ГЛАВА 3

Языки описания дискретных устройств

3.1. Общие положения

3.1.1. HDL-программа как модель проектируемого устройства

Языковое описание устройства — это текст, сохраняемый в некотором файле или нескольких файлах, которые в совокупности составляют представление разработчика о проекте и используются на всех последующих этапах проектирования, в том числе при синтезе устройства и его моделировании. Тексты описаний в большинстве языков проектирования дискретных устройств по составу синтаксических конструкций и по интерпретации результатов их исполнения (физического или модельного) очень схожи с традиционными языками программирования. Поэтому часто такое текстовое описание называют программой на языке проектирования, или, коротко, HDL-программой, а конструкции, описывающие способ формирования результатов, — операторами.

HDL-программа может рассматриваться как *знаковая модель* дискретного устройства [1, 34]. Знаковыми моделями называют способы представления реальных или проектируемых объектов, которые не имеют физического или геометрического подобия с объектами моделирования, а отражают лишь существенные свойства объектов с использованием принятых формальных обозначений. Воспроизведение функционирования и иных свойств объекта по знаковым моделям может выполняться на основе мысленных или машинных экспериментов с ними. Знаковая модель, как всякая другая модель, обеспечивает выделение наиболее существенных для разработчика характеристик. Как правило, составитель HDL-программы абстрагируется от конкретной физической реализации, выделяя, прежде всего, воспроизведение функционирования проектируемого изделия. После выполненных прове-

рочных процедур на основе модели может быть построен объект, т. е. некое физическое устройство, которое подтвердит или (что при хорошем уровне моделирования мало вероятно) опровергнет предположения, заложенные в модель. Естественно, в современных условиях основную работу по интерпретации модели в физический объект выполняет компьютер, но иногда пользователь может вмешиваться в процедуру синтеза, т. е. возможны интерактивные процедуры проектирования.

Современные языки позволяют строить модели (программы), характеризующиеся различной степенью приближения к будущей реализации — от внешнего описания закона функционирования до детального представления проекта на уровне вентилей или макроячеек БИС. Одним из способов сближения описания и фактической реализации является иерархическое проектирование, возможность которого предусмотрена семантикой современных языков проектирования.

Типы данных

Входная информация в дискретных устройствах — цифровые сигналы, которые должны поступать в систему через контакты и далее двигаться по цепям, называемым также *связями*, к блокам, выполняющим те или иные преобразования, и далее на выходные контакты или к другим блокам. Входным, внутренним и выходным данным проектов сопоставляются имена связей, которым соответствуют цифровые сигналы на соответствующих входных и выходных контактах проектируемой схемы или внутренних цепях проекта. Источнику сигнала, поступающего на некоторую связь, в языках сопоставляется оператор, присваивающий значения переменной, представляющей связь или сигнал на этой связи. Такие операторы называют *драйверами*. Различие сигналов от связей выполняется только по контекстным признакам.

Основным типом данных в языках являются данные сигнального типа, по свойствам близкие к обычным логическим данным (часто мы будем называть их просто логическими). Основное отличие логических данных в языках проектирования от логических данных языков программирования заключается в наборах допустимых значений, что более подробно будет изложено в *разд. 3.1.2.* Кроме того, явно или неявно подразумевается, что сигнал обладает временными характеристиками, в частности, изменение значений отображающей переменной может происходить не сразу после присвоения ей нового значения.

Для удобства работы данные могут, как и в обычных языках программирования, структурироваться и объединяться в агрегаты — массивы, векторы, записи (VHDL, Verilog), группы (AHDL). В некоторых оговоренных случаях структурированные логические данные представляются в достаточно "привычных" формах, например как числа или строки. Преобразованиям логи-

ских данных сопоставляются определенные компоненты в структуре проктируемого устройства.

омимо данных сигнального типа применяется ряд служебных типов данных. Из них наиболее важными являются данные арифметического типа и трочные данные. Хотя программы могут предусматривать преобразования ких данных, однако эти преобразования реализуются только на этапе мпиляции и, может быть, моделирования проекта. При реализации в аратуре используются только окончательные результаты преобразований, бычно в форме констант, задающих конфигурацию аппаратных средств.

труктура и поведение

одель, отражающая объект проектирования в форме правил преобразования ходных данных в выходные, называется поведенческой. Такая точка зрения на роект представлена рис. 3.1, *а*. В общем случае можно записать функцию реобразования как

$$Y = F(A, B).$$

спользуя терминологию VHDL, назовем модуль F — PROJECT ENTITY ословный перевод — сущность проекта), входы и выходы назовем портами ORT). Программные модули на языке VHDL, описывающие проекты поведенческой форме, именуют *поведенческими архитектурными телами.*

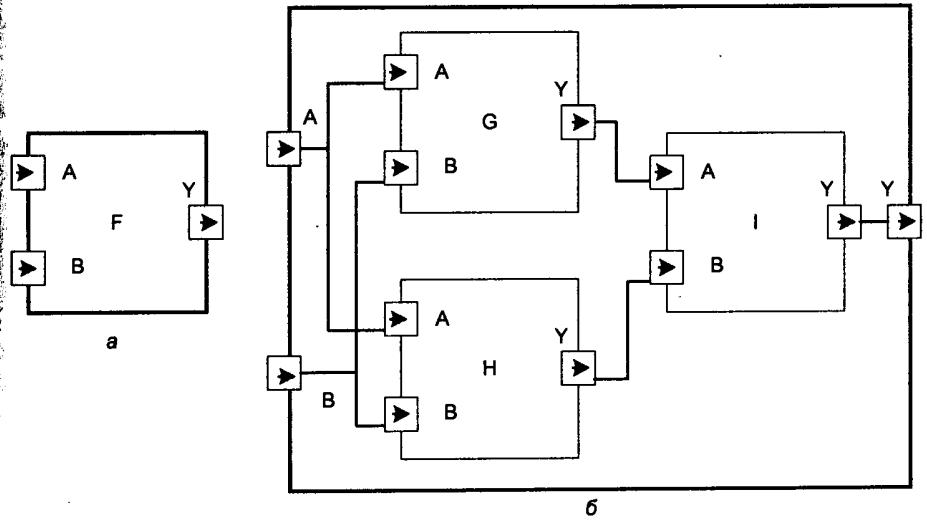

Рис. 3.1. Поведенческое (*а*) и структурное (*б*) представление проекта

Структурная модель описывает проект в виде совокупности модулей, каждый из которых реализует определенную часть задачи, и набора связей меж-

ду ними. Такое представление интерпретируется рис. 3.1, *б*. В данном случае, очевидно, выполнена декомпозиция функции F в виде:

$$F(A, B) = I(G(A, B), H(A, B)).$$

Каждый подблок (в нашем примере блоки G, H и I), в свою очередь, может быть представлен в виде поведенческой или структурной модели.

Структурная модель является естественным способом создания сложных проектов, описание которых целесообразно выполнить по иерархическому принципу. В программном модуле высшего уровня иерархии содержатся объявления портов и внутренних связей, а также так называемые *декларации вхождений* компонентов (INSTANTION DECLARATION), т. е. определенных синтаксисом языка указаний на включенные компоненты и способ их соединений. В VHDL такое описание называют *структурным архитектурным телом*.

На рис. 3.2 представлен типовой набор проектных модулей и их взаимосвязей на примере VHDL. Хотя в других рассмотренных в настоящей книге языках понятие архитектурного тела явно не определяется, общая структура комплекса программных модулей иерархического проекта в разных языках имеет много общего. Рассмотрим процедуру представления иерархических проектов в языковой форме на основе дисциплины нисходящего проектирования.

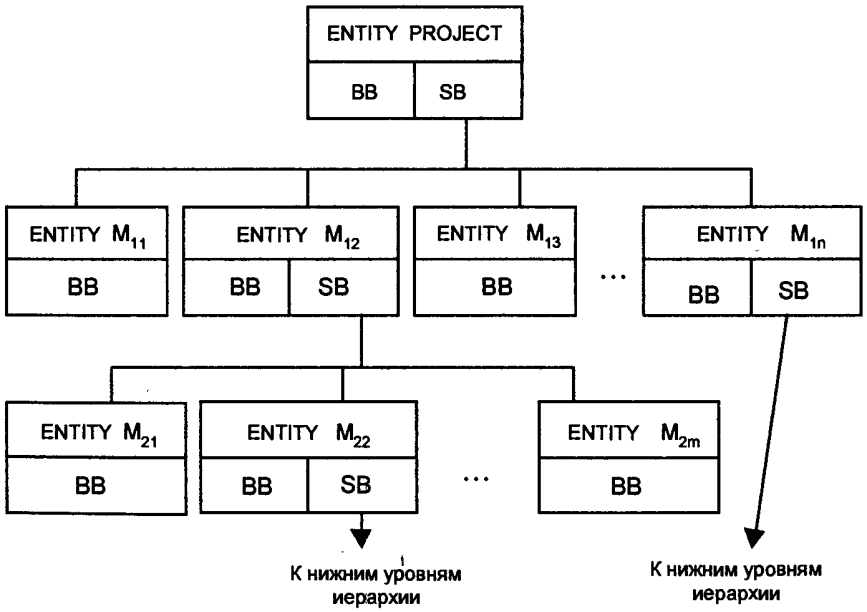

Рис. 3.2. Иерархическая структура проекта

Первым этапом проектирования по дисциплине "сверху вниз" является создание поведенческой модели высшего уровня иерархии. На рисунке этот этап отображен исходным первичным модулем ENTITY PROJECT, описывающим интерфейс проекта, и связанным с ним вторичным модулем BB (Behavioral Body) — описанием его поведения. Средствами используемого пакета проектирования выполняется сеанс моделирования поведения. Часто может потребоваться несколько коррекций до получения соответствия поведенческой модели заданным требованиям.

Если объект является сравнительно простым, то поведенческое описание высшего уровня после отладки на модели может непосредственно передаваться в подсистему компиляции для синтеза аппаратной реализации. Однако, как показывает опыт использования САПР, при интерпретации сложных алгоритмов возникают ситуации, когда компилятор "навязывает" неоптимальные решения. Использование детализации описания позволяет смягчить эффект такого навязывания, усилить влияние разработчика на окончательную реализацию. Некоторые конструкции программ, особенно характеризующиеся большой глубиной вложения операторов, вообще не поддаются в распространенных САПР прямой интерпретации в соответствующее схемное решение. Кроме того, разделение общего описания на фрагменты облегчает реконфигурацию системы за счет замены некоторых блоков другими блоками аналогичного назначения, не говоря о преимуществах связанных с возможностью распараллеливания проектной процедуры между несколькими исполнителями.

При структуризации проекта каждой выделенной структурной единице сопоставляется проектный модуль со своим ENTITY. Модули первого уровня декомпозиции на рисунке обозначены как M_{11}, M_{12}, ..., M_{1n}. Создается структурное архитектурное тело SB (Structural Body), соответствующее верхнему уровню иерархии. SB содержит декларацию выделенных подблоков и описание необходимых межблочных связей. Первичное (поведенческое) описание подблока может быть построено путем "вырезания и копирования" соответствующих фрагментов текста из поведенческого тела предшествующего иерархического уровня с добавлением требуемых стандартом языка заголовков. В общем случае, возможно сохранение описания части подблоков в архитектурном теле высшего уровня иерархии в поведенческой форме. Тогда говорят о смешанных, или структурно-поведенческих телах. Важно отметить, что ENTITY целостного проекта остается без изменений. После тестирования структурированной версии описания можно перейти к структуризации подблоков, которая выполняется подобным образом. При необходимости, после каждого уровня структурная декомпозиция блоков может быть продолжена. В результате получается древовидная иерархия проекта. Процесс проектирования можно рассматривать как перемещение по дереву декомпозиции, а окончательный проект как совокупность однонаправленных путей в дереве от корня, представленного ENTITY целостного проекта, к конечным вершинам. На любом этапе декомпозиции (в том числе на ко-

нечном и на всех промежуточных этапах) конечными вершинами являются только поведенческие тела, а промежуточные вершины представлены структурными телами. Если модуль далее не подлежит декомпозиции, он представляется только поведенческим телом.

В процессе осуществления конкретного проекта полная декомпозиция требуется не всегда. На некотором этапе может оказаться выделенной совокупность операторов, которая воспроизводится некоторым модулем, присутствующим в библиотеке, поставляемой с САПР, используемой при проектировании, библиотеке предыдущих разработок проектировщика или приобретаемых у третьих фирм расширениях библиотек. Тогда достаточно в соответствующем структурном теле сослаться на этот библиотечный модуль. Однако и в этом случае общая структура проекта сохраняется. Просто ссылаясь на библиотечный модуль, проектировщик фактически подключает к своему проекту иерархическое поддерево, вершиной которого является модуль, указанный в ссылке. Само поведенческое или структурное описание подключаемого модуля от проектировщика может быть скрыто.

Тестирование является одной из важнейших проектных процедур. Можно рекомендовать одновременно с разработкой программы описания проектируемого устройства изделия создавать программу для тестирования проекта. Подобная программа представляет структурную модель (SB) тестирующей установки (Test-Bench), компонентами которой, в общем случае, являются: проектируемый модуль, генератор тестирующего воздействия (Stimulator) и анализатор результатов (рис. 3.3).

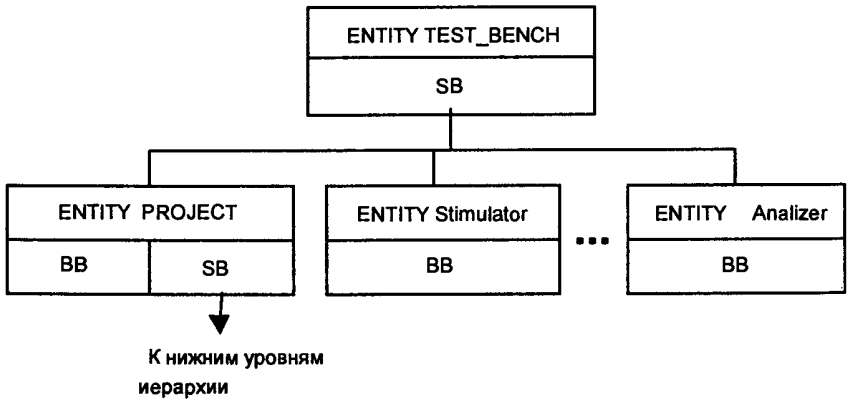

Рис. 3.3. Структура модулей для тестирования проекта

Замечание

В языках VHDL и VerilogHDL можно встраивать описание поведения внешних выводов, фактически, тестового воздействия, непосредственно в программу, представляющую проектируемый модуль. Тем не менее, создание модуля для тестирования как самостоятельной проектной единицы представляется более оправданным как с точки зрения использования Test-Bench на различных этапах реализации проекта, так и с точки зрения отделения описания фактически реализуемых и служебных модулей проекта.

Test-Bench верхнего уровня с весьма незначительными изменениями может использоваться для отладки структурированных версий описания проекта. В сложных случаях могут создаваться свои программные модули для тестирования фрагментов.

Проектирование "снизу вверх" предусматривает объединение простейших модулей в более сложную структуру. Исходные модули — это решения, созданные проектировщиком на более ранних этапах работы, в ходе работ над другими проектами или доступные проектировщику через библиотеки САПР. Специфические языковые конструкции позволяют описывать последовательное объединение компонентов в группы (выделение групп осуществляется исходя из степени сильной алгоритмической и информационной связанности компонентов в целевой системе) с последующим объединением полученных фрагментов в структуры высшего уровня иерархии.

Сложные проекты, как правило, создаются большими группами разработчиков. В этом случае проектирование сверху вниз более оправдано. После любого этапа декомпозиции выделенные фрагменты, для которых уже определен интерфейс и принципы функционирования, могут передаваться для детальной проработки разным исполнителям. Отметим, что важнейший элемент корпоративной работы — это проектная база данных коллективного пользования. В подобной базе должна быть установлена строгая иерархия прав доступа, но в любом случае изменение, внесенное разработчиком одного из разделов, должно быть доступно для разработчиков смежных разделов.

Стили описания проектов

Возвращаясь к понятию поведенческой модели, отметим, что известны различные подходы к описанию поведения в HDL-программах. Для наглядности рассмотрения этих подходов будем использовать понятие информационного графа алгоритма (рис. 3.4). Вершинам информационного графа проектируемой системы сопоставляются элементарные действия (операторы), которые выполняются над исходными или промежуточными данными, а дугам соответствуют информационные связи [10].

Выделим основные подходы к описанию функционирования дискретных систем, иногда называемые *стилями программирования*. Выбор стиля программирования во многом определяется наклонностями и опытом разработчиков, но надо иметь в виду, что часто стиль существенно влияет на порождаемую системой автоматизированного проектирования реализацию.

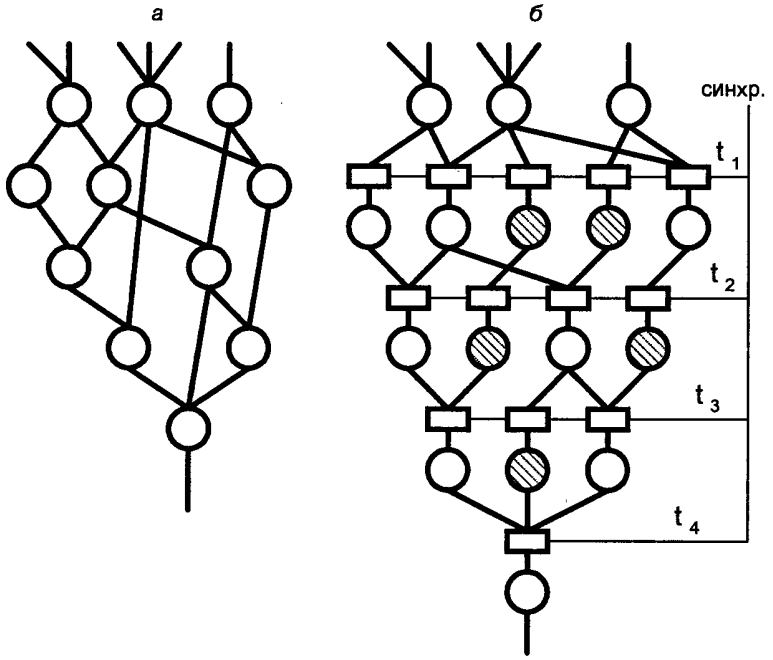

Рис. 3.4. Информационный граф алгоритма: каскадная форма (*а*) и конвейерная форма (*б*)

- Последовательный стиль описания соответствует традиционным подходам к составлению компьютерных программ. Расположение и порядок исполнения операторов в тексте описания соответствует порядку прохождения данных через информационный граф. Любой оператор в тексте записывается после всех операторов, ему инцидентных, т. е. тех, с которыми он имеет связи по входу. Такое описание наиболее точно интерпретируется ярусной формой информационного графа (см. рис. 3.4, *а*), которая получается из исходного представления алгоритма путем упорядочения независимых примитивных операторов таким образом, что вершины каждого яруса связаны с вершинами предыдущих. Желательно, чтобы максимальное число связей осуществлялось вершинами непосредственно предшествующего яруса. Порядок записи операторов одного уровня не имеет значения. Графы с обратной связью в последовательной форме интерпретировать в HDL затруднительно, а иногда и невозможно.
- Параллельный стиль предполагает асинхронное и, по возможности, одновременное исполнение операторов в реализуемом устройстве. Обычно блок, реализующий некоторый оператор, непосредственно реагирует на признак исполнения действия любым из его предшественников. Особенности моделирования поведения параллельного описания в программных системах последовательного типа будут рассмотрены в *разд. 3.1.2.* Для

представления параллельных процессов используются такие формы записи, которые предусматривают исполнение каждого оператора после выполнения хотя бы одного любого из инцидентных ему операторов (в VHDL такие операторы называются *параллельными* и отличаются от последовательных по своей локализации в программе). Порядок записи параллельных операторов безразличен, допустимы обратные связи и петли.

- Потоковый стиль отличается от предыдущего тем, что оператор исполняется после готовности *всех* предшественников данного оператора. В чистом виде потоковый принцип довольно сложен в реализации, за исключением случая линейного алгоритма или блочно-линейного алгоритма (т. е. алгоритма, разделимого на последовательно соединенные блоки, каждый из которых может иметь достаточно сложную структуру). Чаще используется его модификация — конвейер. Операторы, представляющие вершины одного яруса графа, исполняются после реализации всех операторов предыдущего яруса. Операторы одного яруса могут описываться и как последовательные, и как параллельные. Техническая реализация обычно предусматривает синхронную работу ступеней конвейера от общего синхронизирующего сигнала, причем период синхронизирующих сигналов больше максимального времени задержки элементов. На рис. 3.4, *б* представлена реализация того же алгоритма, что и на рис. 3.4, *а*. Прямоугольниками показаны вспомогательные элементы, фиксирующие состояния входов и выполняющие запуск очередной ступени в синхронные моменты времени. На управляющие входы всех вспомогательных элементов подается один и тот же сигнал. Обозначения на управляющем входе отображают относительный порядковый номер порции данных, подаваемых на очередную ступень конвейера. В случае рассинхронизации прохождения потоков данных по разным путям вставляются пустые операторы (элементы задержки), отображенные на рис. 3.4, *б* заштрихованными кружочками. Конвейерный стиль описания порождает конвейерные реализации, которые, как известно, обладают повышенной производительностью, т. к. во время обработки порции данных некоторой ступенью предыдущая ступень может обрабатывать следующую порцию данных. В то же время недостатком конвейера в сравнении с параллельной реализацией может стать большая задержка результатов относительно момента появления порции данных.
- Автоматный стиль, в отличие от предыдущих, опирается не на модель передачи данных, а на модель переключения состояний. Стиль основан на определении некоторого множества состояний проектируемого устройства и правил перехода из одного состояния в другое в зависимости от входных сигналов. Каждому состоянию или переходу соответствует определенный набор действий. Такой подход наиболее эффективен при описании устройств, характеризующихся циклическим выполнением однотипных последовательностей преобразований, в частности управляющих устройств, процессорных модулей.

Целесообразный стиль создания проекта и привлекаемые для его реализации средства языка определяются структурными особенностями проектируемого устройства. В реальных проектах содержатся, как правило, структурные фрагменты различных типов, которые целесообразно описывать с привлечением различных средств, в частности стилей программирования.

3.1.2. Принципы интерпретации поведения дискретных устройств средствами моделирования

Процессы в дискретных устройствах, как и вообще в реальном мире, происходят параллельно во времени, причем изменения на входах в различных частях устройства могут происходить асинхронно и относительно независимо. При моделировании такое поведение должно быть воспроизведено с требуемой степенью точности последовательными алгоритмами, реализуемыми в ЭВМ. Для правильного понимания принципов языкового описания и результатов моделирования следует достаточно четко представлять методы, заложенные в подсистемы моделирования САПР.

Моделирование и реальное время

Известно два противоположных подхода к построению систем моделирования дискретных устройств — *сквозное моделирование и событийное моделирование* [1].

При сквозном моделировании время делится на кванты, длительность которых выбирается (с необходимой точностью) как наибольший общий делитель времен задержек компонентов. Каждый квант реального времени соответствует единице модельного времени и отображается вычислительной процедурой (шагом моделирования), состоящей из двух фаз. Частный и простейший случай — моделирование с константными задержками, т. е. равными для всех компонентов. Тогда в первой фазе последовательно выполняются вычисления состояний всех компонентов на основе сигналов, вычисленных на предыдущем шаге, а результаты сохраняются в буфере предсказанных состояний. В следующей фазе данные из буфера переписываются в рабочие ячейки, сохраняющие значение сигналов для очередного шага.

Если при моделировании предполагаются произвольные задержки компонентов, то используется не простой одноступенчатый буфер предсказанных состояний, а буферы типа FIFO (первый вошел — первый вышел) для каждого компонента. Глубина каждого буфера равна задержке соответствующего компонента, выраженной в числе единиц модельного времени. Сдвиг данных в буфере выполняется после каждого шага моделирования. Таким образом, результаты, поступающие на вход такого буфера, поступят на его выход

через время, точнее, число шагов, соответствующее выбранной модельной задержке.

Некоторые особенности имеются при моделировании без учета задержек. В этом случае каждый шаг моделирования реализуется как итерационная процедура. Буфер предсказанных состояний может и не требоваться. Вычисленные состояния сразу доступны программе и используются при моделировании уже следующего компонента моделируемого устройства. Если после такого прохода по всем компонентам обнаружены изменения по сравнению с результатами предыдущего прохода, процедура повторяется до достижения установившегося состояния или до исчерпания заданного числа итераций. Когда установившееся состояние достигнуто, можно переходить к очередному шагу моделирования. Если после заданного числа итераций на каких-либо связях наблюдаются изменения сигналов, можно сделать вывод о наличии осцилляций и определить эти выходы как неопределенные.

Недостатки сквозного моделирования достаточно очевидны. Это, во-первых, нерациональные затраты машинного времени, вызванные потребностью достаточно мелкой дискретизации времени и необходимостью воспроизведения на каждом шаге поведения всех компонентов, в том числе тех, на входах которых не происходит изменений сигналов. На самом деле такие компоненты программа моделирования могла бы попросту пропускать. Во-вторых, налицо нерациональное использование памяти, а это, в сущности, тоже потеря производительности. При моделировании систем с большим разбросом задержек требуются сравнительно емкие FIFO-буферы предсказанных состояний.

Поэтому практически все развитые системы моделирования используют событийный подход, или, как говорят, *дискретную событийную модель*. На каждом шаге моделирования перевычисляются состояния только тех компонентов, на входе которых в данный момент происходят изменения. *Изменения логических переменных*, как входных, так и промежуточных, *называют событиями*. Любое событие может вызвать цепочку других событий. Моделирование системы производится не для равномерно отстоящих моментов реального времени, а лишь для моментов, для которых ранее были предсказаны события.

Событийная модель кроме совокупности таблиц, представляющих структуру моделируемого устройства, использует две основные структуры данных — календарь событий и поле состояний сигналов.

Календарь событий — это список, каждый элемент которого представляет запись и содержит значение времени наступления события, имя изменяемого сигнала и предсказанное на интервал времени после наступления события значение этого сигнала. Элементы списка упорядочены по возрастанию времени появления событий. Перед началом моделирования в календарь событий заносятся все запланированные в эксперименте события на входах, т. е. изменения входных сигналов, а в поле состояний сигналов —

исходные состояния (в крайнем случае, неопределенные состояния). Каждый шаг моделирования отражает реакцию системы на одно событие и предусматривает следующую последовательность действий:

1. Из календаря выбирается запись, соответствующая очередному событию. В первом цикле выбирается первый элемент календаря событий. В дальнейшем выборке подлежит событие, у которого отметка модельного времени является ближайшей большей по сравнению с отметкой времени события, проанализированного на предыдущем шаге. Устанавливается модельное время в соответствии с указателем времени события, а новое значение сигнала, вызвавшего событие, переписывается в поле текущих состояний сигналов.
2. По имени сигнала из структурных таблиц последовательно выбираются компоненты, на входы которых подан изменяющийся сигнал, и для всех этих компонентов выполняется моделирование, заключающееся в определении характера изменения его выходных сигналов в ответ на это событие. Выполняется предсказание изменений выходных сигналов в будущие моменты модельного времени. Это называют *временным распределением переходов* (scheduling a transaction) выходных сигналов. Отметим, что пока выполняется цикл отработки одного события все сигналы как бы "заморожены".
3. Модификация календаря событий. При этом выполняется запись всех новых предсказанных событий в календарь. Отметка времени каждого нового события вычисляется как сумма текущего времени и времени задержки элемента, который генерирует соответствующий сигнал. Событие помещается в список вслед за имеющимся в списке событием, характеризующимся ближайшей меньшей отметкой времени. Кроме того, в ряде случаев должны удаляться некоторые события, которые имеются в календаре. Алгоритм удалений зависит от используемой модели задержки. Модели задержки, используемые в VHDL, и соответствующие правила удалений приведены в *разд. 3.2.6*. Если изменение входов не приводит к изменению выходов, или, как говорят, происходит поглощение события, то, естественно, календарь не изменяется. Простейший пример такой ситуации — переход "логический ноль—логическая единица " на входе элемента ИЛИ, на другом входе которого уже присутствует единица.
4. Если в календаре нет событий, предсказанных на время, позднее текущего модельного времени, то моделирование прекращается, в противном случае выполняется возврат к п. 1.

