ; msize<="010"; mburst<=HBURST_SINGLE;
            mwrite<='1'; mprot<="0001"; mwdata<=(others=>'0');
            if mgrant='1' and mready='1' then mc_nextstate<=USEB_A;
              else mc_nextstate<=ACKB; end if;
when USEB_A => mbusreq<='0'; mlock<='0';
             if master_id="001" then
                     maddr <="00000000000000000000000000000001";
                else maddr <="00000000000000000000000100000001";end if;
             mtrans<=HTRANS_NONSEQ;
             msize<="010"; mburst<=HBURST_SINGLE; mwrite<='1';
             mprot<="0001"; mwdata<=(others=>'0');
             if mgrant='1' and mready='1' then mc_nextstate<=USEB_D;
               else mc_nextstate<=USEB_A; end if;
when USEB_D => mbusreq<='0'; mlock<='0';
             maddr <=(others=>'0'); mtrans<=HTRANS_IDLE; msize<="010";
             mburst<=HBURST_SINGLE; mwrite<='0'; mprot<="0001";
             mwdata<=res_value;
             if mready='0' then   mc_nextstate<=USEB_D;
               else  if (mresp=HRESP_OKAY) or (mresp=HRESP_ERROR) then
                                mc_nextstate<=WORK;
                           else mc_nextstate<=ACKB;
               end if;
             end if;
when others =>mbusreq<='0'; mlock<='0';maddr<=(others =>'0');
             mtrans<=HTRANS_IDLE; msize<="010"; mburst<=HBURST_SINGLE;
             mwrite<='0'; mprot<="0001"; mwdata<=(others=>'0');
             mc_nextstate<=WORK;
```

```
end case;
end process p_outs;
end architecture rtl;
```

Рассмотрим функционирование блока ведущего устройства. Внутри архитектурного описания блока определен тип m_state_type, множество значений которого соответствует множеству возможных состояний блока. Сигналы mc_state и mc_nextstate используются для хранения значений текущего и следующего состояний. Сигналы res_value и master_id используются для защелкивания значения результата и идентификатора ведущего устройства, которому он должен быть возвращен.

В процессе p_lockstate организован переход из состояния в состояние. Процесс p_lockdata используется для защелкивания данных, поступающих из функционального модуля. Данные защелкиваются по восходящему фронту тактового импульса, если при этом блок ведущего устройства находится в состоянии WORK, т. е. отправка предыдущих данных завершена. В данном устройстве не предусмотрена обработка ситуации, когда к моменту получения следующего результата предыдущий еще не был отправлен из-за того, что компоненту ведущего устройства не была предоставлена шина. В этом случае следующий результат просто пропадает.

Процесс p_outs используется для определения значений выходных сигналов, а также определения следующего состояния. Адрес, по которому выполняется обращение, определяется в зависимости от идентификатора ведущего устройства.

Структурное описание устройства в целом

Структурное описание устройства в целом приведено в листинге 5.15. Для названий портов приняты следующие соглашения: названия портов, образующих интерфейс ведущего устройства начинаются с Hm, далее идет название сигнала в соответствии со стандартом шины AHB, но первая буква н опущена; аналогично образованы названия портов, образующих интерфейс ведомого устройства, они начинаются с Hs.

Листинг 5.15

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
use ms_p.ms_p.all;
```

```
entity ms1 is
port(Hreset:in std_logic;
     Hclk: in std_logic;
     --
     Hmbusreq: out std_logic;
     Hmlock: out std_logic;
     Hmgrant: in std_logic;
     Hmaddr:out std_logic_vector((H_ADDR-1) downto 0);
     Hmtrans:out std_logic_vector(1 downto 0);
     Hmsize:out std_logic_vector(2 downto 0);
     Hmburst: out std_logic_vector(2 downto 0);
     Hmwrite: out std_logic;
     Hmprot: out std_logic_vector(3 downto 0);
     Hmwdata: out std_logic_vector((H_DATA-1) downto 0);
     Hmrdata: in std_logic_vector((H_DATA-1) downto 0);
     Hmready:in std_logic;
     Hmresp:in std_logic_vector(1 downto 0);
     --
     Hssel: in std_logic;
     Hsaddr: in std_logic_vector((H_ADDR-1) downto 0);
     Hswrite: in std_logic;
     Hstrans: in std_logic_vector(1 downto 0);
     Hssize: in std_logic_vector(2 downto 0);
     Hsburst: in std_logic_vector(2 downto 0);
     Hswdata: in std_logic_vector((H_DATA-1) downto 0);
     Hsmaster: in std_logic_vector(3 downto 0);
     Hsmastlock: in std_logic;
     Hsmready: in std_logic;
     Hsready: out std_logic;
     Hsresp: out std_logic_vector(1 downto 0);
     Hsrdata: out std_logic_vector((H_DATA-1) downto 0);
     Hssplit: out std_logic_vector(15 downto 0)
    );
end entity ms1;
architecture structure of ms1 is
component f_unit is
port(freset:in std_logic;
     fclk: in std_logic;
```

```
      fmmaster_id:out std_logic_vector(3 downto 0);
      fres_value:out std_logic_vector((H_DATA-1) downto 0);
      fsend:out std_logic;
      --
      fstate:out f_states;
      ffunc_type: in std_logic_vector((H_DATA-1) downto 0);
      ff_send:in std_logic;
      farg_value: in std_logic_vector((H_DATA-1) downto 0);
      fa_send:in std_logic;
      fsmaster_id:out std_logic_vector(3 downto 0)
);
end component f_unit;
component m_comp is
port(mreset:in std_logic;
      mclk: in std_logic;
      mbusreq: out std_logic;
      mlock: out std_logic;
      mgrant: in std_logic;
      maddr:out std_logic_vector((H_ADDR-1) downto 0);
      mtrans:out std_logic_vector(1 downto 0);
      msize:out std_logic_vector(2 downto 0);
      mburst: out std_logic_vector(2 downto 0);
      mwrite: out std_logic;
      mprot: out std_logic_vector(3 downto 0);
      mwdata: out std_logic_vector((H_DATA-1) downto 0);
      mrdata: in std_logic_vector((H_DATA-1) downto 0);
      mready: in std_logic;
      mresp: in std_logic_vector(1 downto 0);
      --
      mmaster_id: in std_logic_vector(3 downto 0);
      mres_value: in std_logic_vector((H_DATA-1) downto 0);
      msend: in std_logic
);
end component m_comp;
component s_comp is
port(sreset: in std_logic;
      sclk: in std_logic;
      ssel: in std_logic;
      saddr: in std_logic_vector((H_ADDR-1) downto 0);
```

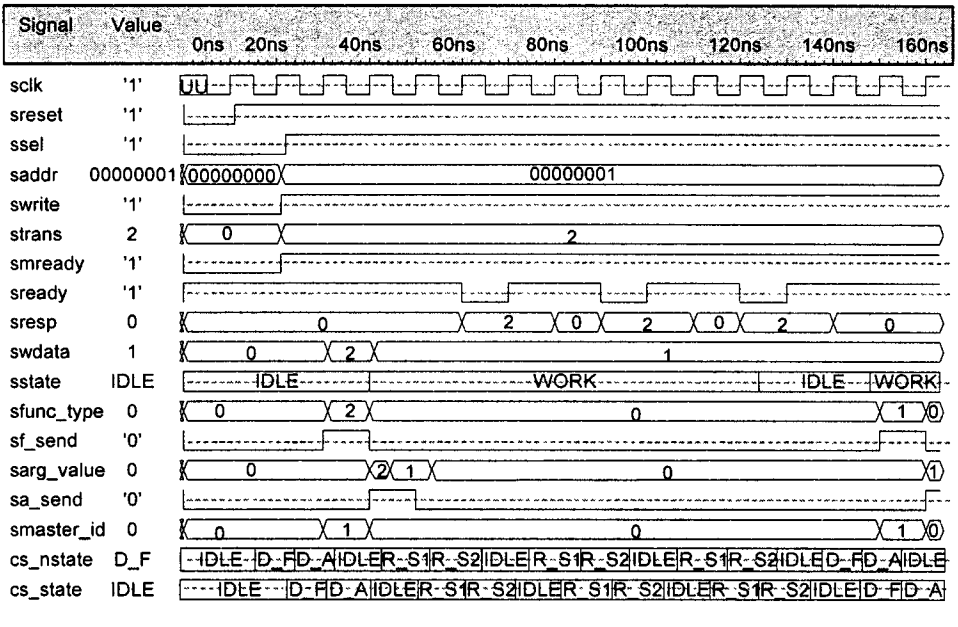
```
    swrite: in std_logic;
    strans: in std_logic_vector(1 downto 0);
    ssize: in std_logic_vector(2 downto 0);
    sburst: in std_logic_vector(2 downto 0);
    swdata: in std_logic_vector((H_DATA-1) downto 0);
    smaster: in std_logic_vector(3 downto 0);
    smastlock: in std_logic;
    smready: in std_logic;
    sready: out std_logic;
    sresp: out std_logic_vector(1 downto 0);
    srdata: out std_logic_vector((H_DATA-1) downto 0);
    ssplit: out std_logic_vector(15 downto 0);
    --
    sstate: in f_states;
    sfunc_type: out std_logic_vector((H_DATA-1) downto 0);
    sf_send: out std_logic;
    sarg_value: out std_logic_vector((H_DATA-1) downto 0);
    sa_send: out std_logic;
    smaster_id: out std_logic_vector(3 downto 0)
);
end component s_comp;
 signal cfmmaster_id: std_logic_vector(3 downto 0);
 signal cfres_value: std_logic_vector((H_DATA-1) downto 0);
 signal cfsend: std_logic;
    --
 signal cfstate: f_states;
 signal cffunc_type:  std_logic_vector((H_DATA-1) downto 0);
 signal cff_send: std_logic;
 signal cfarg_value:  std_logic_vector((H_DATA-1) downto 0);
 signal cfa_send: std_logic;
 signal cfmaster_id: std_logic_vector(3 downto 0) ;
begin
UF: f_unit
port map(freset=>Hreset, fclk=>Hclk,
         fmmaster_id=>cfmmaster_id, fres_value=>cfres_value,
         fsend=>cfsend,
         fstate=>cfstate, ffunc_type=>cffunc_type, ff_send=>cff_send,
         farg_value=>cfarg_value, fa_send=>cfa_send,
         fsmaster_id=>cfmaster_id
        );
```

```
UM: m_comp
port map(mreset=>Hreset, mclk=>Hclk, mbusreq=>Hmbusreq,
         mlock=>Hmlock, mgrant=>Hmgrant, maddr=>Hmaddr, mtrans=>Hmtrans,
         msize=>Hmsize, mburst=>Hmburst, mwrite=>Hmwrite, mprot=>Hmprot,
         mwdata=>Hmwdata, mrdata=>hmrdata, mready=>Hmready,
         mresp=>Hmresp,
         mmaster_id=>cfmmaster_id, mres_value=>cfres_value, msend=>cfsend
        );
US:s_comp
port map(sreset=>Hreset, sclk=>Hclk, ssel=>Hssel, saddr=>Hsaddr,
         swrite=>Hswrite, strans=>Hstrans, ssize=>Hssize,
         sburst=>Hsburst, swdata=>Hswdata, smaster=>Hsmaster,
         smastlock=>Hsmastlock, smready=> Hsmready, sready=>Hsready,
         sresp=>Hsresp, srdata=>Hsrdata, ssplit=>Hssplit,
         sstate=>cfstate, sfunc_type=>cffunc_type, sf_send=>cff_send,
         sarg_value=>cfarg_value, sa_send=>cfa_send,
         smaster_id=>cfmaster_id
        );
end architecture structure;
```

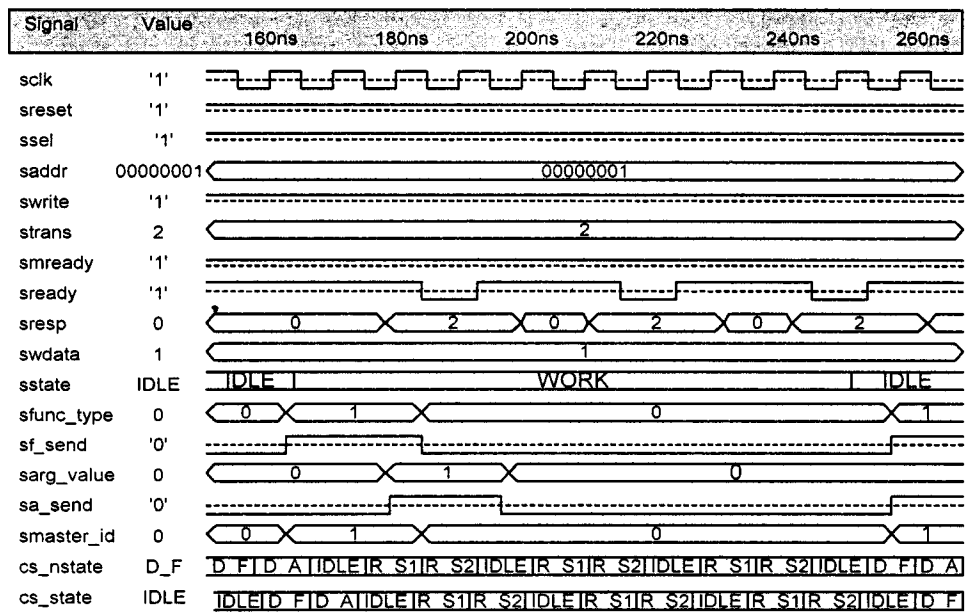
Рассмотрим примеры функционирования вычислителя специальной функции. Сначала рассмотрим функционирование вычислителя, к которому выполняется правильное обращение. Временные диаграммы работы блока ведомого устройства приведены на рис. 5.22 (*а*, *б*). Временные диаграммы работы функционального блока приведены на рис. 5.23 (*а*, *б*). Временные диаграммы работы блока ведущего устройства приведены на рис. 5.24 (*а*, *б*).

В первом такте сигнал сброса установлен в активное состояние, конечные автоматы, соответствующие компонентам, устанавливаются в начальное состояние.

В начале третьего такта сигнал выбора ведомого устройства (для компонента *ведомого устройства* блока спецвычислителя) устанавливается в активное состояние. Начинается передача очередного задания. В конце этого такта ведомое устройство защелкивает значения линий адреса и управления. Организация блока ведомого устройства такова, что адрес можно и не защелкивать, так как он используется только для контроля правильности обращения. Независимо от выставленного адреса, первое передаваемое слово рассматривается как идентификатор функции, второе — как аргумент. Поэтому при передаче аргумента неважно, устанавливается HTRANS=SEQ или сохраняется значение NONSEQ. В конце третьего такта компонент ведомого устройства защелкивает идентификатор ведущего устройства, которое начало обращение. В данном примере идентификатор ведущего устройства равен 1.

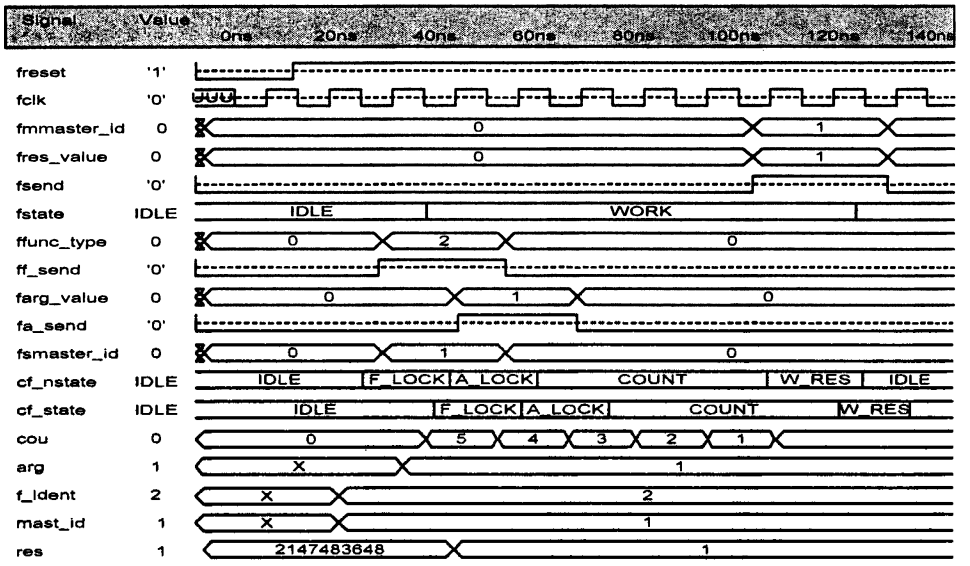


а

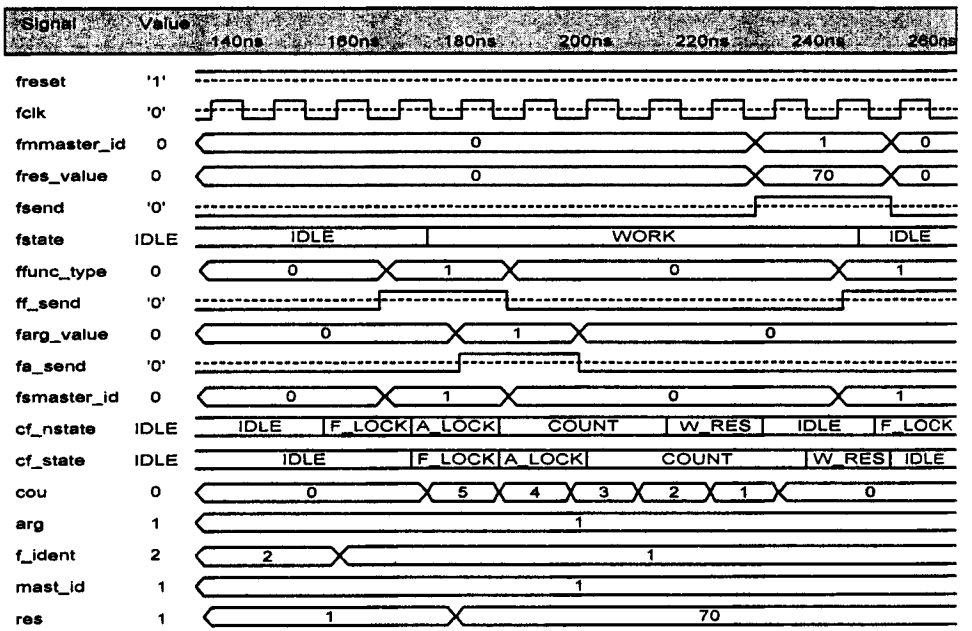


б

Рис. 5.22. Временные диаграммы функционирования блока ведомого устройства. Обработка нормальных обращений (*а* – соответствует модельному времени от 0 до 160 нс., *б* – от 160 нс. до 300 нс.)

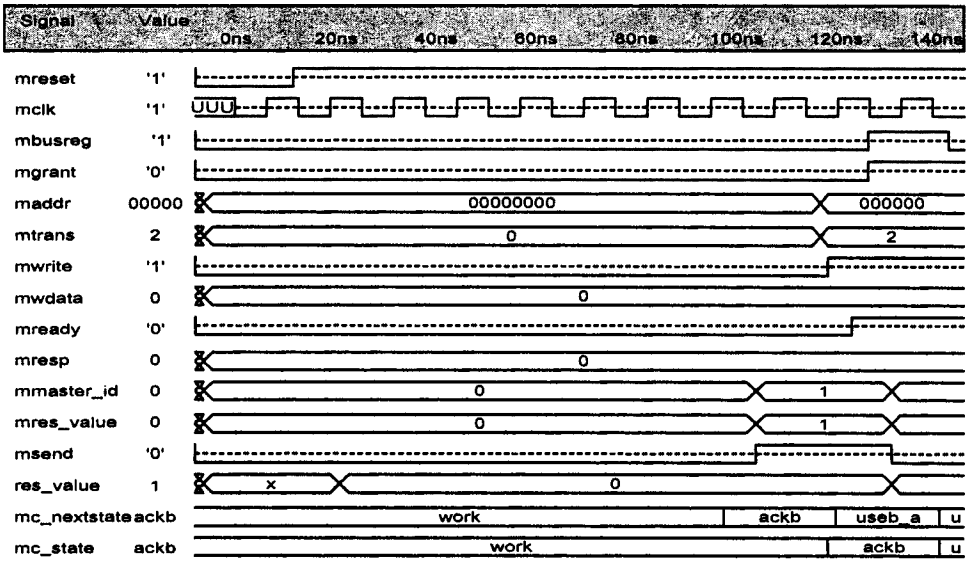


а

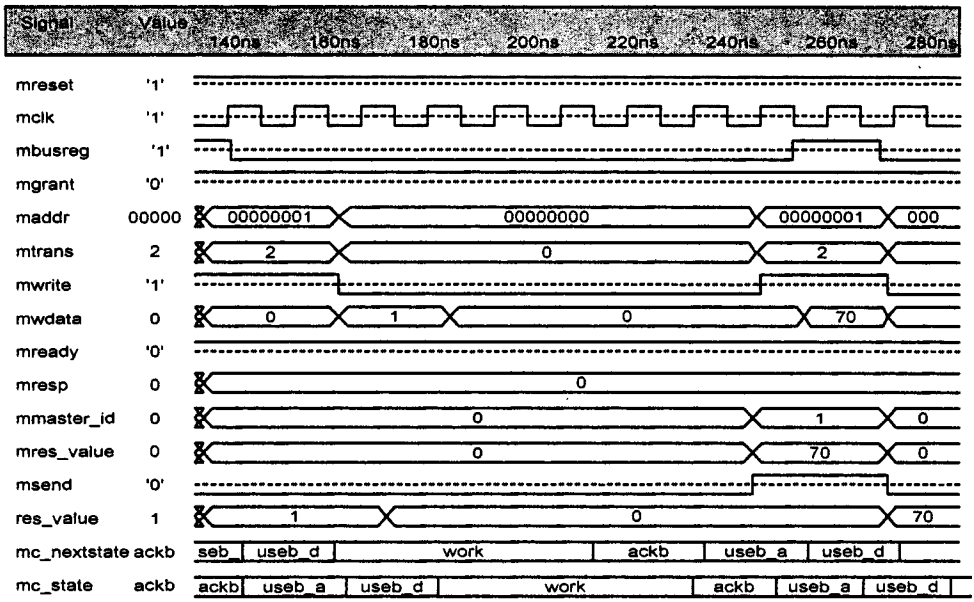


б

Рис. 5.23. Временные диаграммы функционирования функционального блока. Обработка нормальных обращений
(*а* – соответствует модельному времени от 0 до 160 нс., *б* – от 160 нс. до 300 нс.)



а



б

Рис. 5.24. Временные диаграммы функционирования блока ведущего устройства. Обработка нормальных обращений (*а* – соответствует модельному времени от 0 до 160 нс., *б* – от 160 нс. до 300 нс.)

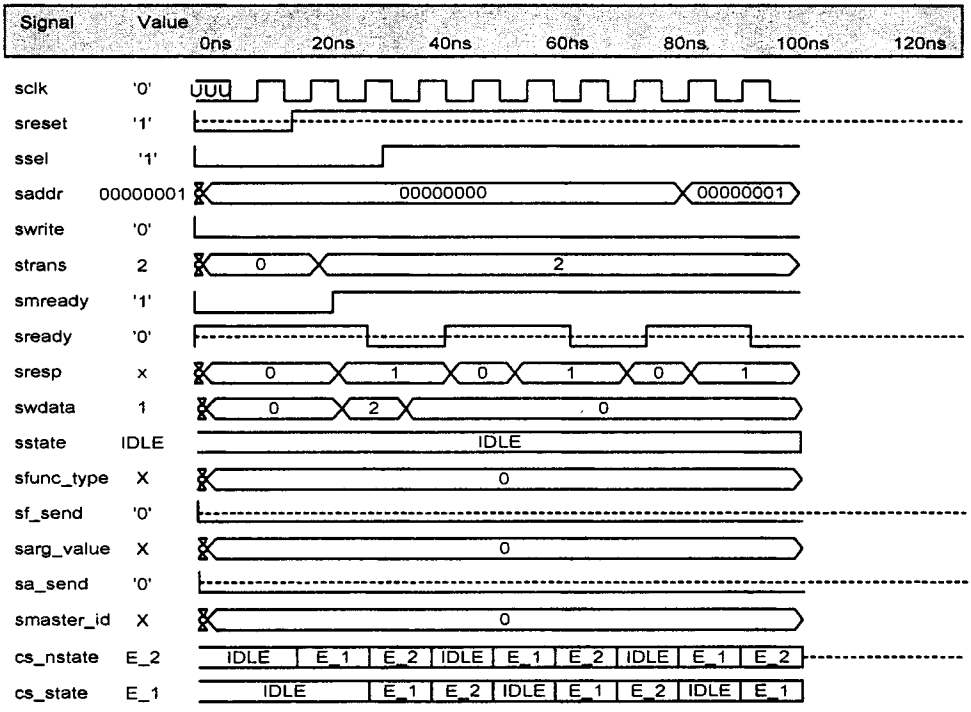
В начале четвертого такта компонент ведомого устройства переходит в состояние D_F. В конце этого такта идентификатор функции должен быть защелкнут в функциональном блоке, поэтому на протяжении этого такта сигнал sf_send установлен в активное значение '1'. В пятом такте компонент ведомого устройства переходит в состояние D_A. В конце этого такта в функциональном блоке должно быть защелкнуто значение аргумента, поэтому сигнал sa_send в этом такте установлен в значение 1. Автомат находится в состояниях D_F и D_A по одному такту, поскольку HTRANS=NONSEQ. Если бы ведущее устройство (здесь имеется в виду процессорное ядро, обращающееся по шине ASB к спецвычислителю) выставило подтверждение BUSY, блок ведомого устройства пребывал бы в этих состояниях в течение большего количества тактов.

На шестом такте компонент ведомого устройства переходит в состояние IDLE. Сигнал выбора этого ведомого устройства по-прежнему активен. Однако теперь функциональный блок находится в состоянии WORK, поэтому блок ведомого устройства выдает в седьмом и восьмом тактах подтверждение RETRY, указывающее, что в данный момент запрос не может быть выполнен. Функциональный блок находится в состоянии WORK до 13-го такта включительно. В этот период все обращения к блоку ведомого устройства заканчиваются подтверждением RETRY. В тринадцатом такте блок ведомого устройства находится в первом состоянии выдачи этого подтверждения, поэтому, несмотря на то, что в 14-м такте функциональный блок уже готов к выполнению следующего задания, блок ведомого устройства в этом такте находится в состоянии R_S2. Только на 15-м такте он переходит в состояние IDLE и защелкивает адрес и управляющую информацию следующего обращения.

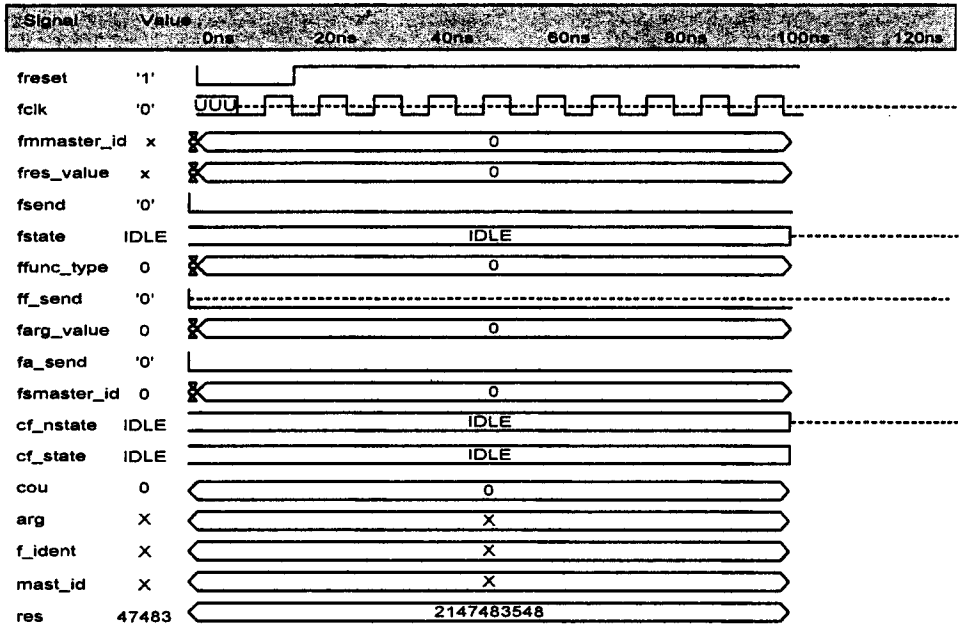
Рассмотрим, как в это время работает компонент *функциональный блок*. После прихода сигнала сброса он переходит в состояние IDLE. В 4-м такте этот блок получает сигнал ff_send='1', следовательно, в конце этого такта он должен защелкнуть значение идентификатора функции и идентификатора ведущего устройства, инициировавшего этот запрос. Они защелкиваются в f_ident и mast_id соответственно. Значение идентификатора функции в этом примере — '2', идентификатора ведущего устройства (инициировавшего обращения к блоку спецвычислителя) — '1'. В 5-м такте, когда fa_send='1', — выполняется защелкивание аргумента в arg. В данном примере значение аргумента равно 1. После этого начинается вычисление результата. В модели результат получается в 7-м такте, но принято, что для его вычисления в моделируемом устройстве необходимо еще шесть тактов, таким образом, результат считается полученным только к 13-му такту. В 13-м такте функциональный блок переходит в состояние W_RES и записывает результат в компонент ведущего устройства блока спецвычислителя. В этом

такте сигнал f_send установлен в активное состояние. В 14-м такте функциональный блок переходит в состояние IDLE и готов к выполнению следующего задания.

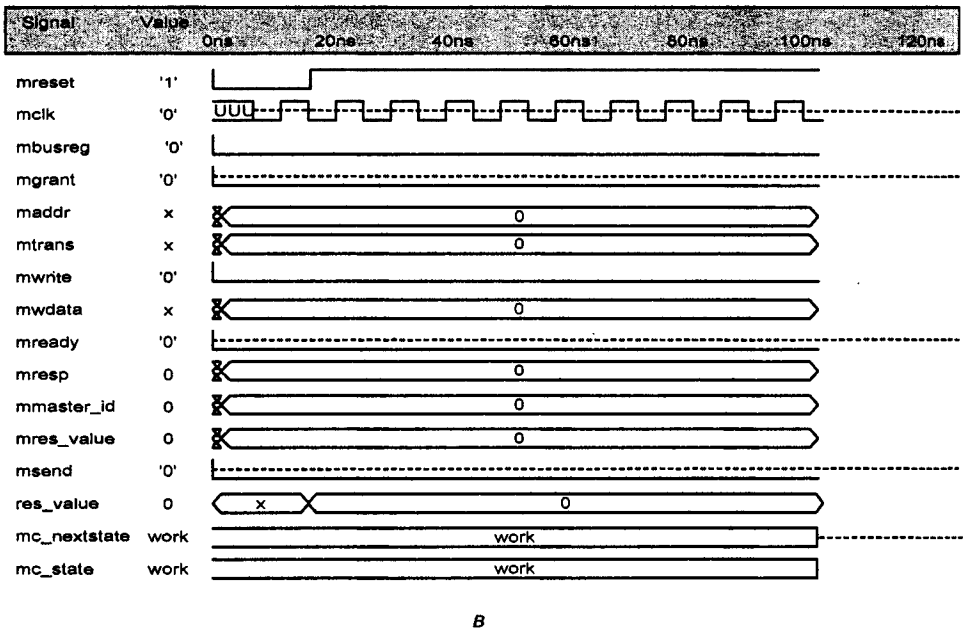
Рассмотрим теперь функционирование компонента *ведущего устройства*. В 13-м такте сигнал msend='1', что указывает на необходимость в конце этого такта защелкнуть результат, полученный функциональным модулем, и начать его передавать. В 14-м такте блок ведущего устройства запрашивает шину AHB, и она ему предоставляется. В 15-м такте выставляется адрес и команда. Адрес обращения определяется в соответствии с идентификатором ведущего устройства, от которого было получено задание. Предполагается, что модуль, выдавший задание, должен иметь компонент ведомого устройства, которому будет возвращен результат. Поскольку HREADY='1', то в 16-м такте на линии данных выставляется результат. HREADY сохраняет значение '1', что указывает на то, что результат успешно записан, поэтому в следующем такте этот компонент переходит в состояние WORK. В этом состоянии компонент ведущего устройства ожидает получения следующего задания от функционального компонента.



а



б



в

Рис. 5.25. Временные диаграммы функционирования блока ведомого устройства — (а), функционального блока — (б), блока ведущего устройства — (в). Обработка неправильного обращения

Рассмотрим теперь обработку неправильного обращения к вычислителю специальной функции (временные диаграммы представлены на рис. 5.25). В этом примере обращение к блоку ведомого устройства начинается также в 3-м такте. Сигнал шины AHB, `HWRITE='0'` (в модели компонента ведомого устройства — сигнал порта `swrite`), осуществляется попытка чтения. Блок ведомого устройства идентифицирует такое обращение как неправильное. Вследствие этого, в 4-м и 5-м такте он выдает код подтверждения `ERROR` (рис. 5.25, *а*), после чего снова переходит в состояние `IDLE`. Остальные блоки вычислителя специальной функции так и остаются в своих начальных состояниях.

Организация шины AMBA AHB для взаимодействия модулей в системах-на-кристалле. Блок связей на основе мультиплексоров

Рассмотрим вариант организации Блока связей модулей для систем-на-кристалле на базе шины AHB. Будем проектировать модуль коммуникационной системы таким образом, чтобы он мог использоваться в системах с различным количеством ведущих и ведомых устройств.

Внутри Блока связей можно выделить два основных узла: Арбитр и Мультиплексор.

К Мультиплексору поступают выходные сигналы от всех ведущих устройств, на их базе формируются входные сигналы для всех ведомых устройств. Аналогично в этот узел поступают выходные сигналы от всех ведомых устройств; на их базе формируются входные сигналы для ведущих устройств. Мультиплексор функционирует под управлением Арбитра.

Арбитр определяет пару ведущее/ведомое устройства, которая будет участвовать в обмене, и формирует управляющие сигналы для блока мультиплексирования. В соответствии с перечнем функций, которые возлагаются на арбитра шины AHB, Арбитру необходимо отслеживать ход выполнения запросов и определять конец очередного запроса.

Арбитр можем представить в виде совокупности следующих компонентов:

- компонент определения следующего ведущего устройства;
- компонент определения следующего ведомого устройства;
- компонент, выполняющий контроль расщепленных транзакций;
- компонент, определяющий конец очередного запроса.

Каждый из компонентов Блока связей должен проектироваться таким образом, чтобы этот модуль мог использоваться в системах-на-кристалле с различным количеством ведущих и ведомых устройств шины AHB. Количество устройств определяется на этапе синтеза. Чтобы проектировщик, не редак-

тируя описание самого модуля, мог определить количество устройств в системе, для которой синтезируется модуль Блока связей, будет использован пакет AMBA_AHB_parameters.vhd (листинг 5.16), в котором описаны две константы Nm — количество ведущих устройств в системе и Ns – количество ведомых устройств в системе. Необходимость наличия такого пакета, а не использование механизма generic, связано с тем, что на базе значений параметров Nm и Ns определяется размер массивов в пакете AMBA_AHB_p.vhd.

Параметризацию разрядности адресов и данных на шине AHB обеспечивают константы H_ADDR и H_DATA, специфицированные в пакете AMBA_AHB_p.

Листинг 5.16

```
Package AMBA_AHB_parameters is
constant Nm: integer:=3;
constant Ns: integer:=4;
end package AMBA_AHB_parameters;
```

Узел Мультиплексор

Узел Мультиплексор описываем на VHDL как один компонент. Рассмотрим описание интерфейса и организацию внутренней структуры Мультиплексора. К Мультиплексору поступают выходные сигналы от всех ведущих и ведомых устройств. Интерфейс компонента должен быть описан таким образом, чтобы количество ведущих и ведомых устройств могло быть различным, количество входных сигналов должно быть функцией от параметров Nm и Ns.

Рассмотрим, как на базе этих двух параметров может быть описан интерфейс Мультиплексора. К Мультиплексору поступают выходные сигналы от всех ведущих устройств. Одноименные сигналы от всех ведущих устройств можно сгруппировать в массивы. Для этого будут использованы типы H1 – HD, описанные в пакете AMBA_AHB_p.vhd (см. листинг 5.8). Эти типы описаны как массивы неопределенной длины. Для того чтобы их использовать при описании портов объекта, необходимо определить размер массива. Для того чтобы размеры массивов соответствовали конфигурации конкретной системы, их размер должен определяться на основе значения константы Nm. Например, описание порта для линий адреса может выглядеть следующим образом:

```
mhaddr: in HA(1 to Nm);
```

Линии адреса являются выходными для ведущих устройств, а для Мультиплексора — входными. Ведущему устройству с номером *j* соответствует элемент массива с номером *j*.

Аналогичным образом к Мультиплексору подключаются выходные сигналы от ведомых устройств. Для определения размеров массивов используется константа Ns.

Рассмотрим, какие управляющие сигналы должны поступать к узлу Мультиплексора от компонента Арбитра. Мультиплексоры, на которые поступают выходные сигналы от ведущих устройств, можно разделить на две большие группы: мультиплексоры адреса и управления, и мультиплексоры данных, записываемых в ведомые устройства. В рамках каждой из этих групп мультиплексоры управляются одним и тем же сигналом, но сигналы управления для каждой из этих групп — это два разных сигнала. Такое разделение связано с тем, что последняя фаза данных одного запроса может перекрываться первой фазой адреса следующего запроса. В результате линии данных могут использоваться одним ведущим устройством, а линии управления и адреса — другим. Мультиплексоры, на которые поступают выходные данные от ведомых устройств, образуют одну группу, работающую под управлением одного сигнала.

Таким образом, для управления Мультиплексором должно использоваться три сигнала:

- номер ведущего устройства, использующего линии адреса и управления;
- номер ведущего устройства, использующего линии данных;
- номер ведомого устройства, использующего линии данных и подтверждения.

Описание компонента Мультиплексор c_muxs приведено в листинге 5.17.

Листинг 5.17

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
entity c_muxs is
port(mhtrans: in H2(1 to Nm);
     mhsize: in H3(1 to Nm);
     mhburst: in H3(1 to Nm);
     mhwrite: in H1(1 to Nm);
     mhprot: in H4(1 to Nm);
     mhaddr: in HA(1 to Nm);
     mhwdata: in HD(1 to Nm);
     --
     mhready: out std_logic;
     mhresp: out std_logic_vector(1 downto 0);
```

```
      mhrdata: out std_logic_vector((H_DATA-1) downto 0);
      --
      shready: in H1(1 to Ns);
      shresp: in H2(1 to Ns);
      shrdata: in HD(1 to Ns);
      --
      shaddr: out std_logic_vector((H_ADDR-1) downto 0);
      shwrite: out std_logic;
      shtrans: out std_logic_vector(1 downto 0);
      shsize: out std_logic_vector(2 downto 0);
      shburst: out std_logic_vector(2 downto 0);
      shwdata: out std_logic_vector((H_DATA-1) downto 0);
      --
      ccAUmast:in natural range 1 to Nm;
      ccDlmast: in natural range 1 to Nm;
      ccSlave: in natural range 1 to Ns
    );
end entity c_muxs;

architecture rtl of c_muxs is
begin
p_m_addr_contr: process(mhaddr,mhwrite,mhtrans,mhsize,mhburst,ccAUmast)
begin
shaddr<=mhaddr(ccAUmast) ;
shwrite<=mhwrite(ccAUmast) ;
shtrans<=mhtrans(ccAUmast) ;
shsize<=mhsize(ccAUmast) ;
shburst<=mhburst(ccAUmast) ;
end process p_m_addr_contr;

p_m_data: process  (mhwdata,ccDlmast)
begin
shwdata<=mhwdata(ccDlmast) ;
end process p_m_data;

p_slave: process (shready,shresp,shrdata,ccSlave)
begin
mhready<=shready(ccSlave) ;
mhresp<=shresp(ccSlave) ;
```

```
mhrdata<=shrdata(ccslave) ;
end process p_slave:;
end architecture rtl;
```

Рассмотрим описание поведения Мультиплексора, включающее в себя три процесса, каждый из которых соответствует одной из групп мультиплексоров.

Процесс `p_m_addr_contr` соответствует группе мультиплексоров, формирующих выходные сигналы от ведущих устройств на линии адреса и управления. Сигнал `ccAUmast` определяет номер ведущего устройства, которое в данный момент времени является собственником этих линий. Совокупности сигналов от ведущих устройств сгруппированы в массивы. На выходы подаются значения элементов массивов, номера которых соответствуют номеру `ccAUmast` ведущего устройства, выбранного Арбитром для доступа к линиям адреса и управления шины AHB.

Процесс `p_m_data` соответствует группе мультиплексоров, формирующих выходные сигналы от ведущих устройств на линии данных. Сигнал `ccDlmast` определяет номер ведущего устройства, получившего доступ к линиям данных шины AHB.

Процесс `p_slave` соответствует группе мультиплексоров, мультиплексирующих выходные сигналы от ведомых устройств. Сигнал `ccSlave` определяет номер ведомого устройства, значения выходных линий которого подаются Мультиплексором на соответствующие линии шины AHB.

Описание Мультиплексора можно выполнить и по-другому: вместо каждого из трех процессов можно использовать отдельный компонент. Это не повлияет на поведение модели и результат синтеза.

Узел Арбитр

Рассмотрим интерфейс узла Арбитр. К этому блоку поступают сигналы `HGRANTx` и `HLOCKx` от всех ведущих устройств системы. Эти сигналы могут быть сгруппированы в массивы, размер которых определяется константой `Nm`. Компонент Арбитра формирует сигнал `HGRANTx` для каждого ведущего устройства, группа этих сигналов так же может быть объединена в массив. Арбитр формирует значение сигнала `HSELx` для всех ведомых устройств; эта группа сигналов может быть объединена в массив, размер которого определяется константой `Ns`.

Арбитр формирует значения управляющих сигналов для узла Мультиплексор. Блок Арбитра формирует значения сигналов `HMASTLOCK` и `HMASTER`. Для того чтобы арбитр мог отслеживать ход выполнения запросов, он должен отслеживать значения сигналов адреса и управления от ведущего устройства, которое в данный момент использует линии адреса и управления, т. е. сигналы адреса и управления с соответствующих выходов узла Мультиплексор. Арбитр должен отслеживать значения сигналов подтверждения от ведомого устройства,

которое участвует в текущем обмене, т. е. сигналы подтверждения с соответствующих выходов блока мультиплексирования. Отслеживание значений этих сигналов необходимо для определения моментов завершения выполнения транзакций. Для контроля выполнения расщепленных транзакций Арбитр должен отслеживать также значение сигнала HSPLIT.

Рассмотрим структуру узла Арбитр (рис. 5.26).

Блок определения конца запроса указывает моменты времени, в которые должны быть зафиксированы номера (идентификаторы) ведущих и ведомых устройств, которые будут владеть шиной. Он формирует два выходных сигнала — selnewmast и gonewmast. Если сигнал selnewmast установлен в '1', это указывает, что должен быть зафиксирован номер ведущего устройства, которое будет являться собственником линий адреса и управления. Установка в '1' сигнала gonewmast указывает, что должен быть зафиксирован номер ведущего устройства, которое будет владеть линиями данных, и номер ведомого устройства, которое будет владеть линиями данных и подтверждения от ведомого устройства. Если выполняется пакетный запрос неопределенной длины, то его конец определяется по сбросу сигнала запроса шины от ведущего устройства, владеющего линиями адреса и управления. Пусть BR – сигнал запроса шины от ведущего устройства, владеющего линиями адреса и данных. Он поступает к блоку определения конца запроса от блока определения ведущего устройства.

Блок определения ведущего устройства формирует сигналы HGRANT для ведущих устройств, значения сигналов HMASTLOCK и HMASTER, а также сигналы AUmast и Dmast, предназначенные для управления группами мультиплексоров, определяющих значения выходов ведущих устройств на общую шину и сигнал BR. В ходе своего функционирования этот блок анализирует значения сигналов HBUSREQ, HLOCK от ведущих устройств и значение сигнала MMACK от блока контроля расщепленных транзакций. На основе значений этих сигналов блок определения ведущего устройства определяет ведущие устройства, которые являются собственниками линий адреса, управления и данных. В соответствии со значениями сигналов selnewmast и gonewmast определяются моменты времени, в которые происходит защелкивание номеров ведущих устройств — собственников шин.

Блок определения активного ведомого устройства формирует сигналы HSEL для всех ведомых устройств, а также сигнал cSlave, предназначенный для управления группой мультиплексоров, определяющих значения выходов ведомых устройств на общую шину. В ходе функционирования этих сигналов он анализирует значение, выставленное на шину адреса, в соответствии с чем блок определяет ведомое устройство, к которому обращается ведущее устройство, владеющее линиями адреса и управления. По значению сигнала gonewmast, поступающему от блока определения конца запроса, определяются моменты времени, в которые происходит защелкивание номера ведомого устройства — собственника шины.

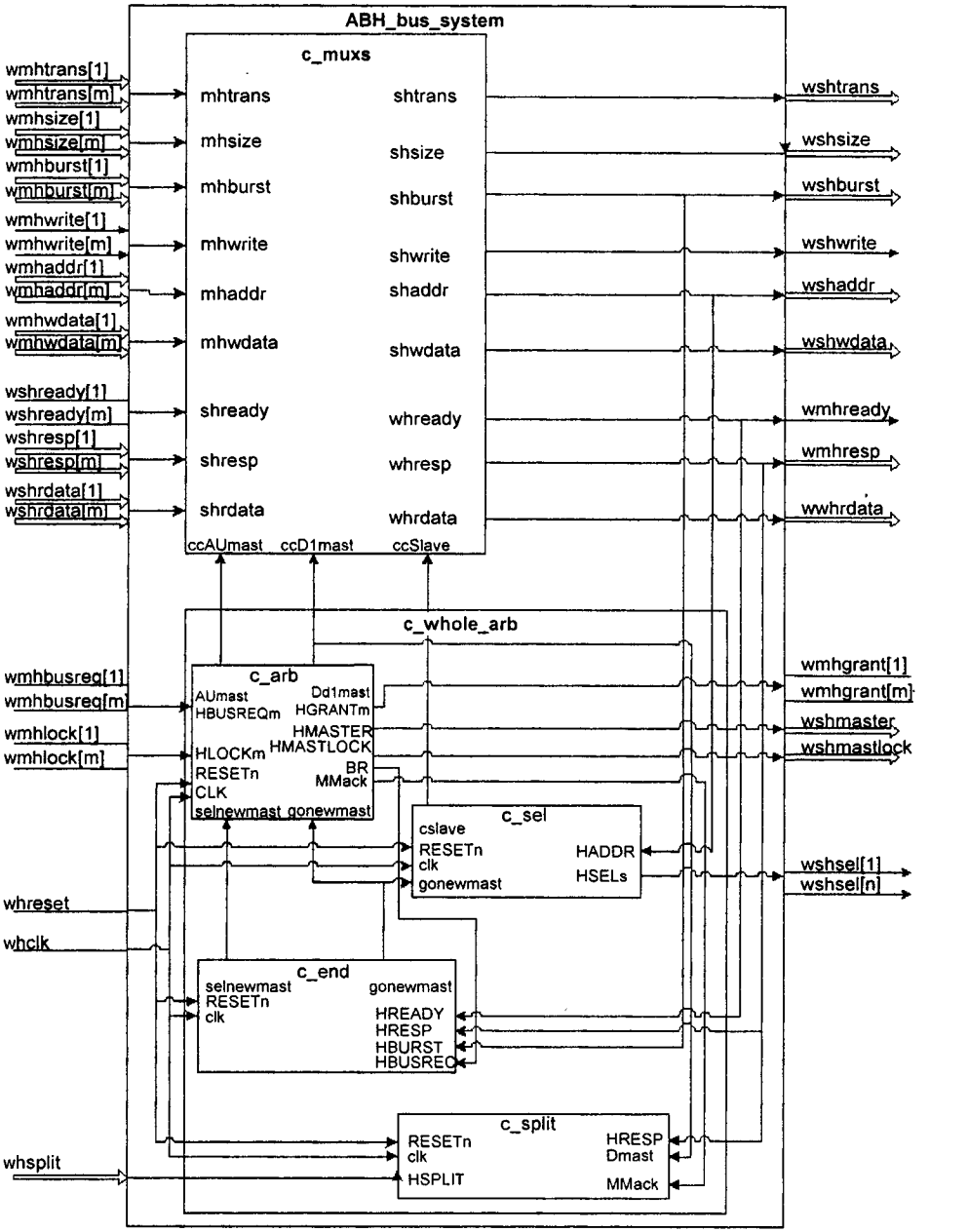


Рис. 5.26. Структура Арбитра

Блок контроля расщепленных транзакций определяет моменты времени начала и завершения расщепленных транзакций. Он формирует сигнал `Mmack`, опреде-

ляющий номера ведущих устройств, которые участвуют в незавершенных расщепленных транзакциях. В ходе функционирования, этот блок анализирует значения сигналов Dmast от блока определения следующего ведущего устройства, а также HRESP – от ведомого устройства, владеющего шинами данных и подтверждения. На основе этих сигналов он определяет моменты начала расщепленных транзакций и ведущие устройства, участвующие в этих транзакциях. По сигналу HSPLIT блок контроля расщепленных транзакций определяет моменты завершения расщепленных транзакций.

Узел Арбитр опишем на VHDL структурно, как совокупность компонентов, сопоставляемым перечисленным блокам.

Компонент определения конца запроса

Рассмотрим функционирование блока определения конца запроса. Этот блок должен определить момент времени, в который линии адреса и управления переходят к другому собственнику (это последняя фаза адреса текущего запроса), а также момент времени, в который линии данных переходят к другому собственнику (это последняя фаза данных текущего запроса).

Последняя фаза данных текущего запроса начинается в следующем такте, после завершения последней фазы адреса текущего запроса. В последнем такте фазы данных ведомое устройство устанавливает HREADY='1'. Таким образом, после обнаружения момента завершения последней фазы адреса очередного запроса, этот блок, при обнаружении HREADY='1', определяет момент завершения последней фазы данных очередного запроса.

Рассмотрим теперь, как определяется последняя фаза адреса текущего запроса. Блок определения конца запроса анализирует значение на шине HBURST. В соответствии с этим, он определяет, выполняется ли в данный момент фаза адреса одиночного запроса, пакетного запроса неопределенной длины или пакетного запроса фиксированной длины. Если выполняется одиночный запрос, то его первая фаза данных является и последней. Последний такт фазы адреса определяется так же, как и последний такт фазы данных, по HREADY='1'.

Если выполняется пакетный запрос неопределенной длины, то на начало последней фазы адреса указывает сигнал сброса запроса шины от ведущего устройства, которое в данный момент использует линии адреса и управления (сигнал HBUSREQ). Вообще говоря, это несколько упрощенная схема, здесь не учитывается возможность того, что ведущее устройство может выполнить несколько запросов подряд. В этом случае, в последней фазе запроса неопределенной длины, ведущее устройство не сбрасывает сигнал запроса шины. При данной схеме определения конца запроса неопределенной длины, конец запроса в этой ситуации не будет обнаружен, и шина останется в собственности прежнего ведущего устройства. Это может не соответствовать выбранной схеме арбитража, если по завершении очередно-

го запроса шина должна быть предоставлена ведущему устройству, имеющему наивысший приоритет среди ведущих устройств, запросивших шину. Для того чтобы точно определить конец запроса, необходимо еще отслеживать значение HTRANS. HTRNS=NONSEQ указывает на то, что адрес очередного обмена не связан с адресом предыдущего обмена, т. е. начинается новый запрос.

Если выполняется пакетный запрос фиксированной длины, то в блоке определения конца запроса просто выполняется подсчет количества выполненных обменов, на основе которого и отслеживается конец запроса.

В соответствии с этим, конечный автомат блока определения конца запроса может иметь два состояния: st1 — выполняется первая адресная фаза очередного запроса и stn — выполняется не первая адресная фаза пакетного запроса.

Описание блока определения конца запроса приведено в листинге 5.18.

Листинг 5.18

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
 entity c_end is
 port (RESETn: in std_logic;
       clk: in std_logic;
       HREADY: in std_logic;
       HBURST: in std_logic_vector(2 downto 0);
       HRESP: in std_logic_vector(1 downto 0);
       HBUSREQ: in std_logic  ;
       selnewmast: out std_logic;
       gonewmast: out std_logic
     ) ;
 end entity c_end;
architecture rtl of c_end is
 type c_end_state is (st1,stn);
 signal c_state, c_nextstate: c_end_state;
 signal countburst: natural range 0 to 15;
 begin
p_state: process  (RESETn, clk, HREADY)
```

```
 begin
 if RESETn='0' then c_state<=st1  ;
 else
   if rising_edge(clk) then
     if HREADY='1' then c_state<=c_nextstate; end if;
   end if;
 end if;
 end process p_state;

p_countburst: process (RESETn, HREADY, clk, HBURST)
 begin
 if RESETn='0' then countburst<=0;
 else
  if rising_edge(clk) then
   if HREADY='1' then
     if c_state=st1 then
      case HBURST is
       when HBURST_WRAP4|HBURST_INCR4 =>  countburst<=2;
       when HBURST_WRAP8|HBURST_INCR8 =>  countburst<=6;
       when HBURST_WRAP16|HBURST_INCR16 =>  countburst<=12;
       when others =>         countburst<=0;
      end case;
     end if;
     if countburst>0 then countburst<=countburst-1; end if;
   end if;
  end if;
 end if;
 end process p_countburst;

p_out: process (c_nextstate,clk,HREADY,HBURST,countburst,HBUSREQ)
begin
if HREADY='0' then selnewmast<='0'; gonewmast<='0';
  else
    if HBURST=HBURST_SINGLE then selnewmast<='1'; gonewmast<='1';
                                 c_nextstate<=st1;
       else if HBURST=HBURST_INCR then
                 if HBUSREQ='1' then selnewmast<='0'; gonewmast<='1';
                                     c_nextstate<=stn;
                    else selnewmast<='1'; gonewmast<='1';
```

```
                    c_nextstate<=st1;
                end if;
            else if countburst>0 then selnewmast<='0';
                                     gonewmast<='1';c_nextstate<=stn;
                    else selnewmast<='1'; gonewmast<='1';
                        c_nextstate<=st1;
                end if;
        end if;
    end if;
end if;
end process p_out;
end architecture rtl;
```

Рассмотрим функционирование этого блока. Внутри архитектурного описания блока описан тип `c_end_state`, определяющий множество состояний автомата. Сигналы `c_state` и `c_nextstate` используются для хранения текущего и следующего состояния автомата. Сигнал `countburst` используется для подсчета количества уже выполненных обращений в рамках пакета.