Событийное моделирование без учета задержек также имеет определенную специфику. Модель и в этом случае предполагает наличие задержки, но очень малой, называемой дельта-задержкой (δ-задержка). Дельта-задержка не несет информации о реальном времени передачи сигналов, а отражает причинно-следственные связи в объекте моделирования. Если на какомлибо из элементов при выполнении п. 2 описанного порядка моделирования

предсказывается изменение сигнала, то соответствующее событие заносится в календарь после всех событий, имеющихся в данный момент в календаре и предсказанных на это же время. Отметка времени у предсказанного события такая же, как и у инициирующего события. Важным для понимания механизмов и результатов моделирования является то, что это предсказанное событие не влияет на исходные данные для воспроизведения поведения остальных компонентов в циклах моделирования, предшествующих циклу отработки этого события.

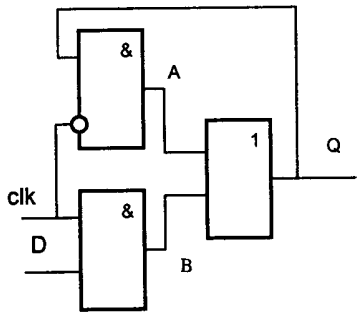

Рис. 3.5. Однофазный D-триггер

В качестве иллюстрации в табл. 3.1 показаны результаты моделирования однофазного D-триггера, схема которого приведена на рис. 3.5. В таблице представлено содержание календаря событий (одна строчка — одно событие) до моделирования, а также после моделирования с учетом задержек, принятых равными 1 нс, и без учета задержек. В исходном состоянии все сигналы нулевые.

Знак <- означает изменение сигнала. Таблица построена на основе файлов отчета сеансов моделирования VHDL-описания в системе Model Technology. Обратите внимание, что одно и то же время может быть присвоено нескольким событиям. События, предсказанные на одно и то же время, не упорядочены, а записываются одно за другим по мере обнаружения переходов при обработке предыдущих событий.

Таблица 3.1. *Результаты моделирования однофазного D-триггера*

Исходный календарь		Календарь после моделирования с учетом задержек		Календарь после моделирования без учета задержек	
Время	**Событие**	**Время**	**Событие**	**Время**	**Событие**
0		0		0	
10	d <- 1	10	d <- 1	10	d <- 1

Таблица 3.1 (окончание)

Исходный календарь		Календарь после моделирования с учетом задержек		Календарь после моделирования без учета задержек	
Время	**Событие**	**Время**	**Событие**	**Время**	**Событие**
15	clk <- 1	15	clk <- 1	15	clk <- 1
20	clk <- 0	16	b <- 1	15	b <- 1
25	d <- 0	17	q <- 1	15	q <- 1
30	clk <- 1	20	clk <- 0	20	clk <- 0
35	clk <- 0	21	a <- 1	20	a <- 1
40	clk <- 1	21	b <- 0	20	b <- 0
45	d <- 1	25	d <- 0	25	d <- 0
50	d <- 0	30	clk <- 1	30	clk <- 1
55	clk <- 0	31	a <- 0	30	a <- 0
		32	q <- 0	30	q <- 0
		35	clk <- 0	35	clk <- 0
		40	clk <- 1	40	clk <- 1
		45	d <- 1	45	d <- 1
		46	b <- 1	45	b <- 1
		47	q <- 1	45	q <- 1
		50	d <- 0	50	d <- 0
		51	b <- 0	50	b <- 0
		52	q <- 0	50	q <- 0
		55	clk <- 0	55	clk <- 0

Алфавит моделирования

Важной характеристикой метода моделирования цифровых устройств является количество различимых состояний сигнала. *Каждому состоянию сопоставляется индивидуальный символ, совокупность символов составляет алфавит моделирования.* Естественно, каждое состояние специфически воспринимается приемниками сигналов, поэтому системе моделирования определяется набор правил преобразования сигналов типовыми цифровыми элементами.

Простейший алфавит — двоичный, содержащий набор {'0', '1'}. Функционирование элементов описывается по правилам алгебры логики. Моделирование на базе этого алфавита весьма экономично, но возможности моделирования ограничены. Невозможно описание шинной логики, в том числе схем, имеющих высокоимпедансное состояние на выходе (Z-состояние), схем с открытым коллектором и подобных. Затруднено воспроизведение сбойных ситуаций, например, вызванных подачей управляющих сигналов на триггеры во время, когда информационные сигналы еще не установлены.

Весьма распространен алфавит из четырех символов {'0', 'X', '1', 'Z'}. Здесь 'X' означает неопределенное состояние. Такой символ присваивается, в частности, сигналу на выходе логического элемента во время переходного процесса. Неопределенное состояние принимает выход триггера после подачи активизирующего сигнала на синхронизирующий вход при запрещенной или неопределенной комбинации сигналов на информационных входах триггера. Символ 'Z' представляет высокоимпедансное состояние порта или отключенную линию. Алфавит {'0', 'X', '1', 'Z'} является единственным, используемым при записи и моделировании программ, написанных на языке AHDL в системе MAX+PLUS II.

Дальнейшее расширение возможностей — девятиэлементный алфавит, в котором приняты следующие символы для представления состояний связей:

- 'U' — не инициализировано (сигналу в программе вообще не присваивались другие значения; обеспечивает контроль корректности инициализации);
- 'Z' — отключено (все источники, подключенные к связи в высокоимпедансном состоянии);
- 'X' — активное неопределенное состояние;
- '0' — активный ноль;
- '1' — активная единица;
- 'L' — слабый ноль;
- 'H' — слабая единица;
- 'W' — слабое неопределенное состояние;
- '-' — не важно (разработчик может запрограммировать переход в это состояние, если реализация алгоритма не зависит от результата; выбор конкретного значения предоставляется компилятору с целью оптимизации реализации устройства).

Разница между слабыми и активными состояниями состоит в том, что слабый сигнал формируется от источников (называемых драйверами), имею-

щих повышенное выходное сопротивление по сравнению с активными источниками. В этом случае источник, генерирующий активный сигнал, подавляет слабый, если не отключен. Пример элемента, генерирующего слабую единицу, — буфер с открытым коллектором: на выходе у него может быть активный ноль, но слабая единица.

При записи программ в VHDL пользователь может априорно задать алфавит моделирования тех или иных языковых конструкций, определяя тип сигналов — от простого двоичного, задаваемого как тип BIT (битовый) до девятикомпонентного типа STD_ULOGIC. В принципе, пользователь может создавать свои типы с большим или меньшим числом символов для представления логических данных, или, что то же самое — числом воспроизводимых в модели состояний сигналов.

Язык Verilog в качестве базового использует алфавит {'0', 'X', '1', 'Z'}, однако проектировщик может присвоить драйверу сигнала специальный атрибут "уровень силы" (strength level), который позволяет моделировать вентильные и буферные компоненты, а также ключевые и резистивные схемы с учетом соотношения их выходных сопротивлений.

При выборе алфавита моделирования (если это допускает система моделирования) следует учитывать, что расширенный алфавит, обеспечивая во многих случаях большую адекватность моделирования, требует больших затрат машинного времени на проведения сеансов моделирования.

3.1.3. Соглашение о правилах записи программ

Программы составляются из лексических элементов, к которым относят имена (идентификаторы), ключевые (зарезервированные) слова, специальные символы, числа, текстовые символы (characters) и строки символов. Лексические элементы объединяются в синтаксические конструкции — *выражения* (expression). Выражение — это конструкция, которая объединяет операнды и знаки операции для формирования результата, являющегося функцией от значений операндов и семантического смысла знака операции. Любое выражение может стать операндом в другом выражении. Константные выражения содержат только определенные в языке записи значений данных и объявленные ранее в тексте имена констант (параметров).

Выражения в свою очередь составляют *предложения*. Основные категории предложений это *декларации* (declaration) и *операторы* (statement, дословный перевод — утверждения). Главное назначение деклараций состоит в определении смысла некоторых идентификаторов. Операторы определяют действия, предусмотренные описываемым алгоритмом, иногда в неявной форме.

Представление языковых конструкций в настоящей книге, кроме пояснения их содержательного смысла и примеров использования, предусматривает формальное определение синтаксиса, т. е. правил вхождения лексических элементов в эти конструкции. При записи программ необходимо строго

придерживаться синтаксических стандартов языков. В противном случае компилятор не сможет интерпретировать предложения и будет выдавать сообщение об ошибке.

Синтаксические конструкции определяются с использованием формализованных правил их записи — формул Бэкуса—Науэра (BNF). Такие формулы представляют некоторые трафареты, в которые разработчик подставляет правильно оформленные конструкции.

Определяемая (или уже определенная) конструкция в BNF представляется с названием, в данной книге на русском языке, заключенным в угловые скобки <>. Допускается опережающее использование конструкции, иными словами, определение конструкции далее по тексту после ее первого использования в определении другой конструкции. Сразу за определяемой конструкцией ставится знак ::=, который трактуется как "записывается в форме", после чего записывается трафарет конструкции, состоящий из лексических элементов рассматриваемого языка, определенных конструкций и специальных *метасимволов*, смысл которых поясняется далее.

Метасимвол | определяет альтернативные возможности записи конструкции.

Пара метасимволов « » — кавычки-"елочки" — ограничивают конструкцию, которую можно повторить сколько угодно раз, в том числе ни одного раза. Например:

```
<целое десятичное число>::= <цифра> « <цифра> »
<цифра>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<идентификатор>::= <буква> « <цифра>|<буква> »
```

Последнюю формулу можно трактовать так: "идентификатор записывается в виде любой последовательности букв и цифр, начинающейся с буквы".

Пара метасимволов [ ] — квадратные скобки — будет использоваться для обозначения необязательного элемента синтаксической конструкции VHDL. Например, BNF-форма

```
<вызов функции>::=<имя функции> [ <список ассоциаций>) ]
```

определяет, что список ассоциаций при вызове функции не обязателен.

Дело в языках Verilog и AHDL квадратные скобки используются как основные символы для выделения индексов. Поэтому при определении синтаксиса этих языков использованы неполные начертания скобок: ⌈ и ⌉

Во всех рассмотренных далее языках при записи идентификаторов и ключевых слов допускаются только латинские буквы. Кириллица может использоваться лишь в комментариях и текстовых константах, причем не во всех системах проектирования, поэтому такая возможность требует специальной проверки.

В языках VHDL и AHDL не вводится различий в использовании прописных и строчных букв (за исключением символьных данных). Так, ключевое слово, открывающее декларацию переменных, может быть записано и как VARIABLE, и как variable, а запись переменной box означает то же, что BOX, даже если варианты встречаются в одном контексте. Тем не менее, в данной книге для лучшей читаемости программ на этих языках мы будем придерживаться правила, по которому ключевые слова записываются прописными буквами, а идентификаторы — строчными.

Однако Verilog HDL не допускает прописных букв для отображения ключевых слов. Поэтому в *разд. 3.3*, посвященном изложению основ этого языка, авторы вынужденно отказались от наглядного выделения ключевых слов.

Символ "перевод строки" игнорируется известными HDL-компиляторами. Поэтому разбиение текста программ на строки произвольно. В данной книге авторы старались при записи листингов программ по возможности придерживаться правила "одна строка — одно предложение", хотя в некоторых случаях предложение будет записываться в нескольких строках. Возможны варианты, когда несколько коротких предложений размещены в одной строке.

При описании синтаксиса языка VHDL синтаксические формулы даются в варианте VHDL'93. Опции, которые не поддерживаются в VHDL'87, будут подчеркиваться.

И последнее. Авторы в настоящей работе не ставили своей целью полное описание синтаксиса языков. Приводятся только наиболее важные, по мнению авторов, и часто используемые в проектировании конструкции и принципы составления программ. Некоторые альтернативные BNF-представления намеренно опущены, чтобы вычленить наиболее существенные варианты. Для более детального ознакомления с рядом специфических конструкций потребуется изучение соответствующих стандартов. Однако надо знать, что стандарты, являясь официальными документами, весьма сложны для понимания, если не иметь представление о принципах использования основных конструкций. Кроме того, авторы предполагают, что читатели знакомы с программированием на общеупотребительных языках, таких как Pascal и C.

3.2. Основы языка VHDL

3.2.1. Язык VHDL как программная система

Язык VHDL был разработан в США в начале восьмидесятых годов по заданию Министерства обороны как язык спецификации проектов с целью обеспечения единообразного понимания подсистем различными проектными группами. В 1987 г. спецификация языка VHDL была принята в качестве

стандарта ANSI/IEEE STD 1076-1987, который часто называют VHDL'87. Удобства и относительная универсальность конструкций этого языка достаточно быстро привели к созданию программ моделирования систем на основании их описания в терминах VHDL.

С начала девяностых годов разрабатываются прямые компиляторы VHDL-программ в аппаратные реализации различных классов. Это наряду с необходимостью более адекватного представления в языке современных тенденций в цифровой схемотехнике стало стимулом усовершенствования языка и привело к созданию расширенного стандарта ANSI/IEEE STD 1076-1993, или кратко VHDL'93 [56, 57, 58]. К тому же следует отметить, что стандарты IEEE подлежат обязательному обновлению каждые пять лет. В 1999 г. была утверждена новейшая версия стандарта IEEE STD 1076.1-1999, известная как VHDL-AMS. Наиболее существенным нововведением VHDL-AMS является появление конструкций, обеспечивающих эффективное описание аналоговых и аналого-цифровых устройств. За основу изложения в настоящей книге принят VHDL'93. Учитывая, что VHDL'87 еще имеет достаточно широкое распространение, конструкции, которые не поддерживаются VHDL'87, отмечаются особо (подчеркиванием, как мы предупредили в предыдущем разделе).

В настоящее время трудно найти САПР дискретных устройств, которая не воспринимает описания устройств на VHDL или хотя бы на его усеченных версиях. В этом смысле надо различать набор языковых конструкций, ориентированных на спецификацию и обеспечение возможности моделирования проекта, и так называемое реализуемое подмножество, т. е. подмножество, которое может непосредственно интерпретироваться схемными компонентами в окончательном проекте в рамках принятой САПР.

VHDL поддерживает широкий набор задач, возникающих при проектировании. Он обеспечивает возможность описания устройств с различной степенью детализации и в различных формах, начиная от внешнего описания общих принципов функционирования до представления в форме элементарных цифровых компонентов (вентилей и триггеров). Разрешается представление через алгоритм функционирования в форме, близкой к традиционным алгоритмическим языкам (так называемое "чистое поведение"). В то же время присутствуют средства, описывающие проект как набор компонентов и связей между ними, допустимы и смешанные формы описаний. Поэтому обеспечивается возможность организации проектной процедуры как последовательной декомпозиции абстрактных спецификаций. Значительное место в языке отведено средствам моделирования, позволяющим проектировщику быстро обнаруживать ошибки и сравнивать альтернативные варианты до сравнительно дорогостоящей фактической реализации проекта. Все это уменьшает объем рутинной работы, "позволяя разработчику сконцентрироваться на стратегических решениях и сократить время доставки изделия на рынок" [34].

В соответствии с вышесказанным средства языка VHDL можно отнести к одному из двух разделов [52, 56] (см. рис. 3.6):

- *общеалгоритмический компонент*, определяющий набор типов данных и операторов, обеспечивающих общее описание алгоритма функционирования;
- *проблемно-ориентированный компонент*, включающий такие специфические и важные для описания аппаратных средств разделы, как специфические для аппаратуры типы данных, средства для описания процессов с учетом их протекания в реальном времени, средства для структурного представления проекта.

Общеалгоритмический компонент по составу, смыслу и принципам использования ее составляющих мало отличается от состава традиционных языков программирования, да и форма записи (синтаксис и семантика языковых конструкций) весьма близка к традиционным языкам. Предварительно остановимся на некоторых составляющих проблемно-ориентированного компонента.

В числе проблемно-ориентированных типов данных прежде всего следует отметить физический тип, используемый для моделирования поведения реальных цифровых систем. Здесь присутствует предопределенный тип — время. Пользователь может определить дополнительные типы данных, отражающих электрические (напряжения, токи, сопротивления и т. п.) или механические свойства носителя информации.

Многозначная логика формально в языке не определена. Однако входящие в любой комплекс моделирования на VHDL стандартные пакеты, например `std_logic_1164`, определяют целую совокупность допустимых алфавитов представления сигналов на основе их декларации как данных перечислимого типа, а также определяют правила их преобразования.

Ограниченные типы данных присутствуют в ряде традиционных языков программирования, например PASCAL. Но следует иметь в виду существенное отличие. Если в PASCAL ограничение множества допустимых значений служит лишь для контроля исполнения программы, то в VHDL такое ограничение задает разрядность устройств и связей, представляющих соответствующие данные.

Специфическим понятием языка VHDL является подтип. Данные, отнесенные к подтипу, сохраняют основные свойства данных базового типа и совместимы с ними в выражениях. Но для них определяются дополнительные ограничения по сравнению с базовым типом, и, возможно, дополнительные функции, которые относятся только к данным, отнесенным к подтипу.

Среди средств представления поведения системы в реальном времени следует, прежде всего, отметить средства представления параллелизма в реальной системе: понятие сигнала, как единицы передаваемой информации между

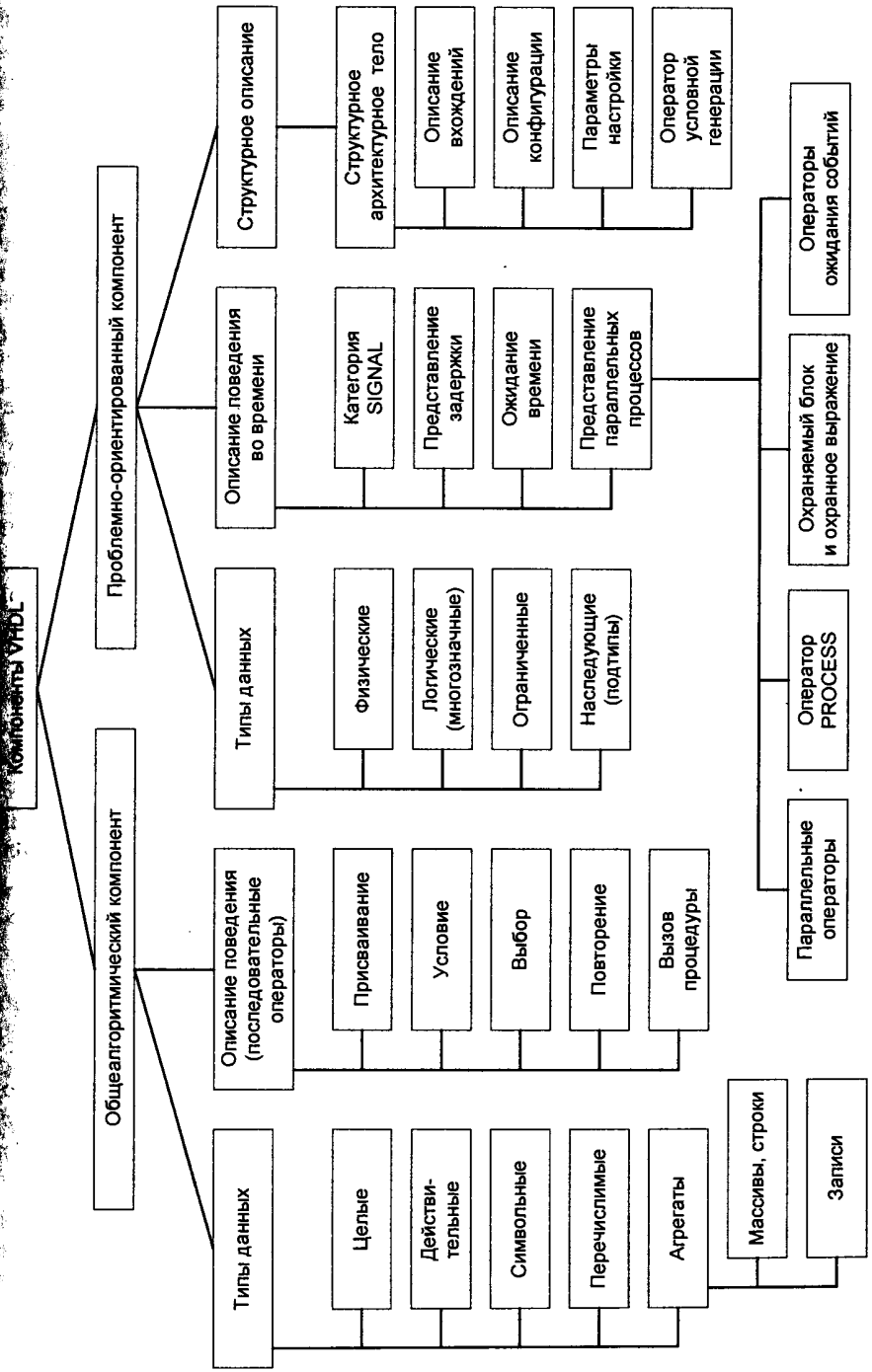

Рис. 3.6. Язык VHDL как программная система

параллельно работающими компонентами; так называемые параллельные операторы, отражающие непосредственное взаимодействие компонентов; понятие процесса, как совокупности действий, инициируемой изменениями сигналов.

При моделировании параллельный оператор интерпретируется таким образом, что он исполняется при любом изменении сигналов, являющихся его аргументами, точнее, при отработке реакции на соответствующее событие. Отметим, что моделирование на основе VHDL-описания должно выполняться на базе дискретной событийной модели. Кроме средств описания параллельных процессов определены конструкции, явно указывающие поведение объекта проектирования во времени — выражения задержки AFTER, оператор приостановки WAIT и ряд других.

Средства структурного представления проекта включают оператор вхождения компонента (Component Instance Statement), задающий тип используемых структурных компонентов и способы их соединений, декларации конфигурации (Configuration Declaration), с помощью которых можно выбирать вариант реализации включаемого компонента, и ряд других конструкций языка.

Из представленного предварительного обзора можно видеть, что VHDL представляет собой развитую алгоритмическую систему, позволяющую описывать разнообразные структуры и явления в информационных системах.

3.2.2. Структура проекта. ENTITY и архитектурные тела

Проект в системе проектирования на основе VHDL представлен совокупностью иерархически связанных текстовых фрагментов, называемых проектными модулями.

Различают первичные и вторичные проектные модули, при этом

```
<первичный модуль> ::=
    <декларация ENTITY> | <декларация пакета> | <декларация конфигурации>
<вторичный модуль> ::= <архитектурное тело> | <тело пакета>
```

Декларация ENTITY — определяет имя проекта и, необязательно, его интерфейс, т. е. порты и параметры настройки. Подчиненное ENTITY *архитектурное тело* описывает тем или иным способом функционирование устройства и (или) его структуру.

Пакет — набор объявлений вводимых пользователем типов, переменных, констант, подпрограмм и т. п. (подобно h-файлам в языке C). *Декларация пакета* определяет только смысл глобальных для проекта идентификаторов. *Тело пакета* содержит расшифровку порядка вычисления процедур и функций пакета, определение локальных имен.

Каждому ENTITY сопоставляется одно или и несколько архитектурных тел. Декларация пакета не требует обязательного присутствия тела пакета. Несколько вторичных модулей, соответствующих одному первичному, составляют набор возможных альтернативных реализаций объекта, представленного первичным модулем. Например, одному ENTITY может соответствовать несколько архитектурных тел, отличающихся степенью детализации описания и даже алгоритмом преобразования данных, определенных в ENTITY.

Первичные и соответствующие им вторичные модули могут сохраняться в различных файлах или записываться в одном файле. Важно лишь, чтобы они были скомпилированы в общую проектную библиотеку, причем первичный модуль компилируется раньше подчиненного ему вторичного. При записи первичного и вторичного модуля в одном файле первичный модуль записывается ранее соответствующего ему вторичного.

Типовой текст программы на языке VHDL имеет следующую структуру:

```
<VHDL программа>::=
 «
  « <объявление библиотеки> » « <объявление использования> »
  <первичный модуль>
 »
 « <вторичный модуль> »
<объявление библиотеки> ::= LIBRARY <имя библиотеки>;
```

Иными словами, текст представляет произвольный набор первичных и вторичных модулей, причем каждому первичному модулю может предшествовать указание на библиотеки и пакеты, информация из которых используется для построения этого модуля. Указание использования (Use Clause) применяется для различных целей с определенными модификациями синтаксиса. В данном контексте используется конструкция

```
<объявление использования>::=
 USE <имя библиотеки>.<имя пакета>.<имя модуля> |
 USE <имя библиотеки>.<имя пакета>.ALL
```

Эта запись определяет место хранения используемой информации. Вариант записи USE – ALL обеспечивает доступ ко всем модулям объявленного пакета.

Отметим, что даже если несколько первичных модулей в одном программном файле ссылаются на одинаковые библиотеки и модули, декларация использования предшествует каждому первичному модулю индивидуально.

Упрощенная форма синтаксиса декларации ENTITY имеет вид:

```
<декларация ENTITY> ::=
  ENTITY <имя проекта> IS
   [<объявление параметров настройки>]
      [<объявление портов>]
  END [ ENTITY ] <имя проекта>;
```

```
<объявление параметров настройки> ::=
   GENERIC (<имя>:<тип> [ :=<выражение> ]
           «;<имя>:<тип> [ :=<выражение> ] »
     );
<объявление портов> ::=
   PORT (<имя>:<режим><тип> [<выражение>]
        «;<имя>:<режим><тип> [ =<выражение>]»
   );
<режим> ::= IN | OUT | INOUT
```

Объявление параметров настройки включается в ENTITY для создания проектов, которые предполагается использовать как фрагменты в разнообразных других проектах, причем возможна модификация некоторых свойств встраиваемого фрагмента, точнее выбор параметра из множества значений, определенного типом параметра.

Определение портов задает имена входных (IN), выходных (OUT) и двунаправленных (INOUT) линий передачи информации и тип данных, передаваемых через порты. Объявление портов, как следует из представленных правил синтаксиса, не обязательно. Но возникает вопрос: зачем может понадобиться устройство, не имеющее входов и выходов? Оказывается, такая конструкция позволяет эффективно сочетать описание собственно проектируемого устройства и алгоритм его тестирования. Программы, создаваемые для отладки и называемые Test-Bench, обычно включают описание как самого проектируемого устройства, так и модель внешней среды, в частности генератор тестового воздействия. Такая система внутренне определена.

Для параметров и входных портов можно задавать значение по умолчанию, представляемое выражением, отделенным от типа знаками :=. Эти значения принимаются, если соответствующим единицам информации не присвоены другие значения в модулях высшего уровня иерархии.

Архитектурные тела представляют содержательное описание проекта. Архитектурное тело в некотором смысле подчинено соответствующему ENTITY. Различают *структурные архитектурные тела* (описывающие проект в виде совокупности компонентов и их соединений), *поведенческие архитектурные тела* (описывающие проект как совокупность исполняемых действий) и *смешанные тела*. Формальных признаков, по которым архитектурное тело можно было бы отнести к определенному типу, не вводится — различие в составе используемых в программе операторов. Определены следующие правила записи архитектурных тел:

```
<Архитектурное тело> ::=
   ARCHITECTURE <имя архитектуры> OF <имя ENTITY> IS
      <раздел деклараций>
   BEGIN
      <раздел операторов>
   END [ ARCHITECTURE ]  <имя архитектуры>;
```


Здесь имя архитектуры — это любое индивидуальное имя проектного модуля. Имя ENTITY задает первичный модуль, которому подчиняется архитектурное тело. В разделе деклараций объявляются локальные для этого модуля информационные единицы — типы данных, сигналы, подпрограммы и т. д. Раздел операторов описывает правила функционирования и (или) конструирования устройства в терминах языка.

Одному ENTITY может быть сопоставлено несколько архитектурных тел. Рассмотрим в качестве примера совокупность проектных модулей, представленных в листинге 3.1. Программа описывает устройство, реализующее подсчет числа разрядов входного шестнадцатиразрядного кода input, установленных в состояние логической единицы с выдачей результата в числовой форме на выход output. Одному ENTITY bit_count сопоставлено три архитектурных тела, представляющих функционирование устройства с различной степенью детализации. Формальный синтаксис операторов будет определен далее, но представляется, что читатель, знакомый с программированием в общеупотребительных языках, сможет понять суть записанных преобразований. Следует сделать только два вводных замечания. Во-первых, всякий фрагмент, начинающийся с двойного тире (--) до конца текущей строки является *комментарием*. Во-вторых, присвоение значения в зависимости от контекста задается либо знаком :=, либо знаком <=, детали представлены в последующих разделах.

Архитектурное тело, названное pure_behave (чистое поведение), определяет алгоритм функционирования в самом общем виде и в традиционной форме. Алгоритм описан как последовательный просмотр битов слова с прибавлением единицы к результату, если очередной бит есть логическая единица. Хотя компиляторы доступных САПР способны подготовить по этому описанию процедуры моделирования и даже прямую интерпретацию устройства БИС, фактическая реализация, как показывает практика, не является оптимальной. В данном случае наблюдались излишние затраты на элементы суммирования и неэффективность реализации по быстродействию. Дело в том, что сумма формируется последовательно (заметим, что последовательно не просто во времени, шаг за шагом, как было бы в программных реализациях, а за счет передачи данных через последовательность устройств, прибавляющих единицу к коду).

В то же время "чистое поведение" является эффективным способом спецификации устройств и улучшения взаимопонимания между разработчиками. Кроме того, использование описания в форме "чистого поведения" значительно повышает скорость выполнения моделирования сложных проектов. Тогда отдельные фрагменты, которые в данный момент не интересуют проектировщика, представляются в самой общей форме, а разработчик сосредоточивается на фрагментах, требующих детальной отладки. Для практической реализации целесообразно выполнить декомпозицию устройства. При реализации сложных проектов такая декомпозиция становится еще более акту-

альной, позволяя разработчику временно ограничить поле интересов наиболее критичными участками и организовать их детальную проработку.

Архитектурное тело four_channel_behave предусматривает параллельный подсчет единиц в группах по четыре разряда каждая и суммирование промежуточных значений. При сохранении самого алгоритма компилятору "подсказывается" реализация, обеспечивающая повышение быстродействия за счет частичного распараллеливания выполняемых действий.

И наконец, в архитектурном теле comb_logic предпринимается попытка приблизить описания к реализации в структурах программируемой логики, построенных на основе ячеек типа "четырехвходовая таблица истинности". Функция ones_in_thetrade (число единиц в тетраде) здесь будет реализоваться как набор из трех таких ячеек, т. к. каждый разряд кода результата является функцией четырех разрядов входного кода. Кроме того, в архитектурном теле используются конструкции операторов присваивания, в явном виде задающие задержки компонентов, чем обеспечивается подготовка проекта к моделированию с учетом временных параметров.

Листинг 3.1

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL; -- пакет, содержащий определение
                -- преобразований данных в многозначном алфавите
USE ieee.std_logic_util.ALL; -- пакет, содержащий функции преобразования
                -- форматов из битового вектора
                -- в эквивалентное число и наоборот, и т. п.
ENTITY bit_count IS
PORT( input : IN std_logic_vector (15 DOWNTO 0); -- битовый вектор
                -- в многозначном представлении
     output: OUT integer RANGE 0 TO 15); -- целое в объявленном диапазоне
END  bit_count;

ARCHITECTURE pure_behave OF bit_count IS
-- декларация встроенной подпрограммы --
FUNCTION bits_in_word( x:std_logic_vector (15 DOWNTO 0))
          RETURN  integer IS -
VARIABLE  i,z :integer RANGE 0 TO 15;
BEGIN  z:=0;
  FOR i IN 0 TO 16 LOOP     -- цикл просмотра всех разрядов
     IF  x(i)='1' THEN      -- если разряд установлен в единицу
          z:=z+1;           -- прибавить к результату
     END IF;
  END LOOP;
 RETURN z;
END bits_in_word;
--  конец декларации подпрограммы --
```



```
BEGIN
  output<= bits_in_word(input);  -- вызов подпрограммы
END pure_behave;

ARCHITECTURE four_channel_behave OF bit_count IS
 SIGNAL v1,v2,v3,v4: std_logic_vector (3 downto 0);
 FUNCTION ones_in_word(n:integer;
                       x:std_logic_vector (3 downto 0))
   RETURN integer IS
  VARIABLE i,z : integer;
  BEGIN  z:=0;
    FOR  i IN 0 TO n LOOP
     IF  x(i)='1' THEN
          z:=z+1;
     END IF;
   END LOOP;
  RETURN z;
 END ones_in_word;
BEGIN
  v1<=input(3 downto 0); -- "разрезание" аргумента
  v2<=input(7 downto 4);
  v3<=input(11 downto 8);
  v4<=input(15 downto 12);
- четырехкратный вызов функции подсчета единиц в коде
- и суммирование результатов
  output<=ones_in_word(4,v1)+ones_in_word(4,v2)+
          ones_in_word(4,v3)+ones_in_word(4,v4);
END four_channel_behave;

ARCHITECTURE comb_logic OF bit_count IS
FUNCTION ones_in_thetrade( x:std_logic_vector (3 downto 0))
    RETURN integer IS
 VARIABLE z:integer RANGE 0 TO 4;
 TYPE LUT  IS ARRAY (0 to 15) OF integer;
 CONSTANT result: LUT:=
 (0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,4); -- таблица истинности для функции
                                    -- "число единиц в четырехразрядном коде"
BEGIN
   z:=conv_integer(x);   -- преобразование код - номер в таблице
     return  result(z); -- выборка значения из таблицы
END ones_in_thetrade;
 SIGNAL number1,number2, number3, number4: integer RANGE 0 to 4;
BEGIN
  number1<=ones_in_thetrade (input (3 downto 0)) AFTER 2 ns;
  number2<=ones_in_thetrade (input (7 downto 4)) AFTER 2 ns;
```

```
    number3<=ones_in_thetrade (input (11 downto 8)) AFTER 2 ns;
    number4<=ones_in_thetrade (input (15 downto 12)) After 2 ns;
    output<=(number1+number2)+(number3+number4)  after 10 ns;
END comb_logic;
```

Функция bits_in_thetrade определяет правило работы некоторой подсхемы. Альтернативным способом представления "законченных" фрагментов является объявление *вхождения компонента* (Component Instance), которое содержит ссылку на ENTITY включаемого библиотечного компонента и указание порядка его подключения. Формальные правила включения библиотечных модулей в проект рассмотрены в *разд. 3.2.11.*

В общем случае, архитектурное тело включает сигналы, процессы и компоненты (точнее их декларации). Процессы, т. е. взаимодействующие объекты, в архитектурном теле могут быть представлены специальным оператором PROCESS или так называемыми параллельными операторами. Включаемый компонент в свою очередь представлен в библиотеке архитектурным телом, которое содержит сигналы, операторы и включаемые компоненты. При построении общей модели компилятор выполняет детализацию описания путем увязывания сигналов и компонентов встраиваемых блоков с сигналами модуля высшего уровня иерархии. Если встроенный компонент содержит декларации вхождений других компонентов, детализация продолжается. Детализация проекта заканчивается, когда достигнута декомпозиция на архитектурные тела, представленные "чистым поведением", т. е. совокупностью сигналов и процессов.

3.2.3. Типы данных

Язык VHDL основан на концепции строгой типизации данных, т. е. любой единице информации в программе должно быть присвоено имя, и для нее должен быть определен тип. Определение информационной единицы размещается в разделе деклараций программного модуля, в котором оно используется, или иерархически предшествующего модуля. Тип данных определяет набор значений объектов, отнесенных к этому типу, а также набор допустимых преобразований этих данных. Данные разных типов несовместимы в одном выражении.

Данные, используемые в программах, относятся к одной из категорий: *константы, переменные и сигналы.* Различие между сигналами и переменными определяется в следующем разделе. Декларация объектов имеет следующий синтаксис:

```
<декларация объектов> ::=
    <категория> <имя>«,<имя>» :<тип> [ :=<выражение>];
<категория> ::= CONSTANT |VARIABLE | SIGNAL
```


а декларация может определять несколько объектов. Выражение в декции должно совпадать по типу с декларируемым объектом и задает знаия константы либо начальные значения сигналов и переменных. Пропримеры:

```
TANT a: integer:=15;
ABLE b,c: BIT;
AL  d,e : DOUBLE_WORD;
```

оследнем примере предполагается, что тип DOUBLE_WORD был ранее оплен пользователем.

декларируемый объект является агрегатом, то в одной декларации но определить все компоненты агрегата. Например, пусть определен тип к как массив из восьми целых. Для того чтобы объявить переменную типа и задать ее начальное значение, можно записать:

```
ABLE Stack_instantion: stack:=(4,5,7,129,64,7,87,67);
```

VHDL предопределяет некоторый базовый набор типов данных, котоне требуют объявления в программе пользователя. Кроме того, пользоль может определять свои типы данных. Различают скалярные типы

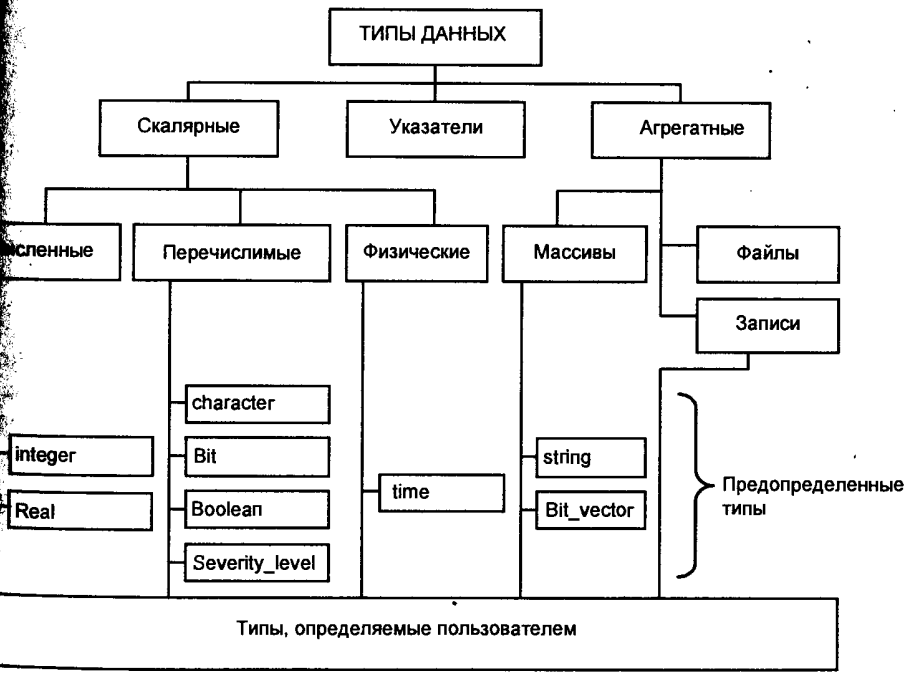

Рис. 3.7. Типы данных языка VHDL

данных и агрегатные типы. Объект, отнесенный к скалярному типу, рассматривается как законченная единица информации. Агрегат представляет упорядоченную совокупность скалярных единиц, объединенных одинаковым именем. Классификация типов данных VHDL приведена на рис. 3.7.

Предопределенные типы данных

Сначала остановимся на предопределенных типах.

```
<предопределенные типы>::=
INTEGER | REAL | BIT | BOOLEAN | CHARACTER | STRING | TIME | BIT_VECTOR|
SEVERITY_LEVEL | FILE_OPEN_STATUS | FILE_OPEN_KIND
```

Типы INTEGER и REAL определяют численные данные — целые и действительные, соответственно. Диапазон представления чисел может зависеть от реализации, но стандартными считаются диапазоны $\{-2^{31}+1, +2^{31}-1\}$ для типа INTEGER и $\{10^{38}, -10^{38}\}$ для REAL. Как уже упоминалось, числа в проектном модуле могут представлять как конфигурационные параметры, так и собственно обрабатываемую в проектируемом устройстве информацию. В последнем случае надо иметь в виду, что число фактически является укороченной записью обрабатываемых кодов, но над этими кодами определены арифметические операции:

- `+` — сложение или повторение;
- `-` — вычитание или инверсия;
- `*` — умножение;
- `/` — деление;
- `mod` — число по модулю (`5 mod 3 = 2`);
- `rem` — остаток от деления;
- `abs` — модуль (абсолютное значение числа);
- `**` — возведение в степень.

Определены для этих кодов также и операции арифметического отношения =, /=, <, <=, >=, >, которые дают результат типа BOOLEAN. В арифметических выражениях предполагаются традиционные способы определения старшинства операций, включая использование скобок.

Данные типа BIT могут принимать значения из множества `('0', '1')`. На данных типа BIT определены логические операции:

- NOT — инверсия;
- OR — операция ИЛИ;
- NOR — операция ИЛИ-НЕ;
- AND — операция И;

- NAND — операция И-НЕ;
- XOR — неравнозначность;
- XNOR — равнозначность.

Замечание

Операции XOR и XNOR в версии VHDL'87 не определены.

Операции определены по правилам положительной логики (`a AND b` дает значение `'1'`, только если оба члена выражения равны `'1'`, а `a OR b` — если хоть один из операндов равен `'1'` и т. д.).

Данные типа BOOLEAN также могут принимать два значения: {TRUE, FASLSE}, и для них определены те же операции, что и над данными типа BOOLEAN. Разница между типами BIT и BOOLEAN состоит в том, что первые используются для представления уровней логических сигналов в аппаратуре, а вторые для представления обобщенных условий, например результатов сравнения. Так, если переменная `select` определена как бит, то нельзя записать условный оператор в виде

```
IF select  THEN ...
```

следует записывать

```
IF select='1' THEN...
```

если бы переменная `select` была определена как BOOLEAN, то, наоборот, первый вариант был бы допустим, а второй нет.

Данные разных типов несовместимы, поэтому недопустимо выражение

```
'0' AND TRUE
```

Тип CHARACTER объединяет все символы, определенные в используемой операционной системе — буквы, цифры, специальные символы. VHDL'87 допускает применение только первых 128 символов кодов ASCII (латинские буквы, цифры, специальные символы). В тексте программы символьная константа записывается как стандартный символ, заключенный в одинарные кавычки (`'a'`, `'b'`, `';'` и т. п.). Отметим, что символы `'0'` и `'1'` имеют двойное назначение — и как символ, и как логическое значение. В каждом конкретном случае тип определяется по контексту.

Тип TIME — время — используется для задания задержек элементов и времени приостанова процессов при моделировании. Запись временной константы имеет вид

```
<целое> <единица измерения времени>
```

Определены такие единицы измерения времени: `fS` — фептосекунда, `pS` — пикосекунда, `nS` — наносекунда, `uS` — микросекунда, `mS` — миллисекунда, `S` — секунда.

Тип SEVERITY_LEVEL задает следующее множество значений: (note, warning, error, failure) и используется для управления работой компилятора или программы моделирования. С помощью переменных и констант этого типа в операторах ASSERT определяются действия, которые следует выполнить при обнаружении некоторых условий: просто выдать сообщение (note или warning), прервать моделирование или компиляцию (error и failure).

Типы FILE_OPEN_STATUS и FILE_OPEN_KIND обеспечивают возможность контроля процедур обмена между программой моделирования с файловой системой инструментального компьютера.

Типы STRING и BIT_VECTOR относятся к агрегатным и фактически определены как неограниченный массив символов и массив битов, соответственно. Более подробно правила использования массивов и их элементов рассмотрены далее. В тексте программы строковая константа заключается в двойные кавычки.

Например, пусть module_type — параметр настройки типа STRING. Тогда запись

```
IF (module_type="СУММАТОР") GENERATE
```

откроет последовательность операторов, которая будет исполняться, только если задано соответствующее значение параметру.

Пользователь имеет возможность определить собственные типы, используя *декларацию типа*:

```
<декларация типа> ::= TYPE <имя типа> IS <определение типа>;
<определение типа> ::=
    <определение перечислимого типа>
    |<определение целого типа>|<определение действительного типа>
    |<определение физического типа>
    |<определение типа массивов>| <определение типа записей>
```

Скалярные типы, вводимые пользователем

Определение *перечислимого типа* имеет вид:

```
<Определение перечислимого типа> ::=
    (перечислимое значение «, перечислимое значение »)
<перечислимое значение> ::=
    <идентификатор> | <символьная константа>
```

Примеры:

- `TYPE state IS (S0,S1,S2,S3)`

  Может представлять, например, набор допустимых состояний системы, для каждого состояния определяются выполняемые действия и правила перехода в другое состояние.

`TYPE colour IS (white, black, red, green, blue, yellow, argenda)`

Набор цветов. Переменные этого типа могут использоваться, например, для управления выводом на дисплей как в сеансах моделирования, так и в реальных устройствах.

```
TYPE std_ulogic IS ( 'U', 'X','0','1','Z','W','L','H','-');
```

Тип std_ulogic и порождаемый на его основе подтип std_logic используются для представления сигналов в девятизначном алфавите *(см. разд. 3.1.3)*. Хотя формально эти типы не относятся к предопределенным, их определение включено в пакет std_logic_1164, являющийся неотъемлемой частью всех современных интерпретаторов языка. Иными словами, эти типы, равно как соответствующий векторный тип std_logic_vector, можно фактически считать предопределенными.

пределение *численных типов пользователя* целесообразно, во-первых, для нтроля совместимости данных в программах, а во-вторых, для точного дания разрядности слов, представляющих данные в проектируемом объек-. В общем случае определение ограниченного типа подчиняется синтаксиому правилу:

```
пределение ограниченного типа> ::=[ <базовый тип> ]  <диапазон>
 <диапазон>::= RANGE <ограничение><направление><ограничение> | RANGE<>
аправление> ::= DOWNTO | TO
```

аправление (TO — увеличение, DOWNTO — уменьшение) должно быть согласовано с соотношением ограничений.

римеры:

```
PE Unsigned_short IS INTEGER RANGE 0 TO 255;
PE  my_data IS INTEGER RANGE  -2**(n-1)+1 TO 2**(n-1)-1;
PE input_level IS -10.0 TO +10.0;
```

ип Unsigned_short объединяет целые положительные числа, которые могут ть представлены в байтовом формате.

ип my_data объединяет целые в диапазоне, который объявляет пользователь рез разрядность данных n. В этом случае пользователь точно указывает мпилятору число разрядов, необходимое для представления данных, обесчивая экономию ресурсов микросхемы по сравнению с неограниченным ипом.

ри объявлении типа input_level базовый тип явно не задан, используется ип ограничений в соответствии с типом их фактических значений.

усть пользователь в одном проекте вводит два типа:

```
YPE data  IS integer RANGE 0 TO 15;
YPE controle IS integer RANGE 0 TO 15;
```

Хотя с точки зрения представления эти типы равноценны, оказывается, что управляющие сигналы controle и сигналы данных data несовместимы, что облегчает контроль корректности описания.

Физические типы

Наряду с предопределенным типом TIME пользователь может определить другие физические типы, которые будут отражать физические (механические, электрические или иные) свойства носителя информации.

<определение физического типа> ::=
```
RANGE <диапазон>
UNITS
  <имя базовой единицы>
   <имя вторичной единицы> = <значение единицы> »
END UNITS [ <имя типа>];
```

Пример:

```
TYPE voltage  IS RANGE -5E6 to +5E6;
  UNITS  uV;       -- базовая единица – микровольт
    mV= 1000 uV;   -- милливольт
    V=1000 mV;     -- вольт
  END UNITS voltage;
```

Введение такого типа позволяет описывать и моделировать сопряжение цифровой логической схемы с аналоговыми источниками. Пусть входной порт analog_input и константа threshhold (порог) объявлены как voltage, а сигнал compare (сравнение) как BIT. Тогда можно записать:

```
compare <= '1' WHEN analog_input>threshhold ELSE
                  '0';
```

Массивы и записи

Массив, как и в других языках, — это набор данных, объединенных общим именем и различаемых по порядковым номерам (индексам). Для того чтобы вводить объект типа массив, необходимо предварительно объявить соответствующий тип на основе следующих синтаксических правил:

<определение типа массива> ::=
```
ARRAY ( <диапазон> «, <диапазон> ») OF
      <тип элемента массива>
```

Диапазон задает множество допустимых значений индекса. Число измерений массива формально не ограничено. Если диапазон задан конструкцией RANGE<>, то это является объявлением неограниченного массива. В этом случае определяется не диапазон значений индекса, а только тип индексной

еременной. Такое определение предполагает задание диапазона при опреении конкретного экземпляра объекта, относимого к такому типу, наимер, при вызове подпрограмм. В подобных случаях диапазон устанавливается динамически в соответствии с диапазоном подставляемого фактичеого параметра.

имеры:

```
PE ram IS ARRAY (length-1 DOWNTO 0) OF integer
                RANGE 2**width-1 DOWNTO 0;
PE ram1 IS ARRAY (length-1 DOWNTO 0,width-1 DOWNTO 0) OF std_logic;
PE ram2 is ARRAY ( integer RANGE<>, integer RANGE <>) OF std_logic;
```

всех приведенных декларациях объявляется в сущности одно и то же, именно матрица ячеек памяти емкостью length слов по width разрядов в ждом, причем предполагается, что эти параметры были ранее определены. нако выполнено это разными способами, а значит, и ссылаться на эти пы следует по-разному. RAM и RAM1 определены как ограниченные типы ссивов, ram — как одномерный массив целых, а ram1 — как двумерный ссив битов. RAM2 определен как неограниченный тип и требует задания аниц индексов при декларации объектов выбранного типа.

кларации объектов, принадлежащих приведенным типам, могут выглядеть едующим образом:

```
RIABLE ram_instance: ram;
RIABLE ram1_instance: ram1;
RIABLE ram2_ins: ram2 (length-1 downto 0,width-1 downto 0);
```

и обращении к элементам массива в программе индексы помещаются скобках следом за именем массива. Тип индексного выражения должен ответствовать типу индекса, объявленного при декларации типа массива. и обращении к элементу многомерного массива индексные выражения писываются через запятые в порядке, определенном в декларации типа.

```
m2_ins(y,5):=x0;      -- x0 определено как бит; y – целое;
=  ram1_instance (Y) -- Y  и  z – целые
```

я одномерных массивов определено несколько групповых операций, которых массив рассматривается как единое целое. Это, прежде всего, опеция конкатенации & (объединение строк). Например, приведенная ниже следовательность операторов присваивает сигналу b значение "11011001".

```
="1001";
= "1101" & a;
```

десь: a и b — строки или битовые векторы, причем a — переменная, b — сигнал.

Операции сдвига определены для одномерных массивов типа BIT или BOOLEAN и записываются следующим образом:

<имя массива> <символ операции сдвига> <целое>

В VHDL'93 определены следующие операции сдвига: логические сдвиги влево и вправо sll и srl, арифметические сдвиги влево и вправо sla и sra, циклические сдвиги влево и вправо rol и ror.

Целое в записи выражения для сдвига определяет число разрядов, на которые осуществляется сдвиг кода.

Замечание

В VHDL'87 операции сдвига не определены.

В составе ряда САПР поставляются пакеты, определяющие арифметические операции над битовыми массивами (кодами). Так, в системе проектирования MAX+plus II арифметические операции над кодами определяются во встроенном пакете std_logic_arith.

Запись — эта структура данных, каждая информационная единица которой, называемая полем записи, имеет индивидуальное имя и может быть индивидуального типа. Обычно записи используются для агрегатирования различных данных, характеризующих один объект. Для использования записей как переменных сначала надо объявить соответствующий тип:

```
<определение типа записи> ::=
    RECORD <список полей записи: тип>;
     «<список  полей записи: тип>;»
    END RECORD;
```

Пример. Определим тип pixel, представляющий цветовые составляющие отображения точки на экране в формате FULL COLOR (полная цветопередача), предусматривающем восьмиразрядное представление трех цветовых составляющих.