В процессе `p_state` реализуется переход из состояния в состояние. При активном значении сигнала сброса, автомат устанавливается в начальное состояние `st1`.

Процесс `p_countburst` используется для подсчета количества обменов данных, выполненных в рамках обработки пакета определенной длины. В нем определяется значение сигнала `countburst`. При активном значении сигнала сброса значение этого сигнала устанавливается в `'0'`. Далее, в конце первой адресной фазы каждого запроса, проверяется, является ли этот запрос запросом определенной длины. Если это так, то сигналу `countburst` присваивается значение, на 2 меньшее, чем количество слов данных, обрабатываемых в рамках запроса. В конце очередной адресной фазы каждого запроса проверяется значение этого сигнала: если оно больше 0, то из него вычитается 1. Например, если размер запроса — 4 слова, то в конце первой адресной фазы значение `countburst` будет установлено в 2, в конце второй — 1, в конце третьей — 0. Четвертая адресная фаза является последней, и значение `countburst=0` укажет на это.

Процесс `p_out` используется для определения значений выходных сигналов и следующего состояния. Если ведомое устройство, участвующее в обмене, установило сигнал `HREADY='0'`, то выходные сигналы `selnewmast` и `gonewmast` установлены в неактивное значение '0', поскольку текущий обмен будет продолжен в следующем такте. Если же `HREADY='1'`, в следующем такте будет выполняться новый обмен данными. Если текущей является адресная фаза одиночного обмена или последняя адресная фаза пакетного обмена, то

устанавливается сигнал selnewmast='1'. Это указывает на то, что в следующем такте линии адреса и управления могут быть переданы другому ведущему устройству; следующее состояние будет st1, состояние начала выполнения запроса. Если же выполняется не последняя адресная фаза пакетного запроса, то линии адреса и управления не могут поменять собственника — устанавливается selnewmast='0' и следующее состояние stn. Во всех этих случаях, сигнал gonewmast установлен в '1', что указывает на то, что в начале следующего обмена линии данных должны быть предоставлены тому ведущему устройству, которое в предыдущем обмене являлось собственником линий адреса и управления.

Компонент выбора ведущего устройства

Организация блока выбора ведущего устройства зависит от используемой в системе схемы приоритетов. Рассмотрим для блока выбора ведущего устройства простейший вариант, при котором ведущие устройства имеют абсолютные приоритеты, определяющиеся их идентификаторами. Ведущее устройство с идентификатором 1 имеет наивысший приоритет и является ведущим устройством по умолчанию. Описание компонента выбора ведущего устройства приведено в листинге 5.19.

Листинг 5.19

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
entity c_arb is
port (RESETn: in std_logic;
      clk: in std_logic;
      HREADY: in std_logic;
      selnewmast: in  std_logic;
      gonewmast: in  std_logic;
      HBUSREQm: in std_logic_vector(1 to Nm);
      HLOCKm: in std_logic_vector(1 to Nm);
      MMack: in std_logic_vector(0 to 15);
      HGRANTm: out std_logic_vector(1 to Nm);
      AUmast: inout natural range 1 to Nm;
      Ddlmast: out natural range 1 to Nm;
      HMASTLOCK: out std_logic;
```

```
      HMASTER : out std_logic_vector(3 downto 0);
      BR: out std_logic
    ) ;
end entity c_arb;
architecture rtl of c_arb is
signal samast: natural range 1 to 16;
signal intlock: std_logic;
signal saum: natural range 1 to Nm;
begin
p_selmast: process (HBUSREQm, MMack)
variable amast: natural range 1 to 16;
begin
amast:=0;
for i in Nm downto 1 loop
  If HBUSREQm(i)='1' and MMack(i-1)='0' then  amast:=i; end if;
end loop;
if amast/=0 then samast<=amast;
 else
  for i in Nm downto 1 loop
  if MMack(i-1) ='0' then samast<=i; end if;
  end loop;
 end if;
samast<=amast;
end process p_selmast;

p_lockDmast: process (RESETn,clk,HREADY,gonewmast,saum)
begin
 if RESETn='0' then Dd1mast<=2;
 else
if rising_edge(clk) then
     if HREADY='1' then
       if gonewmast='1' then Dd1mast<=saum; end if;
     end if;
  end if;
 end if;
end process p_lockDmast;

p_lockmast: process (RESETn, clk,HREADY, gonewmast, selnewmast, samast)
```

```
begin
if RESETn='0' then AUmast<=1; saum<=1;
                   HMASTLOCK<='0';HMASTER<="0001";
 else
   if rising_edge(clk) then
      if HREADY='1' then
        if gonewmast='1' then HMASTLOCK<=intlock; end if;
        if selnewmast='1' then
           if samast>0 then AUmast<=samast; saum<=samast;
            else AUmast<=1;saum<=1; end if;
           if samast>0 then intlock<=HLOCKm(samast);
               HMASTER<=conv_std_logic_vector(conv_unsigned(samast,4),4);
              else intlock<=HLOCKm(1); HMASTER<="0001"; end if;
       end if;
     end if;
   end if;
end if;
end process p_lockmast;

p_grant: process (AUmast)
begin
for i in 1 to Nm loop
 if i=AUmast then HGRANTm(i)<='1'; else HGRANTm(i)<='0'; end if;
end loop;
end process p_grant;
p_BR: process(AUmast, HBUSREQm)
begin
    BR<=HBUSREQm(AUmast);
end process p_BR;
end architecture rtl;
```

Рассмотрим функционирование блока выбора следующего ведущего устройства. Ведущее устройство — кандидат на использование шины, выбирается каждый раз при изменении на линиях запроса шины или изменении сигнала Mmack, отражающего наличие расщепленных транзакций. Если selnewmast='1', то по восходящему фронту сигнала тактирования кандидат на использование шины становится владельцем шины адреса и управления. Если gonewmast='1', то по восходящему фронту сигнала тактирования ведущее устройство, владеющее шинами адреса и управления, получает в свое распоряжение шину данных.

Внутри архитектурного описания используются следующие сигналы. Сигнал samast используется для хранения идентификатора ведущего устройства, выбранного для следующего запроса. Сигнал intlock используется для хранения информации о наличии блокировки в текущем запросе. Сигнал saum используется для хранения идентификатора ведущего устройства, которое в данный момент времени владеет линиями адреса и управления.

Процесс p_selmast используется для выбора ведущего устройства, которому будет предоставлена шина адреса и управления после завершения очередного запроса. Этот процесс выполняется каждый раз, когда происходит изменение на линиях запроса шины от ведущих устройств и изменение сигнала Mmack. Внутри процесса описана переменная amast, используемая при выборе ведущего устройства, которое будет использовать шину. Сначала этой переменной присваивается значение 0. Далее в цикле просматривается наличие запросов от ведущих устройств, начиная с последнего, имеющего самый низкий приоритет. При каждом выполнении цикла, если имеется запрос шины от ведущего устройства, а соответствующий разряд сигнала Mmack='0' (что указывает на отсутствие для данного ведущего устройства незавершенной транзакции), переменной amast присваивается значение идентификатора этого ведущего устройства. Таким образом, после завершения цикла в переменной amast хранится идентификатор ведущего устройства, имеющего наивысший приоритет среди запросивших шину. Если значение amast осталось равным 0, — значит, в данный момент ни одно ведущее устройство не запрашивает шину.

В этом случае шина должна быть предоставлена ведущему устройству по умолчанию. Однако возможна ситуация, когда для ведущего устройства по умолчанию существует расщепленная транзакция, не готовая к завершению. В этом случае шина ему предоставлена быть не может. Пусть, в этом случае, она предоставляется ведущему устройству, имеющему наивысший приоритет среди тех, для которых не существует незавершенных расщепленных транзакций. Для этого используется цикл, аналогичный предыдущему. После того, как выбрано ведущее устройство — кандидат на использование шины, его идентификатор сохраняется в сигнале samast.

Процесс p_lockDmast используется для защелкивания идентификатора ведущего устройства, становящегося собственником линий данных.

Процесс p_lockmast используется для защелкивания идентификатора ведущего устройства, становящегося собственником линий адреса и управления. В этом процессе определяются также значения выходных сигналов HMASTER и HMASTLOCK. При приходе сигнала сброса, в качестве ведущего устройства по умолчанию, выбирается ведущее устройство с номером 1. Если selnewmast='1', то значение сигнала samast по восходящему фронту тактового импульса фиксируется в saum и Aumast. В соответствии со значением samast определяется значение сигнала HMASTER. Значение сигнала HLOCK от ведуще-

го устройства, которое становится собственником линий данных и управления, фиксируется в сигнале intlock. При очередном запуске процесса, после того как это ведущее устройство станет собственником шины данных, значение сигнала intlock выдается на линию HMASTLOCK.

Процесс p_grant используется для определения значений выходных сигналов HGRANT. Эти значения определяются на основе сигнала AUmast.

Процесс p_BR используется для определения значения сигнала BR — запрос шины того ведущего устройства, которое в данный момент владеет линиями адреса и управления. Этот сигнал необходим блоку определения конца запроса для работы с пакетными запросами переменной длины.

Компонент выбора ведомого устройства

Выбор ведомого устройства, участвующего в обмене, осуществляется на базе адреса, выставляемого ведущим устройством. Рассмотрим организацию простейшего компонента выбора ведомого устройства.

Выбор ведомого устройства осуществляется на основе таблицы граничных адресов ведомых устройств, содержимое которой жестко определено в описании компонента. Для каждого ведомого устройства определена пара адресов — начальный и конечный. Все адреса, попадающие в диапазон от начального адреса (включительно) до конечного адреса (не включая его), считаются относящимися к данному ведомому устройству. Функционирование этого компонента подобно функционированию компонента выбора ведущего устройства. Каждый раз, когда на шине данных появляется новый адрес, определяется ведомое устройство, которому этот адрес соответствует. Однако фиксируется номер ведомого устройства, которому будет предоставлена шина, по восходящему фронту тактового импульса и только если gonewmast='1'.

Для того чтобы сократить размер временных диаграмм и облегчить их читаемость, в листинге и на рисунках разрядность линий адреса и данных считается равной 16, а не 32.

Описание компонента выбора ведомого устройства приведено в листинге 5.20.

Листинг 5.20

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
```

```
entity c_sel is
port (RESETn: in std_logic;
      clk: in std_logic;
      HADDR: in std_logic_vector((H_ADDR-1) downto 0);
      gonewmast: in  std_logic;
      HSELs: out std_logic_vector(1 to Ns);
      cslave: inout natural range 1 to Ns
    ) ;
end entity c_sel;
architecture rtl of c_sel is
type  t_addr is array(1 to 2) of std_logic_vector((H_ADDR-1) downto 0);
type t_addr_table is array(1 to 4) of t_addr;
signal addr_table: t_addr_table;
signal ssl: natural;
begin
p_sel_slave: process (HADDR)
variable sl: natural;
begin
sl:=1;
for i in 1 to Ns loop
if (HADDR>=addr_table(i)(1)) and (HADDR<addr_table(i)(2)) then sl:=i;
  end if;
end loop;
ssl<=sl;
end process p_sel_slave;

p_lock_slave: process(gonewmast,ssl,clk)
begin
if RESETn='0' then
addr_table<= ((x"0000",x"0010"),
              (x"0010",x"0100"),
              (x"0100",x"0111"),
              (x"0111",x"1111"));
cslave<=1;
else
  if rising_edge(clk) then
    if gonewmast='1' then cslave<=ssl; end if;
  end if;
```

```
end if;
end process p_lock_slave;

p_hsel: process(ssl)
begin
for i in 1 to Ns loop
      if ssl=i then HSELs(i)<='1'; else HSELs(i)<='0'; end if;
end loop;
end process p_hsel;
end architecture rtl;
```

Рассмотрим функционирование компонента выбора ведомого устройства. Внутри архитектурного описания введены два новых типа: `t_addr`, предназначенный для хранения пары адресов, соответствующих каждому ведомому устройству, и `t_addr_table`, предназначенный для таблицы адресов, соответствующих всем ведомым устройствам. Сигнал `addr_table` используется для хранения таблицы адресов, сигнал `ssl` — для хранения идентификатора ведомого устройства, являющегося кандидатом на участие в обмене.

Процесс `p_sel_slave` используется для определения ведомого устройства — кандидата для участия в обмене. Процесс выполняется каждый раз при изменении адреса. В цикле определяется номер ведомого устройства, которому соответствует выставленный адрес. Этот номер заносится в переменную `sl`. После выполнения цикла значение переменной `sl` присваивается сигналу `ssl`. Такая организация процесса является упрощенной (здесь не учитывается возможность выставления на шину адреса, не соответствующего ни одному из ведомых устройств системы).

Процесс `p_lock_slave` используется для защелкивания идентификатора ведомого устройства при `gonewmast='1'`. При приходе сигнала сброса выбирается ведомое устройство с номером 1 — ведомое по умолчанию.

Процесс `p_hsel` используется для формирования сигналов выборки `HSELs` для всех ведомых устройств, в соответствии с идентификатором ведомого устройства, выбранного для участия в обмене. Процесс запускается, когда меняется номер выбранного устройства (сигнал `ssl`).

Компонент контроля расщепленных транзакций

Этот компонент отслеживает моменты начала и конца расщепления транзакций. Информация об этом отражается в сигнале `Mmack`, в котором каждому ведущему устройству соответствует один бит. Бит 0 соответствует ведущему устройству 1, бит 1 — ведомому устройству 2 и так далее. Если ведомое устройство, участвующее в обмене, выставляет `HRESP=SPLIT`, то бит `Mmack`, соответствующий ведущему устройству, владеющему линиями данных, устанавливается в `'1'`, для этого ведущего устройства начинается рас-

щепленная транзакция. Если по переднему фронту тактового импульса какой-либо разряд сигнала HSPLIT установлен в '1' (ведомое устройство готово завершить расщепленную транзакцию), то соответствующий бит Mmack устанавливается в '0'.

Описание компонента контроля расщепленных транзакций приведено в листинге 5.21.

Листинг 5.21

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
entity c_split is
port(RESETn: in std_logic;
     clk: in std_logic;
     HRESP: in std_logic_vector(1 downto 0);
     HSPLIT: in std_logic_vector(15 downto 0);
     Dmast: in natural range 1 to Nm;
     MMack: out std_logic_vector(0 to 15)
    );
end entity c_split;
architecture rtl of c_split is
begin
process (RESETn,clk,HSPLIT,HRESP)
begin
if RESETn=0 then MMack<=(others=>'0');
  else
    if rising_edge(clk) then
      for i in 0 to 15 loop
       if HSPLIT(i)='1' then MMack(i)<='0'; end if;
      end loop;
      if HRESP=HRESP_SPLIT then MMack(Dmast-1)<='1'; end if;
    end if;
end if;
end process;
end architecture rtl;
```

Структурное описание узла Арбитр

Структурное описание Арбитра приведено в листинге 5.22.

Листинг 5.22

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
entity c_whole_arb is
port (WRESETn: in std_logic;
      Wclk: in std_logic;
      WHBUSREQm: in std_logic_vector(1 to Nm);
      WHLOCKm: in std_logic_vector(1 to Nm);
      WHREADY: in std_logic;
      WHBURST: in std_logic_vector(2 downto 0);
      WHRESP: in std_logic_vector(1 downto 0);
      WHADDR: in std_logic_vector((H_ADDR-1) downto 0);
      WHSPLIT: in std_logic_vector(15 downto 0);
      WHGRANTm: out std_logic_vector(1 to Nm);
      WHMASTLOCK: out std_logic;
      WHMASTER : out std_logic_vector(3 downto 0);
      WHSELs: out std_logic_vector(1 to Ns);
      WAUmast: inout natural range 1 to Nm;
      WD1mast: out natural range 1 to Nm;
      Wcslave: inout natural range 1 to Ns
     );
end entity c_whole_arb;

architecture structure of c_whole_arb is
-- описание компонентов
component c_arb is
port (RESETn: in std_logic;
      clk: in std_logic;
      HREADY: in std_logic;
      selnewmast: in  std_logic;
      gonewmast: in  std_logic;
      HBUSREQm: in std_logic_vector(1 to Nm);
```

```
      HLOCKm: in std_logic_vector(1 to Nm);
      MMack: in std_logic_vector(0 to 15);
      HGRANTm: out std_logic_vector(1 to Nm);
      AUmast: out natural range 1 to Nm;
      Dd1mast: out natural range 1 to Nm;
      HMASTLOCK: out std_logic;
      HMASTER : out std_logic_vector(3 downto 0);
      BR: out std_logic
    ) ;
end component c_arb;
component c_end is
 port (RESETn: in std_logic;
      clk: in std_logic;
      HREADY: in std_logic;
      HBURST: in std_logic_vector(2 downto 0);
      HRESP: in std_logic_vector(1 downto 0);
      HBUSREQ: in std_logic  ;
      selnewmast: out std_logic;
      gonewmast: out std_logic
    ) ;
end component c_end;
component c_sel is
port (RESETn: in std_logic;
      clk: in std_logic;
      HADDR: in std_logic_vector((HADDR-1) downto 0);
      gonewmast: in  std_logic;
      HSELs: out std_logic_vector(1 to Ns);
      cslave: inout natural range 1 to Ns
       ) ;
end component c_sel;
component c_split is
port(RESETn: in std_logic;
    clk: in std_logic;
    HRESP: in std_logic_vector(1 downto 0);
    HSPLIT: in std_logic_vector(15 downto 0);
    Dmast: in natural range 1 to Nm;
    MMack: out std_logic_vector(0 to 15)
   );
end component c_split;
```

```
-- описание внутренних сигналов архитектурного тела
signal s_selnewmast: std_logic;
signal s_gonewmast: std_logic;
signal s_dmast: natural range 1 to Nm;
signal s_mmask: std_logic_vector(0 to 15);
signal s_BR: std_logic;
-- включение и связывание компонентов-экземпляров струкутры
begin
U1: c_arb
port map (RESETn=>WRESETn,     clk =>Wclk,    HREADY =>WHREADY,
       selnewmast =>s_selnewmast,  gonewmast =>s_gonewmast,
       HBUSREQm =>WHBUSREQm,    HLOCKm =>WHLOCKm,   MMack =>s_mmask,
       HGRANTm =>WHGRANTm,     AUmast =>WAUmast,    Ddlmast =>s_dmast,
       HMASTLOCK =>WHMASTLOCK, HMASTER =>WHMASTER, BR=>s_BR) ;
U2: c_end
 port map (RESETn =>WRESETn,  clk =>Wclk,    HREADY =>WHREADY,
       HBURST =>WHBURST,   HRESP =>WHRESP,     HBUSREQ =>s_BR,
       selnewmast =>s_selnewmast, gonewmast =>s_gonewmast) ;
U3: c_sel
port map (RESETn=>WRESETn, clk=> Wclk,  HADDR => WHADDR,
       gonewmast =>s_gonewmast,  HSELs =>WHSELs, cslave =>Wcslave) ;
U4: c_split
port map(RESETn=>WRESETn,  clk=> Wclk,  HRESP=>WHRESP,
     HSPLIT =>WHSPLIT, Dmast =>s_dmast,  MMack =>s_mmask);
WD1mast<=s_dmast;
end architecture structure;
```

Структурное описание Блока связей на основе мультиплексоров

Таким образом, мы имеем описание всех узлов, составляющих Блок связей на основе мультиплексоров: узла Мультиплексор (листинг 5.17) и узла Арбитр (листинг 5.22). Используя их, даем структурное описание Блока связей на основе мультиплексоров, реализующего взаимодействие модулей системы-на-кристалле по стандарту шины AHB (с небольшими упрощениями), листинг 5.23.

Листинг 5.23

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
use IEEE.std_logic_unsigned.all;
use AMBA_AHB_p.AMBA_AHB_p.all;
use AMBA_AHB_parameters.AMBA_AHB_parameters.all;
entity AHB_bus_system is
port(wmhbusreq:in H1(1 to Nm);
      wmhlock: in H1(1 to Nm) ;
      wmhtrans: in H2(1 to Nm);
      wmhsize: in H3(1 to Nm);
      wmhburst: in H3(1 to Nm);
      wmhwrite: in H1(1 to Nm);
      wmhprot: in H4(1 to Nm);
      wmhaddr: in HA(1 to Nm);
      wmhwdata: in HD(1 to Nm);
      --
      wmhgrant: out H1(1 to Nm);
      wmhready: out std_logic;
      wmhresp: inout std_logic_vector(1 downto 0);
      wmhrdata: inout std_logic_vector((H_DATA-1) downto 0);
      --
      wshready: in H1(1 to Ns);
      wshresp: in H2(1 to Ns);
      wshrdata: in HD(1 to Ns);
      --
      wshsel: out std_logic;
      wshaddr: out std_logic_vector((H_ADDR-1) downto 0);
      wshwrite: inout std_logic;--H1(1 to Ns);
      wshtrans: inout std_logic_vector(1 downto 0);
      wshsize: inout std_logic_vector(2 downto 0);
      wshburst: inout std_logic_vector(2 downto 0);
      wshwdata: inout std_logic_vector((H_ADDR-1) downto 0);
      wshmaster: inout std_logic_vector(3 downto 0);
      wshmastlock: inout std_logic;--H1(1 to Ns);
      --
      whreset: in std_logic;
      whclk: in std_logic;
      wwhsplit: in std_logic_vector(15 downto 0)
);
end entity AHB_bus_system;
architecture structure of AHB_bus_system is
component c_muxs is
```

```
port(mhtrans:in H2(1 to Nm);
      mhsize:in H3(1 to Nm);
      mhburst: in H3(1 to Nm);
      mhwrite: in H1(1 to Nm);
      mhprot: in H4(1 to Nm);
      mhaddr:in HA(1 to Nm);
      mhwdata: in HD(1 to Nm);
      --
      mhready: out std_logic;
      mhresp:out std_logic_vector(1 downto 0);
      mhrdata: out std_logic_vector((H_DATA-1) downto 0);
      --
      shready:in H1(1 to Ns);
      shresp:in H2(1 to Ns);
      shrdata: in HD(1 to Ns);
      --
      shaddr: out std_logic_vector((H_ADDR-1) downto 0);
      shwrite: out std_logic;
      shtrans: out std_logic_vector(1 downto 0);
      shsize: out std_logic_vector(2 downto 0);
      shburst: out std_logic_vector(2 downto 0);
      shwdata: out std_logic_vector((H_DATA-1) downto 0);
      --
      ccAUmast: in natural range 1 to Nm;
      ccD1mast: in natural range 1 to Nm;
      ccSlave: in natural range 1 to Ns
          );
end component c_muxs;
component c_whole_arb is
port (WRESETn: in std_logic;
      Wclk: in std_logic;
      WHBUSREQm: in std_logic_vector(1 to Nm);
      WHLOCKm: in std_logic_vector(1 to Nm);
      WHREADY: in std_logic;
      WHBURST: in std_logic_vector(2 downto 0);
      WHRESP: in std_logic_vector(1 downto 0);
      WHBUSREQ: in std_logic  ;
      WHADDR: in std_logic_vector((H_ADDR-1) downto 0);
      WHSPLIT: in std_logic_vector(15 downto 0);
```

```
--
      WHGRANTm: out std_logic_vector(1 to Nm);
      WHMASTLOCK: out std_logic;
      WHMASTER : out std_logic_vector(3 downto 0);
      WHSELs: out std_logic_vector(1 to Ns);
--
      WAUmast: out natural range 1 to Nm;
      WDlmast: out natural range 1 to Nm;
      Wcslave: inout natural range 1 to Ns
     );
end component c_whole_arb;

signal sccAUmast: natural range 1 to Nm;
signal sccDlmast:  natural range 1 to Nm;
signal sccSlave:  natural range 1 to Ns  ;
signal wwwhready: std_logic;
signal wwwaddr: std_logic_vector((H_ADDR-1) downto 0);
signal wwwBURST:  std_logic_vector(2 downto 0);
signal wwwRESP:  std_logic_vector(1 downto 0);

begin
UM: c_muxs
port map(mhtrans=>wmhtrans, mhsize=>wmhsize, mhburst=>wmhburst,
      mhwrite=>wmhwrite, mhprot=>wmhprot, mhaddr=>wmhaddr,
      mhwdata=>wmhwdata,
      mhready=>wwwhready, mhresp=>wwwresp, mhrdata=>wmhrdata,
      shready=>wshready, shresp=>wshresp, shrdata=>wshrdata,
      shaddr=>wwwaddr, shwrite=>wshwrite, shtrans=>wshtrans,
      shsize=>wshsize, shburst=>wwwburst, shwdata=>wshwdata,
      ccAUmast=> sccAUmast, ccDlmast=>sccDlmast, ccSlave=>sccSlave);
UA: c_whole_arb
port map(WRESETn=>whreset, Wclk=>whclk, WHBUSREQm=>wmhbusreq,
         WHLOCKm=>wmhlock, WHREADY=>wwwhready, WHBURST=>wwwburst,
         WHRESP =>wwwresp, WHADDR =>wwwaddr, WHSPLIT=>wwhsplit,
         WHGRANTm=>wmhgrant, WHMASTLOCK=>wshmastlock,
         WHMASTER =>wshmaster, WHSELs=>wshsel,
         WAUmast=>sccAUmast, WDlmast=>sccDlmast,
         Wcslave=>sccSlave);
wmhready<=wwwhready;
wshaddr<=wwwaddr;
```