```
TYPE pixel IS
   RECORD red,green,blue: integer RANGE 0 TO 255;
   END RECORD;
```

Тогда тип "видеопамять" может быть определен как

```
TYPE video_ram IS ARRAY(integer RANGE<>,integer RANGE <>) OF pixel;
```

Экземпляр видеопамяти будет определяться, например, следующим образом:

```
SIGNAL VRAM : video_ram (679 DOWNTO 0, 839 DOWNTO 0);
```

Этот экземпляр может сохранять информацию об изображении размером 680 строк по 840 элементов в строке. Выборка значения красной состав-

ляющей верхнего левого элемента изображения из такой памяти описывается оператором

```
ut_red <=  VRAM (0,0).red;
```

Следующий пример определяет обобщенный тип для представления конечных автоматов. Автомат, как известно, определяется множеством входов, множеством состояний и множеством выходов, а также соответствующими функциями на этих множествах. Значит, можно ввести универсальный тип

```
YPE state_machine IS
   RECORD  s:state;
           x:machine_input;
           y:machine_output;
   END RECORD;
```

Здесь state, machine_input и machine_output — ранее определенные перечислимые типы.

Функции переходов и выходов конкретного экземпляра автомата будут определяться в разделе операторов соответствующего архитектурного тела.

Подтипы

Специфическим понятием языка VHDL является подтип. Объекты, отнесенные к подтипу, сохраняют совместимость с данными типа, из которого выделяется подтип так называемого базового типа. Однако введение подтипа:

- определяет множество допустимых значений данных подтипа как подмножество допустимых значений базового типа;
- позволяет вводить дополнительные функции преобразования, определяемые только для данных подтипа.

Синтаксис декларации подтипа определен следующим образом:

```
<декларация подтипа> ::=
   <имя подтипа> IS [<имя функции разрешения> ]
   <имя базового типа или подтипа> [<ограничение>];
```

Пример:

```
UBTYPE bit_in_word_number IS integer RANGE  31 DOWNTO 0;
```

Определен подтип типа integer. Данные этого подтипа предполагается использовать для индексации бита в 32-разрядном коде. Данные совместимы с данными типа integer. Однако присвоение этим данным значений вне указанного диапазона вызывает сообщение об ошибке.

3.2.4. Сигналы и переменные. Оператор *PROCESS*

Любой проект является описанием явлений в дискретных системах. Эти явления могут представляться тремя различными категориями данных: константы, переменные и сигналы. SIGNAL — это информация, передаваемая между модулями проекта или представляющая входные и выходные данные проектируемого устройства. Сигналу присваиваются свойства изменения во времени. VARIABLE — это вспомогательная информационная единица, используемая для описания внутренних операций в программных блоках. Присвоение значения сигналу отображается знаком <=, а переменной — знаком :=.

Для того чтобы представить различия сигналов и переменных, следует сделать несколько предварительных замечаний. *В языке VHDL введены два типа операторов — последовательные и параллельные.* Последовательные операторы выполняются последовательно друг за другом в порядке записи. Такие операторы во многом подобны операторам традиционных языков программирования и описывают набор действий, которые последовательно выполняются над исходными данными с целью получения результата. К этому классу операторов относят оператор присваивания переменной, последовательный оператор присваивания сигналу, условные операторы, оператор выбора и ряд других.

Исполнение *параллельных операторов* инициируется не по последовательному, а по событийному принципу, т. е. они исполняются тогда, когда реализация других операторов программы создала условия для их исполнения. Параллельные операторы представляют части алгоритма, которые в реальной системе могут исполняться одновременно. Эти части взаимодействуют между собой и с окружением проектируемой системы. Параллельные операторы могут быть простыми и составными. Составной оператор включает несколько простых операторов, для которых определены общие условия инициализации. Такая совокупность операторов называется *телом составного оператора.* Важнейшим составным оператором является оператор процесса PROCESS, синтаксис которого определен следующим образом:

```
<оператор процесса> ::=
  [ <метка процесса>:] PROCESS[ ( <список инициализаторов>) ] [ IS ]
  <раздел деклараций>
  BEGIN
    « <раздел операторов>»
  END PROCESS [<метка процесса>];
<раздел операторов> ::=« <последовательный оператор>»
```

Ключевое слово IS в версии VHDL'93 является необязательным, а в VHDL'87 недопустимо в данной конструкции.

Последовательные операторы могут записываться только в теле оператора PROCESS. При моделировании фрагменты алгоритма, заключенные в оператор PROCESS, будут исполняться друг за другом после возникновения в системе "инициализирующего события" — изменении одного из сигналов, перечисленных в списке инициализаторов, или в заранее определенный момент времени. Параллельные операторы в теле процесса не определены. Переменные могут быть определены только в теле процесса, а сигналы во всем архитектурном теле.

Замечание

Некоторые параллельные и последовательные операторы совпадают по форме записи. В этом случае различие устанавливается по контексту: если оператор локализован в теле процесса, он трактуется как последовательный, а если в другом месте программы — как параллельный.

Наиболее явно разница между сигналами и переменными проявляется при интерпретации операторов последовательных присвоений. Для обоих видов сохраняется общее для последовательных операторов правило начала исполнения: первый оператор в процессе исполняется после выполнения условий инициализации процесса, а каждый следующий сразу после исполнения предыдущего. Однако результат присвоения переменной непосредственно доступен любому последующему оператору в теле процесса. Трактовка оператора последовательного присвоения сигналу существенно отличается от трактовки присвоения переменной или операторов присваивания в традиционных языках программирования. Присвоение сигналу не приводит непосредственно к изменению его значения. Новое значение сначала заносится в буфер, называемый драйвером сигнала, и следующие операторы в теле процесса оперируют со старыми значениями. Фактическое изменение значения сигнала выполняется только после исполнения до конца процессов и других параллельных операторов, инициированных общим событием, или после исполнения оператора останова WAIT *(см. разд. 3.2.6)*.

Общие правила интерпретации оператора PROCESS можно свести к следующим:

- PROCESS "запускается" при изменении любого сигнала, перечисленного в списке инициализаторов.

  Если список инициализаторов пуст, то процесс безусловно исполняется при начальном запуске, а также сразу за исполнением последнего оператора в разделе операторов этого процесса. При этом надо иметь в виду, что оператор процесса без списка инициализаторов обязательно должен содержать в своем теле оператор ожидания WAIT. Иначе исполнение любых других операторов в программе блокируется.

- Все операторы раздела операторов выполняются подряд друг за другом от начала до конца, за исключением случаев приостановки исполнения дей-

ствий оператором WAIT. Тогда после приостановки может быть инициировано исполнение других процессов и параллельных операторов, а реализация операторов, следующих за оператором WAIT, продолжится после наступления события, объявленного в этом операторе.

Раздел деклараций определяет локальные объекты, используемые в следующем за ним программном блоке, в данном случае, в разделе операторов.

Рассмотрим проектный модуль, объявленный ENTITY two_process_example и представленный архитектурным телом test (листинг 3.2). Проект не имеет портов. Это является иллюстрацией реализации Test-Bench, т. е. внутренне определенного модуля, создаваемого для проверки функционирования некоторого узла. Модуль содержит описание исследуемого узла, в данном случае комбинационной схемы, представленной оператором, отмеченным меткой u1, и генератора тестового воздействия, выделенного меткой stimulator. Связи между узлами проекта представлены сигналами x0, x1 и x2. Кроме того, выход логической схемы представлен сигналом z, который может отражаться в качестве результата программой моделирования. Операторы PROCESS включают определение вспомогательных внутренних переменных y1, y2, i и stim_vector. Отметим, что stim_vector — вектор стимулирующего возмущения — является кодовым эквивалентом номера цикла моделирования и получается из него с использованием функции conv_vector, определенной в пакете std_logic_util. Процесс stimulator безусловно запускается в начале сеанса моделирования. Оператор WAIT приостанавливает исполнение последовательности вложенных операторов на 50 nc модельного времени. В этот момент изменения сигналов x0, x1 или x2 инициируют исполнение процесса u1, все вложенные операторы которого безусловно исполняются друг за другом до конца.

Интересно, что единственное изменение переменной stim_vector и соответствующих сигналов может вызвать несколько повторений выполнения операторов в процессе u2. Дело в том, что для процесса u2 изменение любого бита входного вектора является событием, вызывающим его исполнение, а значит, если в векторе меняются несколько битов, процесс инициируется несколько раз. Это можно наблюдать при моделировании в пошаговом режиме в системе интерпретации VHDL-программ фирмы Model Technology.

По истечении времени приостанова, если тест еще не выполнен до конца, происходит повторение операторов процесса stimulator с начала. Заметим, что переменные в VHDL трактуются как статические данные, т. е. при каждом следующем исполнении процесса используются значения, определенные при предыдущем исполнении вплоть до очередного присвоения.

Когда тестовая последовательность исчерпана (в данном случае i=8, и выданы все необходимые кодовые комбинации), оператором ASSERT выдается сообщение, и сеанс оканчивается оператором "бесконечного останова" WAIT.

стинг 3.2

```
BRARY ieee
E ieee.std_logic_1164.ALL;
E work.std_logic_util.all;
E STD.textio.ALL;

TITY two_process_example IS
D lab1;

CHITECTURE test OF two_process_example IS
 SIGNAL  z, x0,x1,x2:std_logic;
GIN
:PROCESS(x0,x1,x2)
RIABLE y1,y2:std_logic;
GIN
1:=x0 and (not x1) and x2;
2:=x0 and (not x1) and (not x2);
<= y1 or y2;
D PROCESS;

imulator: PROCESS
ARIABLE i:integer:=0;
ARIABLE stim_vector:std_logic_vector(2 DOWNTO 0);
BEGIN
    stim_vector:=conv_std_logic_vector(i,3);
    x0<= stim_vector(0);
    x1<= stim_vector(1);
    x2<= stim_vector(2);
    WAIT FOR 50 ns;
    i:=i+1;
   if (i=8) then
    ASSERT false REPORT "End of Stimulation !"
      SEVERITY NOTE;
      WAIT;
  END IF
END PROCESS;
ND test;
```

Укажем на некоторые наиболее существенные различия сигналов и переменных.

- Переменные меняют значения сразу после присвоения, и новые значения непосредственно учитываются во всех преобразованиях, записанных в теле процесса после такого присвоения.

- Значение сигнала меняется не сразу после выполнения присвоения. Оператору присваивания сопоставляется некий буфер, называемый контейнером или, чаще, драйвером сигнала [6, 34]. Оператор присваивания передает новое значение драйверу сигнала, и лишь после того, как выполнены преобразования во всех процессах, инициированных общим событием, содержание драйвера передается сигналу. Передача значения сигналу может быть еще более задержана, если оператор присваивания содержит выражение задержки AFTER.
- Переменная определена только внутри тела процесса, сигнал — во всем архитектурном теле.
- Переменной можно переприсваивать значение в теле процесса. Сигнал внутри одного процесса может иметь только один драйвер. То есть присвоение значения сигналу может быть выполнено только один раз в теле процесса (на различных несовместимых путях реализации алгоритма могут быть несколько операторов присваивания значений одному сигналу).

Проследим эту разницу на примерах. В листингах 3.3, *а* и 3.3, *б* приведены фрагменты программ, описывающих одну и ту же совокупность преобразований, но в первом случае именем cS обозначен сигнал, а во втором именем cV — переменная.

Листинг 3.3, *а*

```
Signal clk: bit;
signal  cS, b:  integer;
constant a,d: integer:=10;
.....
    process (clk)
     begin       cS<= a;
                   b<=cS+d;
-- cS<=b; --недопустимо
    end process;
```

Листинг 3.3, *б*

```
Signal clk:bit;
signal   b:  integer;
constant a,d: integer:=10;
    process (clk)
 variable cV;
     begin       cV:= a;
                   b<=cV+d;
                   cV:=b
    end process;
```

Если перед исполнением этих процессов (после изменения clk) сигнал cS и переменная cV имели значение 4, а b = 3, то после реализации процесса в листинге 3.3, *а* cS и b примут значение 14, т. к. присвоение значения сигналу cS в первом операторе тела не влияет на следующие операторы.

Во втором случае при тех же исходных данных после завершения процесса получим b = 20 и cV = 20.

2.5. Атрибуты в языке VHDL

рибуты — скаляры, отражающие некоторые свойства объектов, исполь-мых в программных модулях (типов, переменных, агрегатов). Например, рибуты типа используются для сжатого представления информации о ожестве значений, объединенных типом, а атрибуты сигнала — для пред-вления временных свойств сигнала. В разделах операторов нельзя при-ивать значение атрибуту, способ его определения задается декларацией рибута. Атрибуту присваивается имя и тип, имя используется как обычная еменная в выражениях того типа, который присвоен атрибуту. Имя атри-та записывается следующим образом:

```
ия атрибута> ::=
<имя атрибутируемого объекта>'<конструктор атрибута> [(<выражение>)]
```

нструктор атрибута определяет свойство объекта, представляемое атрибу-м. Необязательное выражение может задавать дополнительные данные, пользуемые для вычисления значения атрибута.

вают предопределенные атрибуты и атрибуты, вводимые программистом. данной книге мы ограничимся только представлением наиболее употреби-ьных предопределенных атрибутов.

редопределенные атрибуты типов приведены в табл. 3.2. Здесь T — имя па. N — целое, а X — "вспомогательное" выражение, тип которого совпада-с типом T. Тип перечисленных атрибутов, кроме T'pos и T'image, совпа-ет с атрибутируемым типом. Атрибут T'pos — принимает целое значение, T'image — строка.

***Таблица 3.2.** Предопределенные атрибуты типов*

Вид атрибута	Вычисляемое значение	Атрибутируемый тип
T'left	Левая граница значений T	Любой скалярный
T'right	Правая граница значений T	Любой скалярный
T'low	Нижняя граница значений T	Численный, физический
T'high	Верхняя граница значений T	Численный, физический
T'image(X)	Строка символов, представляющая значение X	Любой
T'pos(X)	Позиция значения X в наборе значений T	Перечислимый
T'val(N)	Значение элемента в позиции N в наборе значений T	Перечислимый, физический, целый
T'leftof(X)	Значение в наборе значений T, записанное в позиции слева от X	Перечислимый, физический, целый

Таблица 3.2 (окончание)

Вид атрибута	Вычисляемое значение	Атрибутируемый тип
`T'rightof(X)`	Значение в наборе значений `T`, записанное в позиции справа от `X`	Перечислимый, физический, целый
`T'pred(X)`	Значение в наборе значений `T` на одну позицию меньшее `X`	Перечислимый, физический, целый
`T'succ(X)`	Значение в наборе значений `T` на одну позицию большее `X`	Перечислимый, физический, целый

Для перечислимых типов номер позиции значения отсчитывается от нуля, присвоенного крайнему левому значению с инкрементом номера позиции для каждого следующего значения. Для перечислимых типов справедливо:

```
T'leftof = T'pred;                    (3.1)
T'left = T'low;                       (3.2)
T'right = T'high;                     (3.3)
T'rightof = T'succ.                   (3.4)
```

Например, для типа `std_logic` справедливо:

```
Std_logic'low = 'U';
Std_logic'pos('1') =  3;
Std_logic'val (7) = 'H';
Std_logic'succ('Z') = 'W'
```

Пусть совокупность возможных состояний некоторого устройства объявлена типом

```
TYPE state IS :=( s0, s1, s2, s3, s4, s5)
```

а текущее состояние представлено сигналом `current_state`. Рассмотрим поведение фрагмента, представленного процессом:

```
PROCESS
   IF current_state = state'right THEN
        current_state<=state'left;
    ELSE current_state <=state'succ(current_state);
   WAIT 100 nS;
  END IF;
END PROCESS;
```

Через каждые 100 нс происходит инициализация этого процесса и изменение состояния. Устройство поочередно принимает состояния в порядке

записи в списке значений типа от `s0` до `s5`, а из `s5` выполняется переход в начальное состояние `s0`.

Для атрибутов целых типов номер позиции совпадает с фактическим значением вспомогательного выражения, а для атрибутов физических типов — числом базовых единиц в значении вспомогательного выражения. Может показаться, что такое определение является тавтологией. Однако это не так. Позиция, в отличие от аргумента, не несет физического смысла и, как говорят, представлена "анонимным целым", позволяющим объединять в выражениях свойства различных носителей информации.

Пусть требуется определять заряд `Q`, передаваемый по некоторой цепи постоянным током `I` за время `T`. Тогда следует объявить соответствующие типы и переменные:

```
YPE current IS range integer'low TO integer'high;
    UNITS  uA;          -- микроампер;
           mA= 1000 uA; -- миллиампер;
           A=1000 mA;   -- ампер;
  END UNITS current;
YPE charge IS range integer'low to integer'high
    UNITS  pQ;          -- пикокулон;
           uQ= 1000 pQ;
  END UNITS current;
ARIABLE I : current;
ARIABLE Q : charge;
ARIABLE T : time;
```

Тогда оператор вычисления заряда можно записать следующим образом:

```
:= charge'val ( current'pos( I) * time'pos( T) * E-9);
```

Множитель `E-9` согласует размерности единиц измерения. Напомним, что базовой единицей времени является фептосекунда (1 фс = 10^{-15} с).

Смысл конструкторов `leftof`, `pred`, `left`, `low`, `right`, `high`, `rightof`, `succ` для целых и физических типов зависит от направления (`TO` или `DOWNTO`), объявленного в декларации типа. Если объявлено прямое направление, то справедливы правила (3.1—3.4), а если обратное, то `T'left= T'high; T'leftof= T'succ` и т. д.

Предопределенные *атрибуты массивов*, приведенные в табл. 3.3, упрощают запись подпрограмм и описаний настраиваемых модулей. Они, в частности, позволяют записывать границы обработки безотносительно к фактическому размеру массива.

В табл. 3.3 `A` — имя типа массива, а `N` — порядковый номер измерения многомерного массива. Для одномерного массива `N=1`, но можно выражение в скобках при записи атрибута вообще опускать. Тип результата всегда сов-

падает с типом индекса. Смысл конструкторов `left`, `low`, `right`, `high` такой же, как у конструкторов типов.

***Таблица 3.3.** Предопределенные атрибуты массивов*

Имя атрибута	Результат
`A'left(N)`	Левая граница диапазона индексов *N*-й координаты массива A
`A'right(N)`	Правая граница диапазона индексов *N*-й координаты массива A
`A'low(N)`	Нижняя граница диапазона индексов *N*-й координаты массива A
`A'high(N)`	Верхняя граница диапазона индексов *N*-й координаты массива A
`A'range(N)`	Диапазон индексов *N*-й координаты массива A
`A'reverse_range(N)`	Обратный диапазон индексов *N*-й координаты массива A
`A'length(N)`	Диапазон индексов *N*-й координаты массива A

В качестве примера листинг 3.4 представляет декларацию функции `glue`, аргумент которой есть битовый массив типа `arr_type`, который определен в вызывающей программе, а возвращаемое значение — вектор, объединяющий четыре старших и четыре младших бита аргумента.

Листинг 3.4

```
FUNCTION glue (x:arr_type) return bit_vector(7 downto 0) IS
BEGIN
RETURN ( x(arr_type'right downto arr_type'right-3)
        & x(arr_type'left+3 downto arr_type'left));
END glue;
```

Замечание

Конструктор `range` возвращает не одно значение, а множество значений "от–до". Например, для типа массива

```
TYPE byte IS ARRAY (7 downto) OF bit;
```

атрибут `byte'range` принимает значение `"7 DOWNTO 0"`, атрибут `byte'reverse_range` — значение `"0 TO 7"`, а атрибут `byte'length` — значение 8.

Атрибуты сигналов являются эффективным средством анализа поведения сигнала во времени. В табл. 3.4 символ `S` означает имя сигнала.

***Таблица 3.4.** Атрибуты сигналов*

Имя атрибута	Тип атрибута	Значение
`S'DELAYED (T)`	То же, что у `S`	Значение `S`, существовавшее на время `T` перед вычислением атрибута
`S'EVENT`	`BOOLEAN`	Сигнализирует об изменении сигнала
`S'STABLE`	`BOOLEAN`	`S'STABLE = not S'EVENT`
`S'ACTIVE`	`BOOLEAN`	`TRUE`, если присвоение сигналу выполнено, но значение еще не изменено (не кончен временной интервал, заданный выражением `after`)
`S'QUIET`	`BOOLEAN`	`S'ACTIVE = not S'QUIET`
`S'LAST_EVENT`	`TIME`	Время от момента вычисления атрибута до последнего перед этим изменения сигнала
`S'LAST_ACTIVE`	`TIME`	Время от момента вычисления атрибута до последнего присвоения значения сигналу (не совпадает с `last_event` при наличии слова `after` в определяющем выражении)

Например, в конструкции

```
F current_state'event THEN <оператор>;
```

ложенный оператор будет исполняться только, если в момент инициализации исполнения этой конструкции сигнал `current_state` меняет свое значение.

ыражение

```
a'DELAYED (5 ns)
```

стинно, если сигнал `a` не менял свое значение в течение 5 нс модельного ремени до вычисления этого выражения.

.2.6. Последовательные операторы

оследовательные операторы (Sequential Statement) по характеру исполнения подобны операторам традиционных языков программирования. Операторы этого типа обязательно "вложены" в оператор `PROCESS` или подпрограмму и выполняются последовательно друг за другом в порядке записи. Результаты исполнения последовательных операторов недоступны прочим программным модулям по крайней мере до того, как будет выполнен оператор ожидания `WAIT`, или не будут выполнены до конца все процессы, инициированные общим событием. Это можно трактовать так, что с точки зре-

ния "окружения" все операторы, в теле процесса от его начала до оператора WAIT, а при отсутствии WAIT — до конца тела, исполняются одномоментно. При использовании выражения задержки сигнала AFTER изменение сигнала прогнозируется на еще более отстоящий момент времени.

Ниже приведен полный список последовательных операторов языка.

```
<последовательный оператор> ::=
<оператор ожидания>
| <оператор проверки>
| <последовательное сигнальное присваивание>
|  <присваивание переменной>
| <вызов процедуры>
| <условный оператор>
| <оператор выбора>
| <оператор повторения>
| <оператор перехода к новому циклу>
| <оператор выхода из цикла>
| <оператор возврата>
| <пустой оператор>
```

Перейдем к последовательному рассмотрению этих операторов (исключение составят операторы вызова процедуры и возврата, которые будут рассмотрены специально в *разд. 3.2.9*).

Операторы присваивания

Эти операторы уже затрагивались в *разд. 3.2.2* и *3.2.3*. Здесь подробнее рассмотрим *последовательное сигнальное присваивание* (необходимо отличать последовательное сигнальное присваивание от параллельного присваивания, подробно рассмотренного в *разд. 3.2.7*). Синтаксическая формула оператора присваивания значения сигналу имеет вид:

```
<оператор присваивания сигналу> ::=
    <приемник> <= [  <модель задержки>] <прогноз поведения>;
<модель задержки> ::=
    TRANSPORT | [ REJECT <выражение времени> ] INERTIAL
<прогноз поведения> ::= <элемент поведения> «, <элемент поведения> »
<элемент поведения> ::= <значащее выражение> [AFTER <выражение времени>]
```

Приемник — объект сигнальной категории, представленный простым именем или компонентом агрегатного сигнала. Прогноз поведения (в стандартах VHDL — Wave-form, временная диаграмма) задает порядок изменения сигнала после события, инициирующего исполнение этого оператора. При этом временные интервалы для определения переходов задаются относительно времени возникновения инициирующего события.

Значащее выражение — любое выражение, дающее результат того же типа, что и приемник.

Например, сигнал, соответствующий временной диаграмме, приведенной на рис. 3.8, описывается оператором:

```
<='1' AFTER 5ns,'0'AFTER 20ns,'Z' AFTER 50 ns,'0' AFTER 70 ns;        (3.5)
```

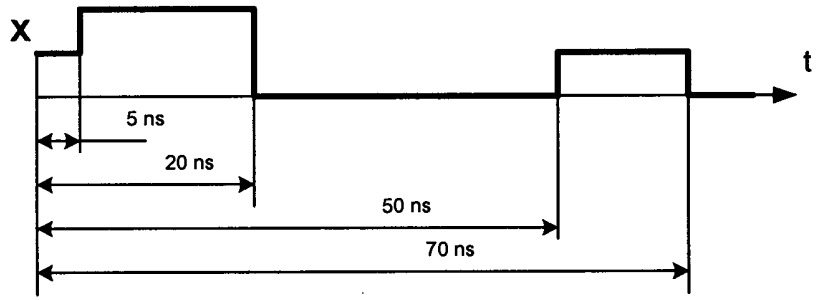

Рис. 3.8. Пример временной диаграммы

Если временное выражение опущено, полагается нулевая задержка (так называемая дельта-задержка): изменение (если оно действительно предсказано при вычислении значащего выражения) заносится в календарь событий той же отметкой времени, что и инициирующее событие.

Если в некоторый момент модельного времени выполняется присвоение сигналу, для которого ранее были предсказаны переходы, часть этих переходов может быть исключена. Такая ситуация возникает, например, когда процесс, содержащий оператор присваивания, инициируется несколькими сигналами, изменения которых отстоят друг от друга на время меньшее, чем определено в прогнозе поведения (напомним, что каждое изменение любого сигнала из списка инициализаторов вызывает исполнение тела процесса).

Порядок исключения зависит от принимаемой модели задержки. Различают транспортную и инерционную модели задержки. Если в операторе присваивания присутствует ключевое слово TRANSPORT, предполагается транспортная задержка. Транспортная модель предполагает идеализацию поведения устройства так, что любой импульс, сколь коротким он бы ни был, воспроизводится на выходе. В этом случае из временной диаграммы (фактически, из календаря событий) исключаются все переходы, которые были предсказаны на время, позднее первого из новых объявляемых переходов, и добавляются новые переходы.

Если через 10 нс после выполнения присвоения (3.5) выполнено

```
x<= transport 'Z' after 35 ns;
```

будет сформирована временная диаграмма, представленная на рис. 3.9. Штриховая линия для сравнения представляет прогноз, сформированный присвоением (3.5).

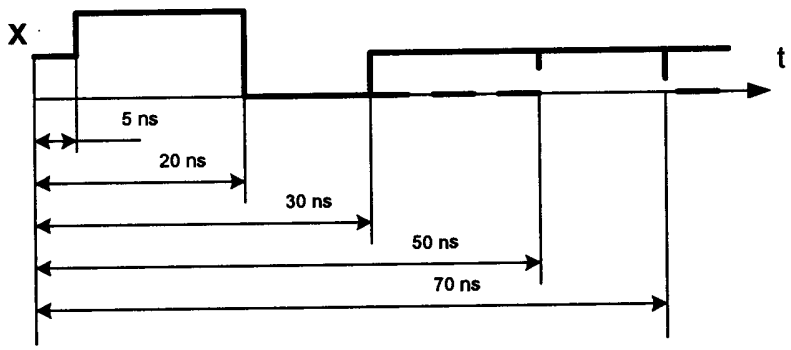

Рис. 3.9. Исключение переходов при транспортной модели задержки

По умолчанию, а также если использовано объявление INERTIAL, предполагается инерционная задержка. В VHDL'87 слово INERTIAL не определено, инерционная задержка выбирается по умолчанию при отсутствии опции "модель задержки".

Инерционная модель используется для описания устройств, не реагирующих на импульсы, длительность которых меньше некоторого наперед заданного значения. В этом случае, подобно транспортной задержке, в календарь событий добавляются новые предсказанные переходы и удаляются все переходы, предсказанные в предшествующих присвоениях на время большее времени нового прогнозируемого перехода. После этого просматривается интервал модельного времени, который предшествует новому предсказываемому переходу и длительность которого определена временным выражением в подстроке REJECT. Все переходы в этом интервале, которые приводят к значению, отличающемуся от нового предсказания, удаляются. Если ключевое слово REJECT отсутствует, то интервал, в котором выполняется такое удаление, определяется значением, указанным после слова AFTER.

Пусть через 10 нс модельного времени после выполнения присвоения (3.5) выполнен оператор:

```
x<= INERTIAL 'Z' AFTER 20 ns;
```

Тогда временная диаграмма модифицируется до вида, приведенного на рис. 3.10. Нетрудно видеть, что короткий переход в состояние логического нуля удален.

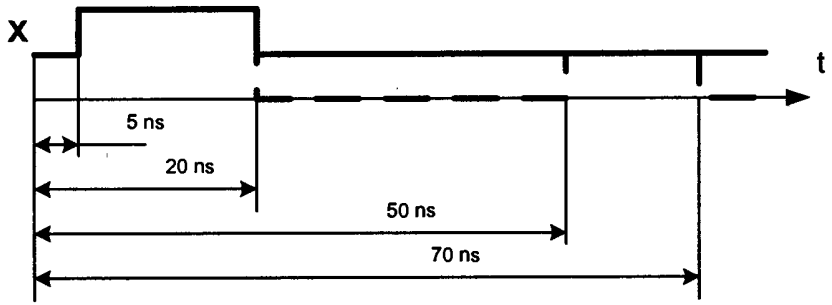

Рис. 3.10. Исключение переходов при инерционной модели задержки

Оператор условия и оператор выбора

Оператор условия IF и оператор выбора CASE позволяют описывать совокупности действий, некоторые из которых исполняются при возникновении определенных условий в реальном устройстве и при моделировании, а иные при тех же условиях не исполняются.

```
<оператор условия> ::=
   IF <булевское выражение>THEN
      <оператор> «<оператор> »
   «ELSIF <булевское выражение> THEN
      <оператор> «<оператор> »
        »
   [ ELSE <оператор> «<оператор> »  ]
   END IF;
```

В качестве операторов в приведенной конструкции могут выступать любые последовательные операторы, в том числе и операторы условия или выбора. В этом случае говорят об иерархическом вложении операторов. Формальных ограничений на глубину вложений не вводится, хотя надо иметь в виду, что некоторые компиляторы могут оказаться неспособны выполнить прямую реализацию в аппаратуре синтаксических конструкций с большим числом уровней вложения. Для пояснения порядка исполнения оператора рассмотрим менее формальную, но достаточно обобщенную форму его записи:

```
IF B1 THEN S1
ELSIF B1 THEN S1
ELSIF B2 THEN S2
. -- и т. д.
ELSIF Bn THEN Sn
ELSE  Sn+1
END IF;
```

где B_1, B_2, B_3, ..., B_i, ..., B_n — булевские выражения; S_1, S_2, S_3, ..., S_i, ..., S_n, S_{n+1} — совокупности последовательных операторов.

Сначала выполняется последовательное, в порядке вхождения в оператор условия, вычисление булевских выражений B_1, B_2, B_3, ..., B_i до тех пор, пока для одного из них (допустим, B_i) не получено значение TRUE. Тогда выполняется одна и только одна совокупность S_i, записанная непосредственно за выражением "ELSIF B_i THEN". Любая совокупность S_i может содержать несколько последовательных операторов или всего один оператор, но не менее одного последовательного оператора. После этого исполнение оператора условия прекращается, т. е. последующие булевские выражения не проверяются и, соответственно, совокупности S_j для $j>i$ не выполняются. Если при каком-либо условии никаких действий не предусмотрено, то все равно после этого выражения проверки данного условия должен размещаться оператор, в данном случае пустой оператор.

Замечание

Пустой оператор NULL, в принципе, может быть записан в любом месте в теле процесса или подпрограммы. Но именно в операторах условия и выбора его использование имеет явно определенный смысл.

Если вычисление ни одного из булевских выражений не дало значения TRUE, выполняется совокупность S_{n+1}, а если при этом ключевое слово ELSE отсутствует, то не выполняется никаких действий.

Разберем несколько примеров.

Сначала рассмотрим простейшую конструкцию без использования ключевого слова ELSE:

```
IF en='1' THEN register_state:= data;
END IF;
```

Здесь при en, не равном логической единице, ничего не происходит.

Процесс, представленный в листинге 3.5, содержит условный оператор в "полной форме". Нетрудно убедиться, что описана реализация двух логических функций:

```
Q= a AND b; P= a XNOR b
```

Листинг 3.5

```
example_IF_1:PROCESS (a,b)
BEGIN
IF a = '1' AND b = '1' THEN
    q <= '1'; P<= '1'
 ELSIF a = '0' AND b = '0' THEN
    q <= '0'; p<='1'
ELSE q <= '0'; p<='0'
END IF;
END PROCESS example_IF_1;
```


следующем примере (листинг 3.6) описан регулятор температуры, входами которого являются сигнал датчика температуры temperature и логический сигнал включения регулировки enable_regulation, а выходами — логические сигналы intensive_heat — сигнал включения интенсивного нагрева, low_heat — слабый подогрев, cooling — охлаждение, например, включение вентилятора.

Листинг 3.6

```
TITY temperature_controle IS
 PORT ( temperature: IN integer ( 50 downto 0);
            enable_regulation: IN bit;
             intensive_heat, slow_heat, cooling: OUT bit)
ND temperature_controle;
RCHITECTURE behave OF temperature_controle IS
EGIN
 PROCESS
BEGIN
 IF Enable_regulation='0' then Intensive_Heat<='0';--прекращение работы
                               Slow_heat<='0';
                               Cooling<='0';
 ELSIF temperature>30 then -- повышенная температура
                    Intensive_Heat<='0';
                    Slow_heat<='0';
                    Cooling<='1';
 ELSIF temperature>20 then  NULL; --   нормальная  температура
 ELSIF temperature>10 then  -- слегка пониженная  температура
                             Intensive_Heat<='0';
                              Slow_heat<='1';
                              Cooling<='0';
  ELSE    Intensive_Heat<='1'; -- очень низкая температура
                            Slow_heat<='0';
 WAIT FOR 1 sec;
 END PROCESS;
D BEHAVE;
```

NF-форма *оператора выбора* имеет вид:

```
ператор выбора> ::=
  CASE <Ключевое выражение> IS
    WHEN <вариант> «|<вариант>» => <оператор> « <оператор> »
  « WHEN <вариант> «|<вариант>»  => <оператор> «<оператор> » »
  END CASE;
ариант> ::= <константное выражение> | <диапазон> | OTHERS
```

В качестве разделителя в списках выбираемых вариантов используется вертикальная черта, т. е. в определении оператора выбора вертикальная черта это не метасимвол, в отличие от любых других определений, а синтаксический элемент определяемой конструкции.

Тип константного выражения или диапазона в записи варианта совпадают с типом ключевого выражения. В частности, в качестве этого выражения может использоваться строка, которая соответствует значению битового вектора или вектора типа std_logic.

При каждом исполнении оператора выбора реализуется единственная последовательность вложенных операторов, а именно та, которой предшествует вариант, совпадающий со значением ключевого выражения в момент исполнения оператора. Если вариант представлен диапазоном, то соответствующая последовательность операторов исполняется при условии, что значение ключевого выражения принадлежит этому диапазону.

Ключевое слово OTHERS определяет операторы, которые исполняются, если значение ключевого выражения не совпадает ни с одним вариантом и не входит в объявленные диапазоны. Если алгоритм предусматривает варианты, при которых не производится никаких действий, то в операторной части таких вариантов записывается пустой оператор NULL.

Практически любой разветвленный фрагмент можно описать с использованием и оператора условия, и оператора выбора. Применение одного из способов записи — дело вкуса программиста. Тем не менее, оператор CASE лучше использовать, когда выбор ветви алгоритма связан с одной переменной, принимающей дискретное множество значений. Иногда набор условий легко приводится к одной такой переменной. Например, фрагмент программы в листинге 3.5 может быть преобразован в эквивалентную программу, представленную в листинге 3.7.

Листинг 3.7

```
example_case_1:PROCESS (a,b)
VARIABLE d: std_logoc_vector (1 DOWNTO 0);
BEGIN
  d:=a & b
  CASE d IS
    WHEN "11" => q <= '1'; P<= '1';
    WHEN  "00" => q <= '0'; p<='1';
    WHEN  OTHERS=>q <= '0'; p<='0';
  END CASE;
END PROCESS;
```

Следующий фрагмент (листинг 3.8) иллюстрирует использование в качестве выражения варианта констант перечислимого типа, а также описание одинаковых действий для нескольких вариантов. Описан порядок формирования адреса памяти address для некоторого процессорного элемента в зависимости от выполняемого цикла обмена. Выполняемый цикл обмена задается сигналом bus_controle, принадлежащим типу bus_operations. Список значений типа и их смысл представлен в листинге. В различных циклах в качестве источника кода адреса могут выступать счетчик адреса команд (program_counter), регистр команд (command_register) или регистр данных (data_register).

Листинг 3.8

```
ARCHITECTURE function_description OF address_generator
TYPE bus_operations IS
  fetch,                          -- цикл выборки команды
  immediate_read,                 -- выборка непосредственного операнда
  direct_read, direct_write,      -- чтение и запись
                                  -- с прямой адресацией памяти
  io_read, io_write,              -- ввод и вывод на внешние устройства
  indirect_read, indirect_write   -- чтение и запись по косвенному адресу

SIGNAL  bus_controle: bus_operations;
SIGNAL program_ counter,        -- счетчик адреса команд
       command_register,        -- регистр команд
       data_register            -- регистр данных
     : std_logic_vector (31 downto 0);
BEGIN
PROCESS(bus_controle)
BEGIN
 CASE bus_controle IS
     WHEN fetch | immediate_read => adress <=program_ counter;
     WHEN direct_read | direct_write | io_read | io_write =>
          adress <=command_register;
     WHEN indirect_read | indirect_write  =>adress <= data_register;
 END CASE;
END PROCESS;
END function_description;
```

Используя задания набора вариантов из диапазона значений, можно операцию выбора в предыдущем примере записать более коротко:

```
CASE bus_controle IS
 WHEN fetch | immediate_read => adress <=program_ counter;
 WHEN direct_read TO io_write =>adress <=command_register;
 WHEN indirect_write DOWNTO indirect_read  =>adress <= data_register;
END CASE;
```

При аппаратной интерпретации оператора условия и оператора выбора в устройстве фактически реализуются подсхемы, исполняющие все возможные альтернативы, из которых в конкретной ситуации инициализируется только одна.

Оператор ожидания

Исполнение операторов, записанных в теле процесса, приостанавливается, если очередной оператор является оператором ожидания (фактически — оператором приостанова) WAIT. При этом результаты исполнения предшествующих операторов заносятся в календарь событий и могут быть инициализированы другие процессы. Прекращения состояния приостанова процесса зависит от условий, определенных в операторе WAIT. Определено несколько модификаций оператора WAIT:

```
<оператор ожидания> ::=
     WAIT;
    |WAIT ON <имя сигнала> «,<имя сигнала> »;
    |WAIT UNTIL <условие>;
    |WAIT FOR <выражение времени>;
```

Вариант оператора WAIT без дополнительных уточняющих конструкций соответствует "бесконечному останову". В этом случае после достижения такого оператора процесс никогда больше не будет исполняться. Указанную версию можно использовать для описания процедур инициализации систем, а также фрагментов, работа которых при некоторых условий прекращается навсегда. Обычно такой оператор завершает программы генерации тестов, означая окончание тестовой последовательности.

Список сигналов в варианте WAIT ON эквивалентен списку инициализаторов процесса: продолжение исполнения будет продолжено после того, как один из сигналов списка изменит свое значение. Так процесс c_1, представленный фрагментом программы в листинге 3.9, эквивалентен процессу example_if_1, представленному в листинге 3.5, но, в общем случае, в теле процесса может быть несколько операторов WAIT, каждый из которых отделяет фрагмент, инициируемый своей совокупностью событий.

Приостанов, заданный конструкцией WAIT UNTIL, заканчивается, когда выполнено заданное оператором условие, т. е. соответствующее выражение принимает значение TRUE. Процесс c_2, представленный листингом 3.10, по составу операторов подобен процессу c_1, но фактическое исполнение существенно отличается. Здесь, например, после выполнения присвоений, предусмотренных строкой, отмеченной комментарием "останов 1", устройство и его программная модель не реагируют ни на какие изменения переменных a и b до тех пор, пока они одновременно не примут значение логического нуля.

Листинг 3.9

```
_1:PROCESS
EGIN
AIT ON a,b;
  IF a = '1' AND b = '1' THEN
   q <= '1';p<='1';
  ELSIF a = '0' AND b = '0' THEN
   q <= '0'; p<='1';
  ELSE q <= '0'; p<='1';
ND IF;
ND PROCESS c_1;
```

Листинг 3.10

```
c_2 : PROCESS
BEGIN
WAIT UNTIL a = '1' AND b = '1';
   q <= '1'; p<='1';-- останов 1
WAIT UNTIL a = '0' AND b = '0';
    q <= '0'; p<='1';
WAIT ON a,b;
q <= '0'; p<='1';
END PROCESS c_2;
```

ариант ожидания по времени иллюстрируется процессом Generator, предтавленным в листинге 3.11. Данный процесс выполняется "бесконечно", риостанавливаясь каждые 50 нс модельного времени, причем перед приосновом уровень сигнала clock меняется на противоположный. В момент риостанова могут быть инициированы параллельные операторы програмы, в том числе другие процессы.

Листинг 3.11

```
nerator: PROCESS
GIN
ock<='0';
IT FOR  50 ns
ock <= not clock;
D PROCESS Generator
```

роцесс, содержащий оператор WAIT, не может иметь списка инициализатов. Это связано с тем, что трудно описать систему, в которой может пройти повторная инициализация действий, в то время как реакция на предущее событие еще не реализована, например, произошло изменение одго из инициирующих сигналов, когда время ожидания еще не вышло. ще раз напомним, что оператор WAIT, как и другие последовательные ераторы, может размещаться только в теле процесса или теле подпроаммы.

ераторы повторения

ераторы повторения LOOP позволяют сокращенно записывать совокупноти однотипных действий.

```
ператор повторения> ::=
   [ <метка оператора повторения> : ] [ <итерационная схема> ]  LOOP
     <оператор> « <оператор> »
```

```
  END LOOP [ <метка оператора повторения> ];
<итерационная схема> ::=
  WHILE  <условие> | FOR <имя переменной>  IN  <диапазон>
```

Последовательность операторов (здесь могут быть только последовательные операторы), заключенная между словами LOOP и END LOOP, называется *телом оператора повторения* или *телом цикла*. Операторы в теле цикла выполняются друг другом в порядке записи, причем такое выполнение повторяется многократно. Число повторений определяется итерационной схемой.

Оператор повторения, не содержащий явного объявления итерационной схемы, предполагает бесконечное повторение последовательностей вложенных в него операторов. Такая модель, в целом, соответствует поведению реальных дискретных устройств, повторяющих некоторую последовательность действий вплоть до отключения питания. В то же время эта конструкция имеет логический смысл, только если тело цикла содержит оператор ожидания WAIT или оператор выхода из цикла EXIT. В противном случае бесконечное безусловное повторение блокировало бы исполнение любых других операторов и процессов. Листинг 3.12 представляет описание двух процессов, каждый из которых содержит бесконечный цикл. Процесс clock_generator описывает устройство, непрерывно генерирующее последовательность прямоугольных импульсов. Процесс execute после начальной установки ждет положительный фронт импульса CLK, и в ответ на это событие выдает на порт очередное значение.

Листинг 3.12

```
ENTITY sequence_of_numbers IS
GENERIC (max_value: integer :=63);
PORT ( out_data: out integer range 0 to max_value;
 Clk : inout bit);
END sequence_of_numbers;
ARHITECTURE behavior OF sequence_of_numbers IS
BEGIN
clock_generator: PROCESS
BEGIN clk<='0';
 LOOP WAIT FOR 50 ns;
 clk<= not clk;
 END LOOP;
END PROCESS clock_generator;

execute : PROCESS
VARIABLE INT_STATE: INTEGER RANGE 0 TO max_value:=0;
BEGIN
 Out_data<=0;
```



```
OOP
AIT UNTIL (clk='1' and clk'event)
F (int_state=max value) then int_state:=0;
 ELSE int_state:=int_state+1;
ND IF;
out_data<= int_state;
ND LOOP;
D PROCESS execute;
D behavior;
```

ератор с ключевым словом WHILE обязательно содержит в теле цикла опеоры, изменяющие описанное в итерационной схеме условие. Операторы кла повторяются, пока при вычислении условия не получается значения LSE. Условие проверяется каждый раз перед исполнением тела цикла. Наимер, оператор

```
LE param>=0 LOOP
 Param :=;  -- некоторое выражение, изменяющее param
D LOOP;
```

полняется, пока переменная param не получит отрицательного значения, ичем если param было отрицательно перед исполнением оператора, то теисполняться вообще не будет.

ератор повторения с ключевым словом FOR повторяется для всех значей переменной из заданного итерационной схемой диапазона.

метим, что присвоенные в теле цикла значения *переменных* могут быть ходными данными для очередного цикла. Если же в цикле выполнено *своение значения сигналу*, то в следующих операторах тела и очередных торениях того же цикла *используются старые значения*, если только тело кла не содержит операторов ожидания.

качестве примера рассмотрим программу в листинге 3.13, представляюю описание синхронной линии задержки на время восьми тактов синонизирующего импульса. Физически это может быть восьмиразрядный гистр сдвига. При каждом переходе сигнала clk в единичное состояние поминающие элементы shift с первого по шестой переходят в состояние, ответствующее состоянию предыдущего элемента перед изменением clk, shift(0) принимает значение входного сигнала. Таким образом, shift(6) держан относительно din на время, соответствующее семи периодам сигала. Сигнал dout задержан еще на один период тактирующего сигнала. нтересно отметить, что если бы вектор shift был определен как переменая, то в силу того, что присвоения переменной непосредственно учитываются в последующих циклах, все биты вектора shift принимали бы значение din после каждого нарастающего фронта сигнала clk.

Листинг 3.13

```
ENTITY delayer IS
port (clk, din:in std_logic;
      dout:out std_logic);
END delayer;
ARCHITECTURE test OF delayer IS
   SIGNAL shift: std_logic_vector (6 downto 0);
BEGIN
PROCESS(clk)
 Variable i: integer range  7 downto 0;
 BEGIN
  IF clk='1' THEN
              shift(0)<=din;
              FOR i IN 1 TO 6 LOOP
                   shift(i)<= shift(i-1);
               END LOOP;
              dout<=shift(6);
  END IF;
 END PROCESS;
END test;
```

Если при моделировании и, в частности, при описании тестовых воздействий смысл операторов повторения практически не отличается от смысла подобных конструкций в традиционных языках программирования, то при интерпретации в аппаратуре имеются существенные отличия. Предусматривается не просто последовательное во времени повторение набора преобразований, а реализация набора устройств, выполняющих однотипные действия, причем эти устройства работают параллельно.

Число устройств определяется итерационной схемой. Для оператора с ключевым словом FOR — это просто число значений переменной в объявленном диапазоне. Для варианта с ключевым словом WHILE условие не может быть связано с сигнальными данными, способными изменяться в реальном устройстве. Например, в материалах фирмы Altera определено, что "операторы повторения должны иметь логически постоянные границы". В противном случае не ясно, сколько повторяющихся блоков в устройстве реально потребуется.

Кроме раздела "итерационная схема" порядок реализации повторений может задаваться дополнительными операторами: оператором перехода к следующему циклу NEXT и оператором выхода из цикла EXIT.

Оператор NEXT блокирует исполнение всех последующих операторов в текущем цикле и обеспечивает автоматический переход к следующей итерации.

Оператор записывается следующим образом:

```
<оператор перехода к следующему циклу> ::=
  [ <метка>: ] NEXT [ <метка оператора повторения> ] [ WHEN <условие>]
```

Фактически, конструкция

```
M1: LOOP      S1
    NEXT WHEN B;
                S2
    END LOOP;
```

де S_1 и S_2 — последовательности операторов, а B — логическое выражение, квивалентна

```
1: LOOP      S1
    IF not  B THEN S2
   END LOOP;
```

ператор EXIT прекращает исполнение не только текущего цикла, но всех ледующих циклов, заданных итерационной схемой исполняемого опера-ра. BNF-форма оператора EXIT имеет вид:

```
ператор прекращения цикла> ::=
 [ <метка>: ] EXIT [ <метка оператора повторения> ] [ WHEN <условие>]
```

апример, оператор вида

```
OP           S1 -- оператор S1 меняет значения аргументов выражения B
EXIT WHEN    B
             S2
D LOOP;
```

вивалентен

```
 TRUE;
ILE not B LOOP
        S1
IF not B then  S2
D LOOP;
```

обязательная метка в операторах NEXT и EXIT используется при записи ложенных циклов. Такая метка указывает, что прерывается не только дан-ый цикл, но и все иерархически предшествующие ему циклы, вплоть до икла, оператор которого помечен этой меткой. Рассмотрим в качестве имера совокупность вложенных операторов повторения, представленных листинге 3.14. Здесь цикл, помеченный m1, содержит две последовательно-ти операторов S_{11} и S_{12}, между которыми включен оператор повторения, омеченный m2. И далее для последующих уровней вложения, mi — это мет-

ка уровня, а S_{i1} и S_{i2} — последовательности операторов, вложенных в оператор повторения *i*-го уровня. Если при вычислении условия B получено значение TRUE, прерывается исполнение циклов предыдущих уровней вложения вплоть до m2. То есть следующей будет выполняться совокупность операторов S_{i2}. Если B = FALSE, то циклы будут завершаться "обычным образом" при возникновении условий, заданных итерационными схемами.

Листинг 3.14

```
m1: LOOP S11
    m2: LOOP S21
      m3: LOOP S31
        .....
            mk: LOOP Sk1
                NEXT m2 WHEN B;
            Sk2
             END LOOP mk;
         .....
            S32
        END LOOP m3;
     S22
        END LOOP m2;
   S12
END LOOP m1;
```

Оператор проверки

Оператор проверки ASSERT относится к категории конструкций, не подлежащих реализации в аппаратуре. Оператор служит для выявления специфических ситуаций, которые могут возникать в процессе компиляции и моделирования (т. е. программной интерпретации описания проекта), и выдачи в этих ситуациях сообщения разработчику. Синтаксис оператора проверки определен следующим образом:

```
<оператор проверки> ::=
    ASSERT <булевское выражение> [ REPORT <строка сообщения> ]
   [ SEVERETY <уровень важности>];
```

При выполнении этого оператора в процессе моделирования проверяется условие, и если получено значение TRUE, выполняется переход к следующему оператору программы. В противном случае на терминал выводится строка сообщения. Если опция "REPORT <строка сообщения>" отсутствует, выдается стандартное сообщение "Assertion violation" (нарушение условий проверки). После этого поведение моделировщика определяется значением

уровня важности. Уровень важности — это выражение (обычно константа) типа SEVERITY_LEVEL. Напомним, что данные этого типа могут принимать четыре значения, причем значения NOTE и WARNING имеют чисто информационный характер, и в этом случае моделирование после выдачи сообщения продолжается, а значения ERROR и FAILURE используются для обнаружения полностью некорректных ситуаций, требующих прекращения моделирования. По умолчанию (т. е. при отсутствии в тексте указания важности) подразумевается уровень ERROR.

Например, если требуется при моделировании просто выводить на монитор информацию о том, что сигнал control_level изменился, и не прерывать моделирование, то можно записать в программе

```
ASSERT NOT control_level'event
  REPORT "Изменение сигнала control_level"
SEVERITY NOTE;
```

Следующий пример иллюстрирует введение сигнализации о возникновении при моделировании недопустимой ситуации — одновременной подачи на входы R и S триггера сигналов логической единицы:

```
ASSERT R='1' NAND S='1';
```

Отсутствие в этой записи ключевых слов REPORT и SEVERITY означает, что используются значения по умолчанию. То есть, если при моделировании возникнет ситуация, когда сигналы R и S одновременно установлены в единицу, на терминал выводится строка "Assertion violation", и моделирование прекращается.

3.2.7. Параллельные операторы

Параллельные операторы это такие, каждый из которых выполняется при любом изменении сигналов, используемых в качестве его исходных данных. Результаты исполнения оператора доступны для других параллельных операторов не ранее, чем будут выполнены все операторы, инициализированные общим событием (а может быть и позже, если присутствуют выражения задержки). В языке VHDL к классу параллельных операторов относятся:

```
<параллельный оператор>::=
<оператор процесса>
    | <оператор параллельного присваивания>
    | <параллельный вызов процедуры>
    | <параллельный оператор проверки>
    | <оператор блока>
    | <оператор вхождения компонента>
    | <оператор генерации>
```

Оператор процесса уже рассматривался в предыдущих разделах. Здесь важно отметить, что этот оператор определен именно как составной оператор параллельного типа. Под составным оператором понимается оператор, имеющий тело, которое содержит несколько вложенных операторов. Оператор процесса начинает исполняться при изменении сигналов, входящих в список инициализаторов (при отсутствии такого списка — безусловно после выполнения всех вложенных операторов), а результаты его исполнения доступны другим параллельным операторам только после исполнения всех операторов, инициируемых теми же событиями, в том числе процессов.

Параллельное присваивание определено в трех различных формах:

```
<Параллельное присваивание> ::=
    [ <метка> : ]<безусловное параллельное присваивание>
  |[ <метка>: ]   <условное присваивание>
  |[  <метка> :]  <присваивание по выбору>
```

По синтаксису и правилам исполнения безусловное параллельное присваивание совпадает с последовательным присваиванием сигналу. Варианты различаются по локализации в программе и характеризуются различными условиями исполнения.

Замечание

Допускается введение ключевого слова QUARDED **перед правой частью оператора присваивания сигналу, о чем см. далее в этом разделе.**

Выделим наиболее существенные различия безусловного параллельного присваивания и последовательного присваивания сигналу:

- параллельное присваивание локализуется в общем разделе архитектурного тела, а последовательное — только в теле процесса;
- последовательное присваивание сигналу выполняется после того, как инициировано исполнение процесса и выполнены все предшествующие операторы в теле процесса;
- оператор параллельного присваивания выполняется сразу (с точки зрения модельного времени) после изменения сигналов в правой части этого оператора.

В обоих случаях результаты присвоения сначала фиксируются в драйвере сигнала и передаются сигналу, т. е. могут влиять на другие операторы, только после исполнения всех операторов и процессов, инициированных одним событием, или через интервал модельного времени, заданный опцией AFTER.

Условное присваивание и присваивание по выбору во многом сходны с условным оператором и оператором выбора, соответственно — описанные действия выполняются при определенных условиях. Различие, кроме единых для

всех параллельных и последовательных операторов свойств, состоит в том, что условный оператор и оператор выбора являются составными, т. е. условие может задавать в них исполнение последовательности действий, а в операторах присваивания — только присвоение одного значения.

```
<Условное присваивание>::=
    <приемник> <= [QUARDED] [<модель задержки>]
  «<прогноз поведения> WHEN <условие> ELSE »
   <прогноз поведения>;
```

Пример. Двухвходовый буфер с тремя состояниями на выходе может быть представлен следующим оператором:

```
z_out<= TRANSPORT  x0 AFTER 2 ns WHEN (adr =  '0' and en='1') ELSE
                   x1 AFTER 2 ns WHEN (adr='1' and  en='1') ELSE
                  'Z' AFTER 5ns;
```

Здесь битовый сигнал adr задает номер включаемого канала, а en — разрешение передачи.

BNF-форма присваивания по выбору имеет вид:

```
<присваивание по выбору> ::=
 WITH <ключевое выражение> SELECT
      <приемник> <= [ QUARDED ] [<модель задержки>]
      «<прогноз поведения> WHEN <вариант>,»
       <прогноз поведения> WHEN <вариант>;
```

Смысл и синтаксис конструкции "вариант" точно совпадает с соответствующим элементом оператора выбора.

Следующий оператор эквивалентен описанию буфера с тремя состояниями, представленному в предыдущем примере:

```
WITH  a & b SELECT
z_out<= TRANSPORT x0 AFTER 2 ns WHEN "01",
                  x1 AFTER 2 ns WHEN "11",
                 'Z' AFTER 5 ns WHEN OTHERS;
```

Важно отметить, что если условный оператор IF и оператор выбора CASE не могут выполняться над данными, вырабатываемыми модулями, представленными различными операторами процесса, то условное присваивание и присваивание по выбору позволяют описывать такие ситуации.

Пример. Пусть три блока (процесса) работают параллельно с выходом на общую шину через буфер с тремя состояниями, причем выбор подключаемого модуля задается сигналом, подаваемым на порт Channel_select. Программный модуль, описывающий такой буфер, представлен в листинге 3.15.

Листинг 3.15

```
ENTITY three_channel IS
PORT ( data_in: IN integer;  -- условные входные данные
           Channel_select: IN integer range 1 TO 3;
           Z: OUT integer);
END three_channel;

ARCHITECTURE skeleton  OF three_channel IS
 SIGNAL data1,data2, data3 : integer;
BEGIN
WITH Channel_select SELECT
    z<=  data1 WHEN 1;
         data2 WHEN 2;
         data3 WHEN 3;
        'Z'    WHEN  others;
  PROCESS.....
       BEGIN   data1<=  ..... END PROCESS;
  PROCESS.....
       BEGIN   data2<= ..... END PROCESS;
  PROCESS.....
       BEGIN   data3<=  ..... END PROCESS;
END skeleton;
```

Параллельные операторы проверки и вызова подпрограмм соотносятся с соответствующими последовательными операторами проверки и вызова подобно соотношению параллельного и последовательного присваивания, а именно: они имеют одинаковый синтаксис и правила выполнения, но различаются локализацией и условиями запуска к исполнению.

Оператор блока

Оператор блока BLOCK, подобно оператору PROCESS, является составным оператором, тело которого включает несколько операторов, но, в данном случае, *параллельных*. Операторы тела блока, как и другие параллельные операторы, обеспечивают возможность представления параллелизма в моделируемой системе. Эти операторы инициируются не по последовательному, а по событийному принципу, а результаты их исполнения становятся доступны другим операторам как включенным в блок, так и размещенным в других блоках или "индивидуально", только после исполнения всех операторов, инициированных одним событием.

В этом смысле операторы, включенные в блок, не отличаются от "индивидуальных" параллельных операторов.

Объединение операторов в блоки обеспечивает следующие возможности:

- структуризация текста описания, т. е. возможность явного и наглядного выделения совокупности операторов, описывающих законченный функциональный узел;
- возможность объявления в блоке локальных типов, сигналов, подпрограмм и некоторых других локальных понятий;
- возможность приписывания всем или некоторым операторам блока общих условий инициализации.

Упрощенные правила записи оператора блока определены таким образом:

```
<оператор блока> ::=
    <метка блока>: BLOCK [ ( охранное выражение) ] [ IS ]
    [ <раздел деклараций блока> ]
 BEGIN
    <раздел операторов блока>
 END BLOCK [ <метка блока>];
```

Наиболее специфическими аспектами блочной организации являются понятия *охранного выражения* и *охраняемого оператора присваивания*.

Охранное выражение — это любое выражение логического типа, аргументами которого являются сигналы. Любое изменение сигналов, входящих в охранное выражение, вызывает вычисление значения этого выражения и присвоение полученного значения предопределенной переменной QUARD. Область действия переменной QUARD — все тело блока, и она может использоваться как обычная логическая переменная во вложенных операторах блока. Например, узел выборки данных из тридцатидвухразрядного регистра на восьмиразрядную линию, в котором транслируется байт, указанный двухразрядным кодом номера byte_sel, может быть представлен таким блоком:

```
select_byte: BLOCK (select='1' and read='1') IS
   BEGIN dbus<=    reg(7 downto 0) WHEN QUARD and byte_sel="00" else
         reg(15 downto 0)  WHEN QUARD and byte_sel="01"  else
         reg(23 downto 16) WHEN QUARD and byte_sel ="10" else
         reg(31 downto 24) WHEN QUARD and byte_sel ="11" else
         "ZZZZZZZZ";
   END BLOCK select_byte;
```

Охраняемый оператор присваивания использует значение переменной QUARD без явного указания условия в программе. Если QUARD ='0', то исполнение операторов присваивания, содержащих ключевое слово QUARDED, в таком блоке запрещено.

Например, два модуля, подключенные к общей шине, могут быть представлены в одном архитектурном теле таким образом, как в листинге 3.16.

Листинг 3.16