```
wshburst<=wwwburst;
wmhresp<=wwwresp;
end architecture structure;
```

Пример функционирования шины AHB на основе мультиплексоров

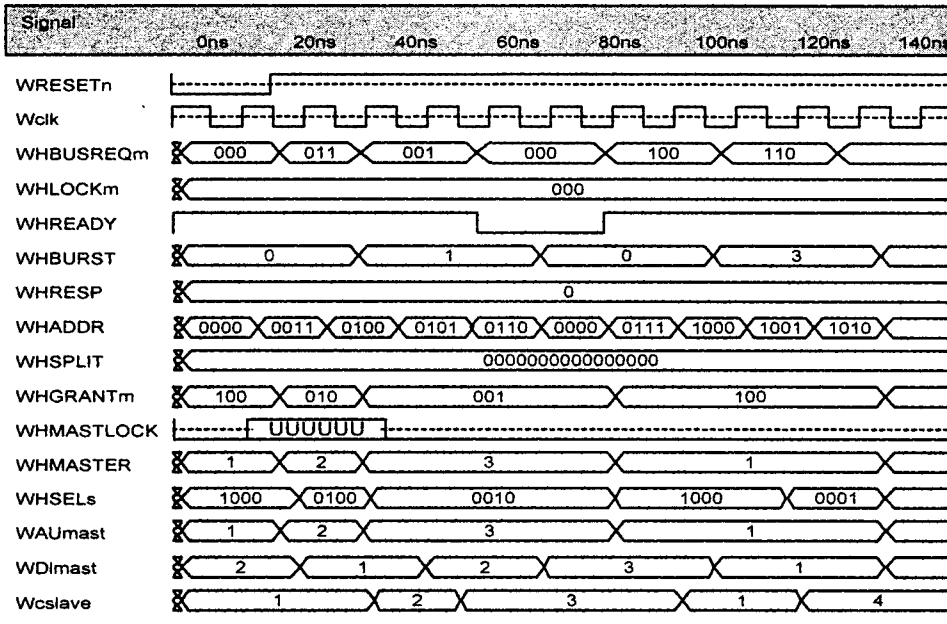
Рассмотрим функционирование шины AHB, реализуемой Блоком связей на основе мультиплексоров, на примере. В 1-м такте сигнал сброса находится в активном состоянии. Во 2-м такте не выполняется никаких действий. В 3-м такте поступают запросы шины от второго и третьего ведущих устройств. В 4-м такте второе ведущее устройство получает шины адреса и управления и выполняет запрос, включающий в себя одно слово данных. В 5-м такте третье устройство получает шины адреса и управления и выполняет запрос неопределенной длины, включающий в себя три слова данных. Выбранное ведомое устройство готово обработать второе слово данных этого запроса только через такт после выставления ведущим устройством его адреса. В 11-м такте первое ведущее устройство выполняет пакетный запрос длины 4. В 12-м такте второе ведущее устройство начинает запрашивать шину для выполнения запроса, состоящего из одного слова данных. При выполнении этого запроса ведомое устройство выдаст код подтверждения `'SPLIT'`.

Этот пример позволяет рассмотреть функционирование Блока связей, описанного нами на VHDL, в следующих ситуациях.

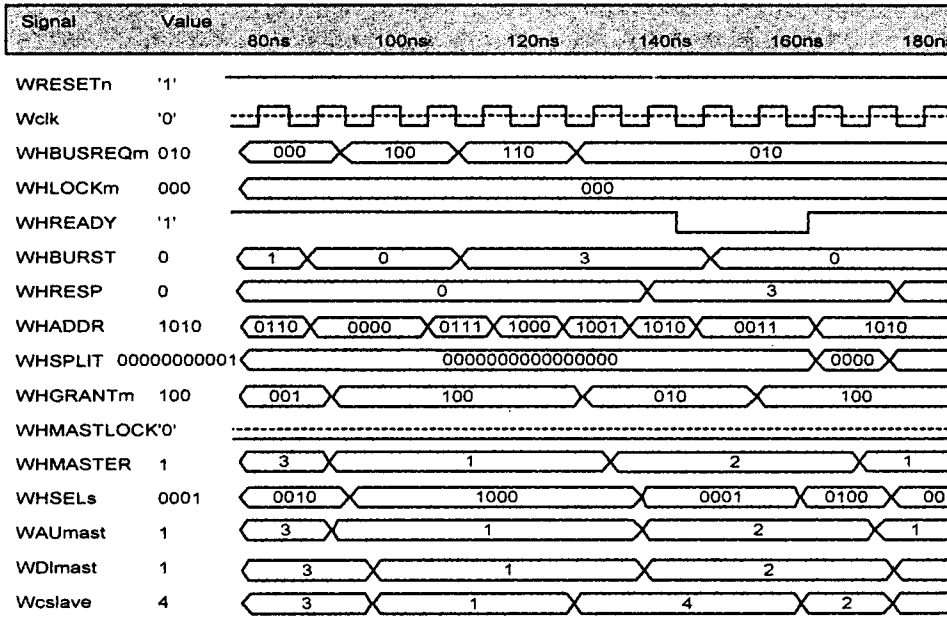
- Одновременный запрос шины от ведущих устройств с различными приоритетами.
- Выполнение запроса, состоящего из одного слова.
- Выполнение пакетного запроса неопределенной длины (3 слова).
- Выполнение пакетного запроса фиксированной длины (4 слова).
- Выполнение расщепленного запроса.

Временные диаграммы представлены на рис. 5.27—5.32.

В первом такте сигнал сброса находится в активном состоянии. В результате, компонент определения конца запроса устанавливается в начальное состояние `st1` (рис. 5.29). Компонент выбора ведущего устройства в качестве устройства, которому предоставлены линии адреса и управления, а также линии данных, выбирает ведущее устройство с номером 1 (т. е. ведущее устройство по умолчанию) (рис. 5.28), в соответствии с чем `Aumast=1`, `Ddlmast=1`, `HMASTER=1`. При этом компонент выбора ведомого устройства также предоставляет линии данных ведомому устройству по умолчанию (т. е. ведомому устройству 1) (рис. 5.31). Блок контроля расщепленных транзакций устанавливает все разряды сигнала `Mmack` в `'0'` (рис. 5.30).



а



б

Рис. 5.27. Временная диаграмма работы компонента whole_arb (*а* – модельное время от 0 до 100 нс., *б* – модельное время от 100 до 220 нс.)

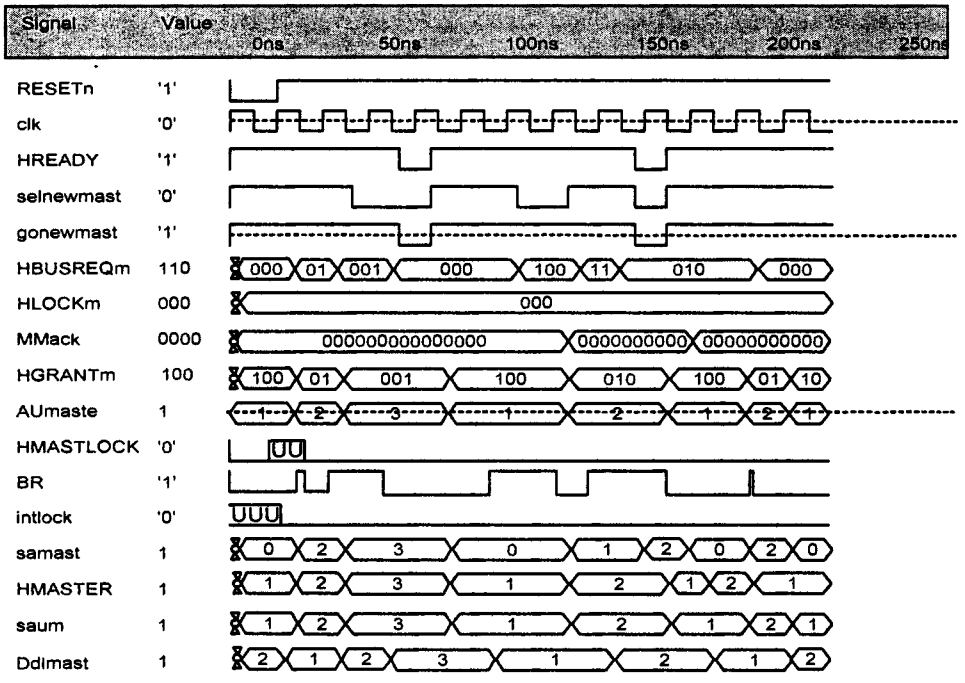


Рис. 5.28. Временная диаграмма работы компонента c_arb

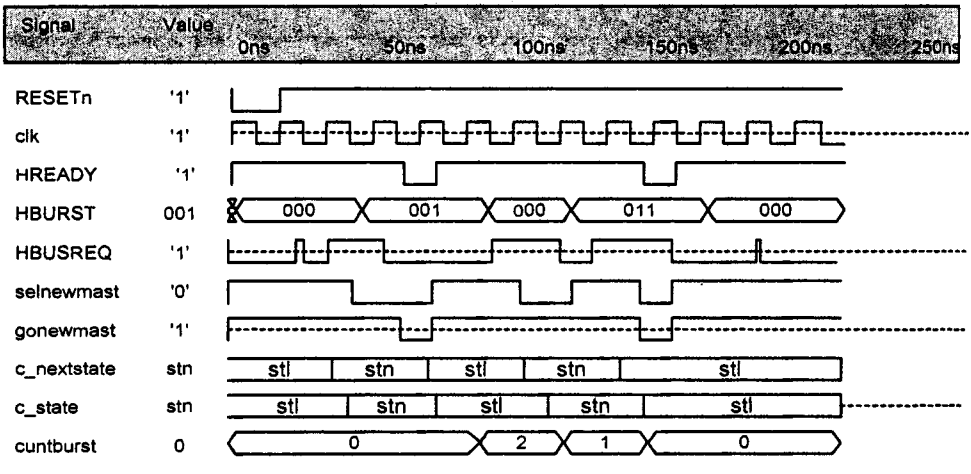


Рис. 5.29. Временная диаграмма работы компонента c_end

В 3-м такте второе и третье ведущие устройства запрашивают шину: `HBUSREQm='011'` (рис. 5.27). В соответствии с выбранной схемой арбитража,

блок выбора ведущего устройства, в качестве кандидата на использование шины адреса и управления, выбирает второе ведущее устройство (samast=2), поскольку оно имеет наивысший приоритет среди устройств, запросивших шину. Поскольку в этом такте блок определения конца запроса находится в состоянии st1, его следующее состояние также st1, а HREADY='1', то сигналы selnewmast и gonewmast установлены в '1'. Это позволяет компоненту выбора ведущего устройства в 4-м такте предоставить шины *адреса и управления* второму ведущему устройству: Aumast=2, HMASTER=2, HGRANTm="0100". Собственником же *линий данных* становится ведущее устройство, которое до этого владело линиями адреса и управления, т. е. первое ведущее устройство.

В 4-м такте второе ведущее устройство выставляет адрес "0011", HBURST="00" (SINGLE) и сбрасывает сигнал запроса шины, т. е. выполняется запрос, включающий в себя одно обращение к ведомому устройству с номером 2, в соотвествии с таблицей адресов. Поскольку выполняется запрос одного слова данных, компонент определения конца запроса остается в состоянии st1. HREADY='1', поэтому selnewmast='1' и gonewmast='1'. В этом такте компонент определения ведущего устройства, в качестве кандидата на использование шин адреса и управления, определяет третье ведущее устройство. В 5-м такте оно становится собственником линий адреса и управления, а второе ведущее устройство становится собственником линий данных. В соответствии с адресом, выставленным в 4-м такте, компонент выбора ведомого устройства определяет кандидата на использование линий данных — ssl=2. Поскольку HREADY='1', и в этом такте HSELm="0100"=4, то сигнал HSEL для второго ведомого устройства устанавливается в '1'. Сигнал gonewmast='1'; это позволяет компоненту выбора ведомого устройства в 5-м такте предоставить линии данных ведомому устройству номер 2 — cslave=2.

В 5-ом такте второе ведущее устройство перестает быть собственником линий адреса и управления, оно становится собственником линий данных. Ведомое устройство 2 участвует в этом обмене и является собственником линий данных. Выполняется фаза данных, единственная в этом запросе. Поскольку HREADY='1', ее длительность составляет 1 такт. В этом такте собственником линий адреса становится ведущее устройство 3. Оно выставляет адрес "0100", HBURST="001" — устройство собирается выполнить пакетный запрос неопределенной длины. В соответствии с этим, блок определения конца запроса выбирает свое следующее состояние — stn, следовательно, selnewmast='0', gonewmast='1'. В соответствии с этим, в 6-м такте ведущее устройство 3 останется собственником линий адреса и управления и еще станет собственником линий данных. Компонент определения ведомого устройства в 5-м такте определяет, в соответствии с выставленным адресом, третье ведомое устройство в качестве кандидата на участие в обмене (ssl=3, HSELm="0010"=2).

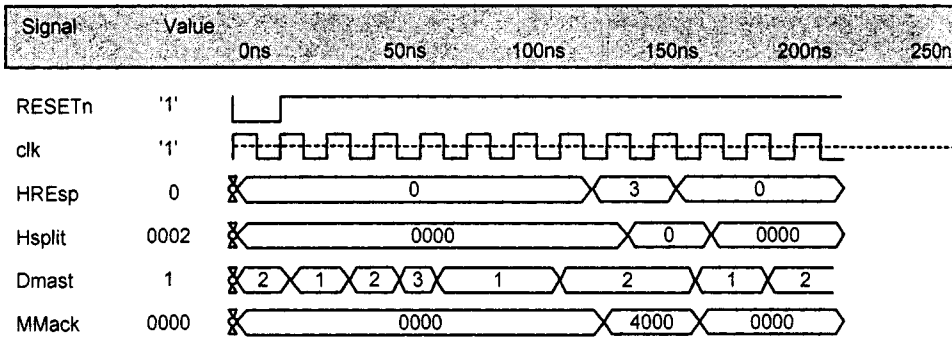
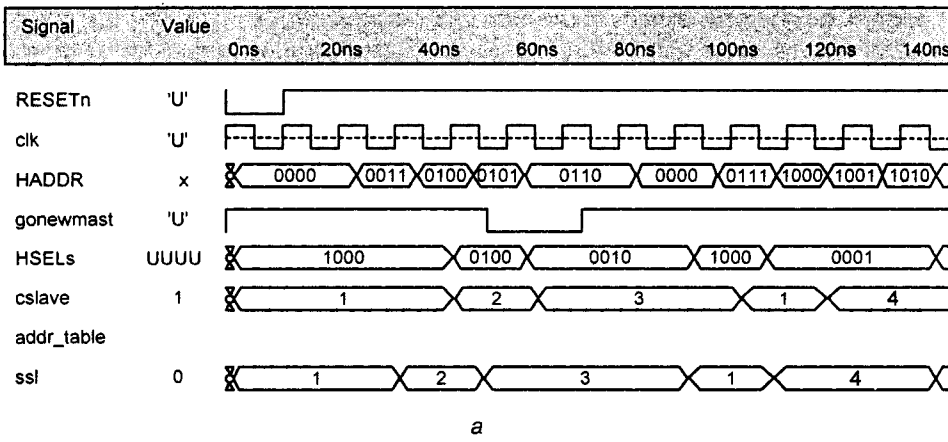
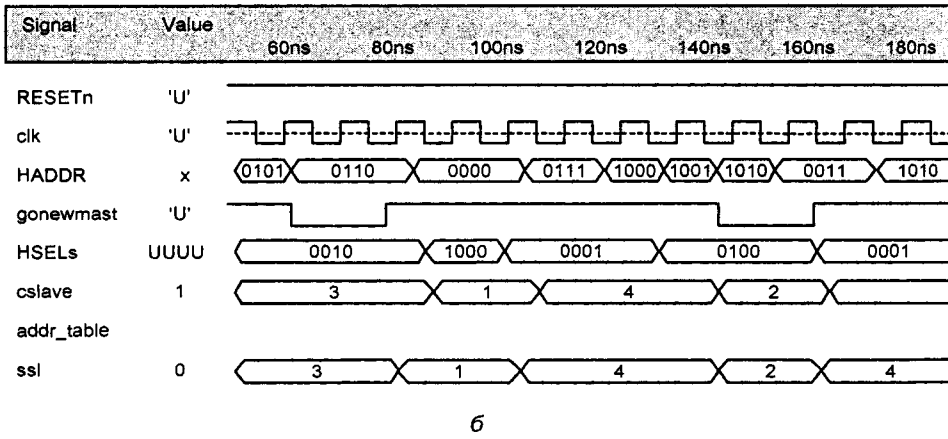


Рис. 5.30. Временная диаграмма работы компонента c_split



а



б

Рис. 5.31. Временная диаграмма работы компонента c_sel.
(*а* – модельное время от 0 до 100 нс., *б* – модельное время от 100 до 220 нс.)

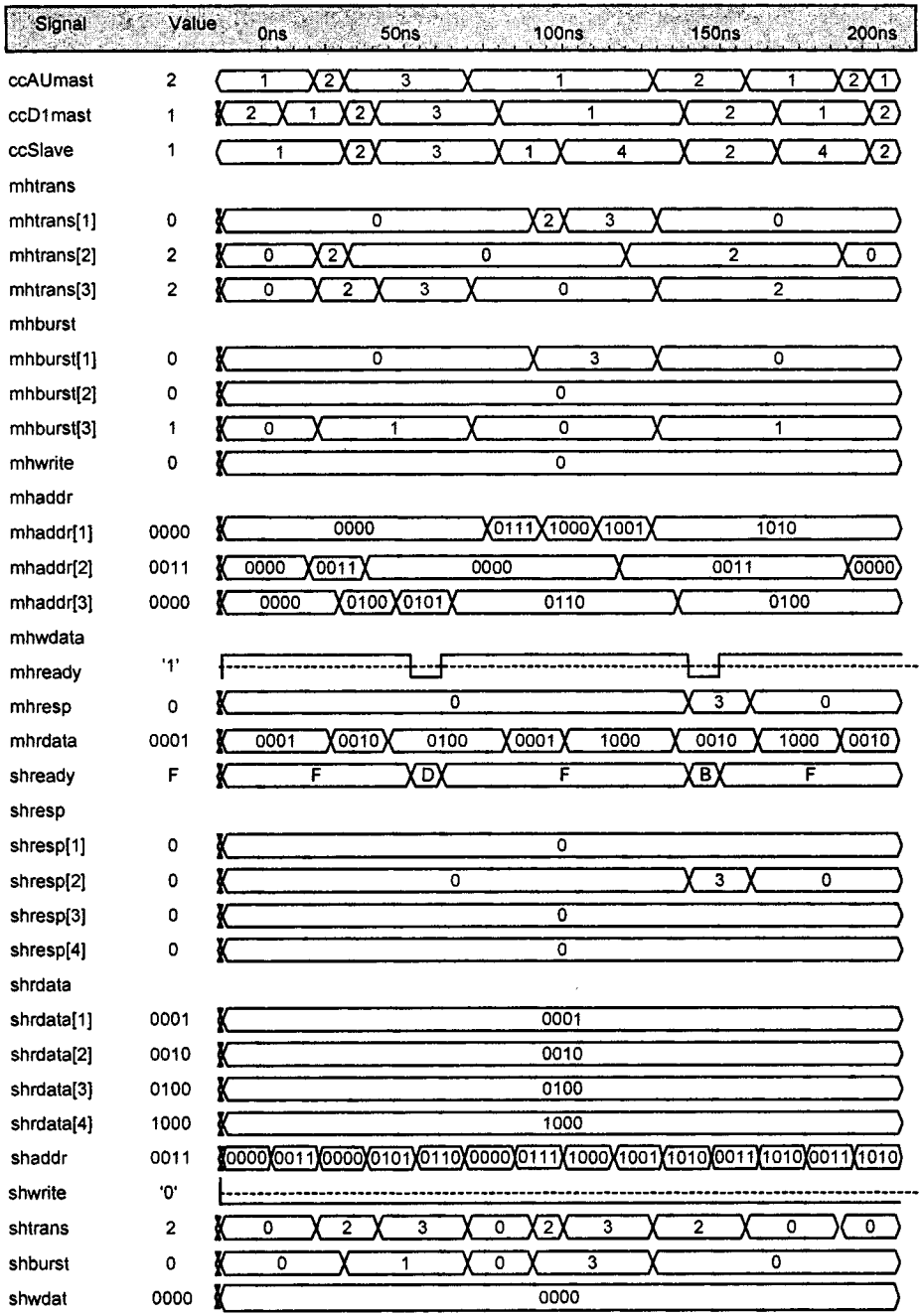


Рис. 5.32. Временная диаграмма работы компонента c_muxs

В 6-м такте осуществляется обмен первым словом данных между третьим ведущим устройством и третьим ведомым устройством. Третье ведущее устройство выставляет на линии адреса адрес второго слова "0101". Поскольку это слово не последнее в пакете, то ведущее устройство 3 продолжает поддерживать сигнал запроса шины в активном состоянии. В соответствии с этим, компонент определения конца запроса выбирает свое следующее состояние — stn. В результате, в 7-м такте ведущее устройство 3 остается собственником линий адреса, управления и данных.

В 7-м такте ведущее устройство номер 3 выставляет на линии адреса адрес третьего (последнего) слова в запросе "0110". Поскольку это слово является последним, ведущее устройство сбрасывает сигнал запроса шины. В этом такте между ведущим устройством 3 и ведомым устройством 3 по линиям данных осуществляется обмен вторым словом данных. Ведомое устройство не готово завершить обмен в этом такте, оно выставляет HREADY='0'. В соответствии с этим, компонент определения конца запроса определяет свое следующее состояние как stn, selnewmast='0', gonewmast='0'. В результате, все шины остаются за их прежними владельцами.

В 8-м такте ведущее устройство номер 3 продолжает выставлять на линии адреса адрес третьего слова данных. На линиях данных продолжается обмен вторым словом данных. HREADY='1' — ведомое устройство готово завершить обмен в этом такте. Сигнал запроса шины от ведущего устройства 3 сброшен, поэтому блок определения конца запроса определяет свое следующее состояние как st1, поэтому сигналы selnewmast и gonewmast устанавливаются в '1'. В этом такте ни одно ведущее устройство не запрашивает шину, кандидата на использование шины нет, samast='0'; в результате, в следующем такте шина адреса и данных предоставляется ведущему устройству по умолчанию.

В 9-м такте ведущее устройство номер 3 обменивается последним словом данных с третьим ведомым устройством.

В 11-м такте первое ведущее устройство выставляет запрос шины; поскольку оно уже получило разрешение на использование шины, то оно сразу же начинает использовать линии адреса и управления. Оно выставляет адрес "0111" и HBURST="011"=INCR4. Ведущее устройство начинает пакетный запрос длины 4 к четвертому ведомому устройству.

В соответствии с этим, компонент определения конца запроса в 12-м такте переходит в состояние stn и устанавливает счетчик countburst=2. В 12-м такте ведущее устройство выставляет адрес второго слова в запросе и осуществляет обмен первым словом данных. Поскольку countburst>0, компонент определения конца запроса определяет свое следующее состояние как stn. В 13-м такте ведущее устройство выставляет адрес третьего слова, осуществляется обмен вторым словом данных. В компоненте определения конца запроса значение счетчика уменьшается на 1, countburst=1, следующее

состояние, соответственно, вновь stn. В 14-м такте ведущее устройство выставляет адрес четвертого (последнего) слова данных и осуществляет обмен третьим словом данных. Значение countburst вновь уменьшается на 1 и становится равным 0. В соответствии с этим, компонент определения конца запроса определяет, что выполняется последняя адресная фаза текущего запроса, его следующее состояние st1.

В 13-м такте второе ведущее устройство начало запрашивать шину, компонент выбора ведущего устройства определил его в качестве кандидата на использование шины адреса и данных. Но в соответствии со значением сигнала selnewmast, собственником линий адреса и управления оно стало только в 15-м такте. Оно выставило адрес "0011" и HBURST="000" — запрос одного слова, адресованный к ведомому устройству 2 (ведомому устройству 2 на шине AMBA приписаны адреса в диапазоне от x"0010" до x"0100", что задано массивом addr_table в программе, листинг 5.20).

В 16-м такте второе ведущее устройство получает линии данных. Второе ведомое устройство, выбранное для участия в обмене, выставляет подтверждение HRESP=SPLIT="11" и HREADY='0'. В соответствии с этим, в 17-м такте компонент контроля расщепленных транзакций устанавливает сигнал Mmack="4000"="0100000000000000". Диапазон значений сигнала Mmack противоположен диапазону HSPLITx. Это не имеет принципиального значения. Можно было задать диапазоны значений этих сигналов одинаково. Разряд, соответствующий второму ведущему устройству установлен в '1'. В соответствии с этим, компонент определения следующего ведущего устройства перестает рассматривать второе ведущее устройство в качестве кандидата на использование шины — samast=0. Вследствие этого, в 18-м такте собственником шин становится ведущее устройство по умолчанию.

В 19-м такте (в момент модельного времени 185) ведомое устройство 2 готово завершить запрос от второго ведущего устройства, оно устанавливает HSPLIT="0000000000000010". В соответствии с этим, в 20-м такте блок контроля расщепленных транзакций устанавливает Mmack="0000". В этом такте компонент определения ведущего устройства вновь начинает рассматривать второе ведущее устройство как кандидата на использование шины. В результате, в 21-м такте второе ведущее устройство получает в свое распоряжение линии адреса и управления, а в 22-м такте — линии данных.

Анализ характеристик шины AMBA AHB для организации связей модулей в системах-на-кристалле

Рассмотрим характеристики коммуникационной системы вычислительного модуля, выполненного как система-на-кристалле с использованием шины AMBA AHB. Предполагается, что вычислительный модуль может включать

в себя до десяти блоков. Обозначим число блоков — N. Присвоим каждому блоку номер от 1 до N. Если блок с номером i может выполнять на шине функции ведущего устройства, то обозначим этот интерфейс Mi. Если блок с номером i может выполнять функции ведомого устройства, обозначим этот интерфейс Si. Не исключается выполнение одним устройством и функций ведущего устройства, и функций ведомого. В этом случае такое устройство j будет иметь два интерфейса — Mj и Sj.

Рассмотрим функционирование системы коммуникаций в общем виде. В систему поступают запросы от ведущих устройств к ведомым. В соответствии с принятым алгоритмом арбитража, ведущим устройствам предоставляются каналы связи для обменов данными с ведомыми устройствами. Канал связи считается занятым на протяжении всего обмена. Если коммуникационная система организована на базе общей шины, то в один момент времени может существовать только один канал связи.

Многие современные процессоры и контроллеры способны формировать следующий запрос к другим блокам, не дожидаясь окончания предыдущего. Однако в рассматриваемых моделях будет считаться, что каждое ведущее устройство может формировать следующий запрос к ведомому устройству только после завершения выполнения предыдущего.

Каждый запрос можно охарактеризовать временем выполнения. Для оценки характеристик анализируемой шины AHB, будем считать, что все ведомые устройства достаточно быстры, и могут обрабатывать одно слово данных за один такт (разрядность слова данных определяется разрядностью шины). На шине AMBA AHB фаза адреса следующего запроса перекрывается во времени с фазой данных предыдущего запроса, за исключением случаев запросов с блокировкой. Рассматриваемая шина является синхронной. Таким образом, время выполнения запроса можно выразить в количестве тактов шины, необходимых для его выполнения. Это количество тактов равно количеству слов в запросе.

Для каждой пары ведущее/ведомое устройство можно найти среднее количество слов в запросе, которое определит и среднее время обслуживания.

Система может быть описана матрицей времен обслуживания Ts. Эта матрица имеет размерность N×N. Пусть строки матрицы соответствуют ведущим устройствам, а столбцы — ведомым. Элемент матрицы Ts с индексами i, j соответствует среднему времени обслуживания запроса от ведущего устройства с номером i ведомым устройством с номером j.

Если блок с номером i не имеет интерфейса ведущего устройства, ему соответствует строка, все элементы которой — нули. Если блок с номером i не имеет интерфейса ведомого устройства, то ему соответствует столбец, все элементы которого — нули.

Предполагается, что поток информации между ведущим и ведомым устройствами, соответствующими одному и тому же блоку, равен нулю, поэтому

на главных диагоналях этих матриц будут стоять нули. Например, для системы, содержащей четыре блока, три из которых могут выполнять и функции ведущего и функции ведомого устройства, а четвертый — только функции ведомого, эта матрица будет иметь следующий вид:

$$Ts = \begin{matrix} 0 & Ts12 & Ts13 & Ts14 \\ Ts21 & 0 & Ts23 & Ts24 \\ Ts31 & Ts32 & 0 & Ts34 \\ 0 & 0 & 0 & 0 \end{matrix}.$$

Например, такую структуру может иметь система-на-кристалле, включающая в себя процессор общего назначения, специальный процессор, контроллер и блок памяти (рис. 5.33).

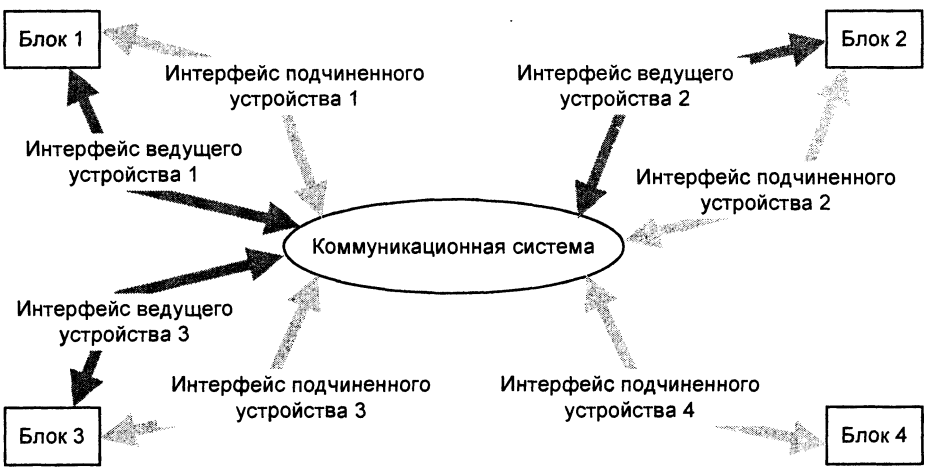


Рис. 5.33. Структурная схема системы-на-кристалле, включающей в себя четыре блока, три из которых могут выполнять функции ведущего и ведомого устройства, а четвертое — только функции ведомого устройства

Каждое ведущее устройство, кроме того, можно охарактеризовать временем между завершением выполнения одного запроса и началом выполнения следующего. Таким образом, система характеризуется вектором $Tm[1..N]$. Для системы, приведенной в примере, он будет иметь следующий вид:

$$Tm = [Tm1, Tm2, Tm3]$$

Стандарт шины AMBA AHB допускает расщепленные транзакции. Расщепленные транзакции моделируются как последовательности отдельных транзакций. То, что ведущее устройство, которое участвует в расщепленной транзакции, до завершения транзакции не может участвовать в других обменах, в модели не учитывается.

Поскольку каждое ведущее устройство может генерировать следующий запрос только после завершения выполнения предыдущего, то для моделирования могут быть использованы замкнутые стохастические сети [24].

Представим функционирование шины AMBA AHB. Модель системы включает в себя обслуживающие приборы, соответствующие ведущим устройствам и обслуживающий прибор, соответствующий каналу между ведущими и ведомыми устройствами. Заявки, перемещающиеся в модели, позволяют отобразить состояния исходной системы. Если заявка покидает систему массового обслуживания (СМО), сопоставленную ведущему устройству, это соответствует формированию запроса от ведущего устройства к ведомому. Из СМО, соответствующей ведущему устройству, заявка перемещается в СМО, соответствующую каналу коммуникационной системы — шины AMBA AHB. После того, как заявка покидает эту СМО, что соответствует завершению выполнения запроса, она возвращается в ведущее устройство, где задерживается на время `Tmi` — среднее время до генерации следующего запроса от этого ведущего устройства.

В общем случае тип заявки определяется номером ведущего устройства, которое генерирует соответствующий запрос, и номером ведомого устройства, которому этот запрос адресован.

Обозначим *Tc* — среднее суммарное время пребывания заявки в коммуникационной системе. Это время складывается из времени пребывания заявки в очереди СМО, соответствующей каналу коммуникационной системы, и времени обслуживания заявки в обслуживающем приборе этой СМО ($Ts[I,j]$).

Математическая модель позволяет оценить следующие характеристики систем.

- Среднее время пребывания заявки в коммуникационной системе — *Tc*.
- Интенсивность потоков заявок через каналы коммуникационной системы — λ.
- Загрузка каналов коммуникационной системы — ρ.

Схема исследуемой системы связей на базе шины AMBA AHB в системе-на-кристалле приведена на рис. 5.34.

Замкнутая стохастическая сеть, используемая для моделирования системы коммуникаций на базе шины, представлена на рис. 5.35.

В этой сети присутствуют четыре обслуживающих прибора. Обслуживающий прибор с номером 0 соответствует каналу связи между ведущим устройством и любым из ведомых устройств. Обслуживающие приборы с номерами 1 — 3 соответствуют ведущим устройствам 1 — 3. Время пребывания заявки в обслуживающем приборе, соответствующем ведущему устройству, равно времени, проходящему между окончанием выполнения предыдущей заявки и формированием следующей. Обозначим среднее время пребывания заявки в обслуживающем приборе 1 — *Tm*1, в обслуживающем приборе 2 — *Tm*2, в обслуживающем приборе 3 — *Tm*3.

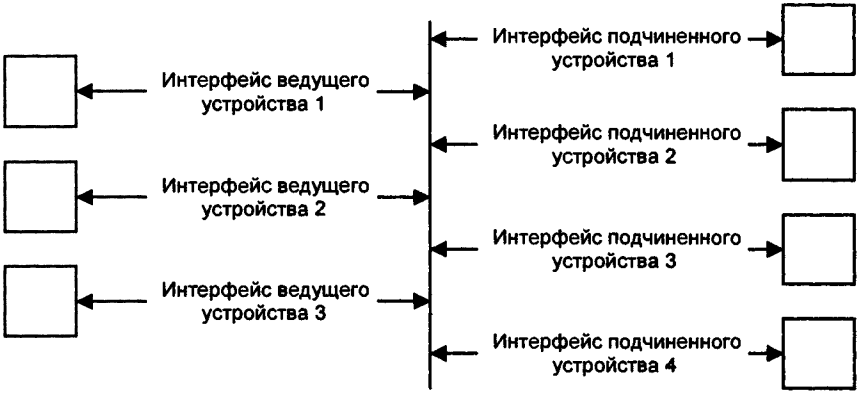


Рис. 5.34. Схема системы коммуникаций на базе шины

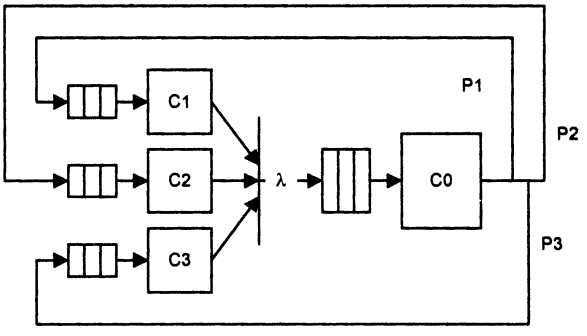


Рис. 5.35. Замкнутая стохастическая сеть — модель системы коммуникаций на базе шины

Для упрощения модели будем считать, что все заявки от одного ведущего устройства, независимо от того, какому ведомому устройству они адресованы, имеют один и тот же тип. Пусть для заявок всех типов время обслуживания в канале связи (C0) равно *Ts*, время пребывания в обслуживающем приборе, соответствующем ведущему устройству так же одинаково $Tm1 = Tm2 = Tm3 = Tm$.

Матрица вероятностей передач для заявок типа 1 для этой сети будет иметь следующий вид:

$P = C1$	$C0$	$C1$	$C2$	$C3$
$C0$	0	1	0	0
$C1$	1	0	0	0
$C2$	0	0	0	0
$C3$	0	0	0	0

Для заявок второго типа:

$P =$	$C0$	$C1$	$C2$	$C3$
$C0$	0	0	1	0
$C1$	0	0	0	0
$C2$	1	0	0	0
$C3$	0	0	0	0

Для заявок третьего типа:

$P =$	$C0$	$C1$	$C2$	$C3$
$C0$	0	0	0	1
$C1$	0	0	0	0
$C2$	0	0	0	0
$C3$	1	0	0	0

Это соответствует тому, что заявки первого типа циркулируют между устройствами $C1$ и $C0$, заявки второго типа циркулируют между устройствами $C2$ и $C0$, заявки третьего типа циркулируют между устройствами $C3$ и $C0$. Поскольку каждое из ведущих устройств в нашей модели может иметь только одну невыполненную заявку, в сети будет циркулировать три заявки.

Обозначим:

- ρ_i — загрузка Ci — вероятность того, что заявка I типа находится в приборе Ci;
- $\rho_i = \lambda i \times Tmi$, где λi — интенсивность потока заявок через Ci.

Для системы возможны следующие векторы состояний:

Тип заявок 1:

$$M11\ (1,0,0,0);\ P11=1-\rho_1$$
$$M12\ (0,1,0,0);\ P12=\rho_1$$

Тип заявок 2:

$$M21\ (1,0,0,0);\ P21=1-\rho_2$$
$$M22\ (0,0,1,0);\ P22=\rho_2$$

Тип заявок 3:

$$M31\ (1,0,0,0);\ P31=1-\rho_3$$
$$M32\ (0,0,0,1);\ P32=\rho_3$$

Вследствие того, что в системе используется бесприоритетная система обслуживания, времена обслуживания заявок всех типов в приборах одинаковы, и пути, по которым циркулируют заявки различных типов, симметричны, $\lambda 1 = \lambda 2 = \lambda 3 = \lambda$. Длительность цикла для заявок каждого типа одинакова. Тогда $\rho_1 = \rho_2 = \rho_3 = \rho = \lambda \times Tm$.

В соответствии с этим, для системы в целом возможны следующие векторы состояния стохастической сети:

*Ms*1 (3,0,0,0); $Ps1=P11\times P21\times P31=(1-\rho)^3$ Длина очереди 2, *C*0 — занят

*Ms*2 (2,0,0,1); $Ps2=P11\times P21\times P32=(1-\rho)^2\times\rho$ Длина очереди 1, *C*0 — занят

*Ms*3 (2,0,1,0); $Ps3=P11\times P22\times P31=(1-\rho)^2\times\rho$ Длина очереди 1, *C*0 — занят

*Ms*4 (2,1,0,0); $Ps4=P12\times P21\times P32=(1-\rho)^2\times\rho$ Длина очереди 1, *C*0 — занят

*Ms*5 (1,0,1,1); $Ps5=P11\times P22\times P32=\rho^2\times(1-\rho)$ Длина очереди 0, *C*0 — занят

*Ms*6 (1,1,0,1); $Ps6=P12\times P21\times P32=\rho^2\times(1-\rho)$ Длина очереди 0, *C*0 — занят

*Ms*7 (1,1,1,0); $Ps7=P12\times P22\times P31=\rho^2\times(1-\rho)$ Длина очереди 0, *C*0 — занят

*Ms*8 (0,1,1,1); $Ps8=P12\times P22\times P32=\rho^3$ Длина очереди 0, *C*0 — простаивает

$$Ps=\sum_{i=1}^{8} Psi=1. \qquad (5.1)$$

Обозначим:

- $\rho 0$ — загрузка *C*0 — загрузка коммуникационной системы;
- $\lambda 0$ — интенсивность потока заявок, проходящего через *C*0.

$$\rho 0=\lambda 0\times Ts. \qquad (5.2)$$

Прибор *C*0 простаивает только в том случае, если система пребывает в состоянии *Ms*8, следовательно:

$$1-\rho 0=\rho^3. \qquad (5.3)$$

$$1-\lambda 0\times Ts=\lambda^3\times Tm^3. \qquad (5.4)$$

Обозначим:

- *L*0 — средняя длина очереди заявок всех типов к *C*0;
- *L*1 — средняя длина очереди заявок типа 1 к *C*0;
- *L*2 — средняя длина очереди заявок типа 2 к *C*0;
- *L*3 — средняя длина очереди заявок типа 3 к *C*0.

Определим среднюю длину очереди к коммуникационной системе. Когда система находится в состоянии *Ms*1, длина очереди равна 2, когда система находится в одном из состояний: *Ms*2, *Ms*3, *Ms*4, длина очереди равна 1. С учетом вероятностей этих состояний можно записать следующее выражение для определения средней длины очереди к *C*0:

$$L0=2\times Ps1+1\times Ps2+1\times Ps3+1\times Ps4=2\times(1-\rho)^3+3\times(1-\rho)^2\times\rho. \qquad (5.5)$$

Рассмотрим среднюю длину очереди заявок первого типа к *C*0. Длина этой очереди может быть отлична от 0, если система пребывает в одном из сле-

дующих состояний: *Ms*1, *Ms*2, *Ms*3, т. е., когда заявка типа 1 находится в СМО *C*0. Когда система пребывает в состоянии Ms1, возможны следующие варианты:

Очередь Обслуживающий прибор

1 2 3 $Psr11=Ps1/6$

2 1 3 $Psr12=Ps1/6$

2 3 1 $Psr13=Ps1/6$

3 2 1 $Psr14=Ps1/6$

1 3 2 $Psr15=Ps1/6$

3 1 2 $Psr16=Ps1/6$

Вероятность этих вариантов одинакова вследствие того, что используется бесприоритетная дисциплина обслуживания. В четырех из этих вариантов заявка типа 1 находится в очереди, следовательно, если система находится в состоянии *Ms*1, заявка первого типа будет находиться в очереди с вероятностью 2/3.

Когда система пребывает в состоянии *Ms*2, возможны следующие варианты:

Очередь Обслуживающий прибор

1 2 Psr21=Ps2/2

2 1 Psr22=Ps2/2

В одном из двух возможных вариантов заявка типа 1 находится в очереди, следовательно, если система находится в состоянии *Ms*2, заявка первого типа будет находиться в очереди с вероятностью 1/2.

Аналогично, когда система пребывает в состоянии *Ms*3, возможны следующие варианты:

Очередь Обслуживающий прибор

1 3 $Psr31=Ps3/2$

3 1 $Psr32=Ps3/2$

В одном из двух возможных вариантов заявка типа 1 находится в очереди, следовательно, если система находится в состоянии *Ms*3, заявка первого типа будет находиться в очереди с вероятностью 1/2.

На основе этого можно записать следующее выражение для средней длины очереди заявок типа 1:

$$L1=(2/3)\times Ps1+(1/2)\times Ps2+(1/2)\times Ps3=(2/3)\times(1-\rho)^3+(1-\rho)^2\times\rho. \qquad (5.6)$$

Для заявок типа 2 и 3, в результате аналогичных рассуждений, получаются такие же выражения. Обозначим среднюю длину очереди заявок *I* типа *L*, $L=L1=L2=L3$.

Обозначим:

- $W0$ — среднее время пребывания в очереди к $C0$ заявок всех трех типов;
- $W1$ — среднее время пребывания в очереди к $C0$ заявок типа 1;
- $W2$ — среднее время пребывания в очереди к $C0$ заявок типа 2;
- $W3$ — среднее время пребывания в очереди к $C0$ заявок типа 3.

Поскольку длительность цикла для заявок всех типов одинакова, $W0 = W1 = W2 = W3 = W$.

$$W0 = L0/\lambda 0. \tag{5.7}$$

$$W = L/\lambda. \tag{5.8}$$

Следовательно, $L0/\lambda 0 = L/\lambda$, или

$$L0/L = \lambda 0/\lambda. \tag{5.9}$$

Подставляя полученные выражения для $L0$ и L, получим

$$L0/L = 3. \tag{5.10}$$

Следовательно, $\lambda 0/\lambda = 3$

Тогда можно записать следующее уравнение:

$$1 - 3 \times \lambda = \lambda^3 \times Tm^3. \tag{5.11}$$

Это уравнение имеет один действительный и два мнимых корня. Значение действительного корня:

$$\lambda = \frac{1}{2} \times \frac{((4 \times Tm + 4 \times \sqrt{\frac{4 \times Ts^3 + Tm^3}{Tm}}) \times Tm^2)^{1/3}}{Tm^2} - 2 \times \frac{Ts}{Tm \times ((4 \times Tm + 4 \times \sqrt{\frac{4 \times Ts^3 + Tm^3}{Tm}}) \times Tm^2)^{1/3}} \tag{5.12}$$

Подставляя это значение, получим

$$\rho = Tm \times (\frac{1}{2} \times \frac{((4 \times Tm + 4 \times \sqrt{\frac{4 \times Ts^3 + Tm^3}{Tm}}) \times Tm^2)^{1/3}}{Tm^2} - 2 \times \frac{Ts}{Tm \times ((4 \times Tm + 4 \times \sqrt{\frac{4 \times Ts^3 + Tm^3}{Tm}}) \times Tm^2)^{1/3}}) \tag{5.13}$$

$$\rho 0 = 3 \times Ts \times \left(\frac{1}{2} \times \frac{\left((4 \times Tm + 4 \times \sqrt{\frac{4 \times Ts^3 + Tm^3}{Tm}}) \times Tm^2\right)^{1/3}}{Tm^2} - \right.$$

$$\left. - \frac{2 \times Ts}{Tm \times \left((4 \times Tm + 4 \times \sqrt{\frac{4 \times Ts^3 + Tm^3}{Tm}}) \times Tm^2\right)^{1/3}}\right).$$

На базе известных загрузок приборов получаем следующее выражение для длины очереди:

$$W = \frac{\frac{2}{3} \times (1 - \lambda \times Tm)^3 + \lambda \times Tm \times (1 - \lambda \times Tm)^2}{\lambda}. \qquad (5.14)$$

На рис. 5.36 приведены графики среднего времени пребывания заявок в коммуникационной системе в зависимости от времени обслуживания заявки в канале. Рассмотрены системы, для которых среднее время между окончанием выполнения одной заявки и началом другой, для каждого ведущего устройства, равно 1, 2, 4, 8 тактам. На этом и последующих рисунках время выражено в относительных единицах — количестве тактов.



Рис. 5.36. Зависимость среднего времени пребывания заявки в системе коммуникаций на базе общей шины от времени обслуживания заявки в ведущем устройстве

На рис. 5.37, *а* приведены графики интенсивностей потоков заявок через коммуникационную систему от ведущих устройств, в зависимости от среднего времени обслуживания заявок в коммуникационной системе.

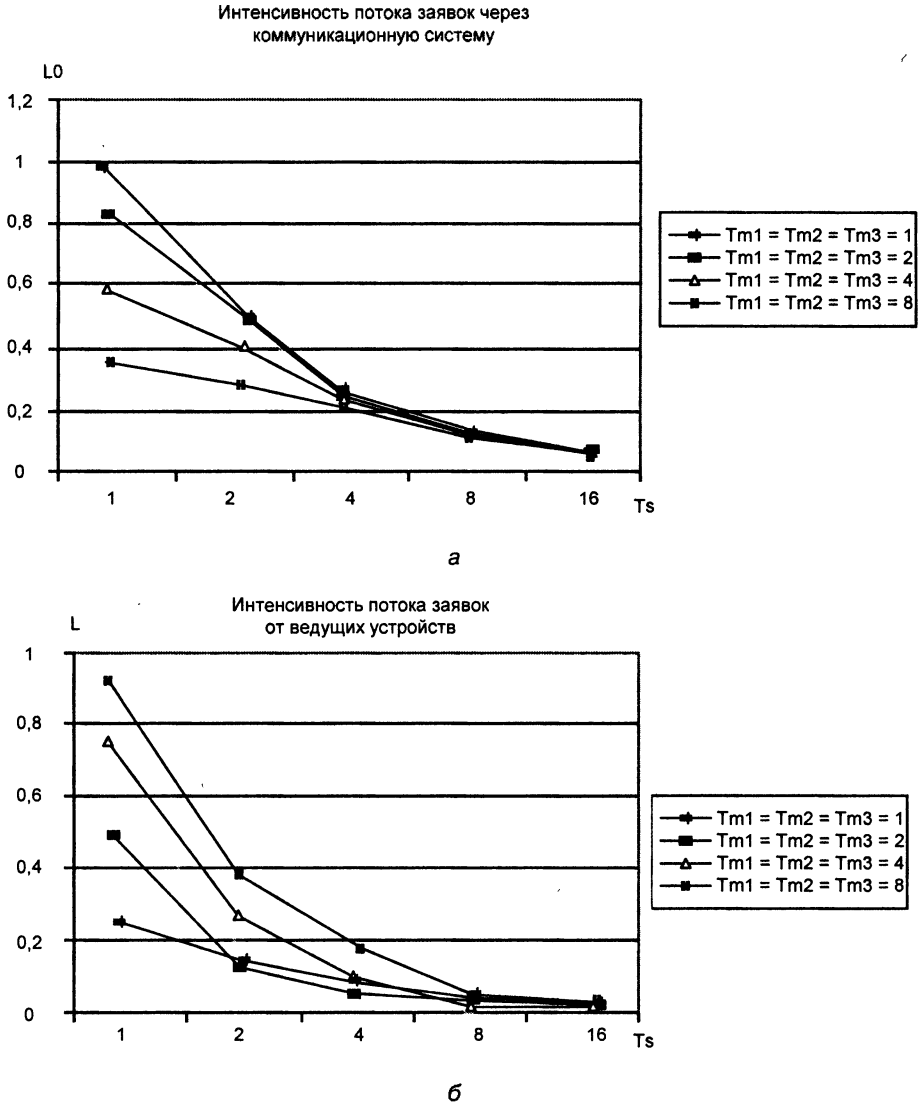


Рис. 5.37. Зависимость интенсивности потоков заявок через коммуникационную систему (*а*), и от ведущих устройств (*б*), от времени обслуживания заявки в коммуникационной системе

Как можно видеть на рис. 5.37, *а*, при *Ts*>4 графики интенсивности потока заявок через коммуникационную систему, при различных временах между поступлениями заявок, практически совпадают, графики интенсивности потока заявок от ведущих устройств так же проходят очень близко.

В то же время, из рис. 5.37, *б* можно видеть, что при *Ts*>4 время пребывания заявки в системе начинает быстро расти — поскольку система замкнутая, интенсивности потоков завок через ведущие устройства резко падают.

Это связано с тем, что при *Ts*>4 коэффициент загрузки коммуникационной системы на базе шины AHB, при заданных нами значениях параметров, становится очень близок к 1, что иллюстрируется графиками рис. 5.38.

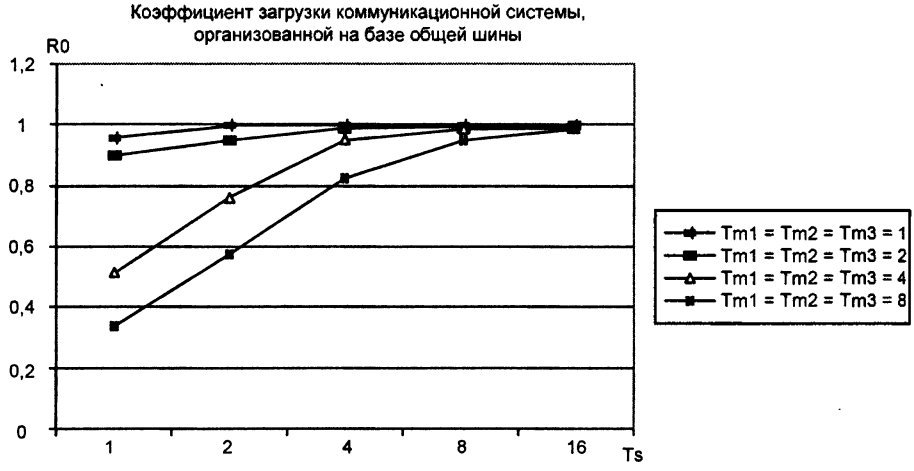


Рис. 5.38. Зависимость коэффициента загрузки коммуникационной системы на базе общей шины от времени обслуживания заявки в ведущем устройстве

Таким образом, если средняя длина пакета данных, которыми обмениваются ведущие и ведомые устройства, становится больше 4, время обслуживания в системе начинает резко возрастать за счет увеличения времени пребывания заявок в очереди к коммуникационной системе, построенной на базе шины AHB.

Для определения характеристик систем, для которых сложно построить математическую модель, используется имитационная модель (она написана на GPSS). С помощью этой модели оцениваются характеристики систем, в которых время обслуживания заявок различных типов — разное. Имитационная модель позволяет также оценить, как будут изменяться характеристики систем при увеличении количества ведущих и ведомых устройств.

На рис. 5.39 представлены графики зависимости среднего времени пребывания заявки в коммуникационной системе, организованной на базе общей

шины, от среднего времени обслуживания заявки ведомым устройством. Среднее время между завершением выполнения одного запроса и генерацией следующего (время пребывания заявки в ведущем устройстве) для всех трех ведущих устройств полагается равным 1. Время обслуживания заявки ведомым устройством и время пребывания заявки в ведущем устройстве распределены по экспоненциальному закону, по нормальному закону или являются постоянными. Как можно видеть из рис. 5.38, законы распределения времени обслуживания заявок мало влияют на среднее время пребывания заявки в системе. Графики практически совпадают.

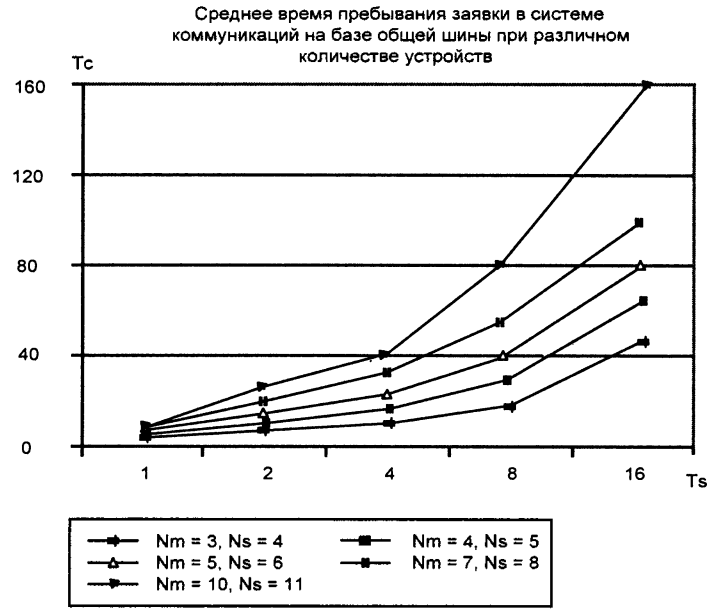


Рис. 5.39. Зависимость среднего времени пребывания заявок в системе коммуникаций, организованной на базе общей шины, от среднего времени обслуживания заявки при различных законах распределения времени обслуживания заявок; ($Tm1=Tm2=Tm3=1$)

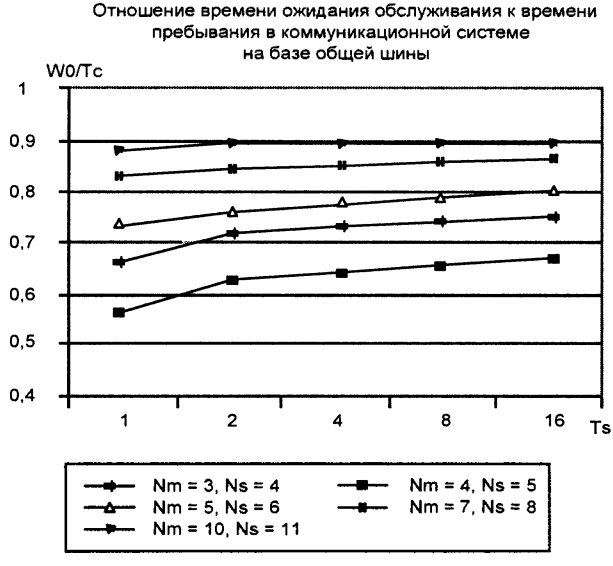
Таким образом, можем считать, что здесь оценка характеристик систем, сделанная в предположении, что времена обслуживания распределены по экспоненциальному закону, будет также довольно точно соответствовать системам, в которых время распределено по другим законам.

Рассмотрим, как влияет на время пребывания заявок в коммуникационной системе количество ведущих и ведомых устройств в системе.

На рис. 5.40, *а* представлены графики зависимости среднего времени пребывания заявки в коммуникационной системе на базе общей шины от времени обслуживания заявки в ведомом устройстве. Время пребывания заявки в ведущем устройстве для всех ведущих устройств здесь полагается равным 1. Рассматриваются следующие системы: (3 ведущих устройства, 4 ведомых); (4 ведущих, 5 ведомых); (5 ведущих, 6 ведомых); (7 ведущих, 8 ведомых); (10 ведущих, 11 ведомых). На рис. 5.40, *б* представлены графики отношения среднего времени пребывания заявок в очереди на обслуживание к времени пребывания в коммуникационной системе (включая время ожидания обслуживания, т. е. доступа к шине, и время обслуживания, то есть собственно передачи информации).



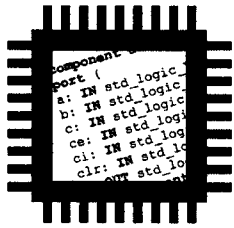
а



б

Рис. 5.40. Влияние количества устройств в системе на среднее время пребывания заявок в коммуникационной системе (*а*); на отношение времени ожидания обслуживания к времени пребывания в коммуникационной системе — (*б*), организованной на базе общей шины

Как можно видеть из рис. 5.40, *а*, с ростом количества устройств в системе среднее время пребывания заявки в системе коммуникаций, естественно, растет. Особенно заметным это увеличение становится с увеличением времени обслуживания заявки. Поскольку время пребывания заявки в коммуникационной системе растет за счет увеличения времени пребывания заявки в очереди на обслуживание, рассмотрим соотношение времени пребывания заявки в очереди к времени пребывания заявки в коммуникационной системе (рис. 5.40, *б*). При количестве ведущих устройств 3, это соотношение составляет около 0.6, то есть заявка находится в очереди чуть больше половины времени ее пребывания в системе, при количестве ведущих устройств 10, заявка находится в очереди 0.9 времени ее пребывания в системе.



Глава 6

Проектирование на VHDL в среде OrCAD Express

Среда OrCAD Express предназначена для построения графических представлений схем устройств, описания устройств на языке VHDL, моделирования работы устройств и создания выходных файлов в форматах фирм-производителей для физической реализации проекта. Эта среда является развитием среды автоматизации проектирования OrCAD и, по сравнению с ней, имеет расширенный набор возможностей. Описание среды дается на основе OrCAD 9.1.

Среда включает в себя следующие основные группы фреймов:

- OrCAD Capture — инструментарий создания принципиальных схем проектов различных типов;
  - OrCAD Capture CIS — OrCAD Capture с возможностью доступа к базам данных компонентов через Интернет.
- OrCAD Express Simulate — инструментарий моделирования цифровых схем. Этот инструментарий может работать с представлениями схем в графическом виде, выполненными в OrCAD Capture, и с описаниями на VHDL;
- OrCAD Express Simulate CIS — OrCAD Express Simulate с возможностью доступа к базам данных компонентов через Интернет;
- OrCAD Express Plus — инструментарий OrCAD Express Simulate, дополненный возможностями разработки семейств Xilinx Virtex и Spartan, Lucent ORCA 3C FPGA.
- OrCAD Layout — инструментарий разработки печатных плат;
- OrCAD Layout Plus — OrCAD Layout и бессеточный автотрассировщик SmartRoute;
- OrCAD Layout Engineer's Edition — программа просмотра печатных плат с возможностью ручного редактирования положения компонентов и прокладки критических цепей. Возможна двухсторонняя передача дан-

ных между OrCAD Layout и P-CAD, Tango, PADS, Protel, AutoCAD, MicroSim, PCBoards, SPECTRA.

- OrCAD Pspice — инструментарий моделирования аналоговых и смешанных аналого-цифровых устройств.

В среде OrCAD приняты следующие соглашения о расширениях используемых файлов:

- ASP — временный файл, в котором сохраняются данные об измененных и не сохраненных проектах;
- BAS — макросы описания диалоговых окон ввода атрибутов компонентов/цепей на языке BASIC;
- BCF — бинарный файл конфигурации Orcad SDT, используемый при трансляции;
- BOM — форматированный список компонентов проекта;
- CFG — SDT-файлы конфигурации, используемые при трансляции;
- CIR — список соединений схемы в формате SPICE;
- DBC — конфигурация базы данных, используемая в CIS;
- DBK — файлы графического представления схемы (для совместимости с предыдущими версиями);
- DCF — файлы списка цепей VST-модели;
- DRC — протокол поиска ошибок в принципиальной схеме;
- DSF — список соединений в формате VST MODEL
- DSN — файлы графического представления схемы;
- EDF или EDN — список соединений схемы в формате EDIF 2.0.0 или файл обратной корректировки;
- ERR — текстовые файлы с информацией об ошибках;
- EXP — перечень параметров компонентов проекта;
- FUS — файл программирования ПЛИС;
- INC — включаемые файлы перечня материалов;
- INF — VST-файлы;
- INF — список соединений схемы в формате DOS-версии ORCAD Digital Simulation Tools 386+;
- INI — конфигурация отдельных фреймов;
- INS — файлы, управляющие созданием списка цепей;
- LIB — файлы библиотек STD;
- LLB — библиотека корпусов компонентов;

- MAP — список соответствий имен цепей (дополнение к списку соединений *.CIR);
- MNL — файлы уровня списка цепей;
- NET — список соединений схемы в формате Pspice;
- OBK — библиотека символов компонентов;
- OLB — файлы библиотек;
- OPJ — файл проекта, содержащий ссылки на все файлы, входящие в состав проекта;
- PIP — файлы создания списка цепей;
- PLD — список соединений в формате OHDL;
- RES —файлы создания списка цепей;
- RPT — файлы отчета обновления свойств;
- SCH — файлы заголовка схемы SDT;
- SDF — файлы описания временных характеристик, используемые для анализа временных характеристик схемы;
- SDT — формат описания стандартных задержек в ПЛИС;
- SNP — файлы описания ножек микросхем и разъемов;
- STM — файлы, содержащие значения входных сигналов для выполнения моделирования;
- TIM — отчет о моделировании с учетом временных задержек;
- TXT — текстовые файлы протоколов сессии;
- UPD — файлы обновления свойств;
- V — файлы списка цепей для Verilog;
- VHD, VHO — VHDL файлы;
- XNF — список соединений в формате Xilinx;
- XRF — список компонентов проекта с указанием координат их расположения в схеме и имен библиотек, в которых они находятся.

Файлы SDT.CFG и SDT.BCF используются при трансляции SDT-файлов.

Проектирование в OrCAD Express

OrCAD Express предоставляет возможности для создания схем проектируемых устройств на базе FPGA (field programmable gate arrays), CPLD (complex programmable logic devises), SPLD (simple programmable logic devises). Отработанный на FPGA проект может быть перенесен из OrCAD Express в САПР заказных или полузаказных СБИС (например, в САПР Cadence).

Можно выделить следующие основные фазы процесса проектирования:

- первоначальное описание схемы (design entry);
- функциональное моделирование;
- логический синтез и оптимизация;
- определение геометрических мест расположения элементов и прокладка связей (place and route);
- временное моделирование;
- генерация документации и архивация.

Создание первоначального описания схемы

На этом этапе может быть создано графическое описание или описание на VHDL проектируемого устройства. Для создания графического описания может быть использован редактор графических представлений схем OrCAD Capture schematic page editor. Для описания проектируемого устройства на VHLD может быть использован редактор программ OrCAD Simulate VHDL programming editor.

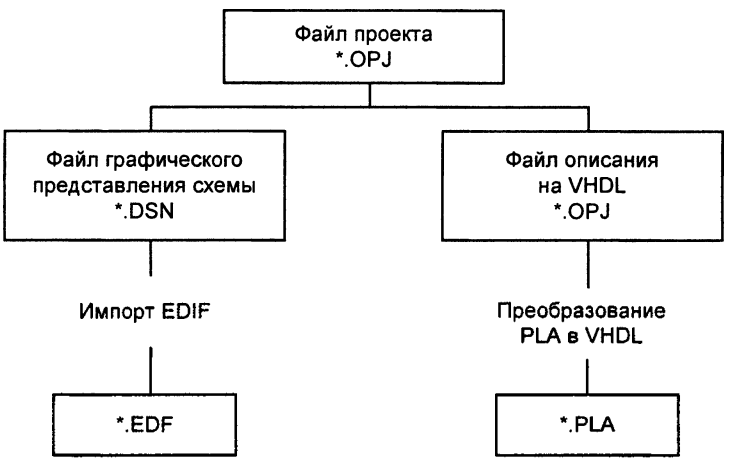


Рис. 6.1. Файлы проекта на этапе первоначального описания схемы

Для создания описания-модели на языке VHDL также может использоваться инструментарий, не принадлежащий OrCAD, в том числе — обычные текстовые редакторы. Кроме того, OrCAD содержит дополнительный инструментарий, позволяющий использовать в моделях файлы рисунков элементов в форматах Graphical EDIF и Open-PLA. OrCAD Capture позволяет также создавать рисунки и описания новых элементов, не только размещать их в уже существующих библиотеках, но и формировать новые библиотеки. Для этого может использоваться редактор представления элементов (part

editor) и редактор библиотек (library editor). Организация проекта на этом этапе представлена на рис. 6.1.

Как правило, первоначальное описание схемы производится без учета особенностей физической реализации.

Функциональное моделирование

На этапе функционального моделирования производится отладка модели и исправление ошибок, а также документирование логики работы схемы. Для выполнения моделирования используется фрейм OrCAD Simulate. Этот фрейм позволяет по заданным значениям входных сигналов определить состояния выходов. Результаты моделирования можно просмотреть в графическом или табличном виде.

OrCAD Simulate позволяет моделировать схемы, представленные в графическом виде в среде OrCAD Capture, описанные на VHDL, а также EDIF-описания. Все эти виды описания схемы могут включаться в один проект в различных сочетаниях. При моделировании схем, представленных в графическом виде, используются VITAL/VHDL библиотеки, содержащие описание поведения элементов, представленных графически. Для моделирования могут быть использованы также PLA и XHF файлы, но они должны быть предварительно оттранслированы средствами OrCAD в VHDL.

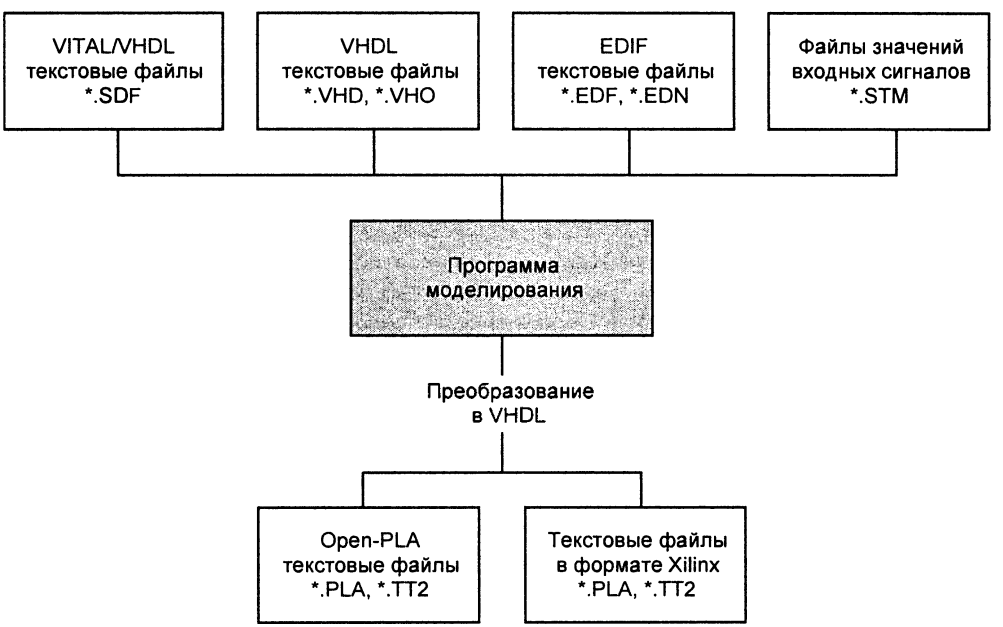


Рис. 6.2. Файлы проекта на этапе функционального моделирования

Для выполнения этого этапа необходимо сформировать наборы значений входных сигналов, применение которых позволяет проконтролировать работу схемы. Наборы значений входных сигналов могут формироваться с использованием соответствующих диалоговых окон или описываться на VHDL. Организация проекта на этом этапе представлена на рис. 6.2.

Если в ходе функционального моделирования были выявлены ошибки, возвращаются к предыдущему этапу и исправляют их.

Логический синтез и оптимизация

На этапе логического синтеза формируется список связей на уровне логических вентилей. OrCAD Express включает в себя Exemplar Logic Leonardo Spectrum logic synthesis для преобразования VHDL моделей в список связей для дальнейшей его обработки инструментарием, выполняющим размещение и маршрутизацию (place and route) фирм-производителей СБИС.

В системах проектирования того класса, который рассматривается в данной книге, термин "логический синтез" (synthesis) соответствует автоматическому преобразованию спецификации проектируемого устройства с уровня RTL-проекта в спецификацию проектируемого устройства на уровне вентилей (gate level representation).

Это преобразование зависит от набора базовых ячеек (basic cells), которые имеются для используемой технологии построения СБИС. Если простые операции (типа сравнения) прямо отображаются в булевские функции и реализующие их логические вентили, то более сложные конструкции VHDL-спецификации на RTL-уровне (например, математические операторы) сначала преобразуются в макроячейки (macrocells). Библиотека таких макроячеек (macrocell library) включает в себя макроячейки для типовых элементов архитектуры системы на СБИС (сумматоры, умножители, регистровые блоки и др.) с готовым их воплощением на уровне вентилей. Обычно библиотеки макроэлементов разрабатываются для конкретных систем автоматизированного проектирования СБИС и конкретного вида СБИС, в которых они воплощаются в конечном итоге. Состав библиотеки макроячеек в значительной степени определяет возможности той или иной программы автоматизированного синтеза. Библиотеки макроячеек, наряду с файлами RTL-проекта, используются как входная информация для подсистемы логического синтеза.

Необходимо учитывать, что язык VHDL в своей полной версии не является полностью синтезируемым (fully syntheziable) языком. Существующие в настоящее время программы автоматизированного синтеза поддерживают различные подмножества языка VHDL. Этим фактором в значительной степени определяются и описываемые в данной книге ограничения версий языка VHDL в системе OrCAD Express.

Ограничения (constrains), используемые программой логического синтеза, включают в себя два типа ограничений: фиксированные ограничения, определяемые выбранной технологией реализации систем на СБИС и возможностями самой программы синтеза; ограничения, которые задает пользователь — разработчик СБИС. Пользователь может задавать ограничения на уровне библиотеки макроячеек, на уровне преобразования булевских уравнений, описывающих устройство, и, для проекта в целом, — на уровне логических вентилей (рис. 6.3). Заданные пользователем ограничения учитываются программой синтеза и оптимизации.

При логическом синтезе, обычными критериями оптимизации генерируемого проекта СБИС на уровне вентилей являются быстродействие и объем аппаратных ресурсов.

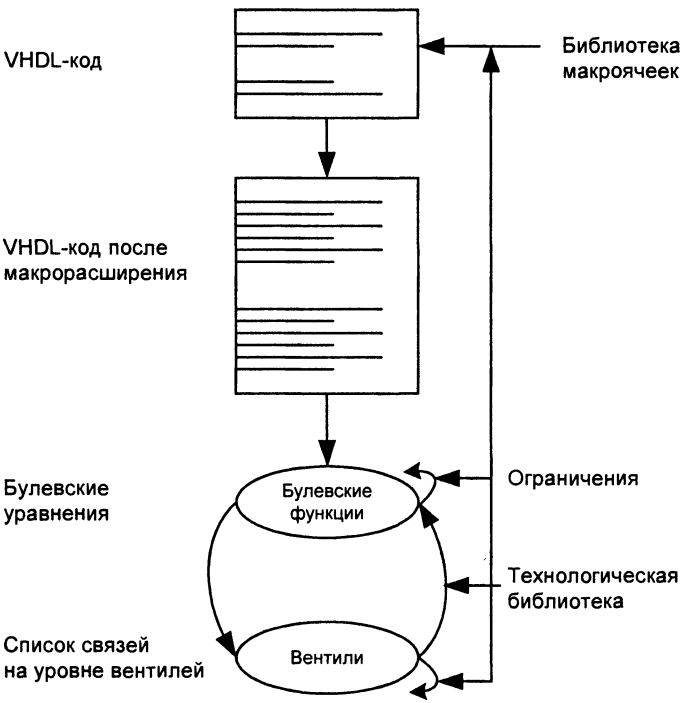


Рис. 6.3. Ограничения (constrains), используемые при логическом синтезе и оптимизации по VHDL-коду

Используемая в OrCAD Express технология логического синтеза позволяет установить баланс между производительностью разрабатываемой схемы и затрачиваемыми ресурсами (площадь, занимаемая топологическим рисунком схемы, количество вентилей) путем настройки параметров синтеза. Для различных частей схемы эти параметры могут принимать различные значения.

Например, в схеме можно выделить блоки, для которых существенно быстродействие, кроме того, OrCAD Express Plus позволяет определить дополнительные требования к частоте.

На результаты синтеза существенное влияние может оказывать стиль спецификации проекта, используемый проектировщиком, стиль программирования на VHDL. Поэтому рекомендуется учитывать ряд дополнительных факторов на заключительных этапах проектирования СБИС на VHDL, на этапе формирования RTL-проекта для последующего автоматического логического синтеза.

Одной из типовых рекомендаций является учет фактора синтезируемости при декомпозиции проекта, разбиении проекта на синтезируемые блоки (на RTL-уровне). Известно, что алгоритмы оптимизации систем логического синтеза дают лучшие результаты при относительно небольших (порядка нескольких тысяч вентилей) синтезируемых блоках. Это стоит учитывать при разработке и декомпозиции RTL-проекта. Так, не следует разделять критические цепи по нескольким синтезируемым блокам.

Некоторую помощь в решении этих проблем дает и система проектирования. Группировка нескольких подмодулей при синтезе позволяет в ряде случаев получить нужный результат без изменения декомпозиции RTL-проекта переписыванием программ на уровне VHDL.

Процесс логического синтеза и оптимизации является итеративным. Количество выполняемых САПР итераций и, соответственно, качество оптимизации можно задать с помощью параметра Maximum operating speed — скорость выполнения (регулирует количество итераций процедур оптимизации). Чем выше скорость выполнения, тем ниже будет качество выполняемой оптимизации.

В конце этапа логического синтеза создается список связей (*.VHD). Этот список создается с учетом требований и возможностей фирмы-производителя, которая будет выполнять физическую реализацию разрабатываемого устройства в СБИС. При создании этого списка, кроме определяемых пользователем ограничений, используются также библиотеки макроячеек и технические библиотеки производителя СБИС. В результате, сформированное на предыдущих этапах описание схемы преобразуется к виду, в котором все функции реализуются с использованием элементной базы выбранной фирмы-производителя СБИС.

Не всегда это преобразование удается выполнить корректно, т. е. таким образом, чтобы функция, выполняемая схемой, после преобразования осталась прежней. Это связано как с особенностями реализации выбранных библиотек макроячеек, так и с особенностями отображения конструкций языка описания. Если полученное описание оказалось некорректным, необходимо или выбрать другие библиотеки макроячеек, или откорректировать исходное описание модели.

Размещение и трассировка

На этапе размещения и трассировки список связей, полученный в результате логического синтеза, обрабатывается программным обеспечением "place and route", соответствующим выбранной фирме-производителю. В ходе его выполнения производится размещение логических элементов внутри архитектурных элементов кристалла и определение геометрического расположения линий межсоединений. В OrCAD включен инструментарий, позволяющий обрабатывать на этом этапе и простейшие PAL/GAL/PROM схемы устройств. Для обработки схем более сложных устройств необходимо инсталлировать специальное программное обеспечение для обработки CPLD или FPGA. Так, для того чтобы выполнить этот этап для Xilinx FPGA необходимо инсталлировать Xilinx Alliance v A1. К возможностям инсталлированного инструментария OrCAD Express предоставит доступ посредством инструмента Build, который включает в себя набор диалоговых окон, позволяющих управлять значениями опций выбранного программного обеспечения от производителя.

В ходе этапа размещения и трассировки выполняется следующая последовательность действий.

1. Трансляция — преобразование информации, содержащейся в файлах списков на уровне вентилей (сформированных на предыдущем этапе), в бинарную базу данных.
2. Отображение — структуры, присутствующие в логическом описании схемы, отображаются в физические структуры, выполняющие заданные функции. Набор используемых физических структур зависит от фирмы-производителя.
3. Размещение и трассировка (Placement and routing) — размещение объектов и прокладка линий связи в физическом представлении схемы. При использовании для реализации СБИС классов CPLD и FPGA в ходе этого процесса производится назначение ресурсов CPLD и FPGA логическим элементам схемы.
4. Генерация выходного файла оптимизации (Implementation) — генерация выходного файла (JEDEC, HEX, POF) в форме, которая необходима для программирования CPLD или FPGA.
5. Генерация файлов для моделирования (Generate simulation output) — преобразование файлов, созданных на предыдущем шаге, в форму, позволяющую выполнить моделирование работы спроектированного устройства.

Традиционное для САПР СБИС разделение этапа логического синтеза и этапа размещения и трассировки, имеющееся также в OrCAD Express, создает определенные проблемы при практическом проектировании СБИС. Классическим примером таких проблем является то, что на этапе синтеза отсутствует информация о размещении элементов на кристалле СБИС и

трассировке связей. В результате длина соединений между элементами — один из критических параметров для временных характеристик — не может быть достоверно оценена на этапе логического синтеза. Появляется необходимость временного моделирования схемы уже *после* этапа размещения и трассировки. Критические же цепи (например, цепи тактирования) часто требуют последующей ручной модификации для обеспечения корректного функционирования проектируемой схемы.

Временное моделирование

На этапе временнго моделирования определяются временные характеристики работы полученной технической реализации схемы.

На этом этапе могут быть использованы наборы входных сигналов, разработанные в ходе функционального моделирования. В процессе временного моделирования программа может занести в отчет информацию о нарушении заданных временных параметров или об эффектах, возникающих в результате задержек распространения сигнала по линиям. Если в ходе моделирования на этом этапе были обнаружены нарушения временных параметров, необходимо вернуться к одному из предыдущих этапов и изменить параметры настройки. После этого процесс разработки должен быть повторен, начиная с данного этапа. Возможно как изменение параметров отдельных элементов, таких как нагрузочная способность выходов, так и изменение элементной базы в целом.

Для выполнения временного моделирования необходимы файлы, содержащие список связей (*.VHD), и файлы, содержащие информацию о задержках распространения сигнала — standard delay format (*.SDF), рис. 6.4.

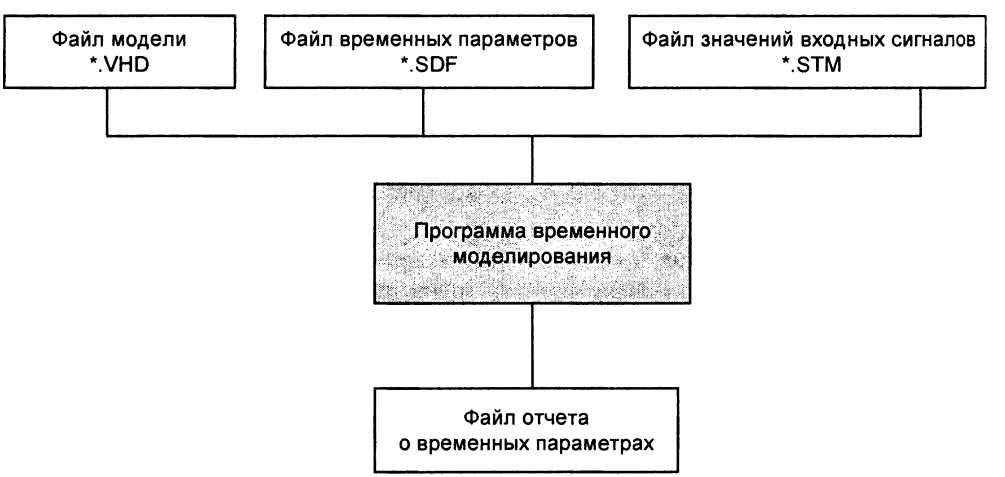


Рис. 6.4. Файлы проекта на этапе временного моделирования

Документирование и архивация

На этом этапе выполняется документирование готового проекта для облегчения его дальнейшего использования и модификации.

Результаты проектирования

Результаты проектирования могут оформляться в виде следующих групп файлов: откомпилированные списки связей, файлы стандартных задержек (*.SDF), списки связей с информацией о временных характеристиках, файлы контактов (pin files), карты пережигания плавких перемычек (fuse maps) и файлы отчетов.

Схема процесса проектирования в обобщенном виде представлена на рис. 6.5.

Компоненты фрейма Express Simulate

Этот фрейм включает в себя следующие компоненты:

- **Session log** (окно сессии);
- **Project manager** (окно менеджера проектов);
- **Main menu** (основное меню);
- **Tool bar** (панель инструментов);
- **Stack Window** (окно информации о стеке);
- **Command line** (командная строка);
- **Status bar** (строка статуса);
- **Рабочая область**.

Видимостью/невидимостью панели инструментов, окна информации о стеке строки статуса и командной строки можно управлять с помощью соответствующих пунктов меню **View** (Вид).

Окно сессии

В окне сессии отображаются все события, связанные с выполнением команд оболочки и их успешным или неуспешным завершением. При возникновении ошибок компиляции в этом окне указывается имя файла, в котором возникла ошибка, если это файл на VHDL, то — номер строки, в которой возникла ошибка и тип ошибки. Для перехода к строке, содержащей ошибку, можно воспользоваться пунктом **Go to** (Перейти) меню **Edit** (Редактирование) или дважды щелкнуть мышью по строке с указанием ошибки в окне сессии. Для того чтобы очистить это окно от информации, применяется пункт **Clear All** (Очистить все) меню **Edit**.

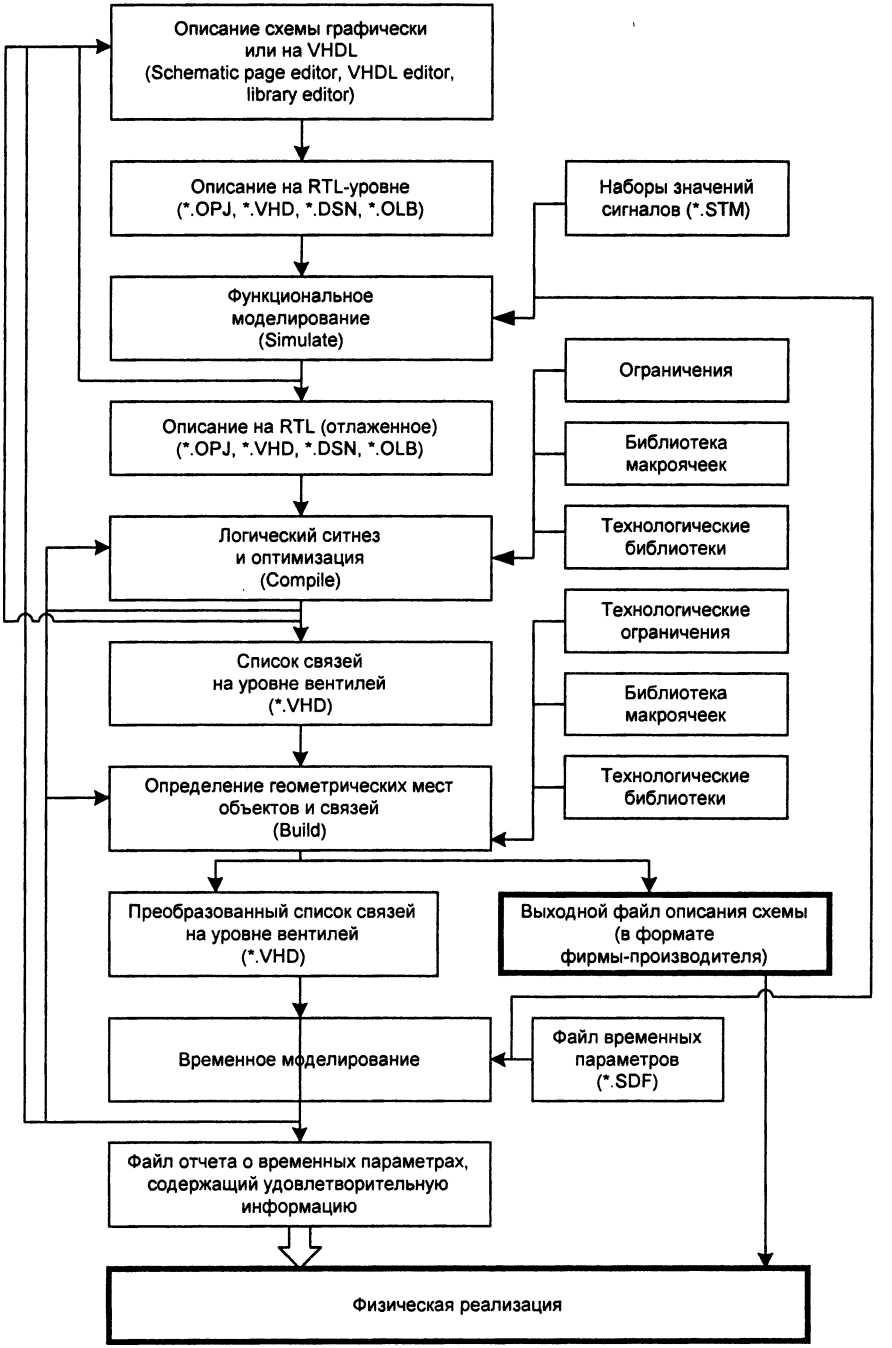


Рис. 6.5. Общая схема процесса проектирования

Менеджер проектов

С помощью менеджера проектов можно включать файлы в проект или исключать их из него. Менеджер может представлять структуру проекта в двух видах:

- file (файловый) — проект представляется в виде совокупности входящих в него фалов;
- hierarhy (иерархия объектов) — проект представляется в виде совокупности входящих в него объектов (entity).

Для перехода от одного представления к другому необходимо воспользоваться соответствующей вкладкой в верхней части менеджера.

Представление проекта в файловом виде

Если проект представлен в файловом виде, то в окне менеджера проектов отображается перечень файлов, входящих в состав проекта. В соответствии со своим назначением они располагаются в следующих папках:

- **Design Resources** (ресурсы графического представления схемы) — файлы описания модели на языке VHDL или графические представления. В этой папке содержится папка, предназначенная для хранения подключаемых библиотек описания элементов — **Library Resources** (библиотеки-ресурсы);
- **Simulation Resources** (ресурсы моделирования) — папка содержит файлы, используемые в процессе моделирования. Она включает в себя как файлы описания моделей, так и файлы входных данных. В эту папку могут включаться также промежуточные результаты моделирования. Разрабатываемое устройство может моделироваться двумя способами. Соответственно, для каждого из них может быть свой набор файлов, используемых в процессе моделирования. Каждый из этих наборов размещается в папке, соответствующей этому способу моделирования: **In design** (функциональное), **Timed** (временное).
  - **In Design** — ресурсы для моделирования схемы на уровне исходных текстов проекта. Это могут быть файлы дизайна схем, VHDL-файлы, списки цепей, файлы сценария моделирования.
  - **Timed** — ресурсы для моделирования с учетом временных характеристик, специфических для выбранного способа реализации. Эта информация может храниться в STD-файлах (Standard Delay Files — файлы стандартных задержек) или в файлах списков цепей, с указанием временных характеристик, связанных с каждой цепью.
- **Outputs** (выходные результаты) — файлы результатов проектирования. Это могут быть списки связей или временные характеристики модели, представленные в формате, требуемом фирмой-производителем;
- **Referenced Projects** (связанные проекты) — ссылки на другие проекты.

Один и тот же файл может быть указан одновременно в нескольких папках.

Представление проекта в иерархическом виде

Если проект представлен в иерархическом виде, то в окне менеджера проекта отображается древовидная структура, корнем которой является сущность, соответствующая моделируемому устройству в целом. Каждому уровню иерархии дерева соответствует перечень блоков, находящихся на этом уровне иерархии устройства.

Информация о блоках, отображаемая в окне менеджера проектов определяется набором установленных флажков из следующего перечня:

- **All** (Все сигналы используемые выделенным блоком);
- **Inputs** (Входные сигналы);
- **Outputs** (Выходные сигналы);
- **Bidirect** (Двунаправленные сигналы);
- **Groups** (Группы сигналов);
- **Ports** (Порты).

Командная строка

Командная строка может использоваться для выполнения большинства команд, присутствующих в меню. С ее помощью можно управлять загрузкой проекта, выполнением моделирования и отладки.

Однако она предоставляет доступ не ко всем возможностям оболочки. Например, с ее помощью невозможно редактирование значений входных сигналов.

Размер окна командной строки можно изменять в процессе работы.

Окно включает в себя две области:

- **область команд**;
- **область информации**.

Обе области снабжены линейками для вертикального скроллинга и имеют общую линейку для горизонтального скроллинга.

Область команд

В область команд можно записывать текстовые строки, содержащие имена команд, полные или сокращенные, и параметры. Для того чтобы вводить команды в область команд, необходимо, чтобы в ней присутствовал курсор. В одну строку можно вводить несколько команд. После ввода очередной команды проверяется ее синтаксис и, если ошибки не обнаружены, она выполняется.

Для размещения комментариев используется разделитель `//`. Часть строки, находящаяся справа от разделителя, рассматривается в качестве комментария.

Одна и та же команда может быть выполнена несколько раз. Для выполнения уже существующей в окне команды ее необходимо выделить и затем нажать клавишу <Enter>. Можно редактировать уже существующие команды. Для того чтобы очистить область команд, используется пункт **Clear All** (Очистить все) меню **Edit** (Редактирование).

Использование командной строки эффективно при многократном выполнении действий по одному и тому же сценарию.

Область информации

В области информации отображаются сообщения об ошибках, сообщения для помощи пользователю, диагностические сообщения.

Окно информации о стеке

Окно информации о стеке предоставляет возможности по отладке подпрограмм, используемых в моделях. В нем отображается информация об активной в данный момент времени подпрограмме. Это окно включает в себя следующие секции:

- **Context** (Контекст) — в этой секции указывается контекст, из которого была вызвана активная подпрограмма. Если подпрограмма была вызвана не из процесса, то контекстом считается верхний уровень проекта;
- **Processes** (Процессы) — в этой секции содержится выпадающий список активных процессов, содержащих вызовы подпрограмм. Процессы в этом списке идентифицируются своими метками. Если в описании модели процессу не была присвоена метка, она синтезируется автоматически. По умолчанию, в этом списке оказывается выбранным процесс, во время выполнения которого, моделирование было прервано;
- **Call Stack List** (Список вызовов) — в этом списке отображается информация об активной подпрограмме: актуальные значения параметров и номер строки подпрограммы, на которой было прервано выполнение. Если вызов подпрограммы был вложенным, то в этом списке содержится информация о вложенности. Если щелкнуть правой кнопкой мыши по этому окну, появится всплывающее меню, которое содержит один пункт **Edit**. Его использование позволяет отредактировать VHDL-файл, содержащий активную подпрограмму;
- **Data List** (Список данных) — в этом списке можно просмотреть текущие значения переменных, констант и сигналов в подпрограмме.

Рабочая область

В рамках рабочей области могут быть расположены окна редактора описания схемы на VHDL, окна временных диаграмм и таблиц, окна для интерактивного определения входных значений.

Работа с проектом

Разрабатываемое в среде устройство представляется в виде *проекта.*

Проекту сопоставляется так называемый *файл проекта.* Каждому проекту соответствует один и только один файл проекта. В этом файле хранятся ссылки на все входящие в состав проекта файлы.

В проект включаются файлы описания устройства (графические и на языке VHDL). Кроме того, в проект могут входить файлы описания входных параметров схемы и файлы результатов моделирования, а также библиотеки и файлы стандартов фирм-производителей.

Поддерживается два типа проектов FPGA (для ПЛИС и полузаказных СБИС) и проекты PCB (для устройств на печатной плате). Они отличаются способами представления выходных данных. Тип проекта выбирается в соответствии с требованиями к выходным данным фирмы производителя проектируемого устройства на СБИС.

Файлы описания устройства, файлы входных параметров и библиотеки могут включаться в несколько проектов. Соответственно, ссылки на них могут присутствовать в нескольких файлах проектов.

Формирование проекта

Описание проектируемого устройства может быть сделано в виде графических схем и на языке VHDL. Причем, если предполагается выполнение моделирования устройства, проект обязательно должен включать файлы описания поведения на VHDL. Это могут быть библиотеки описания элементов или файлы, включающие в себя нестандартные описания элементов.

Для создания нового проекта используется пункт **New** (Новый) в меню **File** (Файл). При этом, тип создаваемого объекта определяется как **OrCAD Project** (Проект). С помощью этого же пункта меню можно создать **Library** (библиотеку графических представлений элементов) или **VHDL file** (файл на VHDL).

Для открытия уже существующего проекта, библиотеки или VHDL-файла используется пункт **Open** (Открыть) меню **File** (Файл).

После того, как создан проект или открыт уже существующий проект, содержимое проекта отображается в **Project manager window** (окне менеджера проектов). Каждому открытому проекту соответствует одно и только одно такое окно. Каждый открытый проект имеет свое собственное окно менеджера проектов. Закрытие этого окна приводит к закрытию проекта.

Внимание

Один и тот же файл может входить в состав нескольких проектов. Если в ходе работы над одним из них он будет модифицирован, то изменится и работа других проектов, включающих в себя этот файл.

После того как создан новый файл проекта или открыт уже существующий, можно создавать новые файлы графического представления схем или описания на VHDL.

Для того чтобы вновь создаваемые или уже существующие файлы графического представления схем, описания схем на VHDL вошли в состав проекта, их необходимо добавить в одну из его папок. Все файлы, которые участвуют в моделировании, необходимо добавлять в папку **Simulation Resources** (Ресурсы моделирования), во вложенную папку **In Design** (функциональное моделирование) или **Timed** (временное моделирование) в зависимости от их предполагаемого использования.

Для того чтобы добавить уже существующий файл в соответствующий элемент структуры (папку проекта), необходимо выделить этот элемент и щелкнуть правой кнопкой мыши, после чего надо указать имя добавляемого файла.

Для удаления файла из проекта необходимо выделить имя удаляемого файла и воспользоваться пунктом **Delete** (Удалить) меню **Edit** (Редактирование). Файл удаляется только из проекта, но на диске остается, не стирается.

В файле проекта указываются пути ко всем файлам, входящим в состав проекта, относительно самого файла проекта, поэтому, если предполагается возможность перенесения проекта с одного компьютера на другой, то целесообразнее размещать все файлы, входящие в состав проекта, в одной папке.

Работа с VHDL-файлами

Файл на VHDL можно подготовить в любом текстовом редакторе, поддерживающем формат "MS-DOS текст" с концами строк. Созданный файл должен иметь расширение VHD.

Создать файл можно и в самой среде OrCAD Express. Для этого необходимо воспользоваться пунктом **New** (Новый) меню **File** (Файл).

При моделировании используются следующие основные типы VHDL-файлов:

- **VHDL source** — (исходный код части модели, находящейся на верхнем уровне иерархии) содержит одну или несколько поведенческих моделей объектов, которые представлены в структуре проекта, одна из этих моделей соответствует верхнему уровню иерархии проекта;
- **VHDL simmodel** — (исходный код части модели) содержит одну или несколько поведенческих моделей объектов;
- **Netlist** — (список связей) список связей между объектами, входящими в состав схемы, и описание входов и выходов объектов;
- **Test Bench** — (тестовый набор) используются для задания значений входных сигналов.

Внимание

Имена одних и тех же сигналов должны в точности совпадать в разных VHDL-файлах, входящих в состав одного проекта.

Если проект включает в себя полное графическое представление схемы, то в него не должен входить файл на VHDL типа **Netlist**, содержащий описание этого же графического представления в папке **Design resources**.

Если по каким то причинам нежелательно, чтобы в проект входил уже существующий файл графического представления схемы, то его можно заменить файлом списка связей. Для генерации этого файла на базе графического представления схемы можно воспользоваться пунктом **Create Netlist** (Создать список связей) меню **Tools** (Инструменты) фрейма OrCAD Capture.

Модель одного объекта должна целиком размещаться в одном файле, однако, если объект состоит из нескольких объектов (компонентов), то модели этих компонентов могут размещаться в различных файлах. Для системы моделирования неважно, содержатся ли поведенческие модели объектов в одном или нескольких файлах.

Для того чтобы изменить тип файла, можно щелкнуть по нему правой кнопкой мыши, в появившемся меню выбрать пункт **Properties** и выбрать из списка нужный тип файла.

Для того чтобы проверить синтаксис программы, написанной на VHDL, можно воспользоваться пунктом **Check Syntax** (Проверка синтаксиса) меню **Edit** (Редактирование). Для того чтобы использовать в модели фрагмент часто встречающегося кода, можно воспользоваться пунктом **Samples** (Примеры) меню **Edit** (Редактирование).

Компиляция проекта

Прежде чем компилировать проект, желательно выполнить проверку синтаксиса. Далее, для выполнения компиляции необходимо воспользоваться пунктом **Reload project** (Перезагрузить проект) меню **Simulate** (Моделирование).

Внимание

Если в ходе работы с проектом в него вносятся изменения, то для того, чтобы они отображались в процессе моделирования, необходимо перекомпилировать проект.

В панели инструментов фрейма OrCAD **Simulate** можно выбрать тип компиляции: **In Design** (функциональная) или **Timed** (временная). Использование типа **In Design** позволяет выполнить моделирование на верхнем уровне; в этом случае используются временные задержки сигналов, которые в явном виде указаны в модели. Использование типа **Timed** имеет смысл только по-

сле выполнения размещения и трассировки СБИС. Оно позволяет определить реальные временные характеристики полученной схемы.

После выбора типа моделирования будет произведена компиляция всех файлов программ на VHDL и файлов рисунков схем, входящих в состав проекта. Для файлов рисунков схем список цепей будет сформирован автоматически.

Если соответствующая папка проекта не пуста, то будут загружены находящиеся в ней файлы описания модели и заданы вопросы о необходимости загрузки файлов входных значений (при их наличии). Если проект не содержит ошибок, то будет выдано сообщение о том, что проект успешно загружен; в противном случае сообщения об ошибках можно просмотреть в окне сессии.

Формирование тестовых наборов

Для работы с наборами значений входных сигналов может использоваться меню **Stimulus** (Тесты).

В OrCAD **Simulate** поддерживается два способа определения входных значений сигналов.

Первый способ — интерактивный. Входные значения формируются и могут в дальнейшем редактироваться с использованием набора соответствующих диалоговых окон. Наборы значений могут сохраняться в бинарных файлах, по умолчанию имеющих расширение STM. Эти файлы называются *файлами сценария.*

Второй способ основывается на возможностях языка VHDL. Фрейм OrCAD **Simulate** генерирует и добавляет в проект VHDL файл, имеющий тип **Test Bench**. Операторы для определения значений входных сигналов могут быть вписаны пользователем в этот файл.

Интерактивный способ формирования тестовых наборов

Для создания нового списка значений входных сигналов используется пункт **New Interactive** (Новый интерактивный тест) меню **Stimulus.** .

Редактирование уже существующего списка осуществляется с помощью пункта **Edit Interactive** (Редактировать интерактивный тест) меню **Stimulus**. Список значений можно сохранить в файле (по умолчанию имеет расширение stm). Файл списка значений можно добавить в папку **Simulation Resources** проекта.

В проекте может быть несколько файлов списков значений, но активным, то есть используемым при моделировании, может быть только один из них. Для того чтобы файл сделать активным, необходимо воспользоваться пунктом **Load Interactive** (Загрузить интерактивный тест) меню **Stimulus**. Для того чтобы файл списка значений перестал быть активным, необходимо воспользоваться пунктом **Unload Interactive** (Выгрузить интерактивный тест) меню **Stimulus**.

При создании или редактировании списка значений входных сигналов открывается диалоговое окно. Это окно содержит три вкладки:

- **Absolute** (абсолютное определение). На этой вкладке можно определить последовательность значений для входных сигналов;
- **Relative** (относительное определение). На этой вкладке можно определить последовательность значений для входных сигналов. Эта последовательность будет иметь циклическую структуру; отдельные повторения могут несколько отличаться друг от друга;
- **Clock** (определение сигналов тактирования). На этой вкладке можно описать значения для сигнала тактирования; этот сигнал может принимать только два значения, которые циклически повторяются.

Последовательность значений для конкретного сигнала может определяться только на одной из этих вкладок.

Внимание

Если сигналу, присутствующему в списке входных значений, в тексте модели присваиваются новые значения, они игнорируются.

Работа с вкладками имеет много общего. Для того чтобы описать сигнал, необходимо или выбрать его из списка сигналов, который вызывается с помощью кнопки **Browse** (Просмотреть), или выбрать его из списка сигналов, уже входящих в этот набор для моделирования (они отображаются в окне **Signals under Stimulus,** расположенного левее этой кнопки). Список сигналов, получаемый с помощью **Browse** (Просмотреть), иерархический. Для работы с ним сначала необходимо выбрать входящий в состав модели объект, которому принадлежит сигнал, после чего появится список сигналов этого объекта, и из этого списка уже можно выбрать нужный сигнал.

После того как сигнал выбран, для него в окне **Stimulus Descriptions** (тестовое описание) отображается существующий список изменений значения. Затем можно модифицировать уже существующую последовательность значений или создавать новую. Это действие несколько отличается на разных вкладках.

Определение значений на вкладке **Absolute**. Для определения последовательности значений сигнала на этой вкладке используется два окна — **Set to** (установить в) и **At time** (в момент времени). В окне **Set to** указывается очередное значение, а в окне **At time** — время, в которое оно должно появиться в модели. Это время рассматривается как абсолютное модельное время. Если значение оказывается меньшим, чем текущее модельное время (оно отображается в верхней части вкладки), то это значение уже не повлияет на текущее моделирование, пока модельное время не будет сброшено в 0. Нажимается кнопка **Add** (Добавить). Таким образом, сигнал описывает последовательность пар значение/время появления. Эта последовательность ото-

бражается в окне **Stimulus Descriptions**. Значение, которое определено в списке последним, при моделировании сохраняется до конца. Если оно должно быть сброшено в конкретный момент времени и дальше сигнал должен оставаться в неопределенном состоянии, то можно воспользоваться кнопкой **Remove** (сброс), вместо того чтобы определять конкретное значение сигнала. Пример содержимого окна **Stimulus Descriptions** приведен в листинге 6.1.

Листинг 6.1