```
ARCHITECTURE  quard_example OF two_block IS

SIGNAL   data_bus: Std_logic_vector (N_-1 downto 0);
 -- n  определяется в разделе GENERIC  проекта two_block;

BEGIN
<описание других блоков системы>
 unit1: BLOCK ( adr='0' and read_data='1')
               SIGNAL data0 : Std_logic_vector (N_-1 downto 0);
               BEGIN
                Data_bus<=QUARDED  data0  AFTER  <выражение задержки>
                PROCESS  BEGIN  <Вычисление data0>
                           END PROCESS;
              END BLOCK unit1;
Unit1: BLOCK ( adr='1' and read_data='1')
               SIGNAL data1 : Std_logic_vector (N_-1 downto 0);
               BEGIN
                 Data_bus<=QUARDED  data1  AFTER  <выражение задержки>
                PROCESS  BEGIN  <Вычисление data1>
                           END PROCESS;
              END BLOCK unit1;
```

Более подробно проблемы описания соединений блоков с использованием шин передачи данных рассмотрены далее в *разд. 3.2.10.*

3.2.8. Описание в VHDL типовых дискретных устройств

Комбинационные логические схемы

Математической моделью комбинационной логической схемы является логическая функция. Известно много способов задания логической функции, из которых наибольшее распространение получили:

- алгебраическое представление;
- табличное представление;
- представление через бинарную декомпозицию, в пределе в виде двоичного дерева решений;
- декомпозиция в априорно заданном базисе функций меньшего числа аргументов.

В принципе, любую логическую функцию можно записать в любой из перечисленных форм, а также в их комбинации. Логическая форма полезна при

сравнительно небольшом числе аргументов (не более пяти-шести). Бинарная декомпозиция позволяет осуществлять разложение логической функции на совокупность функций меньшего числа переменных. Табличное представление является более универсальным, но достаточно трудоемким и трудно контролируемым при большом числе аргументов. В подобных ситуациях применимы смешанные подходы. Декомпозиция в заданном базисе, на наш взгляд, мало продуктивна для проектирования схем на ПЛИС, но, в принципе, может применяться при переводе ранее разработанных устройств на новую элементную базу. Выбор того или иного способа представления может определяться самой процедурой проектирования, точнее, способом создания алгоритма функционирования устройства. Часто сам алгоритм создается по логике "если определенный аргумент (вход) установлен в логическую единицу, то реализуется одна совокупность действий, иначе другая". Выделенные совокупности, в свою очередь, могут быть далее разложены на основе анализа следующих аргументов. Такие рассуждения автоматически приводят к описанию логической схемы в форме двоичного дерева решений.

Часто способ записи логических функций связан просто с опытом и личными предпочтениями разработчика. Язык VHDL представляет возможности использования любой исходной формы задания без необходимости ручного перевода из одной формы в другую.

Реализация комбинационной логической схемы на основе *алгебраической формы записи логической функции* интерпретируется оператором присваивания, в правой части которого записывается эквивалентное логическое выражение. Однако следует обратить внимание на ряд особенностей интерпретации, связанных с взаимной синхронизацией операторов. Один из простейших способов — использование параллельных операторов присваивания. Программа (листинг 3.17) представляет описание комбинационной логической схемы с двумя выходами, приведенной на рис. 3.11, в нормальной форме И-ИЛИ с использованием параллельных присваиваний.

Замечание

Реальная интерпретация программы в аппаратной среде может существенно отличаться от схемного варианта, приведенного на рисунке. Компиляторы САПР проводят автоматическую оптимизацию схемы независимо от формы ее задания. В данном случае при реализациях в ПЛИС, построенных на основе четырехвходовых LUT, функции `F1` и `F2` будут целиком реализованы на одной макроячейке каждая, а сигнал `a_and_b` как таковой не воспроизводится, или, как говорят, поглощается.

Листинг 3.17

```
ENTITY simple_logic IS
PORT (a,b,c,d: in std_logic;
              out1, out2: out std_logic);
END simple_logic;
```

```
ARCHITECTURE concurrent OF  simple_logic IS
SIGNAL a_and_b: std_logic;
BEGIN
  out1<=a_and_b or ( c and d and not b) or ( not a and not b and d);
  out2<=a_and_b or (not a and c and d) or (a and not b and not d)
        or (b and not c and d);
  a_and_b <= a and b;
END concurrent;
```

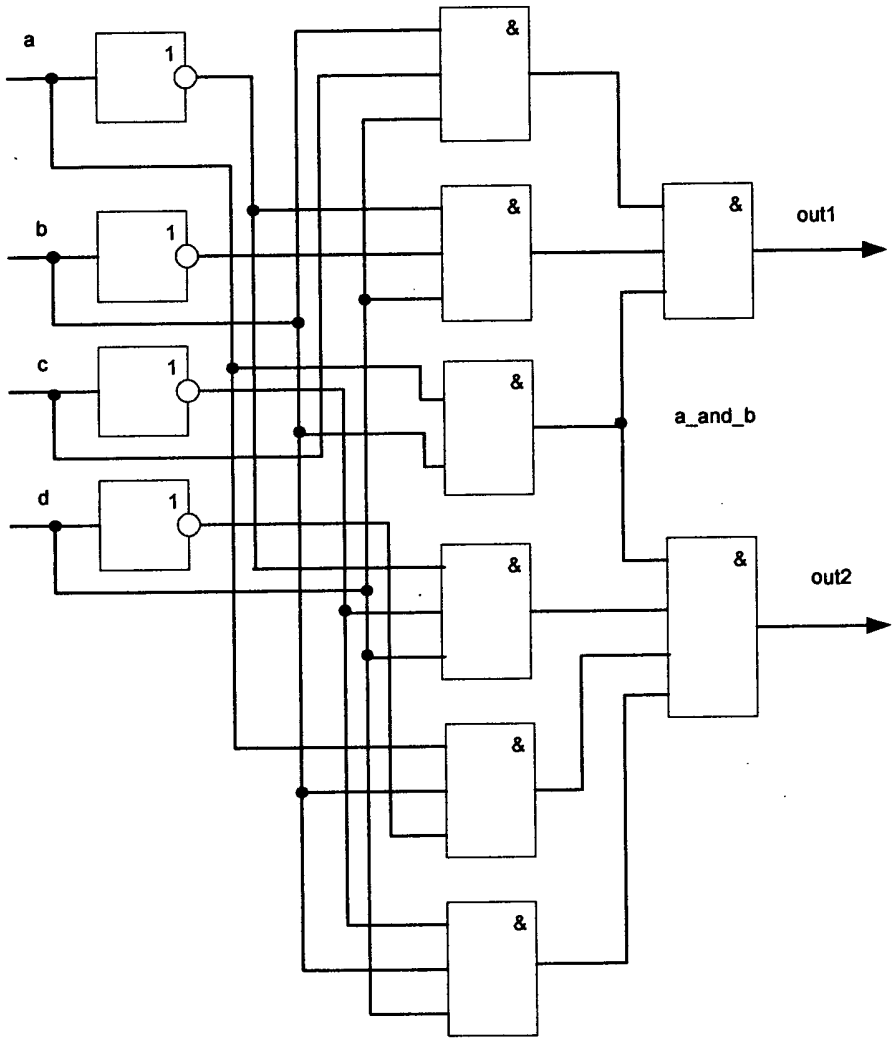

Рис. 3.11. Пример комбинационной логической схемы

Можно применять также и последовательную форму записи правила функционирования с использованием оператора процесса. Архитектурное тело описанного в этой форме устройства, изображенного на рис. 3.11, представлено в листинге 3.18. Здесь важно отметить, что все входные сигналы комбинационной схемы должны быть включены в список инициализаторов процесса с тем, чтобы любое их изменение вызывало исполнение оператора присваивания. Кроме того, в данном случае недопустимо a_and_b декларировать как сигнал. *Это обязательно переменная, причем ее вычисление задается оператором, предшествующим операторам вычисления результирующих сигналов.* В противном случае наблюдается некорректное представление поведения, заключающееся в том, что используются значения не непосредственно полученные в процессе текущего исполнения оператора PROCESS, а значения, вычисленные ранее после предыдущего изменения одного из входных сигналов.

Листинг 3.18

```
ARCHITECTURE sequential OF  simple_logic IS
--SIGNAL a_and_b: std_logic; -- недопустимо в данном контексте
BEGIN
PROCESS (a,b)
 VARIABLE a_and_b: std_logic;
 BEGIN
 a_and_b := a and b;
 out1<=a_and_b or ( c and d and not b) or ( not a and not b and d);
 out2<=a_and_b or (not a and c and d) or (a and not b and not d)
       or (b and not c and d);
END sequential;
```

Если a_and_b это все-таки сигнал (например, необходимый для передачи информации другим операторам программы или на порты), то следует вынести его вычисление в отдельный процесс или параллельный оператор, как показано в листинге 3.19.

Листинг 3.19

```
ARCHITECTURE two_processes OF  simple_logic IS
SIGNAL a_and_b: std_logic;
BEGIN
 PROCESS (a,b)
   BEGIN
     a_and_b <= a and b;
  END PROCESS
 PROCESS (a,b,c,d,a_and_b)
```

```
  BEGIN
    out1<=a_and_b or (c and d and not b) or (not a and not b and d);
    out2<=a_and_b or (not a and c and d) or (a and not b and not d)
         or (b and not c and d);
  END PROCESS;
END two_processes;
```

Табличное представление логической функции можно отобразить в VHDL-программе несколькими способами.

Первый способ состоит в представлении таблицы истинности логической функции в виде константного битового вектора, каждый *i*-й компонент которого представляет значения логической функции на наборе, численный эквивалент которого равен значению *i*. Очевидно, что "вычисление" функции сведется к выборке элемента массива, индекс которого равен численному эквиваленту входной кодовой комбинации.

Формирование такого численного эквивалента выполняется с использованием функций преобразования типов сигналов, которые представлены во всех системах проектирования, способных интерпретировать описания проектов на VHDL. Эти функции могут иметь разные названия и даже синтаксис, что не меняет сути. Мы в дальнейшем будем записывать такие преобразования, используя функции преобразования типов, определенные в пакете `std_logic_util` системы интерпретации VHDL-программ фирмы Model Technology.

В функции `conv_integer (vect)` аргумент относится к типу `bit_vector` или `Std_logic_vector` произвольной длины, а возвращаемое значение — численный эквивалент двоичного кода аргумента (беззнаковое целое в диапазоне от 0 до 2^n-1, где n — разрядность аргумента).

В функции `conv_std_logic_vector (arg1, n)` оба аргумента целые беззнаковые, а результат — n-разрядный двоичный код (`Std_logic_vector`), являющийся двоичным эквивалентом `arg1`.

Листинг 3.20 содержит архитектурное тело, функционально соответствующее программе, представленной в листинге 3.17. В этом варианте определение результата выполняется путем прямой выборки значения из таблицы по индексу, формируемому из входных сигналов, с использованием функции `conv_integer`.

Листинг 3.20

```
ARCHITECTURE table_ presentation OF simple_logic IS
TYPE truth_table_4x1 IS ARRAY (0 to 15) OF std_logic;
CONSTANT function1: truth_table_4x1 :=
  ('0','0','0','1','0','0','0','1','1','0','0','1','1','1','0','1');
```



```
ONSTANT function2: truth_table_4x1:=
 ('0','1','0','1','0','0','0','1','1','0','0','1','1','1','0','1');
IGNAL digital_equivalent: integer range 0 to 15;
EGIN
 Digital_equivalent <= conv_integer (d & c & b & a);
 Out1<= function1(digital_equivalent);
 Out2<= function2(digital_equivalent);
ND table_presentation;
```

озможно использование последовательных операторов присваивания с учеом тех же особенностей, которые были отмечены при изложении интерретации функций с использованием логических выражений. Существенным недостатком присвоения путем выборки из таблицы являются трудноти воспроизведения при временном моделировании поведения устройств, которых переходы 0—1 и 1—0 различны по времени, а также анализ сбойых ситуаций (гонки, риски). Если это в проекте представляется важным, учше на основании таблицы истинности построить оператор присваивания о выбору или использовать последовательный оператор выбора. Для выхода ut1 устройства (см. рис. 3.11) оператор присваивания по выбору, описывающий поведение сигнала во времени, может выглядеть следующим образом:

```
<=d & c & b & a;
TH v SELECT
ut1<='X', '0' AFTER 1 ns WHEN "0000",
     'X', '0' AFTER 1 ns WHEN "0001",
     'X', '0' AFTER 1 ns WHEN "0010",
     'X', '1' AFTER 2 ns WHEN "0011",
     'X', '0' AFTER 1 ns WHEN "0100",
     'X', '0' AFTER 1 ns WHEN "0101",
     'X', '0' AFTER 1 ns WHEN "0110",
     'X', '1' AFTER 2 ns WHEN "0111",
     'X', '1' AFTER 2 ns WHEN "1000",
     'X', '0' AFTER 1 ns WHEN "1001",
     'X', '0' AFTER 1 ns WHEN "1010",
     'X', '1' AFTER 2 ns WHEN "1011",
     'X', '1' AFTER 2 ns WHEN "1100",
     'X', '1' AFTER 2 ns WHEN "1101",
     'X', '0' AFTER 1 ns WHEN "1110",
     'X', '1' AFTER 2 ns WHEN "1111",
     'X' WHEN OTHERS;
```

Использование оператора выбора может обеспечить более компактную запись:

```
:=d & c & b & a;
ASE v  IS
```

```
  WHEN "0000" |"0001"|0010"|"0100"|"0101"|
              "0110"|"1001"|"1010"|"1100"|"1110"
              =>Out1<= 'X', '0' AFTER 1 ns;
  WHEN "0011" |"0111"|"1000"|"1011"|"1100"|"1101"|"1111"
              =>  Out1<= 'X', '1' AFTER 2 ns;
  WHEN OTHERS => Out1<= 'X';
END CASE;
```

При этом еще раз подчеркнем, что такой оператор должен находиться в теле процесса, причем переменные d, c, b и a должны входить в список инициализаторов этого процесса.

Представление функции в форме *бинарного дерева решений* основано на последовательном переходе от представления функции большого числа переменных через суперпозицию функций меньшего числа переменных. Теоретической основой метода является разложение Шеннона:

$$f(x_1, x_2, ..., x_N) = \hat{}x_1 f(0,x_2, ..., x_N) \vee x_1 f(1,x_2, ..., x_N).$$

Тогда вычисление первого терма разложения записывается как одна возможная альтернатива условного оператора или условного присваивания, а вычисление другого — как противоположная:

```
IF x1='0' THEN z<= <подформула, полученная из f заменой x1 на нуль>;
  ELSE z<= <подформула, полученная из f заменой x1 на единицу>;
END IF;
```

Подформулы разложения могут быть, в свою очередь, далее разложены:

$$f(x_1, x_2, ..., x_N) = \hat{}x_1 f(0,x_2, ..., x_N) \vee x_1 f(1,x_2, ..., x_N) = \hat{}x_1 [\hat{}x_2 f(0,0, ...x_N) \vee x_2 f(0,1, ..., x_N)] \vee x_1 [\hat{}x_2 f(1,0, ..., x_N) \vee x_2 f(1,1, ..., x_N)],$$

и т. д. Однако хочется еще раз отметить, что описание логики в виде бинарного дерева или поддерева часто вытекает из самой логики создания проекта.

Оператор процесса, содержащий описание комбинационной схемы, представленной на рис. 3.11 с использованием условного оператора, приведен в листинге 3.21. Здесь вспомогательные переменные pr1 и pr2 введены для возможности представления несимметричных переходных процессов. Если это не существенно, то можно ограничиться присвоением значений сигналам out1 и out2 непосредственно в условном операторе.

Листинг 3.21

```
PROCESS (a,b,c,d)
VARIABLE pr1,pr2:std_logic;
BEGIN
 IF b='1' THEN pr1 := a; pr2:=a or (d and not c);
```



```
 ELSIF a='0' THEN pr1:=d; pr2:=not d or c;
 ELSE   pr1:=c and d;  pr2='0';
ND IF;
ut1<= 'X','1'AFTER 2 ns WHEN pr1='1' ELSE
      'X','0'AFTER 1 ns WHEN pr1='0' ELSE
      'X';
ut2<= 'X','1'AFTER 2 ns WHEN pr2='1' ELSE
      'X','0'AFTER 1 ns WHEN pr2='0' ELSE
      'X';
ND PROCESS;
```

ераторы условного присваивания записываются индивидуально для каждой функции, но структурно подобны представленному условному оператору.

заключение рассмотрения схем комбинационной логики в листинге 3.22 иведены программные модули, описывающие некоторые базовые узлы ифровых устройств: дешифратор (ENTITY decode), мультиплексор (ENTITY ux), схема выделения признака четности (ENTITY parity). Декларация библиотек и пакетов здесь опущена. Значения параметров настройки, определенных в декларации GENERIC (*n* — разрядность входного или выходного ода и delay — время задержки), могут задаваться в модулях высшего уровня иерархии, в которые эти узлы включаются. При автономном использовании проектных модулей decode, mux и parity принимается n = 16 и lay = 2 ns.

Листинг 3.22

```
NTITY decode IS
ENERIC (n: integer:=16);  -- число выходов
ORT (input: IN integer RANGE 0 TO n-1;
     EN: IN std_logic;  -- разрешение
     output:OUT std_logic_vector (n-1 DOWNTO 0));
ND decod;
RCHITECTURE behave OF decod IS
EGIN
ROCESS(en,input)
ARIABLE i: integer RANGE 0 TO n-1;
BEGIN
 FOR i IN 0 TO 15 LOOP
   IF en='1' and i=input THEN
            output(i)<='1' AFTER 2 ns;
     ELSE   output(i)<='0' AFTER 1 ns;
   END IF;
ND LOOP;
ND PROCESS;
ND behave;
```

```
ENTITY mux IS
GENERIC (n: integer:=16;  -- число входов
         delay:time:=2 ns); -- задержка распространения
PORT (adress: IN integer RANGE 0 TO n-1;
      data: IN std_logic_vector ( n-1 DOWNTO 0);
      output:OUT std_logic);
END mux;
ARCHITECTURE behave OF mux IS
 BEGIN
   Output<= data(adress) AFTER delay;
 END behave;

ENTITY  parity IS
GENERIC (n: integer:=16;  -- число входов
         delay:time:=2 ns); -- задержка распространения
PORT (data: IN std_logic_vector ( n-1 DOWNTO 0);
      parit_check: OUT std_logic);
END  parity;
ARCHITECTURE parit_behave OF parity IS
BEGIN
PROCESS (data)
 VARIABLE i:integer RANGE 0 to n-1;
 VARIABLE temp:std_logic;
   BEGIN
     temp:='0';
    FOR i IN 0 TO n-1 LOOP
     temp:= temp xor data(i);
   END LOOP;
parit_check<=temp AFTER delay;
END PROCESS;
END parit_behave;
```

Описание триггеров и регистровых схем

Триггер по определению есть устройство с двумя устойчивыми состояниями (не считая переходных состояний). В триггерных устройствах входные сигналы можно разделить на две категории — управляющие и информационные. Главное свойство триггеров — способность сохранять свое состояние при пассивном состоянии управляющих входов независимо от изменений сигналов на информационных входах. Это свойство удобно описывать в VHDL с помощью оператора PROCESS, причем управляющие сигналы включаются в список инициализаторов или в выражения условий вложенного оператора ожидания WAIT. Состояние триггера после события на управляющем входе определяется его типом (функцией переходов и способом управления), предыдущим состоянием и сигналами на информационных входах.

В практике наиболее часто используются следующие способы управления триггерами:

- асинхронное управление;
- статическое управление, или управление уровнем;
- динамическое управление, или управление фронтом;
- смешанные варианты.

Асинхронное управление характеризуется тем, что события на информационных входах непосредственно вызывают изменение состояния триггера. Единственный нетривиальный триггер с асинхронным управлением это RS-триггер, функциональная модель которого может быть представлена следующим образом:

```
PROCESS (R,S)
 BEGIN   IF R='1' and S='1' THEN
                  ASSERT FALSE REPORT " Illegal input combination"
                                SEVERITY note;
          ELSIF  S='1' THEN Q<='1';
          ELSE  Q<='0';
         END IF;
 END PROCESS;
```

Здесь R, S и Q определены как сигналы или порты типа BIT.

Если требуется выполнить моделирование с учетом возможных переходных состояний входов и выходов, то следует ввести ряд уточнений, например таких, как в программе (листинг 3.23).

Листинг 3.23

```
ENTITY R_S_flip_flop IS
  GENERIC (delay: time:=2 ns;);
 PORT (R,S:IN std_logic;
          Q,NQ: OUT std_logic)
END R_S_flip_flop;
ARCHITECTURE R_S_timing OF R_S_flip_flop IS
BEGIN
 PROCESS  (R,S)
  BEGIN   IF R='1' and  S='0' THEN Q<= 'X','1' AFTER 2*delay;
                                .     NQ<='X','0' AFTER delay;
          ELSIF R='0' and  S='1' THEN Q<='X','0' AFTER delay;
                                      NQ<='X','1' AFTER 2*delay;
          ELSIF R='1' and  S='1' THEN Q<='X','0' AFTER delay;
                                       NQ<='X','0' AFTER delay;
```

```
        ELSIF R='0' and  S='0'
             and Q'DELAYED(delay)='0' and NQ Q'DELAYED(delay)='0'
                    then       Q<='X'; NQ<='X';
        ELSE Q<='X'; NQ<='X';
        END IF;
  END PROCESS;
END R_S_timing;
```

Программа в листинге 3.23 соответствует поведению триггерной ячейки на двух перекрестно соединенных элементах ИЛИ-НЕ. Она учитывает возможности подачи на вход неопределенных сигналов, отражает переход обоих плеч триггерной ячейки в состояние логического нуля при одновременной подаче логической единицы на входы сигналов установки и сброса, а также неопределенность дальнейшего поведения после одновременного перехода этих сигналов из единичного состояния в нулевое. Впрочем, если описывать простой RS-триггер, то легче его показать как соединение пары логических ячеек И-НЕ или ИЛИ-НЕ:

```
Q<=not (NQ or S) after  delay;
NQ<= not (Q or R) after delay;
```

Но описание, подобное приведенному в листинге 3.23, может быть совмещено с объявлением других способов управления при моделировании комбинированных триггерных структур.

В триггерах со статическим управлением при пассивном уровне сигнала на управляющем входе изменение состояния невозможно, а при активном уровне изменения информационных сигналов приводят к изменению состояния. Рассмотрим процесс `d_level`, описывающий D-триггер с потенциальным управлением, иногда называемый *защелкой* (Latch), для которого положительный сигнал является разрешающим.

```
d_level: PROCESS  (clock, data)
           BEGIN   IF clock='1'  THEN  Q<=data AFTER delay;
                    END IF;
           END PROCESS;
```

Тело процесса исполняется как после любого изменения сигнала `clock` (тактирующий сигнал), так и сигнала `data` (информационный сигнал), однако при `clock='0'` изменение состояния `Q` не происходит.

При динамическом управлении изменение состояния триггера происходит только непосредственно после фронта или спада сигнала на тактирующем входе. Новое состояние определяется сигналами на информационных входах в момент изменения управляющего сигнала, а само по себе изменение состояний информационных входов не вызывает изменения состояния триггера.

ператор процесса `d_edge` в листинге 3.24 интерпретирует это правило и личается от оператора `d_level` списком инициализаторов. Указанный роцесс вызывается только при изменении сигнала `clock`, причем изменее состояния может происходить, если произошел переход `clock` в состояе логической единицы. Следующее состояние определяется состоянием нформационного входа. Но если сигнал `data` стабилен перед фронтом `lock` в течение интервала времени, меньшего, чем необходимое для данного иггера время предустановки `delay_d_clk`, следующее состояние не определено.

Листинг 3.24

```
edge:  PROCESS(clock)
BEGIN  IF clock='1'  THEN
       IF data'last_event<delay_d_clk THEN
          Q<='X';
       ELSE Q<=data AFTER delay;
       END IF;
END  PROCESS;
```

ри представлении комбинированных триггерных устройств в число инилизаторов процесса, в котором описывается такое устройство, включают асинхронные входы, и тактирующий сигнал. При составлении программы обходимо учитывать относительные приоритеты управляющих сигналов и совместимость. В качестве примера оператор `PROCESS`, представленный листинге 3.25, содержит упрощенное описание комбинированного JKRS-иггера (переходные состояния в этой модели игнорируются и временные отношения опущены). Входы асинхронной установки и сброса `S` и `R`, как ычно, являются более приоритетными по сравнению с входом синхронии `clock` (в данном случае представлена синхронизация положительным ронтом). Только при нулевых сигналах на входах `R` и `S` выполняется операр выбора `CASE`, интерпретирующий функционирование J-K-триггера.

Листинг 3.25

```
ROCESS (clock,r,s)
  VARIABLE  Qinternal:= BIT;
  VARIABLE variant: bit_vector (2 downto 0);
  BEGIN  variant := j&k;
  IF R='1' and S='1' THEN
           ASSERT FALSE REPORT " Illegal input combination"
                  SEVERITY ERROR;
    ELSIF  S='1' THEN Q<='1';
    ELSIF  R='1' THEN  Q<='0';
```

```
    ELSIF R='0' and S='0' and clock='1' and not clock'stable THEN -- **
        CASE  variant IS
             WHEN "10" => Qinternal:='1';
             WHEN "01"=> Qinternal:='0';
             WHEN "00"=> Qinternal:=Qinternal;
             WHEN "11"=> Qinternal:= not Qinternal;
            END CASE;
     END IF;
  Q<=Qinternal;
END PROCESS;
```

Особо следует обратить внимание на конструкцию `clock='1' and not clock'stable` в условии, помеченном двумя звездочками. Простое условие `clock='1'` оказывается недостаточным для корректного описания поведения. При переходе сигналов `R` или `S` в состояние логического нуля приведенный в листинге процесс будет инициализирован. Если бы отсутствовало дополнительное условие `not clock'stable`, был бы выполнен один из вариантов оператора выбора, даже если в данный момент нет изменений на входе `clock`. Это не соответствует реальному поведению. В целом, во избежание подобных коллизий можно рекомендовать использовать запись вида

```
IF <сигнал синхронизации> =<значение> and
  not <сигнал синхронизации>'STABLE THEN ...
```

всегда, когда описываются устройства с динамически управляемыми триггерами.

Регистр — это набор триггеров, объединенных общими цепями управления. Соответственно регистры в программах удобно представлять процессами, список инициализаторов которых включает управляющие сигналы, а в теле процесса находятся операторы присваивания, определяющие состояние триггеров регистра после изменений управляющих сигналов. Логика анализа условий выполнения операторов в теле этого процесса не отличается от такой же логики для одиночных триггеров.

Состояния триггеров отражаются переменными или сигналами. Можно использовать скалярное представление и индивидуальные имена для всех триггеров регистра, однако чаще используется групповое представление.

Представление в виде битового вектора (типа `bit_vector` или `std_logic_vector`) обеспечивает сочетание простоты описания групповых действий над содержимым регистра с простотой выделения информации о состоянии отдельных разрядов. В качестве примера в листинге 3.26 представлена программа, описывающая преобразователь параллельного кода в последовательный с асинхронной загрузкой кода и синхронным сдвигом.

стинг 3.26

```
BRARY ieee;
E ieee.std_logic_1164.ALL;
TITY Parallel_to_sequential IS
NERIC ( n:integer:=8);
RT (clk,load: IN std_logic;
    din:IN std_logic_vector (n-1 DOWNTO 0);
     dout:OUT std_logic);
D new_del;

CHITECTURE test OF Parallel_to_sequential IS
 SIGNAL reg: std_logic_vector (n-1 DOWNTO 0);
GIN
ut<=reg(n-1);
ROCESS
ARIABLE i: integer n-1 DOWNTO 0;
EGIN
WAIT UNTIL load='1';
reg<=din;   --загрузка кода
j:=0;
WHILE j<n LOOP   --  n сдвигов после загрузки
  WAIT UNTIL clk='1' and not clk'stable;
    FOR i IN n-1 DOWNTO 1 LOOP
        reg (i)<=reg (i-1);
    END LOOP;
  reg(0)<='0';
  j:=j+1;
END LOOP;
ND PROCESS;
D test;
```

ри включении описываемого этой программой устройства (и аналогично, осле начала моделирования) начинается этап ожидания сигнала загрузки oad. Получив `load='1'`, устройство осуществляет `n` сдвигов, каждый из которых выполняется по положительному фронту сигнала `clk`. Каждый сдвиг редставлен в программе как последовательность присвоений состоянию аждого разряда состояния предыдущего в операторе `FOR`. В программе используемы как описания действий с целым вектором при загрузке, так и ействий с отдельными разрядами вектора для описания процедуры сдвига формирования выходного сигнала.

Замечание

Подобное описание соответствует семантике VHDL'87, хотя и не противоречит VHDL'93, где операции сдвига введены как стандартные.

В ряде случаев удобно использовать для представления содержимого регистра, ограниченные целые и перечислимые типы данных. Такое представление может обеспечить большую наглядность и улучшить понимаемость программ. Формально такие типы данных определены как скалярные, а по существу могут в сжатой форме отражать групповой объект — код состояния регистра.

Например, программа (листинг 3.27) представляет описание простого операционного блока, предназначенного для выполнения следующего набора операций с содержимым регистра данных data_register: clear — очистка, nop — сохранение состояния, load — загрузка данных, shift — сдвиг в сторону старших разрядов, inc — инкремент содержимого, dec — декремент. Код операции command для наглядности представлен данными перечислимого типа instruction, а данные — как ограниченные целые. Чтобы соответствующие данные могли быть "понятны" другим проектным модулям, эти типы должны быть объявлены в общедоступном пакете, в данном примере, в пакете OPER_UNIT_DECL. Код операции загружается при единичном сигнале на входе load_command и может сохраняться в регистре instruction_register в течение нескольких следующих операций, инициируемых положительным фронтом сигнала execute.

Данные типа instruction и типа data в аппаратуре представлены многоразрядным кодом. Для доступа до конкретного разряда кодов, отнесенных к такому типу данных, необходимо использовать функции преобразования типов. Функции преобразования "битовый вектор — целое" и обратно определены в ряде стандартных пакетов, например std_logic_util системы ModelSim. Для типов, определяемых пользователем, может потребоваться создание специальных функций преобразования.

Листинг 3.27

```
PACKAGE oper_unit_decl IS
 TYPE instruction IS (clear, nop,load,shift,inc,dec);
 TYPE  data IS RANGE 0 TO 255;
END;

USE work.oper_unit_decl.all;
ENTITY oper_unit IS
PORT ( load_command,execute: IN bit;
       command: IN instruction;
       data_in:IN data;
       data_out:OUT data);
END oper_unit;

ARCHITECTURE test OF oper_unit IS
    SIGNAL data_register: data;
    SIGNAL instruction_register:instruction;
```



```
BEGIN
PROCESS (load_command)
  BEGIN instruction_register<=command;
  END PROCESS;
PROCESS (execute)
  BEGIN  IF execute='1' and not execute'stable THEN
          CASE instruction_register IS
             WHEN load => data_register<=data_in;
             WHEN nop  =>null;
             WHEN clear=> data_register<=0;
             WHEN shift => data_register<=data_register*2;
             WHEN inc   => data_register<=data_register+1;
             WHEN dec   => data_register <=data_register-1;
           END CASE;
          END IF;
  END PROCESS;
END test;
```

На основании вышесказанного можно рекомендовать придерживаться следующих правил.

- Для представления узла без памяти (комбинационной схемы) описание его функционирования удобно разместить в теле оператора PROCESS, в список инициализаторов которого или в условия прекращения ожидания включаются все информационные сигналы.
- Для устройств с памятью логические выражения, определяющие очередное состояние регистровой схемы, следует разместить в теле оператора PROCESS, но в этом случае список инициализаторов включает только управляющие сигналы — тактирующий и, при необходимости, сигналы принудительной установки и сброса.
- В комбинационной схеме недопустимо использовать выходной сигнал в качестве одного из входов этой же схемы. В регистровых схемах это допустимо, однако в таком случае при представлении в программе следует использовать конструкции, которые соответствуют представлению динамически управляемых триггерных устройств.

Простые арифметические узлы

Перечисленные выше правила в полной мере применимы к арифметическим узлам, которые могут, в зависимости от требований проекта, реализоваться и как комбинационные схемы, и как элементы с памятью. Развитием устройств с памятью являются конвейерные реализации, предполагающие наличие нескольких регистров, сохраняющих промежуточные результаты.

Физически входные и выходные данные в узлах арифметического типа это, как правило, многоразрядные коды, которые могут быть представлены в

программе и как двоичные векторы (данные типа bit_vector или std_logic_vector), и как числа. Логика формирования выходной величины арифметического узла может представляться в программе как с помощью логических выражений, так и в арифметической форме. С точки зрения представления поведения варианты эквивалентны, но описание преобразований в арифметической форме интерпретируется в реальном устройстве типовыми узлами библиотеки системы проектирования, а запись в логической форме обеспечивает большую степень вмешательства разработчика в создание технической реализации. Часто форма представления определяется удобствами описания сложных структур, содержащих арифметические узлы.

Описание арифметических преобразований данных, представленных в численной форме, достаточно традиционно — используются арифметические выражения. Относительно операций над кодами отметим, что стандарт языка VHDL не определяет арифметических операций над двоичными векторами. Однако все современные системы интерпретации языка сопровождаются пакетами, содержащими определение функции преобразования "число—битовый вектор" и "битовый вектор—число", а также пакетами, встроенные функции которых позволяют оперировать с битовыми векторами как числами при различных способах кодирования. Один и тот же арифметический символ в применении к преобразованию кодов может интерпретироваться по-разному. Это определяется тем, какой пакет объявлен используемым в программе. Например, в системе ModelSim знак "+" применительно к данным типа std_logic_vector будет интерпретирован различно. Если в программе определено использование пакета std_logic_unsigned, то операция интерпретируется как сложение по правилам беззнаковой двоичной арифметики (старший разряд слева). Если объявлено использование пакета std_logic_signed, то тот же знак задает сложение в дополнительном коде (крайний левый разряд знаковый), а если ни один пакет не указан, знак сложения, примененный к вектору, рассматривается как ошибка. Пакеты по умолчанию скомпилированы в системную библиотеку IEEE.

Листинг 3.28 представляет первичный проектный модуль ENTITY add_subb — сумматор-вычитатель чисел в беззнаковом двоичном представлении, при этом разрядность width является параметром настройки. Этому ENTITY сопоставлено два архитектурных тела. Вариант используемого в конкретном проекте архитектурного тела и разрядность сумматора задаются в модуле высшего уровня иерархии проекта *(см. разд. 3.2.11)*.

В архитектурном теле arithmetic_presentation используется арифметическая форма представления преобразований. Для обеспечения доступа к содержащимся в системной библиотеке функциям, определяющим арифметические преобразования битовых векторов, в программе имеется декларация использования соответствующего пакета (данный пример ориентирован на реализацию в системе отладки ModelSim v.5.5). С целью обеспечения возмож-

ости контроля переполнения использовано "внутреннее" расширение разрядной сетки, т. е. введены вспомогательные переменные a, b и s, причем a и b получаются из входных кодов добавлением нуля со стороны старших разрядов. При сложении и вычитании в расширенном формате наличие переполнения в основных разрядах приведет к появлению единицы в дополнительном разряде результата s, который и используется как сигнал переполнения. Код фактического результата получается переименованием основных разрядов суммы s.

огическая форма представления иллюстрируется архитектурным телом omb_logic. Здесь более явно задана структура проекта — в данном случае редполагается схема с групповым переносом [27]. Сумматор разбивается на руппы по четыре разряда, внутри групп формируются разряды суммы и зрядные переносы по последовательной схеме (переменные carry). Вход ереноса каждой группы (fast_carry) формируется с использованием изстной логики ускоренного переноса на основании входных сигналов предыдущей группы.

стинг 3.28

```
IBRARY IEEE;
E ieee.std_logic_1164.all;
E ieee.std_logic_unsigned."+";
TITY add_subb IS
NERIC (width:INTEGER);
RT ( data_a,data_b: IN std_logic_vector (width-1 DOWNTO 0);-- входы
     direction,                    -- '0'-сложение, '1'-вычитание
     carry_in: IN std_logic;       -- входной перенос
     sum :OUT std_logic_vector(width-1 DOWNTO 0);-- результат
      overflow: OUT std_logic);  -- переполнение
ND add_subb;

RCHITECTURE  arithmetic_presentation of add_subb is
GNAL a,b,s:std_logic_vector( width DOWNTO 0);
GIN
<= '0'& data_a; b<= '0'& data_b;
<=  a + b + carry_in WHEN direction ='0' ELSE
     a - b-carry_in;
um <= s(width-1 DOWNTO 0);
verflow<= s(width);
ND arithmetic_presentation;

RCHITECTURE comb_logic OF add_subb IS
CONSTANT group: INTEGER:= width/4;-- число групп при разбиении по четыре
                                  -- разряда в группе
```

```
BEGIN
PROCESS( data_a,data_b, carry_in)
 TYPE carry_array IS ARRAY (4 DOWNTO 0, group-1 DOWNTO 0) OF std_logic;
 VARIABLE carry: carry_array; -- переносы
 VARIABLE i :INTEGER RANGE 0 TO 3;
 VARIABLE group_number: INTEGER RANGE 0 TO group-1;
 VARIABLE c_b,lb: INTEGER RANGE width-1 DOWNTO 0;
 VARIABLE fast_carry : -- выходы группового переноса
        std_logic_vector (group-1 DOWNTO 0);
 VARIABLE ps,pm,a_carr: std_logic_vector (width-1 DOWNTO 0);
BEGIN
--     --------вспомогательные переменные для вычисления переносов-----
 IF direction='0' THEN a_carr:= data_a;
       else a_carr:= not data_a;
 END IF;
   ps:= a_carr xor data_b; -- Partial Sum — разрядные полусуммы
   pm:= a_carr and data_b; -- Partial multiply — разрядные произведения
--      ----- конец вспомогательных переменных --------------------
    FOR group_number IN 0 TO group-1 LOOP -- цикл по группам разрядов
       lb :=4*group_number;-- номер младшего разряда в группе
       IF group_number=0 THEN carry (group_number,0):=carry_in;
       ELSE carry(group_number,0):= fast_carry (group_number-1);
       END IF;
     FOR I IN 0 TO 3 LOOP  -- анализ разрядов внутри группы
       C_b:= lb+ I; -- абсолютный номер разряда
     sum(c_b) <= data_a(c_b) xor data_b(c_b) xor carry(group_number-,I);
     carry(group_number,i+1) := (a_carr(c_b) and data_b(c_b))
          or ((a_carr(c_b) or data_b(c_b)) and carry(group_number,i));
     END LOOP;
   FAST_CARRY(GROUP_NUMBER):=
                          pm(lb+3)
                       or (pm(lb+2) and ps(lb+3))
                       or (pm(lb+1) and ps(lb+3)and ps(lb+2))
                       or (pm(lb) and ps(lb+3)and ps(lb+2)and ps(lb+1))
or  rry(group_number,0)and ps(lb+3)and ps(lb+2)and ps(lb+1)and ps(lb));
  E  OOP;
 ov  ow<= fast_carry(group-1);
END  ESS;
END c  logic;
```

Программа (листинг 3.28) соответствует комбинационной схеме суммирования. Если же в число входных портов ввести сигнал синхронизации, а в списке инициализаторов процесса записать вместо имен входов данных имя этого сигнала, будет задаваться регистровая схема, которая после небольших модификаций легко преобразуется в сумматор накапливающего типа.

Таким образом процессы, помеченные метками `comb` и `registered` в листинге 3.29, несмотря на подобие выполняемых операций (в обоих процессах выполняются инкремент и сдвиг), представляют совершенно различные алгоритмы функционирования и технические реализации. Процесс `comb` задает комбинационные схемы арифметических преобразований входных кодов, в то время как процесс `registered` определяет счетчик и регистр сдвига. Исходными данными могут быть как целые, так и битовые векторы.

Листинг 3.29

```
..
comb: PROCESS ( data_a, data_b)
  begin
        shift<=  data_a sla 1;
        inc <=   data_b+1;
END PROCESS;
..
registered: PROCESS (clock)
  begin  ...
          If (clock='1') then
              shift1<= shift1 sla 1;
              inc1 <=  inc1+1;
      END IF;
ND PROCESS;
```

Описание конвейерных реализаций рассмотрим на примере программы конвейерного умножителя (листинг 3.30). Реализуется стандартная вычислительная схема для умножения целых чисел, представленных в двоичной форме:

$$product = \sum_{i=0}^{n-1} a \times 2^i \times b(i);$$

где a — множимое, $b(i)$ — i-й разряд множителя, n — разрядность операндов $product$ — результат.

Особенность реализации состоит в том, что исходные аргументы `data_a` и `data_b` продвигаются в каждом такте, инициируемом фронтом тактового сигнала, между ступенями конвейера. В каждой i-й ступени конвейера к накопленной на предыдущей ступени сумме прибавляется произведение множимого, сдвинутого на i разрядов, и i-го бита множителя.

В конвейерной схеме в любой такт времени в каждой i-й ступени конвейера сохраняется и обрабатывается пара сомножителей, поступившая на вход устройства на i тактов ранее текущего такта, и соответствующий этой паре промежуточный результат. Одновременно в конвейере находятся в различных стадиях обработки данные, поступившие на вход в нескольких последо-

вательных тактах. Таким образом достигается повышение производительности устройства по сравнению с комбинационной реализацией. Однако при этом увеличивается время задержки формирования каждого отдельного результата.

Листинг 3.30

```
LIBRARY IEEE;
USE ieee.std_logic_1164.all;
USE work.std_logic_util.all;
ENTITY pipe_mul IS
GENERIC ( width:integer:=8);
PORT ( data_a,data_b: IN integer RANGE 0 TO 2**width-1;
       clock: IN std_logic;
       product :OUT integer RANGE 0 NO 2**(2*width)-1
     );
 END pipe_mul;

ARCHITECTURE behave OF pipe_mul IS
 TYPE shift_array IS ARRAY (width-1 downto 0)
                     OF integer RANGE 0 TO 2*width-1;
 TYPE partial_mult_array IS  ARRAY (width downto 0)
                      OF integer RANGE 0 to 2**(2*width)-1;
 SIGNAL a_shift,b_shift:shift_array;
 SIGNAL P_mult:partial_mult_array;
 BEGIN
PROCESS( clock)
 VARIABLE i:integer RANGE  width-1 DOWNTO 0;
 VARIABLE b,b0: std_logic_vector( width-1 DOWNTO 0);
  BEGIN
IF( clock='1') and not clock'stable THEN
     b0:= to_vector(data_b,width);
     IF ( b0(0) ='1') THEN p_mult(0)<=data_a;
                 else p_mult(0)<=0;
     END IF;
     a_shift(0)<= data_a;
     b_shift(0)<= data_b;
  FOR i IN 1 TO width LOOP
     a_shift(i)<=a_shift(i-1)sra 1;
     b_shift(i)<=b_shift(i-1);
     b:=conv_std_logic_vector(b_shift(i-1),width);
     IF (b(i)='1') THEN
                p_mult(i)<= a_shift(i-1)+ p_mult(i-1);
                ELSE p_mult(i)<=p_mult(i-1);
     END IF;
   END LOOP;
```



```
 product<= p_mult (width);
END IF;
END PROCESS;
END behave;
```

Замечание

Если в списке инициализаторов процесса в программе (листинг 3.30) записать имена входных портов, а промежуточные данные определить как переменные, программа преобразуется в описание матричного умножителя комбинационного типа. Впрочем, в этом случае нет необходимости формировать цепочку элементов задержки исходных данных a_shift и b_shift, т. к. для формирования значений частичных произведений можно непосредственно использовать данные с портов.

Описание цифровых автоматов

Моделью последовательностного дискретного устройства является конечный автомат, т. е. система, способная находиться в одном из состояний из некоторого дискретного множества, изменять состояние в зависимости от входа и формировать выходы в соответствии с определенным алгоритмом. Мы ограничимся представлением синхронных автоматов, изменение состояний которых происходит в моменты появления фронта сигнала на тактирующем входе. Интервал времени между двумя соседними фронтами тактирующего сигнала назовем тактом работы автомата.

Сначала рассмотрим описание автоматов в абстрактной форме, т. е. когда используются обобщенные обозначения состояний, входов и выходов как данных перечислимого типа. Далее показано, как создавать описания более близкие к реальности, определяя входы, состояния и выходы как данные типа битового вектора или именованные сигналы.

Традиционно деление автоматов на автоматы Мили и автоматы Мура. В последнее время набор применяемых моделей, исходя из требований и возможностей практики, расширился — используются автоматы Мили с асинхронными выходами, автоматы Мура с предустановкой и ряд других модификаций.

В автомате Мили состояние и выходной сигнал в очередном такте определяются состоянием и входным сигналом в предыдущем такте работы [15]. Такой модели соответствует структура, проведенная на рис. 3.12.

Пусть автомат Мили задан таблицей переходов и таблицей выходов (табл. 3.5), где в клетках таблицы переходов записаны состояния, в которые переходит автомат из исходного состояния при соответствующем входе, а в клетках таблицы выходов записывается выход при тех же условиях. Нетрудно видеть, что приведенный пример соответствует реверсивному счетчику, причем выходы Y1 и Y2 соответствуют выдаче сигналов переноса.

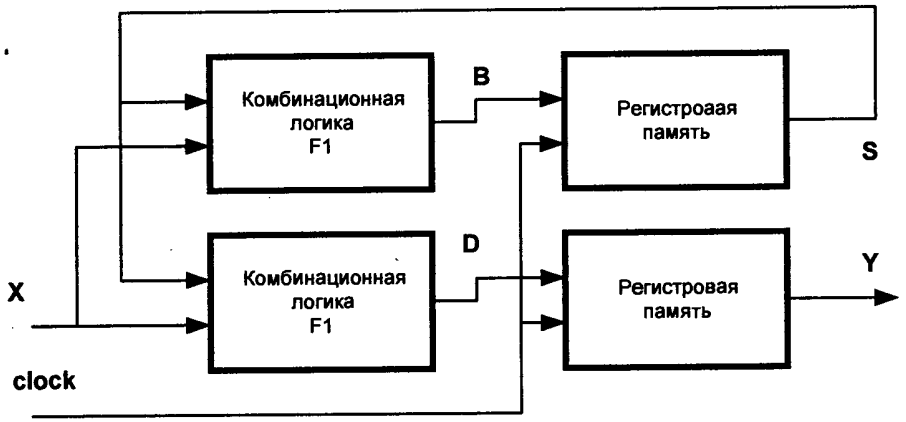

Рис. 3.12. Структура автомата Мили

Таблица 3.5. *Таблицы переходов и выходов автомата Мили*

Таблица переходов				
Вход	**Исходное состояние**			
	S0	S1	S2	S3
X0	S0	S1	S2	S3
X1	S1	S2	S3	SO
X2	S3	SO	S1	S2

Таблица выходов				
Вход	**Исходное состояние**			
	S0	S1	S2	S3
X0	Y0	Y0	Y0	Y0
X1	Y0	Y0	Y0	Y1
X2	Y2	Y0	Y0	Y0

Фрагмент VHDL-программы, описывающий такой автомат, имеет вид, приведенный в листинге 3.31. Предполагается, что перечислимый тип `state` задан списком имен, сигналы `y` и `x` объявлены как порты в декларации `ENTITY`, соответствующей архитектурному телу, в котором определен данный процесс, а их тип задан списком имен — значений.

Процесс после задания исходного состояния (`s<=S0`) входит в бесконечно повторяющуюся петлю, в начале которой помещен оператор `WAIT`. Примененная конструкция оператора соответствует синхронному автомату, состояние которого изменяется по тактирующему сигналу `p_clk`, причем `p_clk` является глобальной переменной проекта.

Важно обратить внимание, что изменения состояний происходит в момент появления нарастающего фронта сигнала `p_clk`, т. к. запускающее событие определено как "появление единицы и наличие переходного процесса на входе `p_clk`".

Использование в качестве условия продолжения процесса выражения "`not p_clk'stable`" соответствует реальной структуре устройства, реализующего автомат. В таком устройстве состояние отображается состоянием регистра. Так как этот регистр является датчиком информации о текущем состоянии и одновременно приемником нового значения, во избежание гонок необходимо использовать регистры с динамическим управлением. После вычисления нового состояния и выходных сигналов процесс переходит в состояние ожидания нового запускающего события. Обратите внимание на то, что сигналы вычисляются на основе состояний, которые были *перед фронтом* тактирующего сигнала, а не вычисленных в текущем цикле.

Листинг 3.31

```
ealey_mach:PROCESS
ARIABLE s: state;
BEGIN
s<=S0;
 LOOP
 WAIT UNTIL (p_clk='1' and not p_clk'stable);
-- Реализация переходов
 CASE S IS
    WHEN S0 => IF x=x0 THEN s<=s0;
               ELSIF (x=x1) THEN s<=S1;
               ELSE s<=s2;
                END IF;
    WHEN S1 => IF x =x0 THEN s<=S1;
               ELSIF x =x1 THEN s<=S2;
               ELSE s<=S0;
               END IF;
    WHEN S2 => IF x =x0 THEN s<=S2;
               ELSIF x =x1 THEN s<=S3;
               ELSE s<=S1;
               END IF;
    WHEN S3 => IF x =x0 THEN s<=S3;
               ELSIF x =x1 THEN s<=S0;
               ELSE s<=S2;
               END IF;
  END CASE;
- Формирование выходов
   IF ( s = s3 and x=x1) THEN y<=y1;
   ELSIF (s =s0 and x=x2) THEN y<=y2;
   ELSE y<=y0;
   END IF;
END loop;
ND process;
```

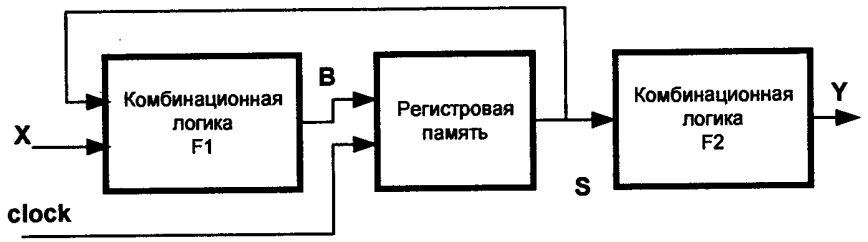

Рис. 3.13. Каноническая реализация автомата Мура

В автомате Мура выходной сигнал зависит только от состояния автомата в текущий момент времени и обычно включается в состав отмеченной таблицы переходов. Такому представлению соответствует структура, приведенная на рис. 3.13. Описание автомата на основании модели Мура отличается не только тем, что значение *x* не входит в выражение для вычисления выхода, но и самой структурой программы. Действительно, при вычислениях по программе (листинг 3.31) аргументом является предыдущее состояние. В модели Мура подобная запись вызывает как бы запаздывание на один такт.

Чтобы корректно представить автомат Мура, целесообразно определение выходов выделить в отдельный процесс, инициатором которого является сигнал, отображающий состояние автомата S. Типовая структура подобной программы приведена на рис. 3.14.

```
ATCHITECTURE <имя архитектурного тела>
            OF <имя ENTITY> IS
 TYPE state IS < идентификаторы состояний>
 SIGNAL S: state;
BEGIN
F2:   PROCESS  (S)    -- комбинационная логика
  BEGIN
    Y<=F2(S);
  END PROCESS F2;
MEM:   PROCESS (clock,reset)  -- регистровая память
VARIABLE B: state;
BEGIN
   IF reset='1' THEN < исходное состояние>
   IF clock='1' and clock'event THEN
    b :=F1(x,S);     -- комбинационная логика
S<=b;
   END IF
  END PROCESS MEM;
END <имя архитектурного тела>;
```

Рис. 3.14. Структура программы описания автомата Мура

В качестве примера в листинге 3.32 представлено описание автомата Мура, заданного отмеченной таблицей переходов (табл. 3.6). Предполагается, что перечислимые типы `mash_in`, `state` и `mash_out` определены в пакете `State_mashine_define`, скомпилированном в рабочую библиотеку `work`.

Таблица 3.6. *Отмеченная таблица переходов автомата Мура*

Вход	Состояния / выходы		
	S0/Y0	**S1/Y1**	**S2/Y0**
X0	S0	S1	S0
X1	S1	S2	S1

Листинг 3.32

```
USE work. State_mashine_define.all
ENTITY moor_example IS
PORT(clock, reset, :IN std_logic;
       X: IN mash_in; -- допустимые значения X0 и X1
       z: OUT mash_out); -- допустимые значения Y0 и Y1
END moor_example;
ARCHITECTURE abstraction OF moor_example IS
SIGNAL moor: state; -- список допустимых значений (S0,S1,S2);
BEGIN
PROCESS (clock,reset)
BEGIN
IF reset='1' THEN moor<=S0;
ELSIF clock='1' and clock'event THEN
CASE moor IS
WHEN S0=> IF x=X0 THEN moor= S0;
ELSE moor= S1;
END IF;
WHEN s1=> IF x=X0 THEN moor= S1;
ELSE moor= S2;
END IF;
WHEN s2=> IF x=X0 THEN moor= S0;
                ELSE moor= S1;
                END IF;
     END CASE;
  END IF;
END PROCESS;
```

```
 PROCESS (moor)
  BEGIN
   IF moor=S1 THEN z<=Z1;
   ELSE z<= Z0;
   END IF;
  END PROCESS;
END abstraction;
```

В реальных системах входы и выходы представлены конкретными кодовыми комбинациями и их представление в абстрактной форме не всегда удобно.

Однако запись функций переходов и выходов при представлении соответствующих данных в форме битовых векторов структурно не отличается приведенных примеров, записанных для абстрактного представления: достаточно указывать в выражениях условий не абстрактные имена входов, а соответствующие коды. Отметим, что любому абстрактному входу соответствует одна и только одна кодовая комбинация. Иногда более компактная и наглядная запись получается, если условия переходов формируются как функции конкретных битов входного кода, а каждому переходу соответствует установка или сброс конкретного бита результата. В качестве примера перепишем представление автомата, заданного табл. 3.6 и представленного процессом `mealey_mach` в листинге 3.33, при использовании следующего кодирования входов и выходов:

`x0="00", x1="01", x2="1x", y0="00", y1="01", y2= ""10".`

Листинг 3.33

```
Mealey_mach:PROCESS
VARIABLE s: state;
Variable prom: std_logic_vector (1 downto o);
 BEGIN
 s<=S0;
  LOOP
  WAIT UNTIL (p_clk='1' and not p_clk'stable);
 Prom="00"
   CASE S IS
WHEN S0 => IF  x(1)='1' THEN s<=s3;  prom(1):='1'; --x=x2
           ELSIF  x(0)='0' THEN s<=s0;             --x=x0
                ELSE s<=s1;                        --x=x1
                 END IF;
     WHEN S1 => IF  x(1)='1' THEN s<=s0;
           ELSIF  x(0)='0' THEN s<=s1;
                ELSE s<=s2;
                 END IF;
```