```
+ Set to '0' at time 0
+ Set to '1' at time 10
+ Set to '0' at time 100, Remove at time 150
```

Знаки "+", стоящие перед каждой строкой, указывают на то, что все они участвуют в моделировании.

Определение значений на вкладке **Relative**. На этой вкладке можно задать несколько наборов значений и количество повторений для каждого набора. При очередном повторении набор может подвергаться действию функций инкремента и декремента.

Рассмотрим, как определяются значения сигнала с помощью этой вкладки. С помощью кнопки **Set to** можно зафиксировать очередное значение сигнала. Собственно значение можно определить в окне, расположенном рядом с кнопкой. Кнопка **Wait for** (ждать) позволяет задать длительность удержания этого значения в наносекундах, оно определяется в соседнем с ней окне. Кнопка **Repeat block** (повторить блок) позволяет внести в описание сигнала границы цикла повторения. Окно **Times** (количество повторений) позволяет определить количество повторений. Для типов с определенными операциями инкремент и декремент, окна, снабженные соответствующими кнопками, позволяют задать приращение или уменьшение значения сигнала при очередном выполнении цикла. Таким образом, сигнал описывается циклически повторяющейся последовательностью пар значение/длительность. От цикла к циклу значения могут подвергаться действию функций инкремент или декремент.

Рассмотрим, например, как с помощью этой вкладки можно определить набор входных значений для сущности, описание которой приведено в листинге 6.2.

Листинг 6.2

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;
```

```
entity test_1 is
port (i1,i2: in std_ulogic_vector(0 to 3);
      o1: out std_ulogic_vector(0 to 3));
end entity test_1;
architecture rtl of test_1 is
 begin
process (i1,i2)
begin
  o1<=i1 and i2;
end process;
end;
```

Пусть, например, входной сигнал i1 изменяет свое значение от 0 до 10 с шагом 1. Длительность сохранения каждого значения 5 нс. Сигнал i2 изменяет свое значение от 12 до 1 с шагом 1. Длительность сохранения каждого значения 6 нс. Рассмотрим, как можно задать такую последовательность значений с использованием вкладки **Relative** на примере сигнала i1. Для того чтобы присвоить сигналу начальное значение 0, необходимо вписать это значение в строку ввода рядом с кнопкой **Set To**, затем нажать эту кнопку. Затем, для определения времени сохранения этого значения, в строке ввода (рядом с кнопкой **Wait for)** необходимо указать значение 5 и нажать эту кнопку, в результате чего в окне **Stimulus Descriptions** появится следующая последовательность строк:

```
+Set to 0 (Dec)
+Wait for 5 ns
```

Затем в окне **Times** необходимо определить количество повторений (10) и нажать кнопку **Repeat Block**. В результате в окне **Stimulus Descriptions** добавятся две новые строки:

```
+Begin Repeat 10
+End Repeat
```

Эти строки определяют границы цикла. По умолчанию курсор установлен на последней строке, в результате все новые строки будут добавляться внутрь цикла, где должны выполняться следующие действия. Значение сигнала должно увеличиваться на 1 и сохраняться таковым в течение 5 нс. Для этого добавим в тело цикла две строки. Для того чтобы значение сигнала увеличивалось на 1, необходимо это число указать в строке ввода рядом с кнопкой **Increment by** (увеличить на) и нажать эту кнопку. Потом необходимо определить время сохранения сигнала, как это было сделано выше. Текст, который в результате наших действий будет в **Stimulus Descriptions**, приведен в листинге 6.3.

Листинг 6.3

```
+Set to 0 (Dec)
+Wait for 5 ns
+Begin Repeat 10
+    Increment by 1
+    Wait for 5 ns
+End Repeat
```

Для сигнала i2 должны быть выполнены аналогичные действия. Текст описания для него приведен в листинге 6.4.

Листинг 6.4

```
+Set to 12 (Dec)
+Wait for 6 ns
+Begin Repeat 10
+    Decrement by 1
+    Wait for 6 ns
+End Repeat
```

Временная диаграмма работы устройства приведена на рис. 6.6.

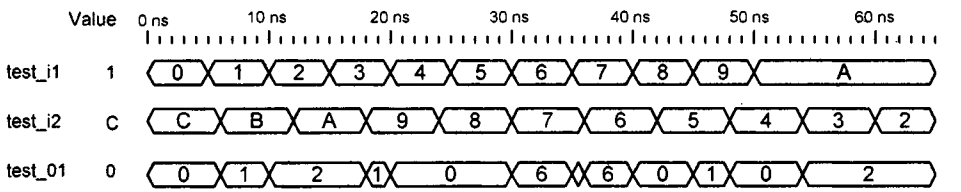


Рис. 6.6. Временная диаграмма работы устройства (листинг 6.4)

Определение значений на вкладке **Clock**. Эта вкладка может использоваться для описания сигналов тактирования. В ней, в окнах **Set to** и **For** первой строки, можно указать значение, которое сигнал тактирования будет иметь в первой части такта, и ее длительность, а в окнах **Set to** и **For** второй строки — значение, которое сигнал тактирования будет иметь во второй части такта, и ее длительность. В окне **Start at** (начать в) можно задать время начала тактирования. Переключатель **Repeat/Repeat forever** определяет, будет ли цикл повторяться определенное количество раз (заданное в окне **Times**) или повторяться постоянно.

После того как для сигнала определено очередное значение или группа значений, в окне **Stimulus decriptions** добавляется строка с описанием введенного значения. Желательно, чтобы строки описания значений стояли в поряд-

ке возрастания времени. Каждое значение сигнала действует до тех пор, пока не наступит время смены сигнала, указанное в этом же наборе, или до окончания моделирования.

В начале вновь введенной строки описания сигнала стоит знак +. Он означает, что строка активна. Для того чтобы строка временно не учитывалась при моделировании, необходимо щелкнуть по ней мышью и нажать кнопку **Disable** (Деактивация), после чего строка будет помечена знаком -. Для активации строки необходимо воспользоваться кнопкой **Enable** (Активировать). На вкладке **Relative** можно активировать и деактивировать только все строки сразу. На этой вкладке имеются кнопки **Disable All** и **Enable All**. Для удаления строки служит клавиша **Delete** (Удалить), а для вставки — **Insert**. Для редактирования строки, ее необходимо выделить мышью, после чего ее содержимое можно отредактировать. После окончания редактирования, для того чтобы изменения вступили в силу, необходимо нажать кнопку **Change** (Изменить).

После того как ввод закончен, надо нажать клавишу <Enter> на клавиатуре или кнопку **OK** в нижней части диалогового окна.

При создании новой строки и при редактировании допустимые значения сигнала можно выбрать из выпадающего списка. Выпадающий список содержит перечень значений, допустимых для выбранного сигнала. Это может быть или список MVL-9, или символическая карта. Если рассматриваемый сигнал является группой, то можно указать специфический для этой группы тип (DEC, HEX, OCT, BIN), в котором будет задаваться значение.

Карта символов. Карта символов состоит из множества пар значение/символ. Каждая пара содержит значение сигнала и строку символов, которая отображается, когда группа сигналов принимает это значение. Если группа сигналов приняла значение, для которого в карте нет описания, то отображается только само это значение.

Каждая Карта символов должна иметь свое уникальное имя. Одна и та же Карта символов может соответствовать многим группам сигналов в модели, но только одной группе в пределах одного окна диаграммы.

Файлы тестов на VHDL (Test Bench-файлы)

Для создания **Test Bench**-файлов (файлов тестов на VHDL) используется пункт **Create Test Bench** (Создать тестовый файл) меню **Stimulus** (Тесты). В диалоге определяется сущность, для которой будет создан тестовый файл, и имя тестового файла. Если установлен флажок **Add To Project** (Добавить в проект), то файл будет автоматически добавлен в проект в соответствующую папку **Simulation Resources** (Ресурсы моделирования). Созданный файл содержит две сущности, предназначенные для выполнения различных вариантов тестирования. Имена этих сущностей формируются следующим обра-

зом: baseentityname_stub и test_baseentityname, где baseentityname — имя сущности, для которой строится тест.

Сущность baseentityname_stub предназначена для выполнения отображения между InDesign test bench и post P&R списком связей. В нем производится отображение векторных портов из представления InDesign в множества скалярных портов, так как они представлены в post P&R, рис. 6.7.

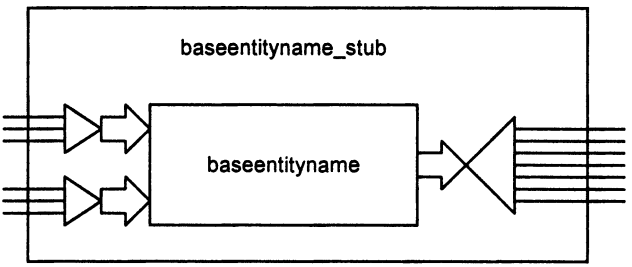


Рис. 6.7. Преобразование портов в сущности baseentityname_stub

Перед описанием этой сущности в файле расположен комментарий, специфицирующий действия, которые необходимо выполнить с описанием перед его использованием.

Рассмотрим сущность baseentityname_stub. Описание портов этой сущности совпадает с описанием портов для baseentityname. В нее включается компонент baseentityname, в описании которого все векторные порты представлены в виде множества скалярных. Архитектурное тело содержит отображение:

```
Dut: baseentityname_hide
```

порты которого соответствуют портам компонента baseentityname.

Прежде чем использовать baseentityname_stub, необходимо в его описании изменить имя устройства, подлежащего тестированию с baseentityname_hide на baseentityname и изменить имя компонента с baseentityname на baseentityname_stub.

Сущность test_baseentityname является оболочкой для выполнения моделирования. Эта сущность не имеет входных и выходных портов. Сущность baseentityname включается в нее в качестве компонента. В ней описываются сигналы, совпадающие по имени и типу с входными и выходными сигналами baseentityname. Это иллюстрируется рис. 6.8.

Архитектурное тело содержит отображение

```
Dut: baseentityname
```

порты которого отображаются на одноименные сигналы, описанные внутри test_baseentityname.

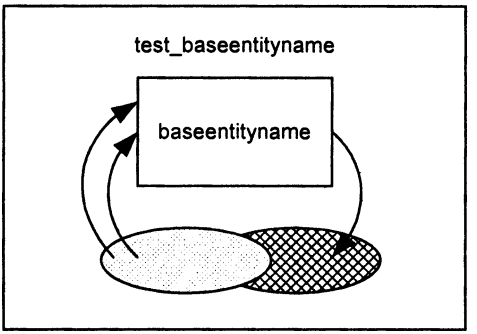


Рис. 6.8. Структура сущности `test_baseentityname`

Перед использованием `test_baseentityname`, в место описания, помеченное комментарием

```
Place stimulus and analysis statements here
```

можно вписать операторы, определяющие значения входных сигналов и анализ выходных сигналов для `baseentityname`. Это может быть совокупность параллельно выполняющихся операторов или совокупность процессов. Значения входных сигналов могут определяться на базе значений выходных сигналов. Этим определяется преимущество использования данного метода перед заданием значений сигналов с помощью **Interactive** (Интерактивного определения). Там значения сигналов определяются безусловно, а в Test Bench возможно их условное определение.

В окне менеджера проектов тестовый файл помечается буквой T и тип его — VHDL Test Bench.

После формирования файла и добавления его в проект, для того чтобы начать его использовать, проект необходимо перезагрузить. При перезагрузке выдается диалоговое окно, в котором запрашивается, какая из двух рассмотренных выше сущностей будет использоваться. Выбранная сущность получает статус *корневой*. Ее пиктограмма в менеджере проектов помечается косой чертой.

Корневую сущность можно определить и вручную. Для этого в менеджере проектов необходимо выбрать эту сущность и воспользоваться пунктом **Make Root** (Сделать корневой) меню **Edit** (Редактирование).

В проекте может быть только одна корневая сущность. Если на момент определения, в проекте уже была корневая сущность, то она перестает быть корневой. Если в проекте в явном виде не определена корневая сущность и нет файла VHDL Test Bench, то корневой считается сущность, находящаяся в файле, имеющем тип VHDL source.

Если описание модели имеет иерархическую структуру, то для выполнения моделирования необходимо также указать сущность, которая будет рассматриваться как сущность, находящаяся на верхнем уровне иерархии. Она не обязательно физически должна находиться на верхнем уровне иерархии, но в этом случае все вышерасположенные сущности в моделировании принимать участие не будут.

Рассмотрим пример использования этого механизма. Описание моделируемой сущности приводится в листинге 6.5.

Листинг 6.5

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity mytest is
 port (in1, in2: in std_ulogic_vector(3 downto 0);
       out1: out std_ulogic_vector(3 downto 0)  ;
       out2: out std_ulogic  );
end mytest;

architecture rtl of mytest is
begin
process (in1,in2)
variable vout: std_ulogic_vector (3 downto 0);
begin
 vout :=in1 and in2 ;
 out1 <=vout after 5 ns;
 out2 <=vout(0) after 5 ns;
end process;
end architecture;
```

В данном примере моделируемая сущность имеет название mytest. Она имеет два входных порта векторного типа и два выходных порта, один из которых так же принадлежит векторному типу. Test Bench-файл, сгенерированный OrCAD Simulate для этой сущности, приведен в листинге 6.6.

Листинг 6.6

```
-- VHDL Stub to map between InDesign testbench and post-P&R netlist
-- This stub file maps vector ports from the InDesign representation
-- to scalar ports in the post-P&R
```

```
-- To use this mapping do the following:
--   1. Change the name of the device under test in the architecture
--      of 'mytest_stub' from 'mytest_hide' to 'mytest'
--   2. Change the name of the component referenced
--      in the test bench below
--      from 'mytest' to 'mytest_stub'

-- Created by OrCAD  Simulate     Mon Mar 18 16:39:07 2002

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mytest_stub is port (
     out2 : out std_ulogic;
     out1 : out std_ulogic_vector(3 downto 0);
     in2 : in std_ulogic_vector(3 downto 0);
     in1 : in std_ulogic_vector(3 downto 0)
  );
end mytest_stub;

architecture mapping of mytest_stub is

-- Declaration of the mapped component
component mytest
  port (
     out2 : out std_ulogic;
     out10 : out std_ulogic;
     out11 : out std_ulogic;
     out12 : out std_ulogic;
     out13 : out std_ulogic;
     in20 : in std_ulogic;
     in21 : in std_ulogic;
     in22 : in std_ulogic;
     in23 : in std_ulogic;
     in10 : in std_ulogic;
     in11 : in std_ulogic;
     in12 : in std_ulogic;
     in13 : in std_ulogic
  );
end component;
```

```
begin
dut :  mytest_hide  port map (
     out2 => out2,
     out10 => out1(0),
     out11 => out1(1),
     out12 => out1(2),
     out13 => out1(3),
     in20 => in2(0),
     in21 => in2(1),
     in22 => in2(2),
     in23 => in2(3),
     in10 => in1(0),
     in11 => in1(1),
     in12 => in1(2),
     in13 => in1(3)
  );
end mapping;

-- ************************************************************************
-- ************************************************************************
-- ************************************************************************
-- Test bench shell
-- Created by OrCAD  Express Simulate
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test_mytest is end test_mytest;
architecture testbench of test_mytest is
-- Declaration of the component under test
component mytest
  port (
     out2 : out std_ulogic;
     out1 : out std_ulogic_vector(3 downto 0);
     in2 : in std_ulogic_vector(3 downto 0);
     in1 : in std_ulogic_vector(3 downto 0)
      );
end component;
```

```
signal   out2 : std_ulogic;
signal   out1 : std_ulogic_vector(3 downto 0);
signal   in2 : std_ulogic_vector(3 downto 0);
signal   in1 : std_ulogic_vector(3 downto 0);

begin
-- Place stimulus and analysis statements here
process
begin
in1 <= "0000", "0001" after 10 ns; wait for 20 ns;
end process;
process(out2)
begin
 if out2='1' then in2<=out1;
  else in2<="1101";
 end if;
end process;
dut :  mytest  port map (
     out2 => out2,
     out1 => out1,
     in2 => in2,
     in1 => in1
      );
end testbench;
```

В приведенном листинге курсивом отмечен фрагмент, который написан пользователем, а не сгенерирован средой автоматически.

Сущность `mytest_stub` содержит компонент `mytest`, порты которого, в отличие от портов исходной сущности `mytest`, не являются векторными. Порты с именами `in10`, `in11`, `in12`, `in13` соответствуют разрядам 0 — 3 порта `in1` исходной сущности, для которой строился тест. Как можно заметить, названия этих портов автоматически формируются на базе названия порта исходной сущности. К названию каждого порта добавляется номер разряда векторного порта, которому он соответствует. Аналогично, порты с именами `in20`, `in21`, `in22`, `in23` соответствуют порту `in2` исходной сущности. Порты `out10`, `out11`, `out12` и `out13` соответствуют порту `out1` исходной сущности. Порт `out2` исходной сущности изначально был скалярным, поэтому он не претерпел изменений.

Рассмотрим теперь сущность `test_mytest`. В нее вручную добавлены два процесса, позволяющие определить значения входных сигналов сущности

mytest. В первом процессе определяются значения для сигнала in1. Этому сигналу присваивается значение 0, а через 10 нс. — значение 1. Эти действия повторяются с периодичностью в 20 нс., т. е. сигнал сохраняет значение 1 в течение 10 нс. (20—10 нс.), после чего цикл начинается снова и сигналу присваивается значение 0.

Во втором процессе определяются значения для сигнала in2. Значение этого сигнала определяется в зависимости от значений выходных сигналов. Оно может измениться при изменении выходного сигнала out2 и принять значение сигнала out1 или 13.

Если приведенный в примере набор входных значений для сигнала in1 можно определить с использованием интерактивных механизмов, то для сигнала in2 это не возможно. Временная диаграмма работы модели приведена на рис. 6.9.

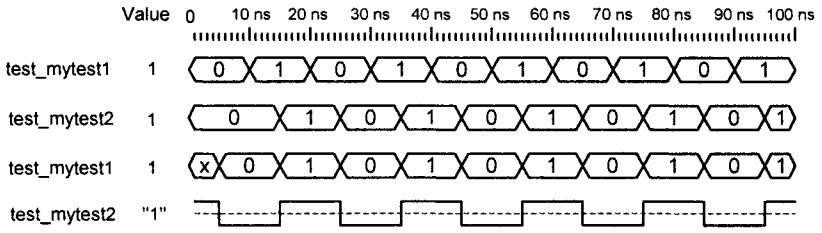


Рис. 6.9. Временная диаграмма работы модели (листинг 6.6)

Выполнение моделирования и просмотр результатов

Прежде чем выполнять моделирование, необходимо сформировать форму для просмотра результатов.

Работа с результатами моделирования

Результаты моделирования можно просматривать в виде списка значений или в виде временной диаграммы.

С помощью пунктов **New List Window** (Новое окно списка) или **New Wave Window** (Новое окно временной диаграммы) меню **Trace** (Трассировка) можно сформировать список сигналов, которые будут отображаться, соответственно, в списке сигналов или на временной диаграмме. После выбора одного из этих пунктов, из имеющихся в схеме сигналов, можно выбрать те, которые будут отображаться, а также выбрать соответствующий им тип, в соответствии с которым будет отображаться каждый сигнал. Выбор осуществляется в режиме диалога.

Изменить перечень отображаемых в окне сигналов можно пунктом **Edit signal traces** (Редактировать список сигналов) меню **Trace** или соответствующим пунктом выпадающего меню, которое появляется при щелчке правой кнопкой мыши в окне временной диаграммы.

По мере выполнения моделирования, в созданных таким образом окнах можно просматривать результаты. Если уже после выполнения моделирования в эти окна будут добавлены новые сигналы, то их значения можно будет просмотреть, только осуществив моделирование заново.

В среде может быть открыто несколько окон временных диаграмм и таблиц значений сигналов одновременно.

Wave Window (Окно временной диаграммы). Все окна временных диаграмм, существовавшие в среде моделирования на момент закрытия проекта, автоматически добавляются в проект и могут быть использованы при последующей работе с ним. Информация, содержащаяся в окне временной диаграммы, может быть также сохранена в файле (в формате ASCII текст). Такие файлы могут быть использованы для сравнения результатов моделирования, но информация из них не может быть отображена в окне временной диаграммы.

Окно включает в себя четыре вертикальные области: **Context** (Контекст), **Signal** (Сигнал), **Value** (Значение), **Waveform** (Временная диаграмма). Размеры областей можно изменять в ходе работы. Информация о каждом сигнале расположена в отдельной строке, которая проходит через все эти области. В области **Context** указывается контекст сигнала, т. е. наименование сущности или компонента, которому принадлежит сигнал. В области **Signal** указывается собственно имя сигнала. В области **Value** указывается значение сигнала в момент модельного времени, на который установлен указатель. Область **Waveform** содержит временные диаграммы сигналов. В верхней части этой области расположена шкала времени. Щелчок мыши в любой точке этой области позволяет установить указатель на соответствующее место этой шкалы. Указатель отображается на экране как вертикальная черта черного цвета.

Для того чтобы отметить начало и конец некоторой области временной диаграммы, можно использовать так называемые *дельта-маркеры*. Для того чтобы установить дельта-маркер, необходимо щелкнуть левой кнопкой мыши по соответствующей точке шкалы времени. Дельта-маркер, соответствующий левой границе области, отображается на экране как вертикальная черта синего цвета, а правой границе — красного цвета. Моменты времени, соответствующие текущему положению маркеров отображаются в строке статуса, что позволяет легко определить длительность отмеченного фрагмента временной диаграммы.

Для документирования результатов моделирования можно выделить строки, соответствующие необходимым сигналам, скопировать в буфер, а уже из буфера, как рисунок, перенести их в другие программы, запущенные на

Вашей машине, или в текстовый файл (например, в редакторе Word). Если некоторый фрагмент временной диаграммы отмечен дельта-маркерами, то копируется именно он, что бывает удобно, когда временная диаграмма имеет большую протяженность, в противном случае копируется вся временная диаграмма.

List Window (Окно списка сигналов). В этом окне информация о значениях сигналов в ходе моделирования формируется в виде таблицы. В заголовке таблицы отображаются имена выбранных для просмотра сигналов. Сам заголовок не является прокручиваемой областью. При моделировании каждый раз, когда изменяется значение хотя бы одного сигнала, в эту таблицу добавляется новая строка значений сигналов. В левой части строки указывается момент модельного времени, которому она соответствует.

Выполнение моделирования

Для выполнения моделирования используются команды **Run** (Выполнить) и **Run to** (Выполнить до) меню **Simulate** (Моделирование).

При использовании команды **Run** в диалоге указывается суммарное время моделирования. Если используется команда **Run to**, то в диалоге указывается, до какого времени необходимо продолжать моделирование.

Если эти команды используются несколько раз подряд, то каждый раз моделирование начинается с того момента времени, на котором закончилось предыдущее.

Для того чтобы начать моделирование с момента времени 0, необходимо воспользоваться командой **Restart** (Возобновить). После выполнения этой команды моделирование начинается с момента времени 0 и, соответственно, используются начальные значения входных сигналов из активного списка значений входных сигналов. Если проект был перезагружен, то моделирование его так же начнется с момента времени 0.

Отладка модели

Для отладки модели во фрейме OrCAD Simulate предусмотрены следующие механизмы:

- пошаговое выполнение;
- механизм точек останова;
- механизм событий.

Пошаговое выполнение

Для проверки и отладки частей модели, написанных на VHDL, можно применить пошаговое выполнение (пункт **Step** (Шаг) меню **Simulate**). Модели-

рование можно прервать в любой момент времени, используя пункт **Stop** (Остановить) меню **Simulate** и затем продолжить, при помощи пункта **Continue** (Продолжить). Этот же пункт меню может быть использован для продолжения моделирования после просмотра предупредительного сообщения, инициатором выдачи которого является модель.

Механизм точек останова

Для отладки модели можно использовать механизм точек останова. Предусмотрено два типа точек останова:

- точки останова, устанавливаемые на конкретной строке текста VHDL (устанавливаются с помощью команды **Break on line** меню **Simulate**);
- точки останова, базирующиеся на логическом выражении (устанавливаются с помощью команды **Break on expr** меню **Simulate**).

После того как моделирование будет прервано на точке останова, для продолжения можно использовать пункт **Continue** меню **Simulate** или перейти на пошаговое выполнение моделирования.

Точки останова, устанавливаемые на конкретной строке текста. Точку останова этого типа также можно установить, щелкнув на строке правой кнопкой мыши, и выбрав в появившемся меню пункт **Break on line**. Строка при этом помечается значком **Stop**. Для того чтобы убрать точку останова, можно щелкнуть по ней правой кнопкой мыши и выбрать в появившемся меню соответствующий пункт.

Точки останова, заданные логическим выражением. В логическом выражении присутствует имя сигнала и константа, с которой сравнивается значение сигнала. Каждый сигнал может иметь несколько связанных с ним точек останова. Если логическое выражение принимает значение `true` в какой-то момент моделирования, то в этот момент моделирование останавливается. В этом окне, с помощью кнопки **Browse**, можно выбрать сигнал с точками останова, для которого будет вестись работа.

При формировании новой точки останова, базирующейся на логическом выражении, необходимо произвести следующие действия. В диалоговом окне новой точки останова, в поле **Operator**, необходимо ввести оператор, который будет использоваться для сравнения (`==` — совпадение, `:=` — несовпадение, `>`, `>=`, `<`, `<=`). В поле **Compare** задается значение сигнала, которое вызывает останов. Выпадающий список содержит перечень значений, допустимых для выбранного сигнала. Это может быть или список MVL-9, или Карта символов. Если рассматриваемый сигнал является группой, то можно указать специфический для этой группы тип (`DEC`, `HEX`, `OCT`, `BIN`), в котором будет задаваться значение.

Для работы с точками останова соответствующего типа, необходимо выбрать команду **Break**. В результате появится диалоговое окно, в котором бу-

дут отображены все существующие в данный момент точки останова соответствующего типа.

Для того чтобы добавить новую точку останова, необходимо выделить ту точку останова, перед которой должна быть добавлена новая. Затем воспользоваться клавишей **Insert**. Для активизации, деактивизации, редактирования или удаления точки останова необходимо выделить эту точку, а затем воспользоваться соответственно одной из кнопок **Enable**, **Disable**, **Edit** или **Remove**. Все активные в данный момент точки останова помечены знаком +, а неактивные — знаком -.

Механизм событий

Для того чтобы принудительно определить значение сигнала в конкретный момент времени моделирования, используется механизм событий.

В любой момент времени, в процессе моделирования можно просмотреть список запланированных событий. Для этого необходимо воспользоваться командой **Show Pending Events** (показать отложенные события). В результате будет отображено окно, строки которого содержат описания событий, запланированных к текущему моменту времени, но еще не произошедших. Пример такой строки описания:

```
Obj1.dsum change to 0 at time 10
```

Эта строка означает, что на момент модельного времени 10 запланировано изменение значения сигнала `dsum`, принадлежащего сущности или компоненту `Obj1`. Этому сигналу будет присвоено значение 0.

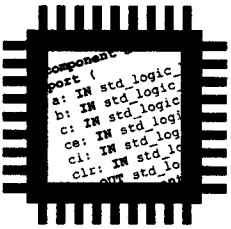
Инструментарий, позволяющий получить дополнительную информацию в ходе моделирования

Использование **Watch Window** (Окно наблюдений). В этом окне отображаются значения выбранных сигналов в текущий момент времени моделирования. Текущее модельное время отображается в верхней части окна. В один момент времени может существовать одно такое окно. Сигналы могут быть перемещены в это окно из **Wave Window**, **List Window** или из вкладки **Hierarchy Tab** окна проекта.

Просмотр свойств сигналов. Для того чтобы просмотреть свойства сигнала, переменной или процесса, может быть использовано **Signal Properties Window** (Окно свойств сигналов). Это окно можно вызвать, щелкнув правой кнопкой мыши по имени объекта в **Wave Window**, **List Window** или на вкладке **Hierarchy Tab** окна проекта. В появившемся меню надо выбрать пункт **Properties**. В этом окне отображается следующая информация об объекте. Для любого объекта отображается его имя, имя файла, в котором он описан и номер строки в файле, в которой выполнено описание. Для сигналов и переменных отображается их тип. Для сигналов отображается ин-

формация о драйверах: указываются имена файлов и номера строк, в которых данному сигналу присваиваются значения. Для них также указываются имена файлов и номера строк, в которых расположены директивы, чувствительные к изменениям сигнала. В один момент времени в этом окне можно просматривать информацию только об одном объекте.

Просмотр предыстории сигналов. Для просмотра предыстории сигналов может быть использована команда **Signal Traceback** (Отслеживание от результата к источнику). В этом случае отображается диалоговое окно, в котором можно выбрать сигнал, предыстория которого будет просматриваться. Затем отображается **Signal Traceback Window** — окно, в котором можно просмотреть предысторию этого сигнала. В этом окне отображается текущее состояние сигнала и сигналы, на базе которых это значение было определено. По умолчанию в окне отображается четыре уровня зависимости сигналов. Если существует более глубокая зависимость по какому-либо из направлений, то оно помечается знаком +. Щелчок по нему позволяет просмотреть еще четыре уровня иерархии.



Глава 7

Проектирование СБИС на языке VHDL в среде Foundation Express

Пакет Xilinx Foundation Express предназначен для проектирования устройств и систем, реализуемых в СБИС фирмы Xilinx классов FPGA и CPLD. В пакете Foundation Express могут быть выполнены все стадии проектирования устройства на СБИС — от разработки описания устройства, выполняемого на языке описания аппаратуры или с помощью рисования схем на базе графических примитивов, до генерации битового файла, на базе которого выполняется прошивка FPGA или настройка CPLD.

Пакет Foundation Express и заложенная в него технология разработки и реализации систем на СБИС ориентированы на проектирование на языках описания аппаратуры высокого уровня — языках VHDL и Verilog. В описаниях, подсказках и диалоговых сообщениях Foundation Express, относящихся к работе с любым из этих языков, часто используется собирательное название — HDL (Hardware Description Language). В нашей книге описание Foundation Express приводится с ориентацией на использование языка VHDL.

Менеджер проектов

Все инструментальные средства пакета Xilinx Foundation Express объединены в интегрированную среду, представленную для пользователя Менеджером проектов (**Project Manager**).

Менеджер проектов выполняет следующие функции:

- автоматически загружает все ресурсы, необходимые для проектирования при открытии проекта;
- проверяет доступность ресурсов, необходимых для проекта, и поддерживает их в актуальном состоянии (`up-to-date`);
- показывает пользователю последовательность действий, соответствующую процессу проектирования;

- позволяет вызывать приложения, необходимые в процессе проектирования;
- позволяет просматривать сообщения о состоянии проекта и об ошибках;
- обеспечивает автоматическую передачу данных между программами, вовлеченными в процесс проектирования;
- предоставляет интерфейс для работы с программами от других производителей;
- предоставляет информацию о состоянии проекта (`design status information`).

Можно выделить следующие основные средства работы с проектом, доступ к которым осуществляется посредством менеджера проектов:

- редактор схем (**Schematic Editor**);
- текстовый редактор (**HDL Editor**);
- редактор диаграмм состояний (**State Diagram Editor**);
- программа моделирования синтезированной схемы на вентильном уровне (**Fast Gate-Level Logic Simulator**);
- программы от сторонних поставщиков (`external third-party programs`), включаемые в среду проектирования Xilinx Foundation Express.

Окно менеджера проектов

Общий вид окна менеджера проектов представлен на рис. 7.1.

Окно менеджера проектов включает три основных области:

- браузер иерархии (**Hierarchy brouser**);
- диаграмма проекта (**Project flowchart**);
- окно сообщений (**Message window**);

и две панели:

- панель инструментов (**Toolbar**);
- полоса статуса (**Status bar**).

Браузер иерархии

Браузер иерархии предназначен для формирования набора ресурсов, т. е. файлов, входящих в состав проекта. Он включает в себя две части — файлы (**files**) и версии (**versions**). На первой из них отображается иерархия файлов, входящих в состав проекта, на второй — информация о реализации проекта на кристалле. Проект может иметь несколько реализаций (версий). На плоскости "Versions" можно выбрать конкретную реализацию, с которой будут выполняться дальнейшие действия.

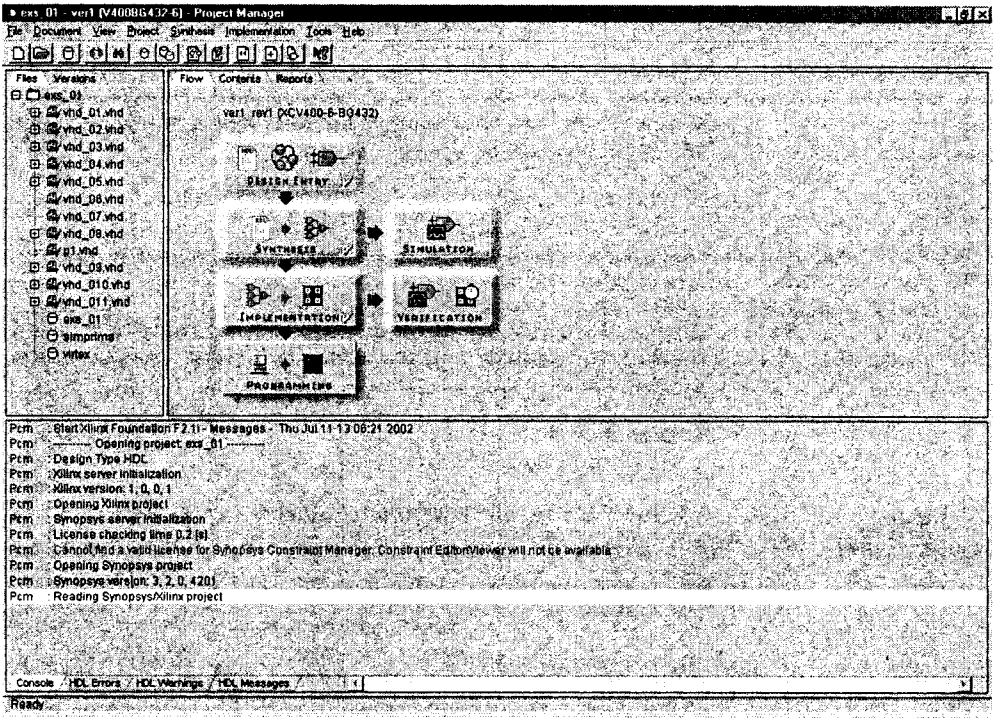


Рис. 7.1. Окно менеджера проектов

Файлы, входящие в состав проекта, отображаются в окне браузера иерархии.

В табл. 7.1 представлены типы файлов, каждому из которых в окне соответствует свой вид пиктограммы.

Таблица 7.1. *Пиктограммы типов файлов в менеджере иерархии*

Пиктограмма	Описание
	Файл описания проекта (всегда располагается на верхнем уровне иерархии)
	Различные текстовые файлы (файл readme.txt, файл пользовательских ограничений)
	HDL-файлы, в которых при анализе не было обнаружено ошибок
	HDL-файлы, содержащие ошибки
	HDL-файлы, которые нуждаются в генерации списка связей

***Таблица 7.1** (окончание)*

Пиктограмма	Описание
	Файлы диаграмм состояний, в которых при анализе не было обнаружено ошибок
	Файлы диаграмм состояний, содержащие ошибки
	Файлы диаграмм состояний, которые нуждаются в генерации списка связей
	Файлы графических представлений схем, которые нуждаются в генерации списков связей. Файлы, для которых сгенерированы динамически обновляемые списки связей, имеют такую же пиктограмму, только знак вопроса заменяется галочкой
	Иерархические макросы (Использование макросов не рассматривается и пиктограмма для них не приводится)
	ABEL-модули (Для них пиктограммы такие же, как и для VHDL-файлов)
	Библиотеки компонентов
	VHDL-библиотеки (им соответствуют такие же пиктограммы как для HDL-файлов)

На вкладке "Versions" браузера иерархии отображается дерево версий реализации. Первоначально, при создании проекта, эта вкладка пуста. Каждый проект может иметь множество версий реализации. Каждая версия имеет свой уникальный номер в рамках проекта. Версии могут отличаться друг от друга стратегией реализации, ограничениями, определяемыми пользователем, типом элементной базы. Все они отображаются на этой вкладке.

В табл. 7.2 представлены группы файлов, отображающиеся на этой вкладке (каждому из них соответствует своя пиктограмма).

***Таблица 7.2.** Пиктограммы типов файлов в менеджере иерархии*

Пиктограмма	Описание
	Представление проекта, находящегося на верхнем уровне иерархии на вкладке **files**
	Версия проекта
	Функциональная, не оптимизированная структура, полученная в результате синтеза, в скобках около названия помечается **functional structure**

Таблица 7.2 (окончание)

Пиктограмма	Описание
	Функциональная, не оптимизированная структура, полученная в результате синтеза, при оптимизации которой возникли замечания, в скобках около названия помечается **functional structure**
	Функциональная, не оптимизированная структура, полученная в результате синтеза, которая должна быть синтезирована вновь, вследствие изменения исходного описания схемы, **functional structure**
	Оптимизированная структура, в скобках около названия помечается **optimized structure**
	Редакция реализации проекта

Для того чтобы добавить или удалить из проекта документ, необходимо воспользоваться пунктами **Add** (Добавить) или **Remove** (Удалить) меню **Document** (Документ).

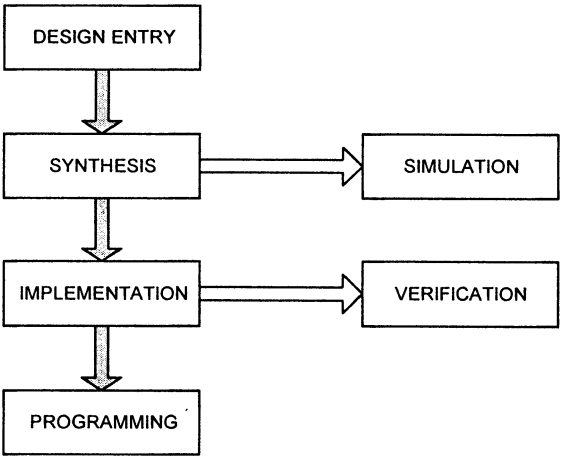
Диаграмма проекта

На диаграмме проекта (рис. 7.2, *а*, *б*) представлены кнопки, соответствующие этапам проектирования. Стрелки между ними указывают естественную последовательность действий, выполняемых при проектировании. Нажатие кнопки приводит к запуску инструментария, необходимого для выполнения выбранного этапа проектирования. Рис. 7.2, *а* соответствует процессу проектирования, если проект содержит HDL-файлы; в противном случае процесс проектирования соответствует рис. 7.2, *б*.

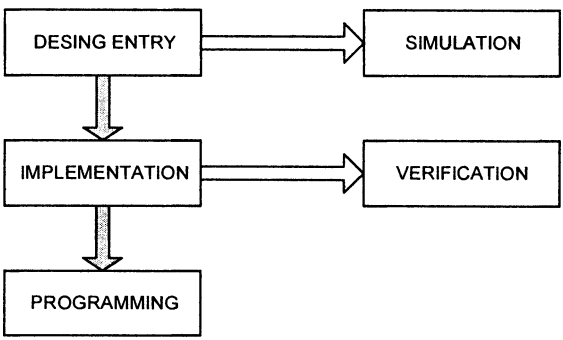
Окно сообщений

В окне сообщений отображаются таблицы (плоскости), содержащие следующие классы сообщений:

- **Console tab** — содержимое **project log** — информация о последовательности действий, выполняемых с проектом, информация о результатах выполнения;
- **HDL errors tab** — сообщения об ошибках, возникающих в ходе анализа HDL-файлов;
- **HDL warnings tab** — предупреждающие сообщения, возникающие в ходе анализа HDL-файлов;
- **HDL message tab** — остальные сообщения.



а



б

Рис. 7.2. Диаграмма проекта

Панель инструментов и полоса статуса

Панель инструментов содержит кнопки, соответствующие наиболее часто употребляемым командам меню.

В полосе статуса отображается информация о состоянии операции, выполняемой в данный момент времени.

Понятие проекта

Проект представляет собой набор (совокупность) файлов, в которых содержится описание проекта. Основные типы файлов, входящих в состав проекта:

- **project documents** (графические описания схемы, файлы HDL описания, файлы диаграмм состояний);
- выходные и промежуточные файлы: списки связей (**netlists**), битовые файлы (**bit stream files**), файлы отчетов и лог-файлы (**report** и **log files**);
- файлы конфигурации.

Каждый проект размещается в своем каталоге, называемом *рабочим каталогом* проекта. Имя этого каталога совпадает с именем проекта (длина — не более 8 символов).

Все файлы, входящие в состав проекта, расположены в рабочем каталоге проекта. Если в проект добавляется уже существующий файл, расположенный в другом каталоге, то менеджер проектов автоматически создает его копию в каталоге проекта. Далее при работе проекта доступна будет эта копия, а не исходный файл.

Для каждого проекта создается *файл описания проекта* — **Project Description File (PDF)**. В нем описывается содержимое и текущий статус проекта. Его имя совпадает с именем проекта и имеет расширение pdf.

Типы проектов

Тип проекта определяет следующее:

- **design flow** (графически представляется диаграммой потока проекта);
- поддерживаемые семейства FPGA;
- используемые системные библиотеки;
- доступные интерфейсы к внешнему программному обеспечению от сторонних производителей;
- используемые форматы списков связей.

Foundation поддерживает тип проектов Foundation Series F2.1i. В этот тип входят два подтипа: HDL и Schematic. Для обоих подтипов поддерживаются следующие семейства Xilinx устройств:

- Virtex;
- Spartan;
- Spartan2;
- SpartanXL;
- XC9500;
- XC9500X (XL/XV);
- XC5200;
- XC4000XLA;
- XC4000E;

- XC4000L;
- XC4000X (EX/XL/XLA/XV);
- XC3000A;
- XC3000L;
- XC3100A;
- XC3100L.

Управление проектом

В меню **File** (Файл) содержатся пункты, соответствующие действиям, которые можно выполнять с проектом:

- **New Project** — создание нового проекта;
- **Open Project** — открытие проекта;
- **Copy Project** — создание копии проекта;
- **Delete Project** — удаление проекта;
- **Archive Project** — создание архивной копии проекта;
- **Restore Project** — восстановление проекта из архива.

Создание нового проекта

В диалоговом окне, открывающемся при создании нового проекта, необходимо указать имя проекта. Оно не должно совпадать с именами библиотек, входящих в состав пакета. Необходимо выбрать также директорий для размещения проекта. Для выбора можно воспользоваться кнопкой **Browse**. Далее необходимо указать тип проекта. (В предлагаемом списке типов присутствует только один тип — F2.1i). Затем выбирается способ описания проекта: в виде графической схемы (**Schematic**) или на языке описания аппаратуры (**HDL**). После этого указываются параметры, которые будут использованы при физической реализации схемы: семейство устройств FPGA (**Family**), конкретное устройство в рамках этого семейства (**Part**) и скорость (**Speed**)

Открытие проекта

При открытии проекта в диалоговом окне необходимо указать имя и местоположение проекта Foundation Express.

Поскольку в один момент времени менеджер проектов может работать только с одним проектом, открытие следующего проекта приводит к автоматическому закрытию предыдущего.

При запуске менеджера проектов автоматически предлагается открыть проект, который был последним в предыдущем сеансе работы.

Копирование проекта

Если возникает необходимость переноса проекта в другое место диска, то копирование каталога проекта средствами файловой системы выполнять не рекомендуется, поскольку в файле проекта указаны абсолютные пути к остальным файлам, и копия, сделанная таким образом, будет неработоспособной.

Копирование проекта необходимо выполнять с помощью специального пункта **Copy Project** меню **Fail** менеджера проектов. При копировании проекта автоматически выполняются следующие действия:

1. Копируется основной файл проекта (**project_name.pdf**).
2. Копируются все файлы, расположенные в рабочем каталоге проекта.
3. Копируются рабочие библиотеки проекта.

Рассмотрим элементы диалогового окна копирования проекта. Поле **Source** (Источник) содержит имя файла проекта (*.pdf), который будет скопирован, по умолчанию там устанавливается имя открытого в данный момент проекта. Для выбора другого проекта можно воспользоваться кнопкой **Browse** (Просмотреть). Поле **Destination** (Место назначения) содержит два подполя: **Name** (Имя) и **Directory** (Папка). В этом поле определяется место, в которое будет скопирован проект.

Удаление проекта

В диалоговом окне, открывающемся при удалении проекта, необходимо указать имя и местоположение проекта. Не рекомендуется удалять каталоги проектов средствами файловой системы. Поскольку информация обо всех проектах хранится в менеджере проектов, эти действия могут привести к нестабильности его работы.

Удаление информации о реализации

При работе с проектом создаются файлы, содержащие промежуточную информацию о моделировании, синтезе и реализации. Если возникает необходимость выполнить моделирование и синтез сначала (с исходного файла описания модели), то необходимо удалить промежуточную информацию, для чего можно использовать пункт **Clear implementation data** (Удалить информацию о реализации) из меню **Project** (Проект).

Сохранение проекта в виде архива

При создании архива проекта запускается мастер архивации, при работе с которым последовательно используется несколько диалоговых окон. В результате, проект и, возможно, дополнительная информация, сохраняется в виде zip-файла. В первом диалоговом окне указывается местоположение

файла архива (по умолчанию имя файла совпадает с названием проекта). В поле **Comment** (Комментарий) можно указать текстовый комментарий к проекту, которой будет сохранен в файле Readme.txt, добавляемом в архивируемый проект. В поле **Password** (Пароль) можно указать пароль, который будет запрашиваться при разархивации. В этом окне так же указывается степень сжатия. Переход к следующему диалоговому окну осуществляется по кнопке **Next** (Далее). Во втором диалоговом окне указывается, какие файлы проекта войдут в состав архива.

Файлы, входящие в состав проекта, разделяются на следующие группы:

- файлы исходного описания проекта (**design source files**);
- файлы синтеза (**synthesis files**);
- файлы реализации (**implementation files**, содержимое папки XPROJ, входящей в состав проекта).

Эти группы файлов могут быть добавлены в архив по отдельности.

В третьем диалоговом окне могут быть указаны имена дополнительных файлов и библиотек, которые также будут включены в архив. Для добавления файлов можно воспользоваться кнопкой **Add Files**, для добавления библиотек — **Add Libraries**. Может оказаться целесообразным добавлять в архив и системные библиотеки, и файлы глобальной настройки системы, такие как Autoexec.bat и Config.sys.

Если файл был выбран случайно, его можно удалить из списка. Для этого его необходимо выделить мышью и воспользоваться кнопкой **Remove Files** (Удалить файлы). После нажатия кнопки **Start** (Начать), расположенной в этом окне, выполняется создание архива. При этом открывается диалоговое окно, в котором отображается процесс создания проекта. После того как архивация закончена, окно можно закрыть кнопкой **Close**.

Восстановление проекта из архива

При восстановлении проекта из архива в начальном диалоговом окне необходимо выбрать имя файла архива, после чего запускается мастер разархивации. В первом диалоговом окне мастера определяется место, куда будет распакован проект (оно указывается в поле **Destination** (Место назначения)). В поле **Comment** (Комментарий) отображается содержимое файла Readme.txt, если он присутствовал в архиве. Если при архивации проекта был задан пароль, то его необходимо указать в поле Password. Если флажок **Open Restored Project** (Открыть восстановленный проект) установлен, то после разархивации проект будет сразу открыт в менеджере проектов. Для перехода к следующему окну мастера надо нажать кнопку **Next**.

Если в проект были добавлены дополнительные файлы, то их список с указанием исходного местоположения будет отображен в диалоговом окне **User**

Files (Пользовательские файлы). Если необходимо изменить место размещения этих файлов после разархивации, то можно воспользоваться кнопкой **Select all** (Выбрать все) для изменения местоположения всех файлов сразу или **Change directory** (Изменить каталог) для изменения местоположения выбранного файла. После нажатия кнопки **Next** запускается собственно процесс разархивации. При перезаписи файла из архива поверх уже существующего выдается соответствующий запрос подтверждения.

Изменение свойств физической реализации проекта

В процессе работы с проектом может возникнуть необходимость изменить тип проекта или элементную базу, используемую для его реализации. Для этого необходимо воспользоваться пунктом **Project type** (Тип проекта) из меню **File** (Файл). В открывающемся диалоговом окне можно изменить все необходимые параметры. Выбор другого устройства в рамках одного семейства, как правило, определяется изменением ограничений, связанных с количеством вентилей (или LUT) или количеством входов и выходов, необходимых для реализации схемы. Переход к другому семейству FPGA связан с подключением других библиотек и может очень существенно повлиять на характеристики реализуемой схемы.

Работа с библиотеками

Библиотека используется как хранилище компонентов, используемых в проектах. Поддерживается два основных типа библиотек:

- системные библиотеки, которые содержат набор объектов, соответствующих конкретному типу проектов;
- библиотеки пользователя, содержимое которых определяется пользователем.

В каждом проекте используется совокупность библиотек, индивидуальная для данного проекта. Как правило, в каждый проект включаются следующие наборы библиотек:

- совокупность *системных библиотек*, определяемых типом проекта. Эти библиотеки автоматически добавляются в проект на этапе его создания. В дальнейшем они могут быть вручную удалены из него и вновь добавлены. Если на момент очередного открытия проекта в нем отсутствуют библиотеки, соответствующие его типу, они автоматически добавляются в него;
- *рабочая библиотека проекта*, в которую добавляются все макросы пользователя, создаваемые при разработке проекта. Эта библиотека также гене-

рируется автоматически при создании проекта, ее нельзя удалять из проекта, поскольку это приведет к нарушению его целостности;

- *дополнительные системные и пользовательские библиотеки*, добавляемые пользователем вручную при необходимости.

В пакет Xilinx Foundation Express входит инструмент **Library Manager** (Менеджер библиотек), который позволяет управлять библиотеками.

Выбор библиотек, входящих в состав проекта

Для добавления или удаления библиотек проекта необходимо выбрать пункт **Project Libraries** (Библиотеки проекта) в меню **File** (Файл). В результате будет открыто диалоговое окно **Project libraries**. В нем отображается список библиотек, входящих в состав пакета — attach libraries. Из этого списка можно выбирать библиотеки и переносить их в состав своего проекта. Дополнительные библиотеки, включенные в состав проекта, отображаются в списке **Project libraries**.

В этом же диалоговом окне можно запустить менеджер проектов с помощью кнопки **Lib manager**. Для того чтобы внести изменения в состав библиотек пакета (подключить дополнительные библиотеки), необходимо нажать кнопку **Attach** (Присоединить) диалогового окна. Библиотека считается добавленной, если ее название появляется в списке библиотек проекта (**attach libraries**). После нажатия кнопки **Attach** открывается диалог **attach library**. В списке **Drivers** можно выбрать устройство, на котором будет осуществлен поиск новых библиотек. Поиск начинается при нажатии кнопки **Update** (Обновить). Список найденных библиотек отображается в окне **Libraries Found**. При нажатии кнопки **OK** библиотеки, выделенные в списке найденных, добавляются в состав пакета Foundation.

Изменение порядка просмотра библиотек

Браузер иерархии отображает список файлов, входящих в состав проекта, в том числе файлов библиотек в виде дерева (возле файлов библиотек слева расположена пиктограмма в виде бочки, см. табл. 7.1).

При поиске используемых элементов, библиотеки просматриваются в порядке, определяемом этим списком, сверху вниз.

Порядок просмотра влияет на время и результаты поиска элементов. Библиотеки, содержащие большинство используемых в проекте элементов, лучше использовать в начале списка. Если элементы с одинаковым именем располагаются в нескольких библиотеках, то использован будет элемент, входящий в состав библиотеки, расположенной в списке первой.

Для изменения местоположения библиотеки в списке нужно выделить ее мышью и, при нажатой левой кнопке мыши, перетащить на нужное место.

Получение информации о библиотеке

Для получения информации о библиотеке, необходимо выбрать мышью библиотеку в браузере иерархии, после этого выбрать пункт **Info** (информация) меню **Document** (Документ), в результате чего откроется диалоговое окно `Library info` (Информация о библиотеке), в котором указано имя библиотеки, ее тип, цифровой идентификатор и местоположение в файловой системе.

Просмотр содержимого библиотеки

Для того чтобы просмотреть содержимое библиотеки, выбранной в браузере иерархии, надо воспользоваться пунктом **Open** (Открыть) меню **Document** (Документ). В результате будет отображен список компонентов, входящих в состав библиотеки. Информация представляется в виде таблицы, для каждого компонента указывается: логическое имя, физическое имя, комментарий (описание элемента), тип объекта, атрибуты и имя библиотеки, к которой принадлежит компонент.

Запуск менеджера библиотек

Для того чтобы запустить менеджер библиотек, надо воспользоваться пунктом **Schematic Symbol Library Manager** (Менеджер библиотек символов схем) из **Utilities** (Утилиты) подменю в меню **Tools** (Инструменты).

VHDL-библиотеки

VHDL-библиотеки используются только при синтезе моделей, описанных на языке VHDL. Сам менеджер проектов с этими библиотеками не работает. Если выполняется синтез модели, описанной на VHDL, то по умолчанию используется VHDL-библиотека WORK.

Пользователь может создавать собственные библиотеки, которые могут содержать фрагменты кода или пакеты. Для того чтобы использовать библиотеку пользователя в составе проекта, ее необходимо добавить в список файлов проекта. Для этого можно воспользоваться пунктом всплывающего меню **Add**. Для вызова этого меню надо щелкнуть правой кнопкой мыши по списку файлов в браузере иерархии.

Создание исходного описания проектируемой системы

В пакете Xilinx Foundation Express исходное описание проектируемого устройства может задаваться в виде описания на языках высокого уровня VHDL или Verilog, в графическом виде — в виде схемы, а также в виде диаграммы состояний синтезируемого автомата.

Для создания исходного описания устройства используются редактор текстовых описаний **HDL editor** (VHDL или Verilog описание), редактор диаграмм состояний **State diagram editor** и редактор схем (**Schematic editor**).

Редактор описаний на языке высокого уровня HDL editor

Редактор описаний на языке высокого уровня (**HDL editor**) является текстовым редактором, ориентированным на тексты, написанные на языках VHDL, Verilog и ABEL. Этот редактор предоставляет интерфейс для обращения к инструментарию синтеза, позволяющему генерировать списки связей на уровне вентилей. Код на языке высокого уровня HDL может представлять всю схему или некоторые элементы графического представления схемы.

Файлы, с которыми работает редактор **HDL editor**, имеют следующие расширения: abl, abv (для ABEL), vhd для VHDL, v для Verilog.

Редактор диаграмм состояний State diagram editor

Редактор диаграмм состояний (**State diagram editor**) предназначен для описания диаграмм состояния автоматов. Он позволяет сформировать описание диаграмм состояний и преобразовать его в код на выбранном HDL языке.

Файлы, с которыми работает этот редактор, имеют расширение asf.

Редактор схем Schematic editor

Редактор схем (**Schematic editor**) позволяет работать с иерархическими графическими представлениями схем, которые могут быть расположены на нескольких листах.

Файлы, с которыми работает этот редактор, имеют расширение sch.

Импорт готовых файлов исходного описания проектируемого устройства

Кроме файлов исходного описания, созданных средствами самого пакета Xilinx Foundation Express, в проект могут быть добавлены также файлы, полученные с использованием других инструментальных пакетов, это — файлы, написанные на языках VHDL, Verilog, ABEL, а также списки связей в форматах XNF или EDIF.

Для того чтобы добавить файл в проект, необходимо в браузере иерархии на вкладке **Files** (Файлы) щелкнуть правой кнопкой мыши, в появившемся всплывающем меню воспользоваться пунктом **Add** (Добавить), затем в диалоговом окне необходимо выбрать файл, который должен быть добавлен в проект. Этот файл будет скопирован в рабочий каталог проекта.

Подготовка исходного описания проекта к синтезу

Для выполнения синтеза схемы все файлы, входящие в ее описание, должны быть преобразованы в списки связей. Поддерживаются следующие форматы списков связей: для графических представлений схем используются EDIF форматы, для HDL-файлов формируются XNF списки, за исключением семейства СБИС FPGA Virtex, для которого используется EDIF формат (как входной для инструментария Place and Route).

Проверка синтаксиса, определение иерархии связей между описаниями и, при необходимости, синтез списков связей осуществляется инструментальными программами анализа проекта. Все файлы, добавляемые в состав проекта, анализируются автоматически.

В процессе анализа не только проверяется синтаксис, но и устанавливается иерархия связей между описаниями.

Если в ходе работы с проектом эти файлы модифицируются, их анализ необходимо выполнить заново. Для того чтобы система проанализировала конкретный файл, необходимо щелкнуть правой кнопкой мыши по его имени в браузере иерархии и в появившемся меню выбрать пункт **Analysis**.

Аналогично выполняется **Force Analysis**. Для того чтобы воздействовать на все файлы проекта из этого меню, нужно выбрать пункт **Update Project**. Результат анализа отображается в виде пиктограммы рядом с именем файла в браузере иерархии. Красная пометка указывает на наличие ошибок или предупреждений.

Синтез проекта

После того как исходные файлы проекта успешно проанализированы (пиктограммы, расположенные рядом с именами файлов, в браузере иерархии имеют пометку зеленого цвета), можно переходить к синтезу проекта.

Процесс синтеза

Для того чтобы начать процесс синтеза, необходимо использовать пункт **Synthesize** из меню **Synthesis** или воспользоваться кнопкой **Synthesis** в окне диаграммы проекта.

В начале процесса будет открыто диалоговое окно **Synthesis/Implementation settings**, рис. 7.3. В нем можно определить параметры, характеризующие конкретную версию реализации: тип FPGA, на базе которой будет выполнена реализация; тактовая частота; коэффициент скорости.

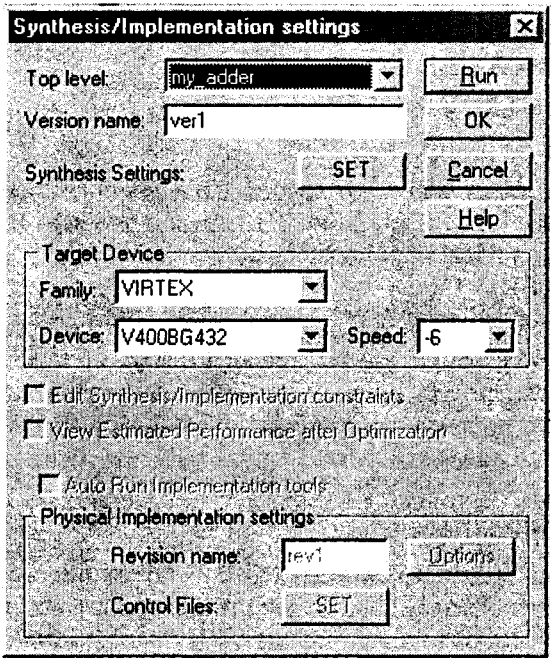


Рис. 7.3. Диалоговое окно **Synthesis/Implementation settings**

Это диалоговое окно содержит следующие поля:

- Поле **Top level** позволяет определить название модуля проекта, находящегося на верхнем уровне иерархии.
- Поле **Version name** — имя вновь создаваемой версии реализации;
- Поля **Family** и **Device** позволяют определить семейство устройств, а в нем конкретное устройство, на базе которого будет выполнена реализация;
- Поле **Speed** позволяет указать коэффициент скорости синтеза. Этот коэффициент может принимать значения −4, −5, −6. Чем меньше его значение, тем большее время затрачивается на синтез, но тем лучше оптимизируется результирующая схема. Таким образом, наилучшей оптимизации можно достигнуть при значении коэффициента −6. Если установлен флажок **Edit Synthesis/Implementation constraints** (редактирование ограничений синтеза/реализации), то перед началом собственно моделирования будет сделана пауза, позволяющая пользователю отредактировать файл ограничений схемы. Если установлен флажок **View Esti-**

mated Performance after Optimization (отобразить производительность после оптимизации), то результаты оптимизации будут отображены;

- Кнопка **SET** позволяет открыть диалоговое окно **Synthesis settings**, в котором можно настроить параметры синтеза.
  В этом окне можно также настроить параметры, связанные с конкретной физической реализацией;
- В поле **Revision Name** необходимо указать имя создаваемой редакции;
- Кнопка **Options** позволяет открыть диалоговое окно **Options**, в котором можно настроить конкретные параметры физической реализации.

После того, как необходимые настройки выполнены, можно нажать кнопку **OK** для запуска синтеза, или кнопку **RUN** — для запуска синтеза, а затем реализации.

Если в диалоговом окне были определены параметры только для версии, то в результате будет создана новая версия, если же были определены параметры и для редакции, то будет создана новая версия, а в ней — новая редакция.

Диалоговое окно **Synthesis settings** состоит из двух вкладок: **Synthesis settings** и **Implementation control files**.

Рассмотрим вкладку **Synthesis settings**. В зависимости от положения переключателя **Optimize for area/Speed** оптимизация выполняется по занимаемой синтезируемым устройством площади или скорости. Переключатель **Effort level High/Low** позволяет сделать выбор между существенными временными затратами на компиляцию (при этом будет получен хорошо оптимизированный результат) и быстрой компиляцией (при этом используется ускоренный метод размещения элементов и связей, что приводит к ухудшению характеристик синтезированной схемы и по занимаемой площади, и по быстродействию). Поле **Target clock frequency** позволяет определить базовую тактовую частоту работы выбранной FPGA. Если флажок **Export timing constraints** установлен, то временные ограничения будут экспортированы. Установка флажка **Insert I/O pads** указывает на то, что в ходе синтеза должны быть добавлены входные/выходные выводы. Если флажок **Preserve design hierarchy** установлен, то выходной список связей будет иерархическим, при условии, что исходное описание модели устройства имело иерархическую структуру.

Рассмотрим вкладку **Implementation control files**. Поле **Use constraints file from** позволяет указать расположение файла, содержащего описание ограничений. Выбранный в этом поле файл будет скопирован в папку, соответствующую текущей редакции. Поле **Copy guide file from** позволяет определить, откуда должен быть скопирован **guide file**. Поле **Copy floorplan files from** позволяет указать, откуда будут скопированы **floorplan** файлы (файлы размещения на кристалле). Если установлен флажок **Enable guided MAP and PAR**, **guide**-файл будет использован в текущей редакции. Если установлен флажок **Enable floorplaning**, то **floorplan**-файлы будут использованы в текущей редакции.

Редактирование ограничений

Ограничения — это специфические требования пользователя к синтезу и оптимизации. Можно выделить следующие группы ограничений:

- **Clocks** — ограничения на сигналы тактирования;
- **Paths** — ограничения на временные характеристики путей прохождения сигналов от входов к выходам;
- **Ports** — ограничения на геометрическое расположение входных и выходных портов;
- **Modules** — характеристики используемых модулей;
- **Xilinx Options** — выбор опций синтеза.

Для того чтобы отредактировать ограничения, связанные с конкретной версией физической реализации, необходимо в браузере иерархии перейти на вкладку **Versions** и щелкнуть правой кнопкой мыши на нужной версии, в появившемся всплывающем меню выбрать пункт **Edit Constraints.** В результате появится диалоговое окно редактора ограничений (`Constrains Editor`).

Этот редактор может работать со следующими типами файлов:

- **UCF (user constraints file)** — такие файлы содержат логические ограничения, записанные пользователем;
- **NGD (native generic database)** — эти файлы являются входными для приложения, создающего базу данных, соответствующую физической реализации устройства.

Редактор ограничений создает выходной файл UCF. Этот файл является входным для приложения **NGDBuild**, которое использует его наряду со списками связей, полученными на базе файлов описания устройства. NGD-файл может быть прочитан приложением **MAP**, которое на его базе генерирует NCD-файл (файл в формате внутренней базы данных для описания схемы) и PCF-файл (файл физического описания ограничений). Инструменты **Implementation** используют эти файлы для создания файла битового описания схемы (bit stream file).

Общий вид окна редактора ограничений представлен на рис. 7.4.

Основное окно редактора ограничений включает следующие компоненты:

- полосу заголовка;
- меню;
- панель инструментов;
- полосу статуса;
- окно, в котором отображаются ограничения;
- окно результатов.

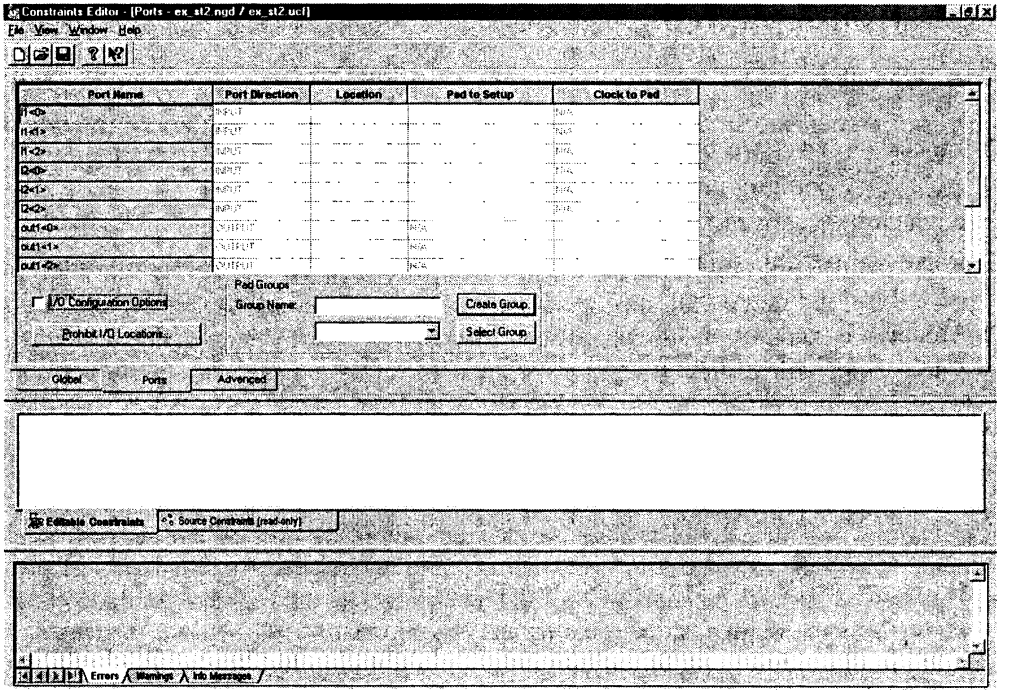


Рис. 7.4. Окно редактора ограничений

При загрузке файла в редактор ограничений в основном окне становятся доступными 3 вкладки:

- **Global**;
- **Ports**;
- **Advanced**.

В сочетании с окном ограничений их можно использовать для редактирования ограничений. Окно результатов расположено в нижней части окна редактора. В нем отображаются сообщения об ошибках и другая информация. В полосе статуса отображается информация об исполняемой команде.

Для сохранения информации, внесенной с использованием конкретной вкладки, перед переходом на другую вкладку, необходимо воспользоваться пунктом **Save** меню **File**.

Вкладка Global

На вкладке **Global** отображается информация обо всех присутствующих в схеме сигналах тактирования. Для каждого сигнала могут быть заданы:

- **Period** (период);
- **Pad to setup** (**Pad to setup patch time** — это максимальное время между моментом, ког а этот сигнал от входа кристалла достигает входа элемента, на

который подается сигнал тактирования, и моментом прихода на этот элемент активного фронта тактового сигнала);

- **Clock to pad** (это максимальное время, проходящее между моментом, когда измененное значение сигнала на выходе элемента, на который подается сигнал тактирования, достигает выхода кристалла и временем прихода следующего активного фронта сигнала тактирования).

Эта информация на экране представляется в виде таблицы.

Кроме того, имеется поле, в которое можно ввести ограничение **Pad to pad** — максимальное время, проходящее с момента поступления на вход кристалла входных сигналов, до момента, когда на его выходе появятся вызванные ими выходные значения. Это время имеет лишь косвенное отношение к характеристикам сигнала тактирования.

Ограничения на период тактового сигнала задаются следующим образом. Двойным щелчком мыши в ячейке таблицы, соответствующей выбранному сигналу, открывается диалоговое окно **Clock Period**. Оно включает в себя следующие поля. Поле **Timespec** позволяет определить имя временной спецификации для сигнала, по умолчанию оно образуется из имени сигнала добавлением к нему префикса **ts_**. В поле **Clock signal Definition** можно указать тип длительности периода сигнала тактирования. Он может задаваться как абсолютно, так и на базе длительности периодов других сигналов тактирования. Если задается абсолютное значение времени, то, в поле напротив, необходимо выбрать из списка единицу измерения, а в поле **Start High** или **Start Low** указать начальное значение тактового сигнала в периоде. Если указано относительное задание, то в списке **Reference time spec** необходимо выбрать название сигнала, на базе которого будут определяться характеристики данного, а в поле **Multiple by** (Умножить на) или **Divide by** (Разделить на) — ввести соответствующий коэффициент.

Открытие диалогового окна **Pad to setup** аналогично **Clock period**. В нем необходимо выбрать подходящий модуль из списка **Units pull-down**.

Диалоговое окно **Clock to pad,** при открытии его из этой вкладки, содержит следующие элементы: пункт **Time requirements** (Временные требования), который позволяет определить (**value representing the clock-to-out of the synchronous element plus the propagation time to the pad**); пункт **Units** — единица измерения времени.

Диалоговое окно **Pad to pad** содержит следующие элементы: **Time spec name** (имя временной спецификации), **Time** (максимальное значение задержки), **Units** (единица измерения).

Вкладка Ports

Информация на вкладке **Ports** (Порты) также представлена в виде таблицы. Каждому порту, определенному пользователем в исходном описании, соот-

ветствует строка в таблице. Для каждого порта отображается его имя, направление, местоположение, **pad to setup-** и **clock to pad**-ограничения. Редактирование этих параметров осуществляется так же, как и на вкладке **Global**.

Диалоговое окно **Location** позволяет ввести идентификатор вывода устройства. В нижней части окна расположены кнопки, позволяющие сконфигурировать I/O порты, задать ограничения на местоположение, задать поле для создания и выбора групп сигналов. Конфигурирования I/O портов и задание ограничений на местоположение осуществляется с использованием PAR- и FPGA-редакторов.

При конфигурировании I/O портов в диалоговом окне можно определить следующие параметры:

- **FAST/SLOW** позволяет определить скорость передачи сигнала;
- **PULLUP/PULLDOWN** позволяет определить тип резистора, связанного с этим контактом;
- **DRIVE** — определяет электрические характеристики сигнала в миллиамперах;
- **IOSTANDARD** (только для СБИС FPGA семейства Virtex) — стандарт порта.

Вкладка Advanced

На вкладке **Advanced** предоставляются возможности определения ограничений (тех же, что и на двух предыдущих вкладках) для групп элементов. На этой вкладке расположены две области: **Grouping** и **Timing Constraints**. В первой из этих областей расположены кнопки **Create**, позволяющие создавать группы элементов по различным признакам: возможна группировка элементов, связанных одной линией связи, группировка по именам элементов (instance name), по имени выходной цепи. Во второй области расположены кнопки **Specify**, позволяющие определить наборы ограничений для групп.

Организация работы с окном ограничений

Для просмотра и редактирования ограничений необходимо выбрать **Editable constraints** в нижней части окна ограничений. Для редактирования выбранного ограничения необходимо дважды щелкнуть мышью по выбранному ограничению. Для того чтобы просмотреть ограничения, которые редактировать нельзя, необходимо выбрать **Source constraints** в нижней части окна ограничений. Если в области окна ограничений щелкнуть правой кнопкой мыши, то появится всплывающее меню, содержащее следующие пункты:

- **Allow Docking** — позволяет перемещать окно по всему экрану;
- **Hide** — скрывает окно; для того чтобы вновь отобразить его, — необходимо воспользоваться пунктом **Constraints window** меню окна;

- **Disable** — отключение выбранного ограничения (в начале строки, соответствующей отключенному ограничению ставится символ #). Если меню было вызвано на ограничении, которое уже отключено, то этот пункт заменяется на **Enable** — включить;
- **Delete** — удаляет выбранное ограничение.

Моделирование и проверка функционирования проекта

Моделирование работы проекта осуществляется с помощью программы **Logic Simulator**. Моделирование позволяет проверить правильность работы модели как на функциональном уровне, так и на уровне временных характеристик. Результаты моделирования отображаются в окне программы моделирования в виде временных диаграмм.

Загрузка описания проекта в программу моделирования Logic Simulator

При вызове программы моделирования в нее загружается описание модели в виде линейных или иерархических списков связей в форматах EDIF. ALDEC или XNF. Формат XNF при загрузке автоматически преобразуется в формат EDIF. Если загружаемый список связей имел иерархическую структуру, она сохраняется.

Внимание

Если после запуска программы моделирования Logic Simulator, в описание проекта были внесены изменения, списки связей необходимо загрузить в него заново, вручную. Для этого может быть использован пункт **Load Netlist** (Загрузить список связей) меню **File** (Файл).

Если при загрузке списка связей в нем оказывается элемент, который не может быть сопоставлен **Logic Simulator** ни с одним из компонентов, входящих в библиотеки проекта, то генерируется сообщение об ошибке. Однако эта ошибка не является фатальной. Вместо нераспознанного компонента программа моделирования генерирует так называемую "пустую модель", выходные сигналы которой всегда находятся в Z-состоянии.

Во время загрузки модели в **Logic Simulator** может возникнуть три основные группы ошибок, сообщения о которых выдаются пользователю:

- *критические ошибки.* Моделирование схемы, список связей которой содержит такие ошибки, невозможно (сообщения о них выделяются красным цветом);

- *некритические ошибки.* Моделирование схемы, содержащей такие ошибки, возможно, но результат может оказаться непредсказуемым (сообщения о них выделяются синим цветом);
- *информационные сообщения.* Моделирование схемы может быть выполнено без проблем (текст сообщения — черного цвета).

Все сообщения, генерируемые при загрузке списка связей, сохраняются в файле Netlist.log и могут быть просмотрены с помощью пункта **View Netlist Log** меню **Tools** (Инструменты).

Основное окно программы моделирования

Рассмотрим элементы, которые содержит основное окно программы моделирования. В верхней части окна располагается основное меню и основная панель инструментов. В нижней части экрана расположена строка статуса и двоичный счетчик. В строке статуса выводятся сообщения программы моделирования, они отображаются так же в окне сообщений менеджера проектов. Шестнадцатиразрядный двоичный счетчик может быть использован в качестве источника входных сигналов (**stimulator**). Состояние битов счетчика отображается кружками: зелеными, если бит установлен в 0, и красным, если бит установлен в 1. На рис. 7.5. представлен общий вид окна программы моделирования.

В средней части окна программы моделирования расположены окна временных диаграмм. Одновременно может существовать несколько таких окон. В них отображаются одни и те же сигналы, однако, масштаб и конкретная область временной диаграммы могут быть различными. В верхней части окна расположены панели инструментов. Основная часть окна разделена на две вертикальные части. В левой части расположена постоянная информация о сигналах, а в правой — временные диаграммы. Постоянная информация о сигналах разделена на следующие колонки:

- *тип сигнала*:
  - I — входной сигнал;
  - o — выходной сигнал;
  - b — двунаправленный сигнал;
  - *l* — метка;
  - B — шина;
  - V — Vss;
  - G — Gnd;
  - R — эталонное напряжение (`Reference voltage`);
  - H — высокий уровень (`high voltage`);
  - G — общий сброс (`global reset`).

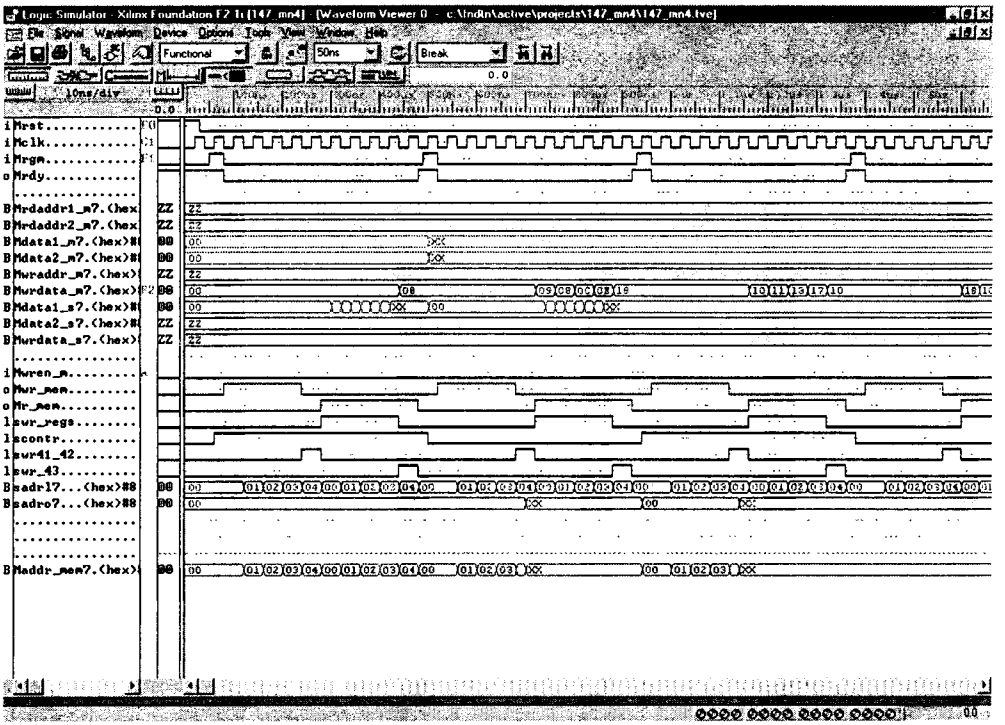


Рис 7.5. Окно программы моделирования

- *имя сигнала*. Если это шина и включен режим отображения шины, то в скобках указывается тип отображения: бинарное, восьмеричное, десятичное, шестнадцатеричное. Если режим отображения шины не включен, то старший разряд шины помечается символом M, а младший — L;
- *источники входных значений* сигнала (stimulators). В этой колонке отображаются состояния источников входных значений сигнала, ассоциированных с сигналами. Цвета, которыми отображаются символы, означают следующее:
  - красный — источник находится в режиме `Override`;
  - черный — источник находится в режиме `Chip control`;
  - серый — источник отключен.

В правой части окна информация представлена в виде следующих колонок:

- **State** — состояние сигнала в момент времени, отмеченный на диаграмме вертикальной синей чертой;
- **Tag** — `tag signal conditions`;
- Колонка собственно временной диаграммы.

Панель инструментов окна временной диаграммы содержит кнопки, представленные на рис. 7.6.

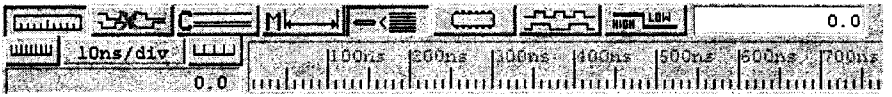


Рис. 7.6. Панель инструментов окна временной диаграммы

- кнопка включения/выключения линейки;
- кнопка удаления всех временных диаграмм;
- кнопка включения/выключения отображения комментариев;
- кнопка включения/выключения измерений; (`Measurements On/Off`) эта кнопка позволяет выполнять и отображать точные измерения интервалов между изменениями состояний/значений сигналов, независимо от включенного масштаба отображения;
- кнопка переключения режима просмотра шин: значение каждого сигнала отдельно/значение всей шины в целом;
- кнопка выбора компонентов — позволяет выбрать сигналы и выводы элементов, состояния которых будут отслеживаться на временных диаграммах;
- кнопка выбора источника стимулирующих воздействий, `stimulators` — позволяет назначить `stimulator` или вектор тестирования конкретному сигналу.

Выбор сигналов, состояния которых будут отображаться программой моделирования

Имена сигналов и выводов компонентов схемы, состояния которых необходимо просматривать, можно определить в программе моделирования с помощью кнопки **Select component** (Выбрать компонент) или воспользовавшись пунктом **Add signals** (добавить сигналы) меню **Signal** (Сигнал).

В результате появится диалоговое окно **Component selection for waveform viewer** (Выбор компонентов для просмотра в окне временной диаграммы). Это диалоговое окно состоит из трех частей:

- **signal selection** — имена сигналов и выводов компонентов, которые могут быть выбраны;
- **chip selection** — компоненты, доступные на выбранном уровне иерархии;
- **scan hierarchy** — иерархическая структура модели.

Для того чтобы выбрать конкретный сигнал, необходимо выполнить следующие действия. Если модель имеет иерархическую структуру, нужно в

scan hierarchy выбрать нужный уровень иерархии (один из уровней, на котором этот сигнал доступен для просмотра), затем выбрать сигнал в **signal selection** и нажать кнопку **Add**.

Шина при моделировании может рассматриваться как набор составляющих ее сигналов или как единый объект. Для выбора формы представления можно воспользоваться опцией **Combine** из **Signal |Bus** меню.

Для выбора вывода элемента, необходимо выполнить следующие действия. Если модель имеет иерархическую структуру, надо выбрать нужный уровень в **scan hierarchy**, дважды щелкнуть мышью по выбранному имени компонента в **chip selection**; в результате чего в **signal selection** будут отображены выводы этого компонента. Для того чтобы состояние этого вывода отображалось в программе моделирования, необходимо дважды щелкнуть по его имени мышью или же, выбрав его мышью, нажать кнопку **Add**.

В области **Signal selection** все, выбранные для просмотра, сигналы помечаются пиктограммой в виде галочки. Если шина рассматривается как единый объект, то ее имя отображается жирным шрифтом.

Для просмотра результатов моделирования рекомендуется выбирать *имена сигналов*, а не выводы элементов. Это связано с тем, что имена выводов элементов состоят из имени элемента (имени ссылки) и собственно имени вывода, а имя элемента в ходе работы с рисунком схемы может измениться, что часто и случается.

Для того чтобы исключить сигнал из числа просматриваемых в окне временных диаграмм, достаточно мышью вытащить его имя за пределы окна.

Программа моделирования позволяет проследить пути соединения сигналов через все уровни иерархии. Для того чтобы проследить конкретный сигнал, необходимо выполнить следующие действия. Выделить этот сигнал щелчком мыши. В меню **Signal** (Сигнал) выбрать подпункт **Connections** (Соединения), в результате чего будет отображено диалоговое окно соединений. В левой части этого окна расположена область сигналов и выводов (**connected signals and pins**), в правой части окна, напротив имен сигналов, указаны их состояния (**node/conv/model/stim**). Эта информация сгруппирована в соответствующие колонки:

- **Node** (Узел) отображает результирующий сигнал в узле;
- **Conv** отображает, как модель интерпретирует данный сигнал (только для входных сигналов);
- **Model** — отображает то, что модель выдает на выход (только для выходных сигналов);
- **Stim** — показывает наличие источника значений входного сигнала, определенного вручную, стимулирующего воздействия (внешнего относительно модели).

В нижней части окна расположены управляющие кнопки:

- **Close** — закрыть окно;
- **Connections** — указать соединения для выбранного сигнала;
- **Hierarchy** — иерархия;
- **States** — используется, если для сигналов в правой части окна не отображается информация о состояниях;
- **Select** — позволяет выбрать сигнал.

Информацию о выбранном множестве сигналов в файле можно сохранить, воспользовавшись пунктом **Signal set** (Множество сигналов) меню **Signal**. Загрузка этой информации в модель в дальнейшем осуществляется пунктом **Load Waveform** (Загрузить временную диаграмму) из меню **File** (Файл); при этом необходимо выбирать ASC-формат.

Определение значений сигналов и работа с результатами моделирования

На различных стадиях процесса проектирования для моделирования могут использоваться временные диаграммы сигналов (**signal waveforms**) или тестовые векторы (**test vectors**).

Signal waveforms представляют собой горизонтальные временные диаграммы, которые начинаются в момент времени T0 и заканчиваются в момент времени T*n*.

Test vectors представляют собой логические состояния всех сигналов в конкретный момент времени (вертикальный срез значений всех линий сигналов). Сигнал, который расположен в верхней части временной диаграммы, записывается в левой части вектора:

- значения входных сигналов: высокий уровень — 1, низкий уровень — 0;
- значения выходных сигналов: высокий уровень — H, низкий уровень — L.

Временные диаграммы удобно использовать в тех случаях, когда необходимо рассмотреть поведение сигнала во временной области, в течение некоторого интервала времени.

Тестовые векторы позволяют определить соотношения между сигналами в конкретные моменты времени.

Foundation Express предлагает пять способов генерации временных диаграмм для сигналов, стимулирующих входные воздействия.

- с помощью клавиатуры. Клавиши <A>—<Z> могут быть назначены линиям сигналов. С их помощью можно управлять значениями сигналов непосредственно в ходе моделирования;

- с помощью прямых выходов бинарного счетчика (**Bc**). Эти 16 желтых ламп представляют выходы бинарного счетчика. Они могут использоваться для задания значений входных сигналов (**basic design stimulus**) и для управления сигналом тактирования;
- с помощью инверсных выходов бинарного счетчика (**NBc**), их использование аналогично использованию прямых выходов;
- может использоваться описание формирования сигнала во времени на базе формулы (**Formula**);
- может использоваться описание на базе формулы тактирования (**Clock**). Это описание является развитием описания на базе формулы, оно позволяет организовать повторяющееся выполнение того, что описано с помощью формулы.

Диалоговое окно выбора стимулирующих входных воздействий может быть открыто с помощью кнопки выбора источника стимулирующих воздействий, расположенной на панели инструментов окна временной диаграммы. Его вид представлен на рис. 7.7.

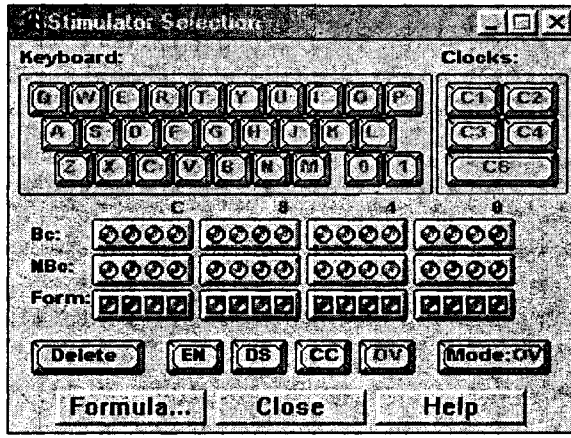


Рис. 7.7. Окно выбора стимулирующих воздействий

Есть несколько способов назначения источников стимулирующих входных воздействий (**Stimulis**) сигналам или выводам элемента схемы. Можно щелкнуть мышью по выбранному источнику и перетащить его на изображение нужного сигнала. Можно, наоборот, выделить значение сигнала, и после этого выбрать соответствующий источник в **Stimulator selection window** (Окно выбора источников входных сигналов). При моделировании могут использоваться следующие управляющие кнопки этого окна:

- **Delete** — позволяет удалить связь источника с конкретным сигналом;
- **DS** — отключает стимулятор, но не удаляет связь с ним;

- **EN** — разрешает работу источника, который был предварительно отключен;
- **CC** — отключает функцию перекрытия значения сигнала, получаемого в модели, сигналом, приходящим от внешнего источника (**override signal function**); используется при определении значений сигналов, которые не являются входными в модели;
- **OV** — позволяет перекрыть значение, получаемое в результате моделирования;
- **Mode CC/ Mode OV** — устанавливает режим источника по умолчанию;
- **CS** — устанавливает использование имеющейся временной диаграммы как входного сигнала;
- **0** — устанавливает моделируемый сигнал в 0;
- **1** — устанавливает моделируемый сигнал в 1.

Для того чтобы включить эти управляющие кнопки, необходимо выполнить следующие действия: назначить сигналам источники входных воздействий, затем активизировать нужный сигнал щелчком мыши, а следующим щелчком — активизировать нужную кнопку. Можно также, при нажатой левой кнопке мыши, перетащить изображение управляющей кнопки на сигнал. В результате она будет сопоставлена сигналу, цвет отображения сигнала в этом случае меняется следующим образом:

- **EN** — подсвечивает сигнал повышенной яркостью;
- **DS** — серый;
- **CC** — черный;
- **OV** — красный.

Работа с бинарным счетчиком

В программе моделирования имеется шестнадцатиразрядный бинарный счетчик. Индикатор этого счетчика расположен в нижней части окна программы моделирования. Он представляет собой набор из 32-х лампочек, расположенных в два ряда. Лампочки верхнего ряда соответствуют прямым выходам счетчика, лампочки нижнего ряда — инверсным. Если разряд счетчика имеет значение 1, то соответствующая его прямому выходу лампочка имеет желто-зеленый цвет, а инверсная — красный. В противном случае (разряд имеет значение 0) — наоборот.

Каждый из выходов счетчика может быть сопоставлен одному или нескольким сигналам.

Работа счетчика управляется тактовым сигналом, установить который можно с помощью диалогового окна **References — Simulation**. Половину длительности такта младший разряд счетчика имеет значение 0, вторую половину — 1.

В диалоговом окне можно установить период младшего разряда счетчика. В нем необходимо указывать половину длительности периода. Например, если желаемая длительность периода 10 нс., то в диалоговом окне необходимо установить значение 5 нс. Длительность периода может варьироваться от 1000 Гц до 100 ГГц. Частота срабатывания каждого последующего разряда счетчика вполовину меньше предыдущего.

В ходе моделирования, текущее состояние счетчика может быть изменено в любой момент времени. Для изменения значения конкретного разряда счетчика достаточно щелкнуть по соответствующей лампочке левой кнопкой мыши. Значения отдельных позиций счетчика изменяются независимо.

Использование инструмента Formula

В модели может быть назначено 16 источников на базе описания формул. Каждый из них представлен квадратной кнопкой. Эти кнопки расположены в ряд непосредственно под индикатором бинарного счетчика. Для того чтобы определить стимулятор на базе бинарного счетчика, необходимо воспользоваться кнопкой **Formula** (Формула) в диалоговом окне **Stimulator Selection** (Выбор источника).

В результате появится диалоговое окно **Set Formula** (Установить формулу). В левой части этого окна расположен список источников, каждый из которых представлен своим номером F0—F15. Если для источников определены формулы, то они также представлены в этом списке.

Для того чтобы определить формулу для источника, надо щелкнуть левой кнопкой мыши по строке, соответствующей этому источнику, в появившемся окне — вписать текст формулы, затем нажать кнопку **Accept** (Принять). С помощью кнопки **Delete** (Удалить) можно стереть выбранную формулу. С помощью кнопки **Delete all** (Удалить все) можно стереть все формулы.

Если закрыть это диалоговое окно с помощью кнопки **Close** (Закрыть), то изменения вступят в силу, если закрыть его с помощью кнопки **Cancel** (Отменить), то изменения будут отменены.

Описание структуры формулы. В формуле могут присутствовать следующие символы:

- H, L — высокий и низкий логические уровни (1, 0);
- X — неизвестное состояние;
- Z — высокоимпедансное состояние;
- 0..9 — цифры, на базе которых определяется продолжительность состояний и количество повторений;
- () — круглые скобки используются для определения границ подвыражений;

- Ps, ns, us, ms — единицы времени для указания продолжительности состояний (по умолчанию — ns);
- [] — квадратные скобки используются для указания шестнадцатеричных значений шин.

Например:

H40L10 — сигнал будет иметь значение 1 в течение 40 нс., затем — 0 в течение 10 нс.

(H40L10)20 — то же, что и в предыдущем примере будет повторено в течение 20 раз.

H4usL1us — сигнал будет иметь значение 1 в течение 4 мкс. и значение 0 — в течение 1 мкс.

((H10L10)20X30)10 — сигнал будет иметь значение 1 в течение 10 нс., затем 0 — в течение 10 нс., эта подгруппа значений будет повторена 20 раз, после чего сигнал будет иметь неопределенное значение в течение 30 нс.; вся группа будет повторена 10 раз.

[02]40[AF]10 — шина будет иметь значение 2 (шестнадцатеричное) в течение 40 нс., AF — в течение 10 нс.

Приведенный инструмент является наиболее удобным для определения значений шин, так как с его помощью можно указывать в квадратных скобках шестнадцатеричные значения всей шины в целом. Одиночные символы H, L, X позволяют присвоить соответствующие значения всем сигналам шины сразу.

Использование инструмента Clock

Для моделирования могут быть использованы четыре инструмента **Clock** (Формула тактирования), которые действуют независимо друг от друга.

Временная диаграмма, заданная с помощью этих инструментов, автоматически повторяется в ходе моделирования. Определение значений этих инструментов также осуществляется с помощью диалогового окна **Set Formula**. Список этих инструментов расположен в левой части окна, непосредственно над списком инструментов — формул, которые обозначены C0—C3. Значения их определяются так же, как и значения формул.

Для того чтобы сопоставить конкретному сигналу значение такого инструмента, необходимо сначала задать для него формулу, после чего выбрать нужный сигнал и воспользоваться кнопкой **Select Stimulator**.

Принудительная установка (Forcing) значения сигналов

Каждой тестовой точке модели можно сопоставить символ клавиатуры от A до Z. Для этого сопоставления надо выбрать сигнал, затем в диалоговом окне

Stimulator Selection определить соответствующую ему клавишу. Сопоставленная сигналу (тестовой точке) клавиша отображается в окне просмотра после имени сигнала. Начальное состояние сигнала в выбранной точке — 1. Нажатие соответствующей ему клавиши переключает значение из 1 в 0, а из 0 в 1.

Если для моделирования необходимо использовать другие значения сигналов, то после того, как точке сопоставлена клавиша, надо вновь активизировать этот сигнал и перейти в диалоговое окно **Test Vector State Selection** (Выбор состояния тестового вектора). В этом диалоговом окне можно выбрать нужное состояние из предлагаемого набора. Если в дальнейшем вновь нажать соответствующую клавишу, то произойдет переключение значения в 0 или 1.

Удаление связи между внешним источником и сигналом

Для того чтобы удалить связь между источником и сигналом, необходимо: активизировать сигнал, которому соответствует этот источник, открыть диалоговое окно **Stimulator Selection** и нажать кнопку **Delete**, после чего диалоговое окно можно закрыть.

Взаимодействие модели и временных диаграмм внешних стимулирующих воздействий

Внешние временные диаграммы сигналов воздействуют на модель в соответствии со следующими правилами:

- когда они применяются к некоторым входным контактам устройства, то их воздействие принимается безусловно. При этом они не влияют на другие контакты устройства, находящиеся в том же узле;
- когда они применяются к выходным контактам устройства в неперекрывающем (non-override) режиме, то влияют на состояние узла, только в случае, когда эти выходные контакты устройства, в соответствии с работой схемы, находятся в высокоимпедансном состоянии;
- если внешний сигнал устанавливается в перекрывающий (`override`) режим, то он перекрывает (`override`) любые контакты устройства в зависимости от его силы (см. табл. 7.3). Процесс перекрытия сигналов распространяется до самого нижнего уровня иерархии.

Для того чтобы переключить временную диаграмму сигнала в перекрывающий (`override`) режим, надо выбрать этот сигнал в поле сигналов и нажать кнопку **OV** в окне **Simulator Selection**.

По умолчанию все стимуляторы находятся в режиме перекрытия. Для того чтобы переключить стимулятор в **Chip Controlled** режим, необходимо воспользоваться кнопкой **CC**.

Когда имени сигнала сопоставлена кнопка клавиатуры, то она управляет всеми входами в узле, если для нее установлен перекрывающий (override) режим.

Использование временных диаграмм в качестве источников

Временные диаграммы сигналов, уже представленные в окне просмотра временных диаграмм, можно сохранить и в дальнейшем использовать в качестве источников значений.

Для этого необходимо выделить сигнал, временную диаграмму которого необходимо сохранить, затем открыть **Stimulator Selection** окно и нажать кнопку **CS**.

Временные диаграммы можно сохранять в файле (бинарный формат TVE) и в дальнейшем загружать в проект.

Формулы сохраняются автоматически в файле **project_name.frm**.

Редактирование временных диаграмм

Временные формы представления сигналов можно редактировать. Для этого пользуются пунктом **Edit** (Редактировать) меню **Waveform** (Временная диаграмма). Будет открыто окно **Test vector state Selection**. В этом окне представлен набор значений сигналов, среди которых можно выбирать.

Кнопка **More** (Больше) позволяет перейти к полному набору значений, кнопка **Less** (Меньше) — вернуться к минимальному набору значений. Можно выбрать часть временной диаграммы и удалить ее с помощью клавиши **Del** (Удалить).

Редактирование временных диаграмм можно осуществлять в одном из двух режимов: **Normal** (Обычный) или **Fast** (Быстрый). В режиме **Normal** возможно редактирование временных диаграмм путем вырезания и копирования квадратных блоков сигналов. Режим **Fast** хорошо подходит для создания новых временных диаграмм на базе уже существующих. Если щелкнуть по временной диаграмме сигнала мышью, то автоматически будет выделена вся область временной диаграммы, правее этого места, после чего этот фрагмент можно переносить.

Каждому сигналу, временная диаграмма которого редактируется, по умолчанию присваивается **CS** маркер, указывающий на то, что этот сигнал необходимо при моделировании использовать как вход.

Внимание

Если временная диаграмма сигнала была сгенерирована при помощи инструмента Formula(Формула), то редактировать ее можно только с использованием этого же инструмента (копировать фрагменты временной диаграммы нельзя).

В любую точку временной диаграммы может быть вставлена формула. Для этого необходимо воспользоваться пунктом **Insert Formula** (Вставить формулу) меню **Waveform**.

Между строками, соответствующими сигналам, можно добавлять пустые строки, для чего используется пункт **Empty Row** (подпункт **Insert**) меню **Signal**.

Для работы с большими (длинными) диаграммами можно применять *маркеры*.

В модели может присутствовать два маркера. Для установки маркера необходимо выбрать из меню Waveform пункт **Markers** (Маркеры) и подпункт **Set Marker 1** (установить маркер 1) или **Set Marker 2** (Установить маркер 2). В дальнейшем возможен быстрый переход на то место диаграммы, которое помечено маркером. Для этого можно воспользоваться подпунктами меню **Waveform Markers Jump to marker 1** (или 2) — перейти к маркеру 1 или 2 соответственно.

Сохранение временных диаграмм в файлах. Временные диаграммы можно сохранять в файле и потом загружать. Для этого могут быть использованы пункты **Load waveform** (Загрузить временную диаграмму) и **Save waveform** (Сохранить временную диаграмму) из меню **File** (файл).

Тестовые векторы могут сохраняться в файлах в ASCII или бинарном формате. Формат ASCII более удобен для последующего просмотра и редактирования обычным текстовым редактором. Бинарный формат является более компактным.

Инструментарий Foundation Express ориентирован на работу с тестовыми векторами, сохраняемыми в файлах бинарного формата. В них сохраняются имена всех сигналов, представленных на временных диаграммах, а также собственно временные диаграммы. Сохраняется также информация о масштабе отображения временных диаграмм в окне просмотра, об используемых для моделирования формулах.

Если временная диаграмма была сохранена с помощью пункта **Save Waveform** меню **File**, то, при последующей работе, она может быть загружена в том же самом виде с помощью пункта **Load Waveform** меню **File**.

Для того чтобы обеспечить возможность взаимодействия с другими инструментами проектирования, Foundation Express поддерживает файлы временных диаграмм в ASCII формате. Сохранение и загрузка файлов в этом формате осуществляется с помощью тех же самых пунктов меню, но необходимо указать тип файла ASC.

Фрагменты временной диаграммы, видимые в окне просмотра, можно скопировать в буфер в графическом виде (bmp), что позволяет использовать их в дальнейшем в других приложениях, например, для документирования результатов проектирования. Для этого можно использовать пункт **Copy Bitmap** (Копировать изображение) из меню **Waveform**.

Выполнение моделирования

Моделирование в пошаговом режиме

Моделирование может выполняться в пошаговом режиме. Каждый следующий шаг выполняется при нажатии кнопки **Step** (Шаг), находящейся на панели инструментов программы моделирования. Рядом с этой кнопкой расположен выпадающий список, который позволяет определить длительность одного шага моделирования.

Возможно выполнение моделирования в режиме до ближайшего события. При этом используется кнопка моделирования до однократного события (панель **CS Probes**). В этой же панели (рис. 7.5) присутствует кнопка **CS**, позволяющая вернуть процесс моделирования к предыдущему событию.

Точки останова

Точки останова определяются совокупностью условий, которым должны удовлетворять сигналы. Выполнение в ходе моделирования условий, соответствующих точке останова, может сопровождаться следующими действиями:

- остановка процесса моделирования;
- размещение маркера на временной диаграмме;
- размещение **Milestone;**
- сохранение тестового вектора;
- загрузка нового тестового вектора;
- модификация существующего тестового вектора.

Для того чтобы разрешить использование точек останова в ходе моделирования, необходимо установить флажок **Enable Breakpoints** на вкладке **General** в диалоговом окне **Preferences**.

Чтобы отобразить диалоговое окно **Breakpoints Conditions** необходимо выбрать пункт **Breakpoint editor** из меню **Tools**. В этом окне можно определить до 16-ти условий точек останова с номерами 0 — f. В левой части этого окна расположен столбец списка сигналов, затем расположены 16 столбцов условий. В столбце условия можно определять конкретные значения сигналов, соответствующих этому условию (этой точке останова).

Для определения условия необходимо выполнить следующие действия: выбрать мышью столбец, соответствующий этому условию (он будет помечен красным), щелкнуть по имени сигнала, который будет участвовать в этом условии (он будет помечен синим). Для того чтобы выбрать значение или событие перехода для этого сигнала, необходимо воспользоваться кнопкой **States**. Чтобы определить последовательность действий, выполняемую при возникновении в модели данной точки останова, надо воспользоваться

кнопкой **Edit**, в результате чего появится диалоговое окно **Breakpoint Edit**, в котором можно описать программу действий.

Информацию о точках останова можно сохранить в файле с помощью кнопки **Save**, а загрузить из файла — с помощью кнопки **Load**. Информация о точках останова сохраняется в файлах с расширением brk. Кнопка **Bus** позволяет переключаться между представлением шин в шестнадцатеричном или посигнальном виде (отображается значение каждой битовой линии, входящей в состав шины).

Создание программ действий, выполняемых при возникновении точек останова

Программы действий, выполняемых при возникновении точек останова, могут включать в себя условные операторы: **if**, **then**, **else**, **end if**

В программах могут выполняться действия из следующего перечня:

- **Mark Breakpoint** — на временной диаграмме помечается место возникновения точки останова;
- **Stop Simulation** — остановка моделирования;
- **Delay** — задерживает действие, следующее за этой командой на указанное время;
- **Count time** — с этого момента начинается подсчет времени;
- **Set Trigger** — отмечает определенные условия по состоянию сигналов или ветвлениям программ;
- **Clear Trigger** — сбрасывает установленные триггеры;
- **Save Milestone** — сохраняется состояние схемы;
- **Save test vector** — информация в точке останова сохраняется в виде тестового вектора;
- **Load new test vectors file** — загружается новый тестовый вектор из файла;
- **Load Additional test vectors** — загружается информация о дополнительных сигналах (не входящих в текущий тестовый вектор);
- **Append test vectors** — загружаются тестовые векторы, начиная с момента модельного времени, помеченного синим курсором.

Моделирование в режиме длительного промежутка времени

Для того чтобы начать моделирование в течение длительного промежутка времени, необходимо воспользоваться пунктом **Start Long Simulation** в меню **Options**. В открывшемся диалоговом окне необходимо указать модельное время, в течение которого должно быть выполнено моделирование.

Для начала моделирования надо нажать кнопку **Start** в этом диалоговом окне. Чтобы остановить моделирование, не дожидаясь завершения процесса, необходимо воспользоваться кнопкой **Stop** основной панели инструментов.

Внимание

Демонстрационные версии Foundation Express могут иметь ограничения на количество событий в ходе выполнения одного моделирования, по достижении которого моделирование останавливается.

Иногда моделирование занимает значительно больше времени, чем ожидает пользователь. О том, что процесс идет успешно (не произошло зацикливания программы моделирования), можно узнать, воспользовавшись настройками на вкладке **General** в диалоговом окне **Preferences**. На этой вкладке можно установить флажок **End of Step Estimation**, в результате чего через каждые 10 секунд моделирования будет выдаваться сообщение о предполагаемом времени, оставшемся до окончания моделирования.

Вехи моделирования Milestones

При использовании режима длительного моделирования важнейшей становится проблема анализа результатов. Программа моделирования для решения этой проблемы предоставляет механизм **Milestones** (Вехи моделирования).

Программа моделирования сохраняет данные только для тех тестовых точек, которые отображаются на экране. Состояния остальных тестовых точек сохраняются только в течение цикла моделирования; в следующем цикле они перекрываются последующими данными. Для того чтобы скомпенсировать эту утрату данных, программа моделирования имеет опцию **Milestones** (Вехи моделирования), которая позволяет восстановить состояние схемы, которое было в выбранном цикле моделирования.

Использование опции **Milestones** (Вехи моделирования) предоставляет следующие возможности:

1. Возможность выполнить рестарт моделирования не с момента времени t0, а с любого другого прошедшего цикла моделирования;
2. Возможно добавление новых тестовых точек при моделировании, начатом с любого цикла моделирования;
3. Можно внести изменения в схему и просмотреть их влияние, начиная с любого момента времени, помеченного **Milestone**;
4. При последующих сеансах моделирования тестовые векторы могут быть изменены, что скажется на поведении системы. Можно заново провести моделирование схемы, начиная с цикла, непосредственно предшествующего ошибке.

Поддерживается три механизма создания **Milestones** (Вехи моделирования):

- ручной;
- автоматический (периодический);
- на базе точек останова.

Ручной механизм. Можно вручную сохранить любое выбранное условие — совокупность значений сигналов (**design condition**) как **Milestone**. Для этого необходимо воспользоваться кнопкой **Save** в окне **Milestones**. Для вызова этого диалога необходимо выбрать пункт **Milestones** меню **Options,** в результате чего, соответствующая точка будет сохранена. Она отображается в списке **Active Milestones** окна **Milestones**.

Автоматический механизм. Можно выбрать опцию **Automatic milestones**, которая позволяет устанавливать **Milestones** через заданные промежутки времени.

Для включения этого режима необходимо установить переключатель в диалоговом окне **Milestones** в положение **On**. Для выхода из этого режима — переключатель установить в положение **Off**.

При использовании этого режима, в диалоговом окне **Milestones** необходимо указать интервал времени между установками Milestones и максимальное количество этих точек, которое не должно превосходить 32. Если в ходе моделирования это количество будет превышено, то каждая последующая **Milestone** будет записываться поверх самой старой из уже существующих.

Механизм на базе точек останова. В Foundation Express существует возможность устанавливать **Milestones** в моменты времени, когда значения выбранных сигналов в схеме удовлетворяют определенному условию. Для установки **Milestone** в точку останова, необходимо выполнить следующие действия. В меню **Tools** надо выбрать пункт **Breakpoints** (Точки останова), в нем подпункт **Edit**, в результате чего появится диалоговое окно **Breakpoints Conditions** (Условия точек останова). В столбце условий (0 — F) необходимо щелкнуть по выбранному условию (после чего оно будет отображаться красным), затем в поле сигналов щелкнуть мышью по названию сигнала и нажать кнопку **States**. После этого необходимо выбрать состояние сигнала, которое будет соответствовать точке останова. Если для сигнала определить два состояния, то точке останова будет соответствовать переход сигнала из одного состояния в другое.

После того, как условие возникновения точки останова определено, необходимо воспользоваться кнопкой **Edit**. В результате, появится диалоговое окно **Breakpoint Edit**, которое позволит воспользоваться дополнительными инструкциями для работы с точками останова. Присвоить инструкции уникальное имя позволяет инструкция **Mark BreakPoint**. Заканчивается работа с этим диалоговым окном при помощи кнопки **Close**.

Диалоговые окна **Breakpoint Edit** и **Breakpoints Conditions** необходимо закрыть с использованием кнопки **OK**. В результате к списку точек останова будет добавлена новая точка останова, и при каждом ее возникновении в ходе моделирования будет сохраняться новая **Milestone**. Необходимо учитывать, что одна сохраненная **Milestone** может занимать очень много места на диске, поэтому в диалоговом окне **Milestones**, при использовании этого режима, можно указать максимальное количество **Milestones** в поле **Break-Points Milestones**, которое будет ограничивать пользователя и не позволит создавать большее число вех **(Milestones)**.

Список всех доступных для модели **Milestones** отображается в диалоговом окне **Milestones** в списке **Active Milestones**. Для того чтобы установить модель в конкретную **Milestone** (Веха моделирования), необходимо выбрать ее мышью в списке и нажать кнопку **Load**.

Можно удалить выбранную **Milestone** с помощью кнопки **Delete** или удалить все существующие в этот момент **Milestones** с помощью кнопки **Delete All**.

Сохранение и загрузка файлов моделирования

Файлы моделирования (DEC) содержат информацию о моделируемой схеме на момент, непосредственно предшествующий записи файла.

Эти файлы хранят таблицы, содержащие следующее: информация о соединениях в схеме, временные диаграммы, состояния триггеров (элементов памяти), информация о настройках моделирования.

Сохранение этой информации осуществляется посредством пункта **Save Simulation State**. Загрузка файлов моделирования осуществляется с использованием пункта **Load Simulation State**. Процесс загрузки может быть выполнен успешно только в случае, если описание схемы устройства не претерпело никаких изменений с момента сохранения загружаемого файла.

Анализ результатов моделирования.

В ходе моделирования сигналы могут принимать состояния, представленные в табл. 7.3.

Таблица 7.3. *Состояния сигналов*

Наименование состояния	Описание
`Low`	Состояние, точно соответствующее логическому 0 в выбранной элементной базе (например, состояние выхода TTL элемента)
`High`	Состояние, точно соответствующее логической 1 (например, состояние выхода TTL элемента)
`High Impedance`	Z — состояние

***Таблица 7.3** (окончание)*

Наименование состояния	Описание
`Unknown`	Неопределенное состояние (например, состояние триггера в начале работы)
`Resistive Low`	Состояние слабого логического 0 (например, `Open Emitter output in Low state`)
`Resistive High`	Состояние слабой логической 1 (например, `Open Collector output in High state`)
`Resistive Unknown`	Неопределенное состояние
`Output Conflict`	Указывает на наличие конфликта (в одной точке одновременно оказывается состояние логической 1 и логического 0)
`High Voltage`	На контакте — высокий уровень напряжения (например, +12 В, -5 В, и др.)
`Reference voltage`	Используется в ECL технологии
`Unknown activity Low`	Состояние с неизвестной активностью, низкий уровень. (`Low or Resistive Low or High Impedance`). Это состояние генерируется выходами схем с тремя состояниями (`tristate ICs`), когда значение на контакте управления состоянием (`tri-state control pin`) не определено
`Unknown activity High`	Состояние с неизвестной активностью, высокий уровень. (`High, Resistive High or High Impedance`)
`SV High`	Высокий уровень напряжения питания (например, `VCC`)
`SV Low`	Низкий уровень напряжения питания (например, `GND`)
`SV X`	Неизвестный уровень напряжения питания

Сигналы в Foundation Express помечаются разными цветами в зависимости от их статуса:

- входные сигналы отображаются *черным;*
- активные выходные сигналы отображаются *синим;*
- сигналы, участвующие в конфликтах, отображаются *красным;*
- сигналы, перекрывающие другие в течение редактирования, отображаются *зеленым;*
- если во временную диаграмму, полученную в результате текущего моделирования, были внесены изменения, они отображаются *малиновым.*

Измерение временного промежутка между событиями

Для переключения в режим измерений необходимо воспользоваться меню **Waveform,** пунктом **Measurements,** подпунктом **Measurements on**.

При переключении в этот режим изменяется форма курсора. Для того чтобы измерить временной промежуток между двумя событиями, необходимо щелкнуть мышью по первому из них, затем по второму. В результате каждое из выбранных событий помечается зеленым цветом. В окно временной диаграммы добавляется пустая строка, в которой будет отображено измеренное значение (оно отображается красным цветом). Если значение времени оказывается очень мало, то, при выбранном масштабировании, оно не отображается, в этом случае необходимо увеличить масштаб.

Для того чтобы выйти из режима измерения временного промежутка между событиями, необходимо вновь воспользоваться подпунктом **Measurements on**.

Отслеживание signal conditions

Тег (**Tag**) — это совокупность **signal conditions**. Если тег (**Tag**) определен, то можно автоматически отыскать все соответствующие ему места в результатах моделирования схемы.

Чтобы колонка тегов отображалась в программе просмотра временных диаграмм, надо в меню **View** установить опцию **Tag**. Для определения тегов необходимо выбрать из меню **Tools** пункт **Set Tag Conditions**, в результате чего будет открыто диалоговое окно **Set Tag Conditions**. После этого в окне программы просмотра необходимо активизировать сигнал, для которого будет определено условие, затем в окне **Set Tag Conditions** воспользоваться кнопкой, соответствующей проверяемому значению. Если необходимо проверять переход из одного значения в другое (событие), то сначала необходимо щелкнуть кнопку, соответствующую значению до перехода, затем соответствующую значению после перехода. Если в тег должны входить условия для нескольких сигналов, то надо повторить эти действия для каждого из них. Эти условия объединяются логикой "И", т. е. условие, соответствующее тегу в целом, считается выполненным, если выполнены все условия, входящие в его состав.

Для того чтобы отыскать места на временной диаграмме, соответствующие выполнению тега, необходимо выполнить следующие действия. На основной панели инструментов, в выпадающем списке **Search Anchor**, выбрать **Tag**, затем в окне программы просмотра мышью выбрать момент времени, начиная с которого будет вестись поиск. Это место будет помечено вертикальным курсором синего цвета. Далее, с помощью кнопок основной панели инструментов **Search Left** и **Search Right** можно перемещаться на очередные места выполнения комплекса условий тега (влево или вправо, соответственно). При этом синий вертикальный курсор перемещается в

очередное место выполнения тега. Если в определение тега входят только условия на значения сигналов, но нет событий, то курсором отмечается начальная точка выполнения условия. Поскольку условие может выполняться в течение некоторого промежутка времени, то при поиске справа налево и слева направо начальные точки могут не совпадать.

Комментарии во временных диаграммах

Временные диаграммы можно снабжать комментариями.

Для добавления комментария необходимо воспользоваться подпунктом **Add** пункта **Comment** меню **Waveform**, в результате чего появляется диалоговое окно **Add Comment**, в котором можно записать текст комментария.

Для редактирования или удаления уже существующих комментариев можно воспользоваться, соответственно, подпунктами **Edit** и **Del** этого же пункта меню.

Моделирование устройств памяти

Моделирование устройств памяти в Foundation Express отражает их реальную работу достаточно подробно. Моделируется и фактическое хранение значений данных (представляются в шестнадцатеричных кодах) в моделируемом устройстве памяти. Данные могут быть в действительности записаны в моделируемую память и считаны из памяти при выполнении соответствующих циклов обмена.

Содержимое моделируемой памяти хранится в структурах данных системы моделирования, и, при значительных объемах адресного пространства моделируемого устройства памяти, может занимать большой объем оперативной памяти инструментальной машины. Если размер моделируемой памяти составляет 1 Мбайт, то для ее моделирования занимается 1 Мбайт оперативной памяти компьютера.

Однако в большинстве случаев при моделировании фактически требуется содержимое только фрагмента памяти. В Foundation Express предусмотрена возможность моделирования хранения информации не во всем моделируемом устройстве, а только в некоторой его части. Моделируется хранение данных в двух областях адресного пространства моделируемой памяти — в нижней области адресного пространства, и в верхней области адресного пространства (рис. 7.8). Размеры моделируемых, с точностью до содержимого, участков адресного пространства задает пользователь.

Для указания этих параметров используется вкладка **Simulation** диалогового окна **Preferences**. На ней, в области **Memory range,** можно задать значения параметров **Lower Mem Address** (Граница нижней области памяти) и **Upper Mem Address** (Граница верхней области памяти). Параметр **Lower**

Mem Address определяет количество байтов, начиная с адреса 0, которые будут моделироваться; иначе говоря — верхнюю границу моделируемой области адресного пространства, от адреса 0 до адреса **Lower Mem Address** — 1. Параметр **Upper Mem Address** определяет количество байтов, начиная от самого старшего адреса адресного пространства памяти, которые будут промоделированы; иначе говоря — нижнюю границу верхней моделируемой области адресного пространства (рис. 7.8). На рисунке серым цветом отмечены моделируемые области памяти.

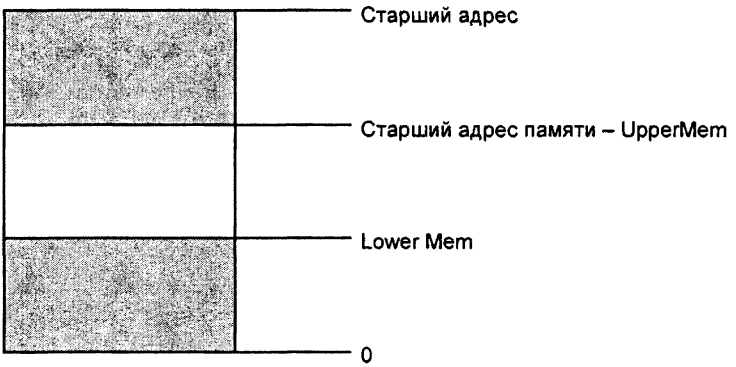


Рис. 7.8. Определение моделируемых (с точностью до содержимого) областей памяти

Если в моделировании участвует несколько блоков памяти, то для них эти параметры имеют одни и те же значения.

Возможен также просмотр и редактирование состояний памяти. Для выполнения этой операции необходимо воспользоваться подпунктом **Edit** пункта **Memory Contents** меню **Device**.

Загрузка и сохранение содержимого блоков памяти в файлах

В моделируемый элемент памяти можно загрузить содержимое шестнадцатеричного файла.

Для загрузки содержимого файла в моделируемый кристалл памяти (**memory chip**) необходимо выполнить следующие действия. В меню **Device** надо выбрать пункт **Memory contents**, а в нем — подпункт **Load Contents**, в результате чего появится диалоговое окно **Select Memory Chip**, где, в списке **Scan Hierarchy** необходимо выбрать уровень иерархии, на котором находится элемент памяти. Затем, в списке **Chip Selection** выбрать нужный элемент памяти, после чего надо нажать кнопку **Select**. В результате появится диалоговое окно **Load Hex File**, в котором можно выбрать загружаемый файл.

Для загрузки содержимого файла в моделируемый блок памяти необходимо выполнить следующие действия: в меню **Device** выбрать пункт **Memory contents**, а в нем — подпункт **Load Block Contents**. В появившемся диалоговом окне **Select memory block**, в списке **Block Selection** надо выбрать наименование нужного блока, если он уже определен.

После того как выбран блок памяти, в который будет загружаться информация из файла, открывается диалоговое окно **Load Hex File**. В нем можно выбрать имя файла, содержимое которого будет загружено в блок.

Для того чтобы выполнить определение нового блока памяти, необходимо воспользоваться кнопкой **Define Block** в диалоговом окне **Select memory block**. В результате появится диалоговое окно **Define Memory Block**, в котором необходимо определить имя блока (**Memory block name**), количество слов, входящих в блок (**Number of words**) и длину слова (**Length of word**). Если работа с этим окном завершается нажатием **OK**, то открывается диалоговое окно **Memory Configurator** — конфигурирование блока памяти. Для того чтобы сконфигурировать блок памяти, необходимо выполнить следующие действия: для выбора кристаллов памяти для данного блока — **Select**, в результате чего появится диалоговое окно **Select Chips for memory block**, в котором можно сделать этот выбор. Выбранные кристаллы помещаются в список **chips for current block**, расположенный в левой части диалогового окна. Оттуда их можно перетаскивать в таблицу, которая соответствует блоку памяти. Она расположена в правой части диалогового окна (уже использованные кристаллы помечаются синим цветом).

Кнопка **New** позволяет определить новый блок.

Кнопка **Hierarchy** служит для отображения иерархии проекта.

Просмотр и редактирование содержимого кристаллов и блоков памяти

Для просмотра и редактирования содержимого кристаллов памяти и блоков памяти необходимо выбрать подпункт **Edit** пункта **Memory contents** в меню **Device**. В результате, откроется диалоговое окно **Select Block/Chip for memory Editor**, в котором можно выбрать блок или кристалл памяти, содержимое которого необходимо просмотреть или отредактировать.

После выбора объекта открывается диалоговое окно **Memory Editor**. Отображение содержимого памяти организовано в виде строк слов памяти. В левой части каждого ряда отображается адрес первого слова, входящего в этот ряд. Длина слова зависит от ширины окна **Memory editor** на экране

Для того чтобы изменить значения слова памяти, надо дважды щелкнуть по нему мышью, затем ввести новое значение, после чего нажать клавишу **Enter**.

Кнопка **Display** позволяет открыть диалоговое окно **Display Options**, в котором можно менять следующие настройки отображения памяти. Содержимое

памяти может отображаться в бинарном или шестнадцатеричном представлении (переключатель **Memory**: **binary/hexadecimal**); адреса памяти могут быть представлены в бинарном или шестнадцатеричном представлении (переключатель **Address: binary/hexadecimal**); отображение может осуществляться в виде обычного содержимого памяти или управляющих слов (команд) (переключатель **Display: memory contents/control words**).

Кнопка **New** позволяет открыть новое окно редактора.

Режимы моделирования

В зависимости от стадии проектирования моделирование может осуществляться в различных режимах.

Возможны следующие режимы моделирования:

- **Functional** — функциональное моделирование; при моделировании в этом режиме задержки распространения сигнала предполагаются равными 0;
- **Timing** — временное моделирование; при моделировании в этом режиме можно получить наиболее точные временные характеристики;
- **Glitch** — функциональное моделирование с регулируемой единичной задержкой (`adjustable unit delay`);
- **Unit** — функциональное моделирование с единичной задержкой, устанавливаемой по показателям изготовителя СБИС (`factory-set unit delay`).

Если программа просмотра запускается с помощью кнопки диаграммы проекта, то тип моделирования определяется автоматически. Пользователь может также выбрать режим моделирования вручную, с помощью выпадающего списка, присутствующего в панели инструментов программы моделирования.

Изменение режима моделирования мгновенно отражается на процессе моделирования схемы: фрагмент временной диаграммы, получаемый после изменения режима, формируется в соответствии с выбранным режимом. Фрагмент временной диаграммы, предшествующий изменению режима, модификации не подвергается.

Функциональное моделирование (Functional)

При функциональном моделировании изменение сигналов на входах схемы мгновенно отражается на ее выходах. Этот режим позволяет проанализировать работу схемы достаточно быстро и просто, но только в самом общем виде.

Если значения с какого-либо выхода схемы поступают на вход этой же схемы, то возникают осцилляции. В этом случае пользователь получает соответствующее сообщение, моделирование прерывается, и пользователь может выбрать другой режим моделирования, при котором временные задержки учитываются (**`Timing`** или **`Unit delay`**).

Моделирование с единичной задержкой (Unit delay)

При моделировании в этом режиме предполагается, что все модули, входящие в состав схемы имеют одинаковую задержку сигнала. Она равна значению, установленному параметром **Current simulation precision**. Этот режим может использоваться для функционального моделирования схем, в которых сигналы передаются с выхода на вход. В этом режиме информация об ошибках временных характеристик (**timing errors**) не выдается, но некоторые кратковременные броски сигналов — **glitches,** отображаются в окне программы просмотра, а соответствующие сигналы выделяются цветом.

Временное моделирование (Timing)

При моделировании в этом режиме учитываются действительные задержки всех элементов схемы с точностью до 100 пс. При моделировании в этом режиме можно выявить критические цепи, посмотреть, как влияет на временные характеристики схемы температура, напряжение, нагрузка. В этом режиме выявляются временные ошибки, выдается отчет обо всех ошибках, таких как **insufficient Setup** или **Hold times**. Временное моделирование используется, как правило, на завершающих стадиях разработки схемы.

При работе в этом режиме моделирования можно изменять временные характеристики модулей, входящих в состав схемы. Для этого необходимо воспользоваться пунктом **Edit timing specification** (Редактировать временную спецификацию) меню **Device** (Устройство). В результате, появится диалоговое окно **Edit timing Specification — Select Chip**. В левой части окна расположен список **Scan Hierarchy,** в котором можно выбрать необходимый уровень иерархии. В правой части окна расположен список **Chip Selection**, в котором отображается перечень элементов, расположенных на выбранном уровне иерархии. Если выбрать элемент в этом списке, то в нижней части окна будет отображена краткая информация о нем.

Для просмотра и редактирования временных параметров элемента его необходимо выбрать в списке **Chip Selection,** после чего нажать кнопку **Select**. В результате появится диалоговое окно **Edit Timing Specification**, где информация разделена на шесть колонок. В первой колонке (**Name**) указаны имена временных параметров. Во второй (**Min**) — минимальные значения этих параметров. В третьей (**Avg**) — средние значения этих параметров. В четвертой (**Max**) — максимальные значения параметров. В пятой (**Set**) — текущие значения параметров. В последней колонке (**Description**) — краткое описание параметров. Можно изменить значения всех параметров одновременно. Нажатие кнопки **Min**, **Avg** или **Max** приводит к установке всех параметров в минимальное, среднее или максимальное значение соответственно. В строке ввода **% of Max** можно указать процент от максимального значения, в соответствии с которым будут установлены все параметры. Для того чтобы изменить значение конкретного параметра, необходимо дважды щелкнуть

мышью по его значению в столбце **Set**, затем ввести новое значение, после чего нажать клавишу **Enter**. Для того чтобы сделанные изменения вступили в силу, необходимо в диалоговом окне нажать клавишу **OK**; для отмены изменений — **Cancel**.

Для поиска ошибок можно воспользоваться кнопкой **Search Errors** (Поиск ошибок).

Для моделирования в этом режиме могут быть использованы те же тестовые векторы, что и при функциональном и **Glitch** моделировании.

Задержки в линиях связи (**Line Delays**). Список связей, загружаемый в программу моделирования, может содержать параметры задержки в линиях связи **–Line Delays**. Они представляют собой специальные атрибуты, назначаемые цепям, у которых время передачи в линиях отлично от 0.

Для того чтобы просмотреть и отредактировать эти параметры, необходимо воспользоваться пунктом **Change Line Delays** меню **Device**. При этом появляется диалоговое окно **Set Line Delay**. В нем расположен список **Scan Hierarchy**, в котором необходимо выбрать уровень иерархии, временные параметры цепей которого будут просматриваться. После этого надо нажать кнопу **Select**; появится диалоговое окно **Line Delay Value Selection**, оно организовано аналогично диалоговому окну **Edit timing Specification**. В первой колонке (**Name**) расположены названия кристаллов, выводов и тип задержек. Эта информация имеет следующий синтаксис: **chip_name.pin_name — delay_type**, далее указаны значения задержек.

Точность временных характеристик схемы, **Simulation Precision**. Все программы временного моделирования функционируют с определенной точностью использования и представления временных характеристик. Для большинства таких программ точность представления времени фиксирована, и составляет 100 пс. (пикосекунд, ps) или 1 нс. (ns). При этом общее время моделирования ограничивается длительностью 100 мс. (миллисекунд, ms) и 1 с., соответственно.

В системе Xilinx Foundation Express используется более гибкая организация управления временной точностью моделирования, что позволяет работать с разными схемами, временные характеристики которых различаются на порядки величин. Здесь применяется не фиксированная, а переменная, настраиваемая точность временного моделирования, которая может лежать в диапазоне от 10 пс. до 1 мс. Соответственно, можно моделировать от 40 мс. до 1-го часа реального времени работы устройства.

Для того чтобы установить новое значение точности временного моделирования, **Simulation Precision**, необходимо воспользоваться пунктом **Preferences** меню **Options**. В появившемся диалоговом окне на вкладке **Simulation** можно установить нужное значение этого параметра в выпадающем списке **Simulation Precision**.

После изменения параметра **Simulation Precision**, Foundation Express автоматически устанавливает модельное время в 0. Таким образом, невозможно смешивать (в рамках того же прогона модели) фрагменты временной диаграммы с разными значениями **Simulation Precision**.

Сравнение результатов временного и функционального моделирования

В ходе проектирования устройства используют комбинацию указанных здесь режимов моделирования. На каждом этапе проектирования, используя адекватный ему режим, исследуют и проверяют те или иные функциональные или временные свойства проектируемой схемы. Понятно, что существенное значение имеет возможность сопоставления результаты моделирования на разных этапах, полученные в различных режимах.

Рассмотрим, как можно сравнить результаты временного и функционального моделирования. При функциональном моделировании надо пометить все сигналы маркером **Cs**, затем сохранить временные диаграммы в файле, после чего в программе моделирования можно переключить систему в режим временного моделирования (при этом в программе моделирования должен быть загружен список связей). Затем надо загрузить из файла предварительно сохраненные диаграммы. Если после этого запустить моделирование, то можно увидеть, как изменяются его результаты по сравнению с функциональным моделированием.

Функциональное моделирование с регулируемой единичной задержкой (Glitch)

При моделировании в этом режиме, базовой считается задержка сигнала в одном модуле. Задержка сигнала во всех модулях считается одинаковой и может быть установлена равной любому значению, допустимому при данной точности временных представлений в модели. Эта задержка считается равной **Simulation Step** и может быть установлена путем выбора соответствующего значения в одноименном выпадающем списке.

Этот режим позволяет определить только последовательность событий, но не реальные временные соотношения. Он имеет много общего с режимом моделирования **Unit**.

Выборочное моделирование

Для моделирования можно выбирать не всю схему, а отдельные ее фрагменты, что позволяет существенно сократить время моделирования.

Если моделирование всего проекта в целом занимает очень большое время, то при выборочном моделировании (по частям), время может быть сокра-

щено значительно. При этом временные диаграммы выходов одних блоков могут быть сохранены и использованы для подачи на входы блоков, которые моделируются следующими. Этот прием позволяет промоделировать даже очень большие схемы с высокой точностью.

Выбор моделируемых фрагментов можно делать непосредственно в процессе моделирования. Опция **Selective Simulation** (Выборочное моделирование) может быть установлена в меню **Options**. В этом случае диалоговое окно **Select chip for Selective Simulation** (Выбрать компоненты для выборочного моделирования) отображается автоматически.

В этом окне отображается иерархическое представление моделируемой схемы. Все элементы схемы, которым разрешено участие в моделировании, отображаются черным цветом, остальные — белым. Переключение состояния элемента осуществляется однократным щелчком левой кнопкой мыши по его изображению.

Если щелкнуть в области иерархического представления схемы правой кнопкой мыши, то появится всплывающее меню, которое позволяет включить в моделирование или исключить из него все элементы схемы сразу. Все элементы, исключенные из моделирования, рассматриваются как пустые сокеты. Все их входы и выходы переводятся в высокоимпедансное состояние.

Создание физической реализации проектируемого устройства

После того как в ходе функционального моделирования получены результаты, удовлетворяющие требованиям, предъявляемым к устройству, можно переходить к его физической реализации.

Для активизации инструментария, выполняющего этот процесс, необходимо воспользоваться кнопкой **Implementation** в окне диаграммы проекта.

Версии и редакции проекта

Для обеспечения надежности процесса проектирования можно, при внесении изменений в какой-либо компонент, входящий в состав проекта, создавать новую версию реализации проекта и снабжать ее соответствующими комментариями.

Для создания новой версии реализации проекта необходимо выполнить следующие действия:

1. В браузере иерархии перейти на вкладку версии.
2. Выбрать пиктограмму, представляющую текущую схему, и щелкнуть по ней левой кнопкой мыши.

3. В появившемся всплывающем меню выбрать пункт **Create version** (Создать версию).
4. В появившемся диалоговом окне ввести имя вновь создаваемой версии.

В рамках одной версии возможно несколько способов реализации (**revisions**).

Для того чтобы удалить версию из проекта, надо щелкнуть правой кнопкой мыши по ее названию и воспользоваться пунктом **Delete** всплывающего меню.

В рамках одной версии может существовать несколько редакций (revision) проекта. Как правило, в таких случаях, всем этим редакциям соответствует один и тот же вариант исходного описания схемы, но возможны различные ограничения.

Для того чтобы запустить процесс создания новой или перезаписи уже существующей редакции физической реализации, необходимо воспользоваться подпунктом **Implement Design** меню **Implementation**. Появится диалоговое окно, позволяющее выполнить необходимые настройки.

Процесс создания физической реализации

При запуске процесса физической реализации отображается окно процесса проектирования **Flow Engine**, которое позволяет управлять процессом физической реализации. Общий вид окна представлен на рис. 7.9. Этот процесс включает в себя следующие шаги:

1. Преобразование списка связей в XNF или EDIF формате в формат Xilinx internal database (NGD).
2. Отображение схемы на структуру ячеек кристалла, на базе которого выполняется реализация.
3. Размещение элементов и трассировка.
4. Анализ временных характеристик.
5. Конфигурирование.

По умолчанию процесс продолжается от первого шага до последнего или до возникновения ошибки, делающей невозможным продолжение процесса. Например, реализация схемы оказывается невозможной, если выбранная FPGA не содержит достаточного количества элементов.

Рассмотрим структуру окна процесса проектирования **Flow Engine.** В верхней половине окна расположены пиктограммы, соответствующие этапам работы. Между ними расположены стрелки, указывающие последовательность действий. Стрелки, соответствующие уже выполненным действиям, окрашены в

черный цвет, остальные стрелки — прозрачные. Под изображениями пиктограмм указан статус выполнения соответствующих им действий: **Completed** (Успешное выполнение), **Failed** (Наличие ошибок).

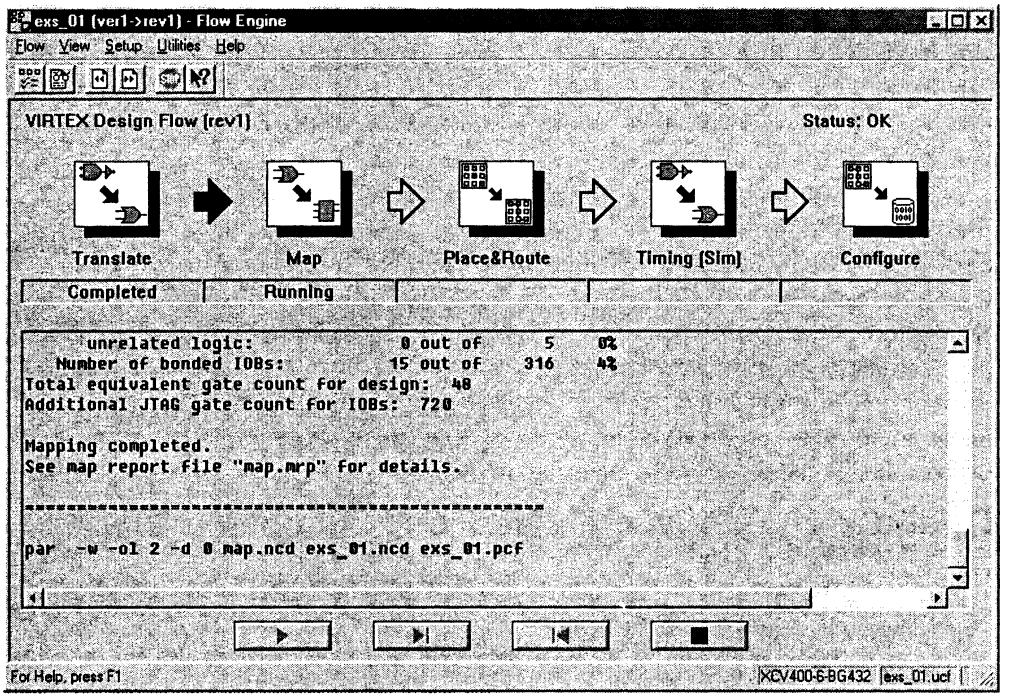


Рис. 7.9. Окно **Flow Engine**

В нижней части окна расположен набор кнопок, позволяющих управлять процессом реализации:

- следующий этап — выполнить первый из еще не выполненных этапов;
- выполнить до конца;
- уничтожить информацию о реализации.

Процесс реализации может быть остановлен пользователем с помощью кнопки **Stop After** (или аналогичного пункта меню **Setup**). В этом случае появляется диалоговое окно, в котором можно выбрать этап, после которого моделирование должно быть остановлено. После указанного этапа, поверх стрелки к следующему этапу, размещается символ **Stop**.

После завершения процесса реализации можно сформировать отчеты о временных характеристиках, используя пункт **Produce timing Report** меню **Utilities**.

Оптимизация схемы

После того как создана реализации схемы (**implementation**) и определены ограничения, можно переходить к оптимизации схемы по занимаемой площади и скорости работы в соответствии с определенными ограничениями.

Для выполнения оптимизации необходимо щелкнуть правой кнопкой мыши по выбранной реализации на вкладке **Versions** браузера иерархии, в появившемся всплывающем меню выбрать **Optimize** (Оптимизировать).

Полоса в верхней части экрана будет отображать ход процесса оптимизации. После завершения процесса пиктограмма, соответствующая вновь оптимизированной реализации, будет размещена на вкладке **Versions** в браузере иерархии, поверх пиктограммы реализации до оптимизации.

Для просмотра отчета об оптимизации необходимо щелкнуть правой кнопкой мыши по оптимизированной реализации, в появившемся меню выбрать **View Synthesis results** (Просмотреть результаты синтеза). В результате будет открыто окно браузера отчетов, в котором будет отображен текст отчета.

После анализа отчета можно изменить ограничения и выполнить оптимизацию вновь, если желаемый результат не достигнут.

FPGA-редактор

Этот графический редактор предназначен для просмотра и редактирования графических представлений FPGA.

Он может использоваться для размещения в FPGA критических элементов и связей перед запуском автоматизированного инструментария **place-and-route** или после запуска этого инструментария, если ему не удалось разместить какие-либо элементы и связи или же они оказались неудачно размещенными.

FPGA-редактор работает с **NCD** (**Native Circuit Description**) файлом описания модели системы Foundation Express. Этот файл содержит описание логики работы схемы в терминах компонентов выбранной элементной базы. В своей работе он учитывает требования к схеме, представленные файлом физических ограничений (PCF).

Окно редактора (рис. 7.10) содержит следующие основные компоненты: заголовок, меню, основная панель инструментов, панель инструментов для работы с уровнями, окно массива, окно списков, пользовательская панель инструментов, панель истории, окно статуса, окно для редактирования логических блоков.

Окно массива содержит графическое представление FPGA-устройства. В нем представлены элементы связи и связи между ними на логическом и физическом уровне.

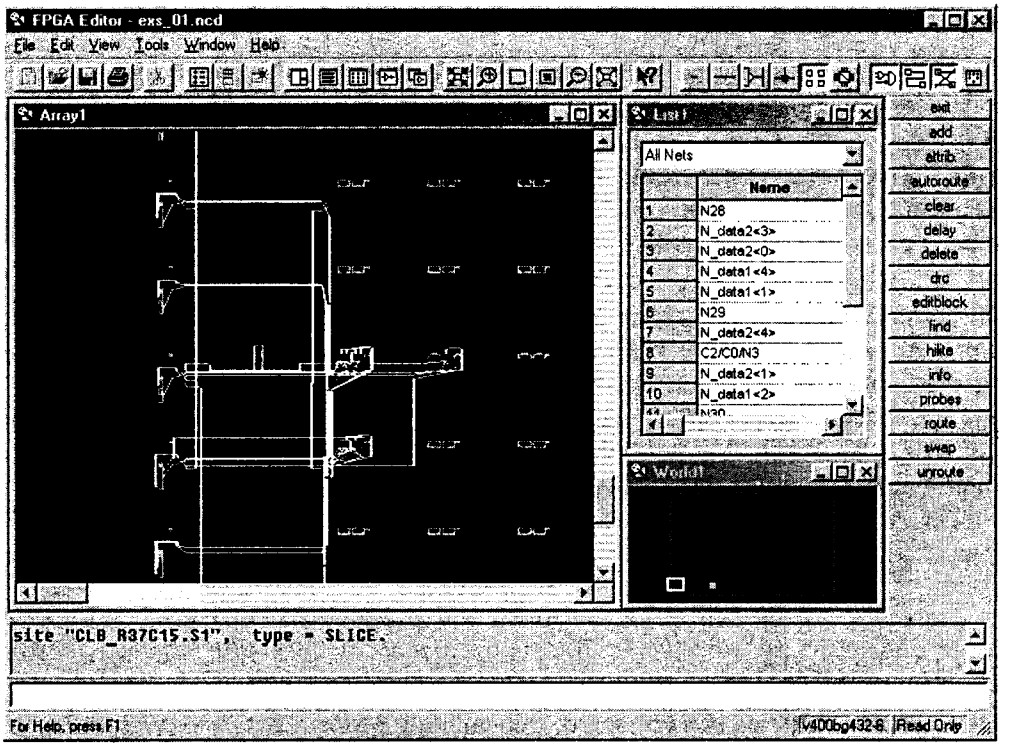


Рис. 7.10. Окно FPGA-редактора системы Foundation Express

Проверка временных характеристик модели после этапа реализации

После выполнения этапа реализации проектируемой схемы в FPGA появляется более достоверная информация о фактических временных характеристиках устройства, которое мы ранее моделировали. Для проверки временных характеристик модели можно воспользоваться одним из следующих инструментов:

- Gate level Timing Simulator;
- Interactive Timing Analyzer.

Активизация обоих инструментов происходит в результате нажатия соответствующих кнопок в фазе верификации (**Verification**) диаграммы проекта.

Программа **Gate level Timing Simulator** является инструментом **Logic Simulator**; запускается в режиме временного моделирования (описан ранее в этой главе).

Анализатор временных характеристик схемы (timing analyzer)

С помощью инструмента **Interactive Timing Analyzer** можно выполнять статический анализ временных характеристик FPGA или CPLD.

Для того чтобы можно было воспользоваться этим инструментом, необходимо, чтобы для FPGA-проекта был выполнен этап отображения схемы на базовую структуру ячеек кристалла и, частично или полностью, выполнены размещение и трассировка (`place and route`). Для CPLD-проекта размещение и трассировка должны быть выполнены полностью.

Статический временной анализ состоит в анализе задержек между точками сгенерированной реализации схемы (`point-to-point analysis`). Векторы входных значений для этого не используются. При этом анализе проверяется, удовлетворяет ли схема заданным временным характеристикам. В ходе анализа отображаются данные, позволяющие определить критические цепочки в схеме, длительность цикла (такта), цепочки с максимальным временем прохождения сигнала.

Анализатор временных характеристик — это статический анализатор, он работает полностью в терминах синхронных схем. Проверку временных характеристик установки и удержания состояний сигналов данный инструмент проводить не позволяет. Для проверки и анализа этих характеристик необходимо использовать инструментарий моделирования.

Окно анализатора включает меню, панель инструментов и строку статуса.

Открытие описания схемы. Для открытия описания схемы можно воспользоваться пунктом Open design меню File (Файл). Если проект реализован на FPGA, то в диалоге необходимо выбрать файл проекта с расширением ncd, если же проект реализован на CPLD, то необходимо выбрать файл проекта с расширением vm6.

В процессе загрузки анализатор временных характеристик читает список связей, содержащийся в ncd-файле, и анализирует временные ограничения, информация о которых присутствует в модели. После завершения загрузки можно формировать отчет о временных характеристиках. Если в проекте имеется файл физических ограничений по умолчанию (этот файл должен иметь имя, совпадающее с именем проекта и расширение pcf), то при открытии описания схемы этот файл так же автоматически открывается.

Открытие файла физических ограничений. Эта операция применима только для FPGA-проектов, поскольку для CPLD-проектов эта информация хранится прямо в vm6-файле. Для открытия файла физических ограничений необходимо воспользоваться пунктом **Open Physical Constraints** из меню **Fail**. Файлы физических ограничений имеют расширение pcf.

Если в анализаторе временных характеристик открыт файл физических ограничений, то информация об имени этого файла отображается в строке статуса.

Для просмотра всех сигналов тактирования, именующихся в открытом описании модели, необходимо воспользоваться пунктом **Clocks** меню **View**. В результате будет создан отчет — информация о сигналах тактирования. Этот отчет может быть сохранен в файле.

Установки. Текущие установки временного анализатора для работы с загруженным в него описанием схемы можно просмотреть с помощью пункта **Settings** меню **View**.

Могут быть использованы следующие установки:

- **OpenPCF** — указывается имя открываемого файла физических ограничений;
- **Speed** — коэффициент скорости выполнения действий (Значение этого параметра устанавливается с использованием пункта **SpeedGrade** из меню **Options**);
- **IncludeNets** — включаемые цепи (если значение этого параметра не указано, то при анализе рассматриваются все цепи);
- **ExcludeNets** — исключаемые из анализа цепи (если значение этого параметра не указано, считается, что все цепи участвуют в анализе);
- **SelectFailingTimeConstraints** — этот параметр может иметь значения **Yes** или **No** (если он имеет значение **Yes**, то формируется информация только о цепочках, не имеющих временных ограничений; по умолчанию этот параметр имеет значение **No**).

Генерация файла прошивки FPGA

Если полученная схема устройства удовлетворяет предъявляемым к ней требованиям, можно перейти к завершающему этапу проектирования — генерации файла для прошивки FPGA. Для выполнения этой операции необходимо воспользоваться кнопкой **Programming** на диаграмме проекта.

Дополнительные возможности

Инструментарий Foundation, с помощью которого генерируются оптимизированные списки связей, организован таким образом, чтобы достигалась наиболее качественная оптимизация комбинационной логики, имеющей в общем случае нерегулярную структуру. При этом он зачастую не позволяет получить достаточно хорошее качество оптимизации таких регулярных структур, как сдвиговые регистры, счетчики, сумматоры, блоки памяти. Списки связей для более крупных блоков с неоднородной структурой также могут быть оптимизированы более качественно с использованием специальных процедур оптимизации, ориентированных именно на этот тип блоков.

В Foundation включены два инструмента, позволяющие генерировать оптимизированные списки связей для большого набора наиболее употребимых компонентов. Применение этих списков связей позволяет сократить площадь кристалла, занимаемую компонентом, и увеличить его быстродействие по сравнению с использованием тех списков связей, которые сгенерированы на базе поведенческого описания.

Инструмент LogiBLOX предназначен для синтеза списков связей компонентов с малой степенью интеграции. Инструмент COREGEN предназначен для синтеза списков связей компонентов с большой степенью интеграции.

Использование LogiBLOX-компонентов

Инструмент LogiBLOX — это графический интерфейс пользователя, который позволяет создавать так называемые LogiBLOX-компоненты высокого уровня (такие как счетчики, сдвиговые регистры, мультиплексоры, и др.). Они используются, как правило, для схем, описанных на языках высокого уровня — VHDL или Verilog.

В состав этого инструментария входит библиотека описаний компонентов. Для каждого компонента определен набор параметров, которые может определять пользователь и правила построения оптимизированного списка связей. Например, с помощью этого инструмента можно сгенерировать оптимизированный список связей для регистра и определить для него такие параметры, как разрядность регистра, возможность асинхронного сброса и асинхронной установки. Наряду с оптимизированным списком связей генерируется поведенческое описание компонента. Это позволяет выполнять поведенческое моделирование устройства, в состав которого включается этот компонент.

Для создания нового LogiBLOX-компонента необходимо выполнить следующую последовательность действий:

1. Запустить **LogiBLOX**-генератор, для чего выбрать подпункт **LogiBLOX module generator** пункта **Design Entry** в меню **Tools**. В результате будет запущена программа **LBGUI** и открыто диалоговое окно **LogiBLOX selector**. Кроме того, открывается диалоговое окно **LogiBLOX GUI Messages**, в котором отображаются сообщения программы **LBGUI.**
2. В окне **LogiBLOX selector** в поле **Module Name** можно выбрать имя одного из уже существующих компонентов или ввести имя вновь создаваемого.
3. В поле **Module type** необходимо указать тип создаваемого модуля.
4. В списке **Bus width** выбрать ширину шины (2, 4, 8,16, 32).
5. Выбрать дополнительные сигналы, необходимые для работы устройства.

Для каждого вновь создаваемого компонента создаются следующие файлы:

- файл графического представления компонента;
- EDIF-список связей;
- MOD-файл, содержащий описание входных и выходных сигналов;
- VHDL- или Verilog-шаблон компонента;
- VHDL- или Verilog-файл описания поведения компонента.

Генерация и использование макроячеек (IP-блоков, CORE-модулей)

Инструмент **Xilinx CORE generator system** позволяет создавать параметризуемые макроячейки, IP-ядра (**cores**), оптимизированные для Xilinx FPGA. С помощью этого генератора могут создаваться как относительно простые макроячейки (сумматоры, умножители), так и блоки со значительно более сложным поведением.

Создаваемые макроячейки генерируются с оптимизацией их размещения на кристалле, таким образом, что их производительность не зависит ни от размеров используемой СБИС FPGA, ни от включения в состав проекта на FPGA других макроячеек, компонентов, IP-ядер. Оптимизированная компоновка улучшает производительность и сокращает энергопотребление сгенерированного компонента. Производительность и показатели использования ресурсов FPGA для макроячеек, автоматически сгенерированных программой CORE generator, близки к характеристикам, которые можно получить при ручном проектировании опытными разработчиками.

Для сгенерированной макроячейки IP-ядра формируется детальное функциональное описание, временные диаграммы, показатели производительности.

В CORE generator используется так называемая *Smart IP-технология*. Он позволяет использовать предварительно разработанные параметризованные IP-блоки. Эти блоки имеют структуру, оптимизированную на физическом уровне в соответствии с выбранной для реализации элементной базой.

Для создания новой макроячейки необходимо выполнить следующие действия:

1. Выбрать пункт **Core Generator** из подменю **Design entry** в меню **Tools**. В результате будет открыто окно **Xilinx Core Generator**.
2. В открывшемся окне надо выбрать тип макроячейки из списка, расположенного в левой части окна.
3. В правой части окна, в появившемся наборе макроячеек, входящих в этот тип, дважды щелкнуть мышью по конкретной макроячейке.

4. В появившемся диалоговом окне (его имя совпадает с именем выбранной макроячейки, в дальнейшем будем называть его Macro_Name) определить параметры этой макроячейки. После нажатия кнопки **Generate** появится последовательность диалоговых панелей **Destination Project**.
5. В диалоге **Destination Project** можно указать проект, в который должна быть добавлена создаваемая макроячейка. По умолчанию проектом назначения является текущий проект.

После нажатия кнопки **OK** автоматически выполняется генерация макроячейки:

1. Создается графическое представление макроячейки;
2. Формируется EDIF-список связей;
3. Генерируется VHDL- или Verilog-шаблон компонента (включает в себя декларацию компонента и конфигурационную спецификацию);
4. Генерируется VHDL- или Verilog-файл описания поведения компонента.

EDIF-список связей автоматически конвертируется в ALR- и ASX-файлы, содержащие бинарное представление списка связей и описания портов. Созданный компонент автоматически сохраняется в рабочей библиотеке проекта.

Если уже созданный с использованием **Core generator** блок необходимо изменить, то в **Core generator** для этого блока может быть использован XCO-файл, который создается при первом создании блока.

Core generator может создавать следующие выходные файлы, содержащие описание блока:

- EDN — EDIF-список связей (используется инструментарием Xilinx implementation);
- VHO — файл VHDL-шаблона (используется для создания компонента-"ядра" - Core (иначе — IP-блока модели), а также для определения ссылки на сгенерированное поведенческое описание, текст которого пользователю не доступен);
- VEO — файл Verilog-шаблона (используется для создания спецификации IP-ядра Verilog);
- EDN — EDIF-список связей;
- XCO — файл параметров блока (создается при его первой генерации);
- MIF — файл инициализации памяти, который автоматически создается для модулей Virtex Block RAM (если для них установлена опция HDL simulation flow).

При создании IP-блока определяется элементная база (семейство FPGA), на которую он ориентирован. Для другой элементной базы его использование

без перегенерации, с соответствующим изменением параметров, будет невозможным.

Определение параметров макроячейки может осуществляться непосредственно в полях диалогового окна **Macro_name**. Параметры могут также загружаться из файла с расширением COE (структура этого файла будет рассмотрена позже). Для выбора файла, из которого будут загружены эти параметры, необходимо воспользоваться кнопкой **Load_Parameters**.

При генерации элементов памяти могут задаваться не только параметры блока памяти, но и начальные значения элементов памяти. Аналогично могут задаваться и коэффициенты FIR-фильтров. Эта информация также может загружаться из файлов COE.

Если для определения каких-либо параметров блока может быть использован COE-файл, то его формат подробно описывается в документации для соответствующего типа блоков.

В общем случае данные в файлах COE представляются следующим образом:

```
KeyWord=Value; Comment
```

Все, что следует в строке после точки с запятой, рассматривается как комментарий.

Для блоков памяти в качестве **KeyWord** используется слово **MemData**, а для блоков памяти Virtex используется **Memory_Initialization_Vector**.

Пример COE-файла параметров для блока RAM приведен в листинге 7.1. Данные в памяти определяются в шестнадцатеричном формате.

Листинг 7.1

```
Component_Name=ram16x12;
Data_Width = 12;
Address_Width = 4;
Depth = 16;
Radix = 16;
memdata=346,EDA,0D6,F91,079,FC8,053,
FE2,03C,FF2,02D,FFB,022,002,01A,005;
```

В этом примере определены следующие параметры:

- `Component_name` — имя компонента;
- `Data_width` — разрядность данных;
- `Address_Width` — разрядность адреса;
- `Depth` — количество слов памяти;
- `Memdata` — содержимое блока памяти.

В листинге 7.2 приведен пример COE-файла для ROM; элементы памяти задаются в десятичном представлении.

Листинг 7.2

```
Component_Name=rom32x8;
Data_Width = 8;
Address_Width = 5;
Depth = 32;
Radix = 10;
memdata=127,127,127,127,127,126,126,126,
125,125,125,4,3,2,0,-1,-2,-4,-5,-6,-8,-9,
-11,-12,-13,-38,-39,-41,-42,-44,-45,-128;
```

В листинге 7.3 приведен пример однопортового ОЗУ (RAM), для которого определяются начальные значения не для всех ячеек. Параметр `Default_Data` позволяет задать значение ячейки по умолчанию.

Листинг 7.3

```
Component_Name = v_spbram;
Depth = 256;
Data_Width = 32;
Radix = 16;
Default_Data = FFF;
MEMORY_INITIALIZATION_VECTOR =
FF0,F0F,0FF,FF4,F4F,4FF,FF8,F8F,8FF;
```

Пример файла спецификации параметров для генерации макроячейки (IP-блока) двухпортового ОЗУ RAM для FPGA семейства Virtex приведен в листинге 7.4.

Листинг 7.4

```
Component_Name=v_dpbram;
Depth_A = 4096;
Data_Width_A = 16;
Depth_B = 1024;
Data_Width_B = 64;
Radix = 2;
Default_Data = 10101010;
```

```
MEMORY_INITIALIZATION_VECTOR=
1111111111111110,
1111111111111101,
1111111111111011,
1111111111110111;
```

Теперь рассмотрим, как осуществляется интеграция сгенерированной макроячейки (IP-блока) в модель проектируемого устройства.

Файлы VEO и VHO, сгенерированные CoreGen, содержат информацию, которая может быть использована пользователем для интеграции блока в модель. В модели проектируемого устройства этот блок будет рассматриваться как "черный ящик". Описание блока на поведенческом уровне содержится в библиотеках Core Generator; указатель на него содержится в VHO-файле (для VHDL) или VEO-файле (для Verilog).

В общем случае VHO-файл имеет следующую структуру:

- XilinxCoreLib LIBRARY Declaration — описание библиотек;
- COMPONENT Declaration — описание компонента;
- Instantiation template — шаблон назначений входных параметров;
- CONFIGURATION Declaration with VHDL generics — конфигурационная декларация.

Рассмотрим процесс интеграции блока в модель на примере поведенческого моделирования, синтеза и реализации проекта с использованием восьмиразрядного сумматора.

Сначала надо выполнить генерацию макроячейки сумматора (IP-блока), предварительно указав значения полей Design Entry, Vendor, и Behavioral Simulation settings в соответствии с выбранным типом проекта.

Далее, в диалоговом окне, открывшемся при выборе элемента — сумматора, нужно определить его параметры — разрядность входов данных (например, 8).

В листинге 7.5 приведен пример VHO-файла для восьмиразрядного сумматора (текст и комментарии сгенерированы автоматически, программой CoreGen).

Листинг 7.5

```
--User: Make sure that these statements appear
--above the top-level entity declaration in your VHDL design...
--LIB_TAG
Library XilinxCoreLib;
Use XilinxCoreLib.null_comp.all;
-- LIB_TAG_END
-- User: Make sure that this statement appears
```

```
-- in the architecture header in your VHDL design...
-- COMP_TAG
component adder8
port (
a: IN std_logic_VECTOR(7 downto 0);
b: IN std_logic_VECTOR(7 downto 0);
c: IN std_logic;
ce: IN std_logic;
ci: IN std_logic;
clr: IN std_logic;
s: OUT std_logic_VECTOR(8 downto 0));
end component;
-- COMP_TAG_END
-- User: Make sure that this statement appears
-- in the architecture body in your VHDL design,
-- substituting your own instance name where shown.
-- Do not forget to change the net names in the port map
-- to your own design's net names.
-- INST_TAG
your_instance_name : adder8
port map (
a => a,
b => b,
c => c,
ce => ce,
ci => ci,
clr => clr,
s => s);
-- INST_TAG_END
-- User: Make sure that this text appears
-- within the top-level configuration body in your VHDL design,
-- for example:
--
-- configuration cfg_top of top_level is
-- for arch_name
-- <Insert text here>
-- end for;
-- end cfg_top;
--
-- CONF_TAG
```

```
for all : adder8 use entity XilinxCoreLib.null(behavioral)
generic map(
signed => true,
input_width => 8);
end for;
CONF_TAG_END
```

Note: The lines between the two markers, -- CONF_TAG, and -CONF_TAG_END:

```
-CONF_TAG
for all : adder8 use entity XilinxCoreLib.null(behavioral)
generic map(
signed => true,
input_width => 8);
end for;
-- CONF_TAG_END
--These lines must be added to a CONFIGURATION statement in your
--VHDL test fixture file or top level design file.
```

После того как блок сгенерирован, для поведенческого моделирования необходимо выделить поведенческую модель блока и включить ее в состав проекта. Созданное генератором CoreGen описание компонента необходимо включить в модель на соответствующем уровне иерархии.

Для этого необходимо скопировать секции `component`, `instance` и `configuration declarations` в текст описания на том уровне иерархии, на который включается компонент. Затем в шаблон секции `instance` и `configuration declarations` необходимо вписать фактические имена сигналов и констант, используемых в модели для присоединения соответствующего компонента. Далее, при необходимости, можно создать тестовый набор (`Declare a configuration in the testbench`) подобно тому, как это выполняется в OrCAD Express *(см. гл. 6)*.

Конфигурационная декларация необходима только для выполнения поведенческого моделирования. Она окружена директивами компилятору, которые позволяют отключить ее при синтезе и реализации, чтобы позволить при реализации включить в модель оптимизированный список связей, а не тот, который может быть получен по приводимому поведенческому описанию.

Пример того, как рассмотренный выше сумматор (листинг 7.5) может быть добавлен в модель, составленную пользователем, приведен в листинге 7.6 (текст и комментарии на английском языке сгенерированы автоматически, программой CoreGen).

Листинг 7.6

```
library IEEE;
use IEEE.std_logic_unsigned.all;
use IEEE.std_logic_1164.all;
ENTITY myadder8_top IS
PORT (ap : IN std_logic_vector(7 downto 0);
bp : IN std_logic_vector(7 downto 0);
ck: IN std_logic ;
cep: IN std_logic;
cip: IN std_logic;
clrp: IN std_logic;
sp: OUT std_logic_VECTOR (8 downto 0));
END myadder8_top;
ARCHITECTURE use_core of myadder8_top IS
------------------------------------------------------------
---- The MYADDER8 core is used in this design. The core
---- must be declared via a 'component declaration';
---- myadder8.vho provides the component declaration
---- which is cut-and-pasted into the design as
---- shown below.
------------------------------------------------------------
-- Описание компонента скопировано из файла, сгенерированного CoreGen

component myadder8
port (
a: IN std_logic_VECTOR(7 downto 0);
b: IN std_logic_VECTOR(7 downto 0);
c: IN std_logic;
ce: IN std_logic;
ci: IN std_logic;
clr: IN std_logic;
s: OUT std_logic_VECTOR(8 downto 0));
end component;
BEGIN
------------------------------------------------------------
---- The core is instantiated into this design.
---- myadder8.vho provides an instantiation
---- template which must be modified
```

```
---- so that it reflects actual signals used in the
---- design, establishing the connectivity between the
---- core and other logic at this level. The instance
---- of the core must also be given an actual label to
---- replace the dummy "your_instance_name" tag. In this
---- example,it is replaced by "myadder8".
-------------------------------------------------------
-- Имена сигналов, выделенные курсивом, определяются пользователем;
-- сам же шаблон назначения может быть скопирован из файла *.VHO,
-- сгенерированного CoreGen.
myadder8_1 : myadder8
port map (
a => ap,
b => bp,
c => ck,
ce => cep,
ci => cip,
clr => clrp,
s => sp);
end use_core;
-------------------------------------------------------------
A partial configuration statement is found in
---- myadder8.vho.It contains information necessary to
---- link the behavior of the core with the its
---- instantiation. This configuration section from
---- myadder8.vho must be reproduced in this parent
---- design, embedded within the configuration statement
---- of parent design. NOTE: For each core that is
---- instantiated in this design, a unique partial
---- configuration statement must be included.
-------------------------------------------------------
-- Значения параметров, выделенные курсивом, определяются пользователем,
-- сам же шаблон конфигурационной декларации может быть скопирован из
-- файла *.VHO, сгенерированного coregen.

-- synopsys translate_off
-- Это директива компилятору, указывающая, что при синтезе дальнейший текст до
 -- директивы synopsys translate_on, т.е. конфигурационная спецификация должна
-- быть проигнорирована (используется только для выполнения поведенческого
-- моделирования)
```

```
CONFIGURATION cfg_myadder8_top OF myadder8_top IS
FOR use_core
for all : myadder8 use entity
XilinxCoreLib.myadder8firVHT(behavioral)
generic map(
signed => 1,
input_width => 8);
end for;
end for;
end cfg_myadder8_top;
-- synopsys translate_on
```

Пример структуры теста для рассмотренного сумматора может иметь вид, приведенный в листинге 7.7.

Листинг 7.7

```
library IEEE;
use IEEE.std_logic_1164.ALL;
ENTITY testbench is
END testbench;
ARCHITECTURE simulate OF testbench IS
-------------------------------------------------------
---- The parent design, myadder8_top, is instantiated
---- in this testbench. Note the component
---- declaration and the instantiation.
-------------------------------------------------------
COMPONENT myadder8_top
PORT (
ap : IN std_logic_vector(7 downto 0);
bp : IN std_logic_vector(7 downto 0);
ck: IN std_logic ;
cep: IN std_logic;
cip: IN std_logic;
clrp: IN std_logic;
sp: OUT std_logic_VECTOR (8 downto 0));
END myadder8_top;
SIGNAL a_data_input : std_logic_vector(7 DOWNTO 0);
SIGNAL b_data_input : std_logic_vector(7 DOWNTO 0);
SIGNAL clock : std_logic;
```

```
SIGNAL clock_enable : std_logic;
SIGNAL carry_in : std_logic;
SIGNAL clear : std_logic;
SIGNAL sum : std_logic (8 DOWNTO 0);
BEGIN
uut: myadder8_top
PORT MAP (
ap => a_data_input,
bp => b_data_input,
ck => clock,
cep => clock_enable,
cip => carry_in,
clrp=> clear,
sp => sum);
stimulus: PROCESS
BEGIN
-- В этой секции выполняется собственно определение значений сигналов для
-- тестирования.
-------------------------------------------------------
-- Provide stimulus in this section. (not shown here)
-------------------------------------------------------
wait;
end process; -- stimulus
END simulate;
-------------------------------------------------------------
--The configuration, cfg_testbench, of the testbench,
---- reminds the simulator to refer to the configuration
---- statement in the parent design. Note that "work" was
---- was the default library into which the testbench
---- and the parent design were analyzed.
-------------------------------------------------------------
CONFIGURATION cfg_testbench OF testbench IS
FOR simulate
for all : myadder8_top
use configuration work.cfg_myadder8_top;
end for;
END for;
END cfg_testbench;
```

После того как модель откорректирована, можно выполнять моделирование, синтез и реализацию.

Описание автоматически сгенерированного IP-блока на поведенческом уровне по умолчанию недоступно для чтения пользователем. Однако оно может быть получено с использованием утилиты **get_models**.

Утилита **get_models** вызывается из командной строки. Она может быть использована как для получения описания конкретного блока, так и для получения описания множества блоков, представленных в библиотеках.

Обращение к утилите **get_models** имеет следующий синтаксис:

```
get_models [ -verilog | -vhdl ] <destination_directory>
```

Здесь `destination_directory` — каталог назначения. Внутри этого каталога генератор создает группу подкаталогов, каждый из которых соответствует одному производителю. Имена этих подкаталогов имеют следующую структуру: <**Vendor**>**CoreLib**; например, **XilinxCorelib**.

Приложение 1

Некоторые особенности использования конструкций языка VHDL при Синтезе

Цель данного приложения — не столько дать справочную информацию, сколько привлечь внимание разработчика к определенным категориям средств языка VHDL при переходе от Моделирования к Синтезу. Конкретные пакеты автоматизации проектирования могут содержать свои ограничения на использование конструкций VHDL в программах-спецификациях, свою специфику их трактовки при Синтезе.

Таблица П1.1

Конструкция языка VHDL	Специфика при Синтезе
Описание сигналов и переменных	Диапазон значений типов сигналов и данных определяет разрядность элементов синтезируемого устройства. Явная спецификация диапазонов значений декларируемых сигналов и данных помогает синтезирующим компиляторам оптимизировать генерируемую реализацию устройства.
Начальные значения в декларации сигналов	При синтезе игнорируются
Начальные значения в декларации переменных	При синтезе игнорируются
Начальные значения в описании портов объекта	При синтезе игнорируются
Перечислимые типы	Использование перечислимых типов, вместо цифровых кодировок, дает лучшие возможности оптимизации генерируемой реализации устройства синтезирующими компиляторами
Тип `real`	Не поддерживается при синтезе
Тип `file`	Не поддерживается при синтезе

***Таблица П1.1** (окончание)*

Конструкция языка VHDL	Специфика при Синтезе
Сигналы, декларированные в пакетах	Не поддерживаются при синтезе
Переменные	Если значение переменной в теле процесса используется прежде, чем присваивается, то переменной при синтезе сопоставляется элемент памяти
Выражения	Синтезирующие компиляторы могут быть чувствительны к форме записи выражения. Использование скобок позволяет явно управлять порядком выполнения операций в выражении

Атрибуты:
`behavior`
`structure`
`last_event`
`last_active`
`transaction`
	Не поддерживаются при синтезе

Операторы присваивания значения сигналу	Указанные в операторе задержки – игнорируются Атрибуты `inertial`, `reject`, `transport` игнорируются. Как правило, игнорируется также атрибут `unaffected`
Условные операторы присваивания значений сигналам	Существенную роль играет то, каким сигналам присваиваются значения в альтернативных ветвях условных конструкций Присваивание значений одинаковым наборам сигналов в каждой из ветвей позволяет избегать появления лишних элементов памяти в схеме Вложение операторов **`if`** ведет к увеличению аппаратных затрат и ухудшению временных характеристик схем
Операторы цикла	Синтезируемы только циклические конструкции, число итераций которых известно при компиляции. Как правило, синтезируемыми являются только циклы **`for...loop`**
Оператор **`assert`**	Игнорируется при синтезе
Список чувствительности процесса	Явно заданный список чувствительности процесса при синтезе игнорируется. При синтезе считается, что список чувствительности составляют *все* входные сигналы

Приложение 2

VHDL, поддерживаемый OrCAD Express 9.1, и его отличия от стандарта IEEE 1076-1993

Таблица П2.1. Объекты моделирования и конфигурации

Имя конструкции	Моделирование	Компиляция	Комментарии
Декларация **Entity**	+	+	
Заголовок **Entity**	+	+	
Generic	+	+	
Port	+	+	При синтезе режим связывания не поддерживается
Декларации в описании (**Entity declarative part**)	+	+	
Операторы в описании (**Entity statement part**)	+	+	
Тело архитектурного описания (**Architecture bodies**)	+	+	
Декларации в описании архитектуры (**Architecture declarative part**)	+	+	
Операторы в описании архитектуры (**Architecture statement part**)	+	+	

Таблица П2.1 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Декларации конфигурации (**Configuration declaration**)	+	+	
Конфигурации блоков (**Block configurations**)	+	–	
Конфигурации компонентов (**Component configurations**)	+	–	

Таблица П2.2. Подпрограммы и пакеты

Имя конструкции	Моделирование	Компиляция	Комментарии
Декларации подпрограмм (**Subprogram declarations**)	+	+	
Формальные параметры (**Formal parameters**)	+	+	
Константы и переменные в качестве параметров (**Constant and variable parameters**)	+	+	
Сигналы в качестве параметров (**Signal parameters**)	+	+	
Файлы в качестве параметров (**File parameters**)	+	–	
Тела подпрограмм (**Subprogram bodies**)	+	+	

Таблица П2.2 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Перегрузка подпрограмм (**Subprogram overloading**)	–	+	
Перегрузка операторов (**Operator overloading**)	+	+	
Сигнатуры (**Signatures**)	–	–	Не поддерживаются
Разрешающие функции (**Resolutions functions**)	–	+	
Декларация пакетов (**Package declaration**)	+	+	При синтезе логики использование глобальных сигналов, не являющихся константами, не поддерживается. При моделировании не поддерживается использование глобальных сигналов в списке связей
Тело пакета (**package body**)	+	+	
Правила согласования (**Conformance rules**)	–	–	Не поддерживаются
Процедуры и функции (**Functions and procedures**)	+	+	

Таблица П2.3. Типы

Имя конструкции	Моделирование	Компиляция	Комментарии
Скалярные типы (**Scalar types**)	+	+	
Перечислимые типы (**Enumeration types**)	+	+	

Таблица П2.3 (продолжение)

Имя конструкции	Моделирование	Компиляция	Комментарии
Предопределенные перечислимые типы (**Predefined enumeration types**)	+	+	
Целые типы (**Integer types**)	+	+	
Предопределенные целые типы (Predefined **integer types**)	+	+	
Физические типы (**Physical types**)	+	–	
Предопределенные физические типы (**Predefined physical types**)	+	+	
Типы с плавающей запятой (**Floating point types**)	–	+	При синтезе типы и объекты с плавающей точкой могут использоваться только в константных выражениях
Предопределенные типы с плавающей запятой (**Predefined floating point types**)	–	+	
Составные типы (**Composite types**)	+	+	
Массивы (**Array types**)	+	+	
Границы изменения индексов и диапазоны (**Index constrains and discrete ranges**)	+	+	
Предопределенные типы-массивы (**Predefined array types**)	+	+	

Таблица П2.3 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Записи (**Record types**)	+	+	
Типы **Access** (**Access types**)	–	+	При синтезе эти типы игнорируются
Неполные декларации типов (**Incomplete type declarations**)	+	–	
Размещение и освобождение объектов (**Allocation and deallocation of object**)	+	–	
Файловые типы (**File types**)	+	+	При синтезе эти типы игнорируются
Операции с файловыми типами (**File operations**)	+	+	При синтезе эти операции игнорируются

Таблица П2.4. Декларации

Имя конструкции	Моделирование	Компиляция	Комментарии
Декларация типов (**Type declaration**)	+	+	
Декларация подтипов (**Subtype declaration**)	+	+	Имена решающих функций в именах типов игнорируются
Объекты (**Object**)	+	–	
Декларация типов (**Object declarations**)	+	+	
Декларация сигналов (**Signal declaration**)	+	+	При моделировании сторожевые сигналы (ключевые слова **register** и **bus**) не поддерживаются
Декларация переменных (**Variable declaration**)	+	+	

Таблица П2.4 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Декларация файлов (**File declarations**)	+	+	Поддерживаются стандарты описания файлов 87 и 93
Декларация интерфейса (**Interface declarations**)	+	+	
Списки ассоциации (**Association lists**)	+	+	
Декларация кличек (**Alias declarations**)	+	+	Кличка не может состоять из одной буквы или операторного символа; указание подтипа не обязательно; спецификация сигнатуры не поддерживается
Клички объектов (**Object aliases**)	+	+	
Клички для конструкций, не являющихся объектами (**Non-object aliases**)	+	–	Не поддерживаются при синтезе
Декларация атрибутов (**Attribute declarations**)	+	+	
Декларация шаблонов групп (**Component declarations**)	+	+	
Декларация интерфейса (**Group template declarations**)	–	–	Не поддерживается
Декларация групп (**Group declarations**)	–	–	Не поддерживается

Таблица П2.5. *Спецификации*

Имя конструкции	Моделирование	Компиляция	Комментарии
Спецификация атрибутов (**Attribute specification**)	+	+	
Индикация связывания (**Binding indication**)	+	–	
Отображение портов (**Port map aspects**)	+	–	
Отображение параметров **generic** (**Generic map**)	+	–	
Индикация связывания по умолчанию (**Default binding indication**)	+	–	
Спецификация отсоединения (**Disconnection specification**)	+	–	

Таблица П2.6. *Имена*

Имя конструкции	Моделирование	Компиляция	Комментарии
Имена (**Names**)	+	+	
Имена простые (**Simple names**)	+	+	
Имена селективные (**Selected names**)	+	+	
Имена с индексами (**Indexed names**)	+	+	
Имена слоев (**Slice names**)	+	+	Слои — одномерные выборки из многомерных массивов
Имена атрибутов (**Attribute names**)	+	+	Сигнатуры не поддерживаются; проверка ошибок ограничена; имена массивов не могут использоваться в качестве префиксов

Таблица П2.7. *Выражения*

Имя конструкции	Моделирование	Компиляция	Комментарии
Выражения (**Expressions**)	+	+	
Операторы (**Operators**)	+	+	
Логические операторы (**Logical operators**)	+	+	
Операторы отношений (**Relational operators**)	+	+	Для векторных типов реализованы только операторы = и /=
Операторы сложения/вычитания (**Adding operators**)	+	+	
Операторы работы со знаком (**Sign operators**)	+	+	
Операторы умножения/деления (**Multiplying operators**)	+	+	
Прочие операторы (**Miscellaneous operators**)	+	+	При синтезе частичная поддержка
Операнды (**Operands**)	+	+	
Литералы (**Literals**)	+	+	
Агрегаты (**Aggregates**)	+	+	При моделировании не поддерживается дискретный интервал в условии выбора списка ассоциации элементов
Агрегаты **Record** (**Record aggregates**)	+	–	Именные ассоциации не поддерживаются
Агрегаты **Array** (**Array aggregates**)	+	+	
Вызовы функций (**Function calls**)	+	+	

Таблица П2.7 *(окончание)*

Имя конструкции	Моделирование	Компиляция	Комментарии
Преобразования типов (**Type conversions**)	+	+	При синтезе поддерживается конвертирование только между связанными типами
Статические выражения (**Static expressions**)	+	+	
Выражения общего вида (**Universal expressions**)	+	+	

Таблица П2.8. *Последовательные операторы*

Операторы	Моделирование	Компиляция	Комментарии
Оператор `Wait`	+	+	Если оператор содержит множество секций, то реально выполняется только первая из них, остальные игнорируются
Операторы `Assert`	+	+	
Операторы `Report`	—	—	Не поддерживается
Оператор присваивания значений сигналам	+	+	При моделировании и синтезе не поддерживаются: `REJECT`, назначения агрегатам, ключевое слово **`UNAFFECTED`**. При компиляции игнорируются задержки
Редактирование выходных временных диаграмм проектов	+	+	При синтезе не поддерживаются: значение `NULL`, rejection ограничения. При моделировании не поддерживаются rejection ограничения
Операторы присваивания значений переменным	+	+	
Операторы присваивания значений переменным—массивам	+	+	
Операторы вызова процедур	+	+	

Таблица П2.8 (окончание)

Операторы	Моделирование	Компиляция	Комментарии
Операторы `If`	+	+	
Операторы `Case`	+	+	
Операторы `Loop`	+	+	При синтезе циклы `for`, которые содержат `wait until`, не поддерживаются
Операторы `Next`	—	—	
Операторы `Exit`	—	—	
Операторы `Return`	+	+	
Операторы `Null`	+	+	

Таблица П2.9. Параллельные операторы

Операторы	Моделирование	Компиляция	Комментарии
`Block`	+	+	При моделировании, в блоках не поддерживаются заголовки
`Process`	+	+	Отложенные процессы не поддерживаются
Операторы параллельного вызова процедур	+	+	
`Assert`, параллельные операторы	+	+	Ключевое слово `POSTPONED` не поддерживается (при исполнении команд компиляции эти команды игнорируются)
Параллельные операторы присваивания значений сигналам	+	+	
Условные операторы присваивания значений сигналам	+	+	
Селективные операторы присваивания значений сигналам	–	–	Не поддерживаются

Таблица П2.9 (окончание)

Операторы	Моделирование	Компиляция	Комментарии
Операторы включения компонентов (**Component instantiation**)	+	+	Ключевые слова **COMPONENT** и **ENTITY** не поддерживаются (не различаются); конфигурации не поддерживаются
Операторы Generate	+	+	Операторы **FOR GENERATE**, которые имеют диапазон, задаваемый выражением "name'**range**", не поддерживаются

Таблица П2.10. Контекст и видимость

Имя конструкции	Моделирование	Компиляция	Комментарии
Секция деклараций (**Declarative region**)	+	+	
Область деклараций (**Scope of declarations**)	+	+	
Области видимости (**Visibility**)	+	+	
Конструкция **Use** (**Use clauses**)	+	+	Некоторые формы, такие как **use library.all,** не поддерживаются
Контекст перегрузки решающих функций (**Context of overload resolution**)	+	+	

Таблица П2.11. Организация проекта

Имя конструкции	Моделирование	Компиляция	Комментарии
Составные части проекта (**Design units**)	+	+	
Библиотеки проекта (**Design libraries**)	+	+	

Таблица П2.11 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Конструкции определения контекста проекта (**Context clauses**)	+	+	
Порядок анализа проекта (**Order of analysis**)	+	+	Порядок появления модулей схем внутри файла должен быть корректным

Таблица П2.12. Выполнение

Имя конструкции	Моделирование	Компиляция	Комментарии
Развертывание иерархии проекта (**Elaboration of a design hierarchy**)	+	+	
Обработка заголовка блока (**Elaboration of a block header**)	+	+	При моделировании отображения портов обобщений в секциях **Block** игнорируются
Конструкция **Generic** в заголовке блока (**Generic clause in a block header**)	+	+	
Конструкция **Generic map** в заголовке блока (**Generic map aspect in a block header**)	+	+	
Конструкция **Port** в заголовке блока (**Port clause in a block header**)	+	+	
Конструкция **Port map** в заголовке блока (**Port map aspect in a block header**)	+	+	

***Таблица П2.12** (продолжение)*

Имя конструкции	Моделирование	Компиляция	Комментарии
Обработка области деклараций в Блоке (**Elaboration of a declarative part**)	+	+	При моделировании атрибут `Foreign` не поддерживается
Обработка деклараций (**Elaboration of a declaration**)	+	+	
Обработка декларации и тела описания подпрограмм (**Subprogram declarations and bodies**)	+	+	
Обработка декларации типов (**Type declarations**)	+	+	
Обработка декларации подтипов (**Subtype declarations**)	+	+	
Обработка декларации объектов (**Object declarations**)	+	+	
Обработка декларации кличек (**Alias declarations**)	+	+	
Обработка декларации атрибутов (**Attribute declarations**)	+	+	
Обработка декларации компонентов (**Component declarations**)	+	+	

Таблица П2.12 (продолжение)

Имя конструкции	Моделирование	Компиляция	Комментарии
Обработка спецификаций (**Elaboration of a specification**)	+	+	
Обработка спецификаций атрибутов (**Attribute specifications**)	+	+	При моделировании — ограниченная поддержка для спецификации атрибутов, определяемых для объектов-массивов
Обработка спецификаций конфигураций (**Configuration specifications**)	+	+	
Обработка спецификаций отсоединения (**Disconnection specifications**)	+	+	
Обработка операторов (**Elaboration of a statement part**)	+	+	
Обработка конструкции **block** (**Block statements**)	+	+	
Обработка операторов **generate** (**Generate statements**)	+	+	
Обработка операторов включения компонентов (**Component instantiation statements**)	+	+	
Другие параллельные операторы	+	+	
Исполнение модели (**Execution of a model**)	+	+	

Таблица П2.12 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Драйверы, источники сигнала (**Drivers**)	+	+	
Распространение значений сигналов (**Propagation of signal values**)	+	+	
Изменение неявных сигналов (**Updating implicit signals**)	+	+	

Таблица П2.13. Лексические элементы

Имя конструкции	Моделирование	Компиляция	Комментарии
Набор символов (**Character set**)	+	+	
Лексемы и разделители (**Lexical elements, separators, and delimiters**)	+	+	
Идентификаторы (**Identifiers**)	+	+	
Абстрактные литералы (**Abstract literals**)	+	+	
Десятичные литералы (**Decimal literals**)	+	+	
Базовые литералы (**Base literals**)	+	+	При синтезе все то, что связано с типами с плавающей точкой, не поддерживается
Символьные литералы (**Character literals**)	+	+	

Таблица П2.13 (окончание)

Имя конструкции	Моделирование	Компиляция	Комментарии
Строковые литералы (**String literals**)	+	+	
Литералы — битовые строки (**Bit string literals**)	+	+	
Комментарии (**Comments**)	+	+	
Зарезервированные слова (**Reserved words**)	+	+	
Допустимые замены символов (**Allowable replacement of characters**)	+	+	

Таблица П2.14. Предопределенное окружение языка

Имя конструкции	Моделирование	Компиляция	Комментарии
Атрибуты `left, right, high, low, ascending, length, delayed, stable, event, last_event, last_value, val, pos, succ, pred, leftof, rightof, range, reverse_range`	+	+	
Атрибуты `behavior, structure, last_event, last_active, transaction`	+	-	При синтезе атрибуты `BEHAVIOR, STRUCTURE, LAST_EVENT, LAST_ACTIVE` и `TRANSACTION` не поддерживаются
Атрибуты `base, image, value, quiet, transaction, active, last_active, driving, driving_value, simple_name, instance_value, and path_name`	+	+	

Таблица П2.14 *(окончание)*

Имя конструкции	Моделирование	Компиляция	Комментарии
Пакет STANDARD	+	+	Поддерживается неполный список единиц измерения времени; командами компиляции не поддерживается функция now
Пакет TEXTIO	+	+	При синтезе операции с файлами игнорируются

Таблица П2.15. *IEEE и VITAL библиотеки*

Имя конструкции	Моделирование	Компиляция	Комментарии
Типы и подтипы EXEMPLAR пакета Exemplar Logic	+	+	
Функции EXEMPLAR пакета Exemplar Logic	+	+	
Процедуры EXEMPLAR пакета Exemplar Logic package EXEMPLAR procedures	+	+	
Типы и подтипы EXEMPLAR_1164 пакета Exemplar Logic	+	+	
Функции EXEMPLAR_1164 пакета Exemplar Logic	+	+	
Процедуры EXEMPLAR_1164 пакета Exemplar Logic	+	+	
Типы и подтипы STDSYNTH пакета Exemplar Logic	+	+	

Таблица П2.15 *(окончание)*

Имя конструкции	Моделирование	Компиляция	Комментарии
Процедуры STDSYNTH пакета Exemplar Logic	+	+	
Функции STDSYNTH пакета Exemplar Logic	+	+	
Типы и подтипы пакета IEEE std_logic_1164	+	+	При синтезе поддерживаются *только* значения "0," "1," "X," и "Z"
Функции пакета IEEE std_logic_1164	+	+	
Пакет IEEE numeric_std	+	+	Не поддерживаются функции и операторы **rem**, **mod**, **rol**, **ror**, **sll**, **srl**, "/", **sla**, и **sra**
Пакеты IEEE vital_primitives и vital_timing (Vital95)	+	–	
Библиотеки Vital libraries	+	–	При моделировании поддерживаются все Vital 95 (IEEE Standard Vital 1076.4) библиотеки

Приложение 3

VHDL, поддерживаемый Foundation Express 2.1, и его отличия от стандарта IEEE 1076-1993

Таблица П3.1. *Объекты моделирования и конфигурации*

Имя конструкции	Анализ	Синтез	Комментарии
Декларация **entity** (**Entity declaration**)	+	+	
Заголовок **entity** (**Entity header**)	+	+	
Константы-параметры (**Generic**)	+	+	Поддерживаются только для типа `INTEGER`
Порты (**Port**)	+	+	Значения по умолчанию игнорируются
Секция декларации в **entity** (**Entity declarative part**)	+	+	
Секция операторов в **entity** (**Entity statement part**)	+	—	Игнорируется
Архитектурное описание (**Architecture bodies**)	+	+	Допускаются множественные архитектуры
Секция деклараций в архитектурном описании (**architecture declarative part**)	+	+	

Таблица П3.1 *(окончание)*

Имя конструкции	Анализ	Синтез	Комментарии
Секция операторов в архитектурном описании (**Architecture statement part**)	+	+	
Конфигурационная декларация (**Configuration declaration**)	+	+	Поддерживаются только для сущностей верхнего уровня иерархии
Конфигурации блоков (**Block configurations**)	+	–	Поддерживаются только для сущностей верхнего уровня иерархии
Конфигурации компонентов (**Component configurations**)	–	–	
Спецификация атрибутов (**Attribute specifications**)	–	–	
Конструкция **use** (**use clauses**)	–	–	
Вложенные конфигурации блоков (**nested block configurations**)	–	–	

Таблица П3.2. *Подпрограммы и пакеты*

Имя конструкции	Анализ	Синтез	Комментарии
Пакеты (**Package**)	+	+	
Библиотеки (**library**)	+	+	Поддерживается раздельная компиляция (separate compilation)
Подпрограммы (**subprogram**)	+	+	Значения по умолчанию для параметров не поддерживаются Рекурсия не поддерживается, если вложенность не ограничивается статическим значением Решающие функции поддерживаются только для монтажной логики (wired-logic) и функций с тремя состояниями (three state functions) Подпрограммы могут декларироваться только в пакетах и в декларативных частях архитектур

Таблица П3.3. *Типы*

Имя конструкции	Анализ	Синтез	Комментарии
Перечислимые типы (**Enumeration**)	+	+	
Целые типы (**Integer**)	+	+	Не поддерживается арифметика с неограниченной точностью (infinite-precision arithmetic) Производится автоматическое преобразование типа Integer в Bit Vector соответствующего размера
Физические типы (**Physical**)	+	–	Игнорируются
Типы с плавающей запятой (**Floating**)	–	–	Использование типов с плавающей точкой не поддерживается, за исключением констант, используемых с атрибутами, введенными в Foundation Express
Массивы (**Array**)	+	+	Диапазон может определяться только на базе INTEGER. Многомерные массивы не поддерживаются, но поддерживаются массивы с элементами типа массив
Записи (**Record**)	+	+	
Указательные типы (**Access**)	+	–	Не поддерживаются
Файлы (**File**)	+	–	Не поддерживаются
Неполные декларации типов (**Incomplete type declarations**)	–	–	Не поддерживаются

Таблица П3.4. *Декларации*

Имя конструкции	Анализ	Синтез	Комментарии
Декларация констант	+	+	Не поддерживаются отложенные декларации констант
Декларация сигналов	+	+	REGISTER и BUS декларации не поддерживаются Решающие функции поддерживаются для wired и three-state функций Начальные значения не поддерживаются

Таблица П3.4 (окончание)

Имя конструкции	Анализ	Синтез	Комментарии
Декларация переменных	+	+	Начальные значения не поддерживаются
Разделяемые переменные (**Shared variable**)	+	+	
Файлы	—	—	Не поддерживаются
Интерфейсная декларация (**Interface**)	+	+	BUFFER автоматически преобразуется в OUT. LINCAGE автоматически преобразуется в INOUT
Псевдонимы (**Alias**)	+	+	Не поддерживаются клички для типов
Декларация компонентов (**Component**)	+	+	Поддерживается только декларация компонентов, ссылающаяся на действительные имена сущностей
Атрибуты (**Attribute**)	+	+	Использование атрибутов, определенных пользователем, не поддерживается

Таблица П3.5. Спецификации

Имя конструкции	Анализ	Синтез	Комментарии
Атрибуты (**Attribute**)	+	+	Пользователь может определять новые атрибуты, но их использование не поддерживается — синтаксической ошибки не выдается, но при моделировании фактически игнорируется
Спецификация конфигурации (**Configuration**)	+	—	Спецификация конфигурации не поддерживается
Спецификация отсоединения (**Disconnection**)	—	—	Спецификация отсоединения не поддерживается

Таблица П3.6. Имена

Имя конструкции	Анализ	Синтез	Комментарии
Простые имена (**Simple**)	+	+	
Селективные имена (**Selected**)	+	+	Поддерживается только внутри USE CLAUSE

Таблица П3.6 *(окончание)*

Имя конструкции	Анализ	Синтез	Комментарии
Операторные символы (**Operator symbol**)	+	+	
Индексированные имена (**Indexed**)	+	+	Не поддерживается для выходного параметра процедуры неограниченного типа
Имена слоев (**Slice**)	+	+	Не поддерживается для выходного параметра процедуры неограниченного типа, если действительный параметр не является идентификатором
Атрибуты (**Attribute**)			Полностью поддерживаются: `base`, `left`, `right`, `high`, `low`, `range`, `reverse_range` и `length`

	+	+	Атрибуты `event` и `stable` поддерживается только в конструкциях типа:
			`If clk'event and clk='1' then`
			`If not clk'stable and clk='0'`
			`Wait until clk'event and clk='0'`
			`Wait until not clk'stable and clk='1'`

Таблица П3.7. *Операторы*

Имя конструкции	Анализ	Синтез	Комментарии
Логические операторы (**Logical**)	+	+	
Операторы отношения (**Relational**)	+	+	
Аддитивные операторы (**Addition**)	+	+	Поддерживается и конкатенация, и арифметическое сложение
Операторы работы со знаком (**Signing**)	+	+	

Таблица П3.7 (окончание)

Имя конструкции	Анализ	Синтез	Комментарии
Мультипликативные операторы (**Multiplying**)	+	+	
Деление (**Division**)	+	+	Операторы /, mod, rem поддерживаются только в том случае, если оба операнда являются константами или же правый операнд (делитель) является константой, кратной 2
Прочие операторы (**Miscellaneous**)	+	+	Abs — полностью поддерживается; ** (возведение в степень) поддерживается, если оба операнда являются константами или же левый операнд равен 2
Перегрузка операторов (**Operator overloading**)	+	+	

Таблица П3.8. Операнды и выражения

Имя конструкции	Анализ	Синтез	Комментарии
Базированный литерал (**Based literal**)	+	+	
Нулевой литерал (**Null literal**)	+	+	Нулевые диапазоны и массивы не поддерживаются
Физический литерал (**Physical literal**)	+	–	Игнорируется
Строка (**String**)	+	+	
Агрегат (**Aggregate**)	+	+	Не поддерживается использование типов для выбора агрегатов. Агрегаты для записей поддерживаются
Вызов функции (**Function call**)	+	+	Function conversions на входных портах не поддерживается, потому что преобразование типов для формальных портов в спецификации соединений (port map) не поддерживается
Преобразование типов (**Type conversion**)	+	+	

Таблица П3.8 *(окончание)*

Имя конструкции	Анализ	Синтез	Комментарии
Статическое выражение **(Static expression)**	+	+	
Выражение общего вида **(Universal expression)**			Операнды с плавающей запятой не поддерживаются, кроме как в определении атрибута, распознаваемого в Foundation Express.
	+	+	Infinite-precision Не поддерживаются выражения с неограниченной точностью.
			Точность ограничивается 32-мя битами; все промежуточные результаты преобразуются к INTEGER

Таблица П3.9. *Последовательные операторы*

Имя конструкции	Анализ	Синтез	Комментарии
`wait`	+	+	Поддерживается только для следующих форм: **`wait`** until *clock* = *VALUE*; **`wait`** until *clock*'event **and** *clock* = *VALUE*; **`wait`** until *not clock*'stable **and** *clock* = *VALUE*; *VALUE* может принимать значения '0' или '1'
`Assert`	+	—	Игнорируется
`Report`	+	—	Игнорируется
Метка оператора (statement label)	+	—	Игнорируется
Операторы работы с сигналами	+	+	Определение значений сторожевых сигналов игнорируется. Множественные присвоения значений сигналам игнорируются
`Variable`	+	+	
Вызов процедуры (procedure call)	+	+	Преобразование типов формальных параметров не поддерживается. Назначение одиночного бита векторному порту не поддерживается

Таблица П3.9 *(окончание)*

Имя конструкции	Анализ	Синтез	Комментарии
`If`	+	+	
`Case`	+	+	
`Loop`			Конструкция `For...loop` не допускает использование внутри тела цикла оператора `wait`.
	+	+	Если используется конструкция `While...loop`, то внутри цикла должен быть хотя бы один оператор `wait`
			Конструкция `Loop`, без указания схемы повторений, поддерживается, но тело цикла должно содержать хотя бы один оператор `Wait`
`Next`	+	+	
`Exit`	+	+	
`Return`	+	+	
`Null`	+	+	

Таблица П3.10. *Параллельные операторы*

Имя конструкции	Анализ	Синтез	Комментарии
Block	+	+	Сторожевые сигналы в блоке поддерживаются
			Порты и Generics не поддерживаются
Process	+	+	Список чувствительности при синтезе игнорируется
Параллельный вызов процедуры **(Concurrent procedure call)**	+	+	
Параллельный оператор **assert (Concurrent assertion)**	+	—	Игнорируется
Параллельное присваивание значений сигналам **(Concurrent signal assignment)**	+	+	Ключевое слово `guarded` — поддерживается Ключевое слово `transport` — игнорируется Множественное присваивание значений не поддерживается

Таблица П3.10 *(окончание)*

Имя конструкции	Анализ	Синтез	Комментарии
Включение компонента (**Component instantiation**)	+	+	Преобразование типов формальных портов в спецификации соединения не поддерживается
Оператор `Generate`	+	+	

Таблица П3.11. *Предопределенное окружение языка*

Имя конструкции	Анализ	Синтез	Комментарии
Тип severity_level	—	—	Не поддерживается
Тип `time`	+	—	Игнорируется
Функция `now`	—	—	Не поддерживается
Пакет TEXTIO	—	—	Не поддерживается
Предопределенные атрибуты (Predefined attributes)	+	+	Частично поддерживаются

Приложение 4

Пакет std_logic_1164

В этом пакете собраны типы, рекомендуемые для моделирования физических сигналов.

```
type std_ulogic is ('U','X','0','1','Z','W','L','H','-');
type std_ulogic_vector is array (natural range <>) of std_ulogic;
```

На базе этих типов, с помощью функции

```
resolved (s:std_ulogic_vector) return std_ulogic
```

определяются вычисляемые типы:

```
subtype std_logic is resolved std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
```

Рекомендуется использовать для описания сигнала вычисляемые типы, даже если сигнал имеет только один источник. Это связано с тем, что производители, как правило, проводят оптимизацию схем, в результате которой сигнал, имевший до этого один источник, будет иметь их несколько.

Функция `resolved` определена следующим образом:

```
type stdlogic_table is array (std_ulogic, std_ulogic) of std_ulogic;
constant resolution_table: stdlogic_table:=
--      'u'   'x'   '0'   '1'   'z'   'w'   'l'   'h'      '-'
(('u',  'u',  'u',  'u',  'u',  'u',  'u',  'u',  'u'),  --'u'
 ('u',  'x',  'x',  'x',  'x',  'x',  'x',  'x',  'x'),  --'x'
 ('u',  'x',  '0',  'x',  '0',  '0',  '0',  '0',  'x'),  --'0'
 ('u',  'x',  'x',  '1',  '1',  '1',  '1',  '1',  'x'),  --'1'
 ('u',  'x',  '0',  '1',  'z',  'w',  'l',  'h',  'x'),  --'z'
 ('u',  'x',  '0',  '1',  'w',  'w',  'w',  'w',  'x'),  --'w'
 ('u',  'x',  '0',  '1',  'l',  'w',  'l',  'w',  'x'),  --'l'
 ('u',  'x',  '0',  '1',  'h',  'w',  'w',  'h',  'x'),  --'h'
 ('u',  'x',  'x',  'x',  'x',  'x',  'x',  'x',  'x'),  --'-'
);
function resolved (s: std_ulogic_vector) return std_ulogic is
  veriable result: std_ulogic:='z';
```

```
begin
  if s'length = 1 then
    return s(s'low);
  else
    for i in s'range loop
      result:=resolution_table(result, s(i));
    end loop;
  end if;
  return result;
end function resolved;
```

Пакет std_logic_1164 содержит также набор дополнительных типов:

```
subtype X01 is resolved std_ulogic range 'X' to '1';
--('x', '0', '1')
subtype X01Z is resolved std_ulogic range 'X' to 'Z';
--('x', '0', '1', 'Z')
subtype UX01 is resolved std_ulogic range 'U' to '1';
--('U', 'x', '0', '1')
subtype UX01Z is resolved std_ulogic range 'U' to 'Z';
--('U', 'x', '0', '1', 'Z')
```

Приложение 5

Пакет std_logic_arith

Этот пакет содержит стандартный набор арифметических, логических функций и функций сравнения для работы с типами SIGNED, UNSIGNED, SMALL_INT, INTEGER, STD_ULOGIC, STD_LOGIC, STD_LOGIC_VECTOR.

Этот пакет содержит определения следующих типов:

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
subtype SMALL_INT is INTEGER range 0 to 1;
```

Типы UNSIGNED и SIGNED имеют такое же определение, как и Bit_vector, но для них разработан набор функций, который позволяет интерпретировать их значения как числовые. Тип Unsigned интерпретируется как двоичное представление числа без знака. Старшие разряды в представлении находятся слева. Тип Signed интерпретируется как двоичное представление числа со знаком. Старшие разряды в представлении также находятся слева. Самый левый разряд представляет знак ('0' — соответствует '+', '1' — соответствует '-').

***Таблица П5.1.** Арифметические операторы*

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
+,—	Unsigned	Unsigned	Unsigned
+,—	Signed	Ssigned	Signed
+,—	Unsigned	Signed	Signed
+,—	Signed	Unsigned	Signed
+,—	Unsigned	Integer	Unsigned
+,—	Integer	Unsigned	Unsigned
+,—	Signed	Integer	Signed
+,—	Integer	Signed	Signed

Таблица П5.1 (окончание)

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
+,–	Unsigned	Std_ulogis	Unsigned
+,–	Std_ulogic	Unsigned	Unsigned
+,–	Signed	Std_ulogis	Signed
+,–	Std_ulogic	Signed	Signed
+,–,*	Unsigned	Unsigned	Std_logic_vector
+,–,*	Signed	Signed	Std_logic_vector
+,–,*	Signed	Unsigned	Std_logic_vector
+,–,*	Unsigned	Signed	Std_logic_vector
+,–	Integer	Unsigned	Std_logic_vector
+,–	Integer	Signed	Std_logic_vector
+,–	Signed	Integer	Std_logic_vector
+,–	Unsigned	Std_ulogic	Std_logic_vector
+,–	Std_ulogic	Unsigned	Std_logic_vector
+,–	Signed	Std_ulogic	Std_logic_vector
+,–	Std_ulogic	Signed	Std_logic_vector
+	Unsigned	–	Unsigned
+, –, ABS	Signed	–	Signed
+	Unsigned	–	Std_logic_vector
+,–,ABS	Signed	–	Std_logic_vector

Таблица П5.2. Операторы сравнения

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
<,<=,>,>=,=,/=	Unsigned	Unsigned	Boolean
<,<=,>,>=,=,/=	Unsigned	Signed	Boolean
<,<=,>,>=,=,/=	Signed	Unsigned	Boolean

Таблица П5.2 (окончание)

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
<,<=,>,>=,=,/=	Signed	Signed	Boolean
<,<=,>,>=,=,/=	Unsigned	Integer	Boolean
<,<=,>,>=,=,/=	Integer	Unsigned	Boolean
<,<=,>,>=,=,/=	Signed	Integer	Boolean
<,<=,>,>=,=,/=	Integer	Signed	Boolean

Таблица П5.3. Операторы сдвига

Оператор	Тип левого операнда	Тип правого операнда (количество позиций)	Тип результата
SHL, SHR	Unsigned	Unsigned	Unsigned
SHL, SHR	Signed	Unsigned	Signed

SHL — оператор сдвига влево. Первый операнд сдвигается влево на количество разрядов, указанное во втором операнде.

SHR — оператор сдвига вправо. Первый операнд сдвигается вправо на количество разрядов, указанное во втором операнде.

Таблица П5.4. Функции преобразования типов

Имя функции	Типы операндов	Тип результата
CONV_INTEGER	ARG: INTEGER	INTEGER
	UNSIGNED	INTEGER
	SIGNED	INTEGER
	STD_ULOGIC	SMALL_INT
CONV_UNSIGNED	ARG: INTEGER; SIZE: INTEGER	UNSIGNED
	ARG: UNSIGNED; SIZE: INTEGER	UNSIGNED
	ARG: SIGNED; SIZE: INTEGER	UNSIGNED
	ARG: STD_ULOGIC; SIZE: INTEGER	UNSIGNED

***Таблица П5.4** (окончание)*

Имя функции	Типы операндов	Тип результата
CONV_SIGNED	ARG: INTEGER; SIZE: INTEGER	SIGNED
	ARG: UNSIGNED; SIZE: INTEGER	SIGNED
	ARG: SIGNED; SIZE: INTEGER	SIGNED
	ARG: STD_ULOGIC; SIZE: INTEGER	SIGNED
CONV_STD_LOGIC_	ARG: INTEGER; SIZE: INTEGER	STD_LOGIC_VECTOR
VECTOR	ARG: UNSIGNED; SIZE: INTEGER	STD_LOGIC_VECTOR
	ARG: SIGNED; SIZE: INTEGER	STD_LOGIC_VECTOR
	ARG: STD_ULOGIC; SIZE: INTEGER	STD_LOGIC_VECTOR

Приложение 6

Пакет Numeric_std

Этот пакет содержит стандартный набор арифметических, логических функций и функций сравнения для работы с типами SIGNED, UNSIGNED, INTEGER, STD_ULOGIC, STD_LOGIC, STD_LOGIC_VECTOR.

В пакете содержатся определения следующих типов:

```
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;
```

Внимание

Вследствие того, что пакеты Std_logic_arith и Numeric_std содержат перекрывающиеся определения типов и функций, одновременное их использование в Foundation Express не поддерживается

Внимание

Foundation Express не поддерживает следующие функции пакета Numeric_std: /, rem, mod, to_01.
Аналогичные функции поддерживаются только для пакета Std_logic_arith

Таблица П6.1. *Арифметические операторы*

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
–,Abs	Signed	–	Signed
+,–,*,/,rem,mod	Unsigned	Unsigned	Unsigned
+,–,*,/,rem,mod	Signed	Signed	Signed
+,–,*,/,rem,mod	Unsigned	Natural	Unsigned
+,–,*,/,rem,mod	Natural	Unsigned	Unsigned
+,–,*,/,rem,mod	Signed	Integer	Signed
+,–,*,/,rem,mod	Integer	Signed	Signed

Таблица П6.2. Операторы сравнения

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
>,<,<=,>=,=,/=	Unsigned	Unsigned	Boolean
>,<,<=,>=,=,/=	Signed	Signed	Boolean
>,<,<=,>=,=,/=	Unsigned	Natural	Boolean
>,<,<=,>=,=,/=	Natural	Unsigned	Boolean
>,<,<=,>=,=,/=	Signed	Integer	Boolean
>,<,<=,>=,=,/=	Integer	Signed	Boolean

Таблица П6.3. Логические операторы

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
NOT, AND, OR, NAND, NOR, XOR, XNOR	Unsigned	Unsigned	Unsigned
NOT, AND, OR, NAND, NOR, XOR, XNOR	Signed	Signed	Signed

Таблица П6.4. Функции и операторы сдвига

Оператор	Тип левого операнда	Тип правого операнда (количество позиций)	Тип результата
SHIFT_LEFT, SHIFT_RIGHT, ROTATE_LEFT, ROTATE_RIGHT	UNSIGNED	NATURAL	UNSIGNED
SHIFT_LEFT, SHIFT_RIGHT, ROTATE_LEFT, ROTATE_RIGHT	SIGNED	NATURAL	SIGNED
sll, srl, rol, ror	UNSIGNED	INTEGER	UNSIGNED
sll, srl, rol, ror	SIGNED	INTEGER	SIGNED

Функции **SHIFT_LEFT** и **SHIFT_RIGHT** выполняют сдвиг первого аргумента на количество разрядов, указанное во втором аргументе, соответственно влево или вправо. Им соответствуют операторы **sll** и **srl**.

Функции **ROTATE_LEFT** и **ROTATE_RIGHT** выполняют циклический сдвиг первого аргумента на количество разрядов, указанное во втором аргументе, соответственно влево или вправо. Им соответствуют операторы `rol` и `ror`.

Таблица П6.5. *Функции изменения размера*

Оператор	Тип левого операнда	Тип правого операнда (количество позиций)	Тип результата
RESIZE	SIGNED	NATURAL	SIGNED
RESIZE	UNSIGNED	NATURAL	UNSIGNED

Таблица П6.6. *Функции преобразования типов*

Имя функции	Типы операндов	Тип результата
TO_INTEGER	ARG: UNSIGNED	NATURAL
	ARG: SIGNED	INTEGER
TO_UNSIGNED	ARG, SIZE: NATURAL	UNSIGNED
TO_SIGNED	ARG: INTEGER; SIZE: NATURAL	SIGNED

Таблица П6.7. *MATCH-функции*

Имя функции	Типы операндов	Тип результата
STD_MATCH	STD_ULOGIC	BOOLEAN
	UNSIGNED	
	SIGNED	
	STD_LOGIC_VECTOR	
	STD_ULOGIC_VECTOR	

Таблица П6.8. *Функции трансляции*

Имя функции	Тип первого операнда	Второй операнд	Тип результата
TO_01	UNSIGNED	STD_LOGIC	UNSIGNED
	SIGNED	Если он не определен, то = '0' (значение по умолчанию)	SIGNED

Результат этой функции формируется следующим образом. Если i-ый разряд первого операнда имеет значение '1' или 'H', то i-ый разряд результата будет иметь значение '1'. Если i-ый разряд первого операнда имеет значение '0' или 'L', то i-ый разряд результата будет иметь значение '0'. Если i-ый разряд операнда имеет значение, отличное от '1', 'H', '0', 'L', то всем разрядам результата будет присвоено значение, равное значению второго операнда, по умолчанию — '0'.

Приложение 7

Пакет STD_LOGIC_UNSIGNED

Таблица П7.1. *Арифметические операторы*

Оператор	Тип левого операнда	Тип правого операнда	Тип результата
+,—,*	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
+,—	STD_LOGIC_VECTOR	INTEGER	STD_LOGIC_VECTOR
+,—	INTEGER	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
+,—	STD_LOGIC	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR
+	STD_LOGIC_VECTOR	—	STD_LOGIC_VECTOR
<,>,<=,>=,=,/=	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	BOOLEAN
<,>,<=,>=,=,/=	INTEGER	STD_LOGIC_VECTOR	BOOLEAN
<,>,<=,>=,=,/=	STD_LOGIC_VECTOR	INTEGER	BOOLEAN

Таблица П7.2. *Операторы сдвига*

Оператор	Тип левого операнда	Тип правого операнда (количество позиций)	Тип результата
SHL, SHR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR	STD_LOGIC_VECTOR

Таблица П7.3. *Функции преобразования типов*

Имя функции	Типы операндов	Тип результата
CONV_INTEGER	STD_LOGIC_VECTOR	INTEGER

Приложение 8

Пакет TEXTIO

Пакет TEXTIO предназначен для работы с текстовыми файлами. Он содержит описания типов и процедуры, необходимые для выполнения операций чтения и записи в текстовые файлы объектов различных типов. В общем случае текстовый файл рассматривается как набор строк. Базовые процедуры ввода и вывода, организованные в пакете, работают с указателями на строки. Кроме того, существует набор процедур, которые позволяют преобразовывать в строки значения других типов (для записи их в файл) и конвертировать сроки в значения других типов (при чтении из файла). Если объект, значение которого записывается или читается, имеет тип, отличный от строки, то используется соответствующая процедура `write` или `read`, которая позволяет конвертировать значение в строку или наоборот. После выполнения процедуры `writeline`, переданный ей указатель на строку сбрасывается в **`null`**. Все процедуры `read` и `write` поступают аналогичным образом. Такие действия выполняются из-за того, что множество процессов может одновременно выполнять операции чтения и записи с файлами, в результате чего строка, прочитанная в память, в любой момент может перестать соответствовать информации, представленной в файле. Сброс указателя в **`null`** приводит к тому, что при каждом обращении к строке она прочитывается заново. Поэтому операции чтения и записи реализованы как атомарные, отдельно от преобразования типов.

Этот пакет содержит следующие определения типов:

```
type LINE is access string;
type TEXT is file of string;
type SIDE is (right, left);
subtype WIDTH is natural;
```

В нем содержатся так же определения стандартных текстовых файлов:

```
file input : TEXT is in "STD_INPUT";
file output : TEXT is out "STD_OUTPUT";
```

Чтение из файла

Базовая процедура чтения из файла:

```
procedure READLINE(variable f: in TEXT; L: inout LINE);
```

Наборы допустимых параметров дополнительной процедуры чтения из файла — **read** приведены в табл. П6.1

Каждая процедура **read** имеет параметр — указатель на строку, из которой читаются данные, и параметр-переменную, в которую эти данные будут записываться. В ходе выполнения процедуры из текстовой строки удаляются все символы, отвечающие за текстовое представление, затем производится преобразование.

Если переменная-параметр имеет тип `character`, то в нее, при первом обращении к процедуре, записывается первый символ строки. Повторное выполнение процедуры возвратит второй символ строки и т. д.

Если переменная-параметр имеет тип `string`, то в нее прочитывается столько символов, сколько умещается. Последующее чтение начинается с того места, где завершилось предыдущее. Если длина строки в файле меньше, чем длина переменной, в которую производится чтение, то последняя остается частично незаполненной.

Если переменная-параметр имеет другой тип, то выполняются следующие действия:

1. Из начала строки удаляются пробелы, неразрывные пробелы и символы горизонтальной табуляции.
2. Производится экстракция стольких символов, сколько может быть интерпретировано в соответствии с данным типом.
3. Последующая операция чтения начинается с того же места.

Например, пусть в файле присутствует следующая строка:

```
12    -4,26
```

Данные из этой строки могут быть прочитаны в переменные типа `integer` и `real` или `real` в результате выполнения двух операций чтения.

Битовые векторы в файле представляются строками нулей и единиц без кавычек.

Объекты типа `time` состоят из двух частей — значения и единицы измерения, разделенных, как минимум, одним пробелом.

Для каждого типа реализовано по две процедуры `read`, в одной из которых присутствует добавочный параметр `good`, который принимает значение `true`, если преобразование прошло успешно, и `false` — в противоположном случае. Если используется процедура без этого параметра, то, в случае не-

возможности преобразования данных, происходит ошибка и останов моделирования.

VHDL 87 содержит дополнительную функцию `endline(L: in line) return boolean;`. Она позволяет определить конец строки, (функция возвращает `true` в случае достижения конца строки).

Таблица П8.1. *Параметры процедуры read*

L	Value	Good
LINE	Bit	BOOLEAN
LINE	Bit	—
LINE	Bit_vector	BOOLEAN
LINE	Bit_vector	—
LINE	BOOLEAN	BOOLEAN
LINE	BOOLEAN	—
LINE	Character	BOOLEAN
LINE	Character	—
LINE	Integer	BOOLEAN
LINE	Integer	—
LINE	Real	BOOLEAN
LINE	Real	—
LINE	String	BOOLEAN
LINE	String	—
LINE	Time	BOOLEAN
LINE	Time	—

Определение конца файла и конца строки

Пакет TEXTIO содержит также две специальные функции, позволяющие определить конец файла и конец строки. Эти функции используются при чтении из файла

```
function ENDLINE(variable L : in LINE) return BOOLEAN;
function ENDFILE (f: in TEXT) return BOOLEAN ;
```

Запись в файл

Базовая процедура записи в файл имеет следующее описание:

```
procedure WRITELINE(f : out TEXT; L : inout LINE);
```

Наборы допустимых параметров дополнительной процедуры записи в файл — `write` приведены в табл. П6.2.

В качестве параметров процедуре передаются указатель на формируемую строку и переменная, содержащая значение, которое должно быть записано в нее; `filed` — максимальное количество символов, которое может быть записано в строку; `justified` — тип выравнивания, если передаваемых символов меньше, чем указано в предыдущем параметре (`left`, `center`, `right`). Если отведенное для записи место меньше, чем действительное количество символов, то записывается столько символов, сколько может поместиться.

Если с одним и тем же указателем выполняется несколько процедур `write` подряд, то их результаты записываются последовательно в одну строку слева направо.

Для чисел с плавающей запятой вводится дополнительный параметр `digits`, указывающий количество знаков после запятой. Если этот параметр установить в `0`, то используется экспоненциальная запись.

Для объектов типа `time_value` используется дополнительный параметр, позволяющий указать единицу измерения, в которой следует представить значение. Например:

```
write(L,40 ns,right,0,ps);
```

В результате будет записано 40000 ps.

Таблица П8.2. *Параметры процедуры* `write`

L	VALUE	JUSTIFIED	FIELD
LINE	bit	right	0
LINE	bit_vector	right	0
LINE	boolean	right	0
LINE	character	right	0
LINE	integer	right	0
LINE	real	right	0
LINE	string	right	0
LINE	time	right	0

Приложение 9

Директивы Foundation Express

Директивы Foundation Express представляют собой специальным образом оформленные комментарии в тексте программы на VHDL, которые влияют на работу инструментария синтеза Foundation Express.

Подобно обычным комментариям, строки, содержащие эти директивы, начинаются с символов '--', поэтому на работу другого инструментария, в частности OrCAD Express, они не оказывают никакого влияния.

В директиве, после символов '--', должно следовать ключевое слово '**pragma**' или '**synopsys**', которое указывает, что данный комментарий следует рассматривать как директиву, после чего следует собственно директива. Если после этих ключевых слов следует набор слов, не распознаваемых Foundation Express как директива, то в соответствующей строке выдается сообщение об ошибке.

Существует три основных типа директив:

- директивы выключения и включения трансляции;
- директивы решающих функций;
- директивы `component implication`.

Директивы включения и выключения трансляции

Эти директивы включают в себя две подгруппы:

Директивы синтеза:

```
-- pragma synthesis_off
-- pragma synthesis_on
```

Фрагмент кода, расположенный между этими директивами, не подвергается синтезу (выполняется только синтаксический анализ). В результате данный фрагмент кода в программе на языке VHDL может быть использован для выполнения моделирования на функциональном уровне, но не реализуется в аппаратуре. Это может быть использовано для отладки модели.

Директивы трансляции:

```
-- pragma translate_off
-- pragma translate_on
```

Эта группа директив работает аналогично предыдущей. Однако мы бы не рекомендовали ее использовать, поскольку эти директивы оказывают влияние на процесс синтеза и могут привести к ошибкам.

Директивы решающих функций

Foundation Express не поддерживает механизм решающих функций в том виде, в котором он описан в стандарте языка VHDL. Однако в нем предоставлен альтернативный механизм определения значений сигналов, которые имеют несколько источников.

Если сигнал имеет несколько источников, то для каждого из них должен быть определен тип решающей функции с помощью соответствующей директивы. Все источники одного сигнала должны иметь один тип решающей функции.

Возможно использование одного из следующих вариантов директивы:

- объединение сигналов от источников по "монтажному И";

```
-- pragma resolution_method wired_and
```

- объединение сигналов от источников по "монтажному ИЛИ";

```
-- pragma resolution_method wired_or
```

- объединение сигналов от источников как сигналов с тремя состояниями.

```
-- pragma resolution_method three_state
```

Директивы component implication

Директивы определения компонентов предназначены для сопоставления готовых реализаций компонентов объектам-entity программы на VHDL при компиляции (т. е. при синтезе реализации проектируемого устройства). Для указанного объекта-entity *entity_name*, эти директивы отключают процедуры синтеза реализации по общим алгоритмам вывода схемы синтезирующим компилятором, вместо них в результирующий файл подставляется готовый фрагмент схемной реализации данного объекта.

```
-- pragma map_to_entity entity_name
-- pragma return_port_name port_name
```

Приложение 10

Зарезервированные ключевые слова

abs	downto	map	register
access	else	mod	rem
after	elsif	nand	report
alias	end	new	return
all	entity	next	select
and	exit	nor	severity
architecture	file	not	signal
array	for	null	subtype
assert	function	of	then
attribute	generate	on	to
begin	generic	open	transport
block	guarded	or	type
body	if	others	units
buffer	in	out	until
bus	inout	package	use
case	is	port	variable
component	label	procedure	wait
configuration	library	process	when
constant	linkage	range	while
disconnect	loop	record	with

В стандарт VHDL 93 добавлены следующие слова:

group	pure	shared	srl
impure	reject	sla	unaffected
inertial	rol	sll	xnor
postponed	ror	sra	

Литература

1. AMBA Interconnection Schemes. Application Note 05. — ARM Limited, 1998. — 36 p.
2. AMBA Specification. Rev 2.0. — ARM Limited, 1999. — 230 p.
3. Ashenden P.J. The Designer's Guide to VHDL. San-Francisco, Morgan Kaufmann Publishers, 1995, 687 p.
4. Biggs J. Design Reuse with Design Reuse with "Virtual Components". IP'97: Design for Reuse Workshop, 1997, 22 p.
5. Burger D., Goodman J. R. Billion-Transistor Architectures. // Computer, 1997, pp. 46-49.
6. Expanding Moore's Law. The Exponential Opportunity. Intel, 2002, 6 p.
7. Gaisler J. The LEON Processor User's Manual. Version 2.4.0. — Noordwijk: Gaisler Research, 2001. — 65 p.
8. Gajski D. Principles of Digital Design. Prentice Hall, New Jersey, 1997, 447 p.
9. IEEE Std 1076.1-1999. IEEE Standard VHDL 1076.1. Language Reference Manual. IEEE, New York, USA, 1999.
10. IEEE Std 1076-1993. // IEEE Standard VHDL Language Reference Manual. IEEE, New York, USA, 1994, 632 p.
11. Kang S., Lebelevici Y. CMOS Digital Integrated Circuits. Analysis and Design. Boston, McGrow-Hill, 1999.
12. Mano M., Kime C. Logic Computer Design Fundamentals. 2nd edition. Prentice Hall, 2001, 650 p.
13. Metz C. Moore's Law Marching On. PC Magazine, 2002, August 19, p. 57.

14. Moore G. E. Cramming more components onto integrated circuits. Electronics, Volume 38, Number 8, April 19, 1965, 4 p.

15. Parnell K., Mentha N. Programmable Logic Design Quik Start Hand Book. Xilinx Corp., 2002, 201 p.

16. Sinander P. The usage of VHDL in the European Space Agency. ESA/ESTEC WSM document, Noordwijk, 1995, 11 p.

17. VHDL Models for Board-level Simulation. ESA/ESTEC, document WSM/SH/010, Issue 1, February 1996, 70 p.

18. Virtex-II Pro Handbook. Xilinx, Inc., 2001.

19. Virtex-II Pro Platform FPGA Handbook, Xilinx Corp., 2002, 437 p.

20. Yalamananchili S. Introductory VHDL: From Simulation to Synthesis. Prentice Hall, New Jersey, 2001, 401 p.

21. Александров Ю.П., Беляев А.А., Глушков А.В., Грибов В.Ф., Никольский В.Ф., Петричкович Я.Я., Солохина Т.В. Отечественная серия ИМС "Мультикор" и ее использование в перспективных бортовых авиационных комплексах. I Всероссийская научно-техническая конференция по проблемам создания перспективной авионики. Тезисы докладов. — М.: Фазотрон-НИИР, 2002. С. 123-124.

22. Володин А.Ю., Горбачев С.В., Шейнин Ю.Е. Шина PCI в высокопроизводительных микропроцессорных системах. Учебное пособие. — СПб.: ГУАП, 1999. — 100 с.

23. ГОСТ Р 50754-95. Язык описания аппаратуры цифровых систем VHDL. Описание языка.

24. Джоунс Г. Программирование на языке ОККАМ. М.: Мир, 1989. 208 с.

25. Майоров С. А., Новиков Г. И., Алиев Т. И. Основы теории вычислительных систем. — М.: Высш.шк., 1978. — 408 с.

26. Мясников В.А. Игнатьев М.Б. Кочкин А.А., Шейнин Ю.Е. Микропроцессоры: системы программирования и отладки. М: Энергоатомиздат. 1985. — 272 с.

27. Суворова Е.А., Шейнин Ю.Е. Язык VHDL для проектирования систем на СБИС: Учебное пособие. / ГУАП, СПб., 2001 — 212 с.

28. Угрюмов Е.П. Цифровая схемотехника. — СПб.: БХВ-Петербург, 2000. — 528 с.