```
WHEN S2 => IF  x(1)='1' THEN s<=s1;
           ELSIF  x(0)='0' THEN s<=s2;
                ELSE s<=s3;
                 END IF;
WHEN S3 => IF  x(1)='1' THEN s<=s1;
           ELSIF  x(0)='0' THEN s<=s3; prom(0)<='1';
                ELSE s<=s0;
                 END IF;
END IF;
    END CASE;
y<=prom;
END loop;
END process;
```

Следующий простой пример иллюстрирует описание управляющего автомата с объявлением индивидуальных имен входных и выходных сигналов, причем в данном примере роль выходных сигналов исполняют непосредственно биты кода состояния автомата. Автомат задан графом переходов, представленным на рис. 3.15.

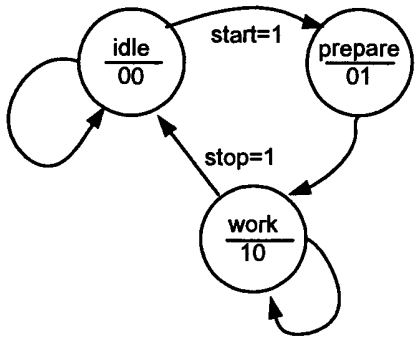

Рис. 3.15. Граф переходов автомата с именованными входами и выходами

На дугах переходов отражены только значения сигналов, вызывающие переход из соответствующего состояния. Алгоритм предполагает, что в исходном состоянии, обозначенном `idle`, устройство ожидает единичного уровня сигнала на входе `START`. После этого в течение одного такта автомат находится в состоянии подготовки (prepare), выдавая на выход сигнал, также обозначенный `prepare`, который сопоставлен младшему биту кода состояния. Переход в состояние работы (work) из состояния подготовки безусловный. Далее, до тех пор пока не получен единичный сигнал на входе `STOP`, автомат сохраняет свое состояние, выдавая единичный управляющий сигнал, кото-

рый сопоставлен старшему биту коду состояния. Сигнал STOP переводит автомат в исходное состояние.

В программе (листинг 3.34), описывающей этот автомат, переменная состояния STATE представлена вектором, "составленным" из выходных сигналов, а логика переходов базируется на анализе только тех входных сигналов, которые важны в соответствующем состоянии.

Листинг 3.34

```
ENTITY direct_inout_asignments IS
PORT( clk,reset:IN bit;
      start,stop:IN bit;
      work,prepare:inout bit);
END direct_inout_asignments;
ARCHITECTURE behave OF direct_inout_asignments IS
 BEGIN
PROCESS (clk,reset)
  VARIABLE state: bit_vector (1 DOWNTO 0);
  BEGIN state:=work & prepare;
   IF reset='1' THEN  work<='0'; prepare<='0';
    ELSIF clk='1' and clk'event THEN
        CASE state IS
         WHEN "00"=> IF start='1' THEN
                       prepare<='1';
                      END IF;
         WHEN "01"=> prepare<='0';work<='1';
         When "10"=> IF stop='1' THEN
                       work<='0';
                     END IF;
         WHEN "11"=>NULL;
         END CASE;
  END IF;
END PROCESS;
END behave;
```

При описании аппаратуры на языке VHDL следует учитывать, что при компиляции программы структура синтезируемой схемы и ее временные характеристики существенно зависят от формы записи автомата.

Варианты, подобные представленному листингом 3.34, позволяют минимизировать запаздывание между выходными сигналами и регистром памяти. В тех же случаях, когда необходимо минимизировать время удержания входных данных, целесообразно использовать структуру автомата Мура с предустановкой в соответствии со схемой, приведенной на рис. 3.16. Типовая структура такой программы представлена на рис. 3.17.

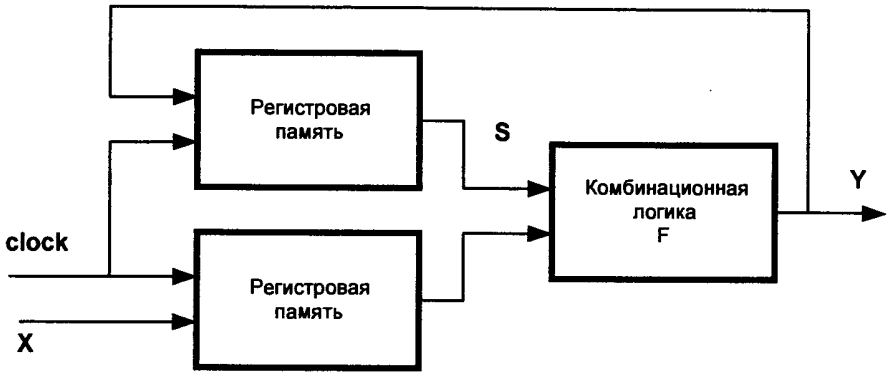

Рис. 3.16. Автомат Мура с предустановкой

```
arhitecture moore2 of State_mashine is
<описания локальных сигналов>
process  -- схемы с памятью
    begin
       <установка исходного состояния>
        wait until  clock='1' and clock'event;
        X1<=X;           -- предустановка входа
           S<=F1(X1,S); -- изменение состояния
    end process;
 process   (S,X1)    --  комбинационная логика
    begin
          Y<=F2(S,X1);
        end process;
end moore2;
```

Рис. 3.17. Структура описания автомата Мура с предустановкой

И, наконец, следует отметить такую модификацию модели последовательностных схем, как автомат Мили с асинхронными выходами, являющийся по сути гибридом автоматов Мили и Мура. В этой модели выходной сигнал является функцией от текущего состояния и состояния входов в текущий момент времени. Структурное представление такого автомата представлено на рис. 3.18. Программа, описывающая указанную реализацию, может быть подобна по структуре рис. 3.14, с той разницей, что в список инициализаторов процесса F2 и в состав аргументов для вычисления выхода войдет не только сигнал состояния, но и входной сигнал.

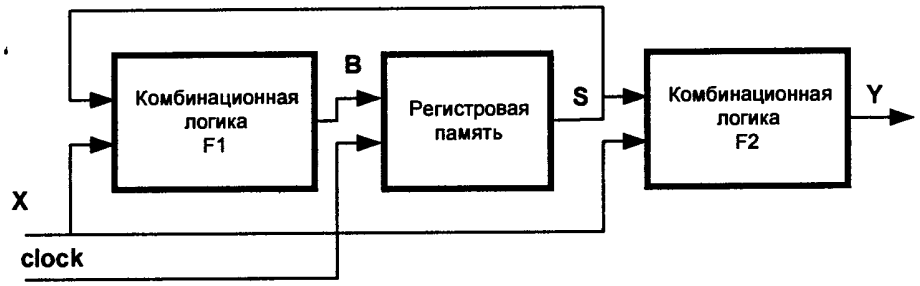

Рис. 3.18. Автомат Мили с асинхронными выходами

3.2.9. Подпрограммы

Подпрограммы в VHDL, как и в других алгоритмических языках, обеспечивают, во-первых, структуризацию описания проекта за счет разделения его на законченные внутренне определенные блоки, а во-вторых, являются средством экономии времени проектировщика, позволяя заменить несколько описаний сходных фрагментов алгоритма одним объявлением подпрограммы и соответствующими ссылками на нее (вызовами) в основном тексте.

Каждая подпрограмма, используемая в проектном модуле, должна быть представлена телом подпрограммы в разделе деклараций этого модуля или проектного модуля, иерархически старшего по отношению к данному. Подробнее концепция видимости деклараций рассматривается далее в *разд. 3.2.12.*

Различают два вида подпрограмм — процедуры (PROCEDURE) и функции (FUNCTION).

Оба вида содержат в своем теле набор *последовательных операторов*, которые задают совокупность действий, исполняемых после вызова этой подпрограммы. Процедура возвращает результаты либо путем непосредственного преобразования объектов, определенных в вызывающей программе (глобальных сигналов или переменных), либо за счет сопоставления объектов через список соответствий. Функция же определяет единственное значение, используемое в выражениях, в которые включен вызов этой функции.

Объявления подпрограмм отображаются в текстах телами подпрограмм, которые подчиняются следующему синтаксическому правилу:

```
<тело подпрограммы> ::=
    <спецификация подпрограммы> IS
    <раздел деклараций подпрограммы>
    BEGIN
        « <последовательный оператор> »
    END [ PROCEDURE | FUNCTION ] <имя  подпрограммы>;
```



```
<спецификация подпрограммы> ::=
    PROCEDURE <имя  подпрограммы> [ (<интерфейсный список>) ]
  | FUNCTION <имя подпрограммы> [ (<интерфейсный список>) ]
    RETURN <тип>
<интерфейсный список>  ::=
    <Элемент интерфейсного списка> «;<Элемент интерфейсного списка>»
<Элемент интерфейсного списка>::=
    [ CONSTANT | VARIABLE | SIGNAL ]
     <формальный параметр> «,<формальный параметр>»
        :<направление> <тип> [ :=<константное выражение> ]
```

Спецификация подпрограммы определяет ее интерфейс — имя, входные и выходные данные. Формальный параметр следует понимать как имя, присваиваемое на время исполнения подпрограммы фактическому параметру, т. е. объекту, сопоставленному этому формальному параметру в списке соответствий оператора вызова. Входные данные подпрограммы специфицируются направлением IN, выходные — направлением OUT, а данные, которые воспринимаются подпрограммой и возвращаются в вызывающую программу измененными, — как INOUT. Указание категории элемента списка (CONSTANT, VARIABLE или SIGNAL) обеспечивает контроль корректности использования подпрограммы. По умолчанию определено, что сопоставляемый объект относится к категории VARIABLE. Несоответствие типа или категории фактического или формального параметра является ошибкой. Необязательное константное выражение, завершающее представление элемента интерфейсного списка, допустимо только для параметров категории VARIABLE. Оно определяет значение по умолчанию, т. е. принимаемое соответствующей переменной, если при каких-либо условиях вызова подпрограммы присвоение иного значения не предусматривается.

В разделе деклараций подпрограммы могут определяться локальные, т. е. определенные только в теле подпрограммы, объекты: вложенные подпрограммы, типы и подтипы данных, переменные, константы, атрибуты. Раздел операторов содержит только последовательные операторы.

Вызов подпрограмм подчиняется единому для процедур и функций синтаксическому правилу:

```
<вызов подпрограммы> ::=
    <имя подпрограммы>[ <список соответствий> ]
<список соответствий>::=
    ([ <Формальный параметр>] =><фактический параметр>
    «,[ <Формальный параметр> ] => <фактический параметр>»)
<фактический параметр> ::=
    <выражение> | <константа> | <имя сигнала> | <имя переменной> |
    <вызов функции>
```

Вызов процедуры записывается в программе как отдельный оператор, а вызов функции используется в выражениях того же типа, что и тип возвращаемого параметра, как обычная переменная.

Язык VHDL, в отличие от традиционных языков программирования, различает последовательный и параллельный вызов подпрограммы. Синтаксически они одинаковы, но различна их локализация и правила исполнения. Вызов функции трактуется как параллельный, если входит в параллельный оператор, чаще всего — в оператор параллельного присваивания, и как последовательный, если входит в последовательный оператор.

Оператор вызова процедуры является последовательным, если локализован в теле процесса или теле другой подпрограммы. В иных случаях оператор вызова подпрограммы интерпретируется как параллельный оператор. Одна и та же подпрограмма может вызываться как параллельным, так и последовательным оператором. Как и другие последовательные операторы, оператор последовательного вызова выполняется после исполнения всех операторов, предшествующих ему в теле процесса или теле подпрограммы. Параллельный оператор вызова исполняется после изменения любого из сигналов, перечисленных в списке соответствий этого оператора. Иными словами, параллельный вызов процедуры эквивалентен процессу, тело которого совпадает с телом процедуры с точностью до обозначений, а список инициализаторов содержит входные фактические параметры оператора вызова.

Например, в листинге 3.35 представлены несколько узлов суммирования-вычитания в различной форме (естественно, это лишь иллюстративный пример — практически в подобных случаях используют одинаковую форму). Предполагается, что все сигналы определены в ENTITY проекта, причем controle определен как бит, остальные отнесены к типу integer.

Листинг 3.35

```
ARCHITECTURE add_examples OF  somethin IS
    PROCEDURE  add_SUBB
            ( signal a,b:IN integer;
              signal Operation : in std_logic; -- '0' -add '1' – subb
              signal Result: out integer) IS
        BEGIN IF operation='0' THEN result<= a+b;
                               ELSE  result<=a-b;
 END IF;
 END add_subb;
BEGIN
   First:  add_subb(x1,y1,controle,z1);
   Second: add_subb( a=>x1,b=>y1, result=>z2, operation=>controle);
   Third: PROCESS ( x2,y2, controle)
```



```
            BEGIN  IF controle='0' THEN z3<= x2+y2;
                      ELSE  z3<=x2-y2;
                   END IF;
         END  PROCESS;
   fourth: PROCESS (controle)
             BEGIN -- .... <дополнительные операторы>
              add_subb( x3,y3,controle, z4);
          END PROCESS;
END add_examples;
```

Параллельные операторы вызова first и second иллюстрируют альтернативные способы записи списков соответствия. Первая форма записи списка ассоциаций соответствует позиционному сопоставлению фактических и формальных параметров подпрограмм, а вторая — сопоставлению по имени. Позиционное сопоставление требует точного совпадения порядка записи фактических параметров в списках соответствия и порядка записи формальных параметров в интерфейсном списке подпрограммы. Если какой-либо параметр не используется или используется значение входа по умолчанию, соответствующая позиция в списке отмечается как пустая. При сопоставлении по имени порядок записи не имеет значения, важно лишь совпадение имени формального параметра с именем, указанным в декларации подпрограммы.

Оператор процесса third реализует такие же преобразования, что и операторы вызова first и second. Обратите внимание: список инициализаторов этого процесса совпадает со списком формальных параметров оператора first. И наконец, вызов подпрограммы в теле процесса fourth, внешне совпадающий с оператором first, будет выполняться только при изменении сигнала controle и после выполнения всех предшествующих ему дополнительных операторов.

Важное значение в подпрограммах имеет оператор возврата RETURN. В процедуре этот оператор прекращает ее исполнение, передавая управление вызывающей программе. Если исполнен оператор RETURN в процедуре, вызванной последовательным оператором, то после него выполняется оператор вызывающей программы, следующий за оператором вызова. После исполнения оператора RETURN в процедуре, вызванной параллельным оператором, интерпретатор программы обращается к календарю событий и инициирует исполнение оператора, связанного со следующим событием в календаре. При отсутствии оператора возврата исполнение процедуры завершается последним оператором в порядке записи.

Оператор возврата в теле функций обязателен. Он также прекращает исполнение подпрограммы, но, кроме того, выполняет присвоение значения результату, который используется в вызывающей программе на месте вызова функции.

Например, программа в листинге 3.36 содержит функцию, возвращающую значение 0, если хоть один элемент во входном массиве равен нулю, и сумму всех элементов этого массива в противном случае. В операторе повторения последовательно проверяются элементы фактического массива, и если очередной элемент нулевой, выполнение подпрограммы прекращается оператором RETURN 0 (возвращается нулевой результат). Если же нулевого элемента в массиве нет, выполняется накопление суммы для всех элементов, и это значение возвращается в вызывающую программу оператором RETUNN sum. Имена result и input должны быть определены в иерархически старшем модуле проекта как данные целого типа.

Листинг 3.36

```
ARCHITECTURE nothing OF something IS
TYPE v IS ARRAY (integer range<>) OF integer;
 FUNCTION sum_until_zero (SIGNAL data:IN v)
                return integer IS
  VARIABLE i,sum:integer;
   BEGIN  FOR i IN data'range LOOP
         IF data(i) =0 THEN  RETURN 0;--обнаружен нулевой элемент
                    ELSE sum:= sum+data(i);
          END IF;
       END LOOP;
   RETURN sum;
 END sum_until_zero;
SIGNAL data_array : v ( 0 TO 15);
 BEGIN
   PROCESS(data_array)
   BEGIN
      result <= input+ sum_until_zero(data_array);
END PROCESS;
END nothing;
```

В связи с рассмотренным примером отметим еще одно обстоятельство. Здесь в описании функции тип формального параметра определен как неограниченный массив целых. Фактический параметр отнесен к тому же типу, но при декларации сигнала определен фактический его размер. Размер массива в подпрограмме автоматически устанавливается равным размеру фактического массива. Атрибут фактического массива data'range использован для задания числа повторений оператора LOOP. Такой прием часто используется для создания подпрограмм обработки массивов произвольной размерности.

3.2.10. Разрешаемые сигналы и шины

В предыдущих разделах мы обходили рассмотрение случаев, когда несколько источников подключаются к одной линии. Если сигнал имеет один драйвер, то его значение определяется достаточно просто. После исполнения или перехода в состояние ожидания всех процессов и параллельных операторов, вызванных общим событием, предсказанные изменения передаются из драйверов сигналов, являющихся, в сущности, программными буферами, в поле данных системы моделирования. Это и определяет новое значение сигнала.

Однако во многих системах к одной линии подключено несколько источников. Например, шина данных компьютера может принимать сигналы от процессора, памяти, периферийных устройств, а к линии данных матрицы запоминающего устройства подключается достаточно много ячеек памяти, а также буферы ввода/вывода. Этому соответствуют программные модели, в которых один сигнал назначается в нескольких параллельных операторах и процессах. В VHDL не любые сигналы способны принимать значение в соответствии со значениями нескольких источников. Сигналы, значения которых автоматически определяются исходя из состояний нескольких источников (драйверов), называют *разрешаемыми* (resolved). Разрешаемым может быть объявлен конкретный сигнал или подтип данных, к которому такой сигнал отнесен при его декларации. Мы ограничимся объявлением сигнала как разрешаемого (resolved) через использование декларации разрешаемого подтипа. Напомним, что декларация подтипа может содержать имя *функции разрешения* (resolution function). Функция разрешения определяет правило вычисления сигнала, формируемого несколькими независимыми источниками. В общем случае функция разрешения зависит от конкретных условий приложения проекта, в том числе технологии изготовления проектируемого устройства, и если в системе интерпретации отсутствует подходящий аналог, требуется разработка соответствующей программы.

Функция разрешения локализуется в том же программном модуле, что и декларация подтипа, и вызывается всякий раз, когда меняет состояние любой из драйверов разрешаемого сигнала. Можно сказать, что по умолчанию предполагается наличие в программном модуле параллельного вызова этой функции, причем драйверы сигналов являются ее фактическими параметрами, а возвращаемое значение присваивается сигналу.

Рассмотрим в качестве примера представление линии, к которой подключаются элементы с выходами типа "открытый коллектор". Если хоть один источник выдает на линию уровень логического нуля, то на линии устанавливается логический нуль. Источники могут быть независимыми и даже представленными в различных проектных модулях. Если принять, что переходные состояния несущественны, можно ограничиться двоичным представлением. Однако тип BIT не может быть использован в подобной ситуа-

ции, ибо является неразрешаемым. Введем тип odw, как разрешаемый подтип типа BIT, и соответствующую функцию разрешения:

```
FUNCTION resolved_odw( s:bit_vector) RETURN bit IS
  VARIABLE c: bit:='1';
  BEGIN  FOR i IN s'range LOOP
     IF s(i)='0' THEN c:='0'; EXIT;
     END IF;
  END LOOP;
  RETURN c;
END resolved_odw;
SUBTYPE odw IS resolved_odw bit;
```

Пусть эти декларации содержатся в пакете wired_logic, скомпилированном в рабочую библиотеку work. Тогда узел передачи на общую линию сигналов от нескольких независимых источников можно описать следующим образом (листинг 3.37).

Листинг 3.37

```
library ieee;
use work.pack_wired_or.all;
ENTITY open_drain is
 PORT (data,selector:in bit_vector(7 DOWNTO 0);
        single:IN bit;
        output:OUT odw);
END open_drain;

ARCHITECTURE example OF open_drain IS
 BEGIN
  output<='0' WHEN single='0'  ELSE '1';
 multiple_assign: PROCESS(data,selector)
  BEGIN  FOR i IN 0 TO 7 LOOP
    IF( data(i)='1' and selector(i)='1') THEN output<='0';
     END IF;
    END LOOP;
  END PROCESS;
END example;
```

В программе (листинг 3.37) описана ситуация, когда к линии output подключено девять источников, и дополнительные источники могут подключаться к этому порту при включении модуля open_drain в иерархические проекты. Сигнал output имеет в модуле open_drain девять драйверов, присваивающих ему различные значения: восемь из них представлены в процессе multiple_assign оператором повторения, а один — параллельным оператором условного присваивания.

Обращаясь теперь к функции resolved_odw, можно обратить внимание на то, что драйверы одного сигнала, являющегося ее фактическим параметром, по умолчанию трактуются как неограниченный массив, объединяющий состояния этих драйверов. После изменения состояния любого драйвера выполняется просмотр всех элементов массива, причем число повторений задается атрибутом s'range, определяющим границы фактического массива. Если обнаружен драйвер, сохраняющий значение логического нуля, цикл прекращается и сигналу присваивается нулевое значение.

К счастью, при проектировании чаще всего не требуется создание собственных разрешаемых подтипов данных. В современных системах интерпретации всегда присутствует пакет std_logic_1164, в котором, в частности, определен подтип std_logic как разрешаемый подтип девятизначного перечислимого типа std_ulogic *(см. разд. 3.2.3)*.

Приведем текст (листинг 3.38) определения подтипа std_logic [34].

Листинг 3.38

```
  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', '0'), -- | 0 |
          ( 'U', 'X', 'X', '1', '1', '1', '1', '1', '1'), -- | 1 |
          ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'Z'), -- | Z |
          ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'W'), -- | W |
          ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'L'), -- | L |
          ( 'U', 'X', '0', '1',' H', 'W', 'W', 'H', 'H'), -- | H |
          ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')  -- | - |
   );
FUNCTION resolved ( s : std_ulogic_vector) RETURN std_ulogic IS
   VARIABLE result : std_ulogic := '-';     -- Значение по умолчанию -
                                            -- самый слабый сигнал
   BEGIN
       IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW);
       ELSE                                 -- просмотр всех драйверов
           FOR i IN s'RANGE LOOP
               result := resolution_table(result, s(i));
           END LOOP
           RETURN result;
   END IF;
END resolved;
SUBTYPE std_logic IS resolved std_ulogic;
```

Функция разрешения resolved устанавливает значение сигнала в соответствии с "наиболее сильным" значением сигнала среди драйверов. Например, активный нуль и активная единица определены как самые сильные и подавляют любые другие сигналы, кроме друг друга. Если присутствуют два драйвера равной силы, но имеющие различные значения, результат считается неопределенным.

Использование подтипа std_logic позволяет облегчить решение многих проблем, возникающих при моделировании шинных соединений. Этот подтип и порождаемый на его основе тип std_logic_vector стали фактическим стандартом. Данные этого типа нередко используют и в случаях, когда сигналы имеют единственный драйвер и даже если при моделировании достаточно воспроизводить меньшее число состояний. Однако платой за такую универсализацию может стать увеличение затрат машинного времени при моделировании.

В листинге 3.39 приведено описание одного разряда матрицы статического запоминающего устройства. Состояние ячеек памяти представлено массивом переменных cell. Если *i*-я ячейка выбрана, т. е. сигнал разрешения en(i) установлен в единицу (имеется в виду, что адрес представлен унитарным кодом или код адреса подвергнут дешифрации в дополнительных узлах), а усилитель записи отключен (Wr='0'), то состояние ячейки передается на разрядную линию данных dat_bus и далее на усилитель чтения. Если же присутствует сигнал записи (Wr='1'), то линия данных и выбранная ячейка принимают значение входного сигнала. Проблема в том, что ячейка в зависимости от состояния линии данных является и приемником, и источником сигнала, причем вход и выход — это одна и та же точка схемы.

Чтобы корректно описать эту ситуацию, следует приписать ячейкам "слабые" состояния, а сигналу на линии данных — разрешаемый тип.

Листинг 3.39

```
LIBRARY ieee;
USE IEEE.std_logic_1164.all;
  ENTITY ram_array IS
     GENERIC ( length:integer:= 64);
     PORT( en:IN std_logic_vector(length-1 DOWNTO 0);-- входы выборки
          wr,rd: IN std_logic;                  -- wr-запись, rd – чтение
            data:INOUT std_logic
      );
  END ram_array;
ARCHITECTURE behave OF ram_array IS
       SIGNAL cell: STD_uLOGIC_VECTOR(LENGTH-1 DOWNTO 0);
        SIGNAL data_bus: std_logic;
```



```
        BEGIN
    data_bus<='Z' WHEN (wr='0') ELSE       -- входной буфер
            '1' WHEN (wr='1' and  data='1') ELSE
            '0' WHEN (wr='1' and data='0') ELSE
             'X';
transmit: PROCESS( rd, data_bus)           -- выходной буфер
     BEGIN
        IF rd='1' THEN
           CASE data_bus IS
             WHEN '1'|'H'=> data<= '1';
             WHEN '0'|'L'=> data<= '0';
             WHEN OTHERS => data<= 'X';
           END CASE;
         ELSE  data<='Z';
        END IF;
     END PROCESS;
cells:PROCESS (data_bus, en)   --  линейка запоминающих ячеек
   BEGIN
     FOR i IN 0 TO length-1 LOOP
      IF en(i)='1' THEN
         CASE data_bus IS
            WHEN '1'|'H'=> cell(i)<= 'H';
            WHEN '0'|'L'=> cell(i)<= 'L';
            WHEN 'Z' => null;
            WHEN others => cell(i)<= 'W';
          END CASE;
      END IF;
     END LOOP;
   END PROCESS;
data_bus_drive: PROCESS (en,data) -- поведение линии данных
                                  -- в режиме чтения
      BEGIN
      FOR i IN 0 TO length-1 LOOP
             IF (en(i)='1') THEN
                  data_bus<=cell(i);
              END IF;
         END LOOP;
  END PROCESS;
END behave;
```

В приведенной программе процесс cells отображает изменение состояния выбранной ячейки в зависимости от состояния линии data_bus. Если к линии данных подключен активный источник, то ячейка переходит в состояние, соответствующее сигналу на линии, с той разницей, что состояние запоминается как слабый сигнал. Варианты оператора CASE, связанные с со-

стояниями 'L' и 'H', определяют подтверждение состояния ячейки, если только одна эта ячейка подключена к линии данных. Если линия данных находится в состоянии 'Z', выбранная ячейка сохраняет свое состояние. Во всех остальных случаях фактическое состояние ячейки в ближайшем будущем непредсказуемо, что отражается символом слабого неопределенного состояния 'W'. Процесс transmit описывает состояние линии входа/выхода данных. Заметим, что тип порта входа/выхода определен как разрешаемый, а значит, в иерархических проектах к этому порту можно подключать другие драйверы.

Сигнал, представляющий линию данных data_bus, имеет драйвер, связанный оператором условного присваивания, и набор драйверов по числу ячеек памяти. Если wr='1', то активный сигнал драйвера оператора условного присваивания подавляет слабые сигналы от подключенной ячейки. Если wr='0', то этот драйвер принимает значение 'Z', и оно подавляется слабыми сигналами ячеек памяти при исполнении функции разрешения, если какие-либо из них выбраны. В случае одновременной выборки нескольких ячеек, находящихся в разных состояниях, data_bus принимает неопределенное слабое значение 'W', что в свою очередь приводит к переходу выбранных ячеек в состояние 'W'.

3.2.11. Структурное представление проекта

В предыдущих разделах рассмотрено несколько языковых понятий, служащих структуризации описания проекта: понятия процесса, блока, подпрограммы.

Язык VHDL предоставляет еще одну возможность структуризации описания — так называемые структурные архитектурные тела (Structural Architectural Body, SAB). Структурное архитектурное тело представляет проект в виде набора используемых компонентов и их связей, т. е. приближает описание к реальной конфигурации проектируемого устройства, как минимум, к представлению разработчика об этой конфигурации. Все используемые в проекте компоненты должны иметь свое ENTITY и однозначно заданное поведение, например поведенческое архитектурное тело, которое следует заранее скомпилировать в библиотеку проекта или иную библиотеку, объявленную перед ENTITY составного проекта.

SAB представляется наиболее эффективным средством создания иерархических проектов. Применение концепции SAB упрощает совместную работу нескольких исполнителей над одним проектом. Упрощается включение в новые проекты ранее созданных модулей, а также использование стандартных библиотек проектных модулей.

По форме структурные и поведенческие архитектурные тела не различаются. Различие состоит в составе используемых операторов и деклараций.

В разделе объявлений SAB, кроме характерных для любых архитектурных тел деклараций, таких как декларации типов, сигналов, констант, включаются специфические подразделы: обязательный подраздел *декларации прототипов используемых компонентов* и необязательный *подраздел объявления конфигураций*. Раздел операторов SAB содержит *операторы вхождения компонентов*, которые, собственно, и определяют порядок соединения компонентов. Могут включаться и другие параллельные операторы, а если таких операторов относительно много, подобное архитектурное тело называют смешанным, или структурно-поведенческим.

Каждому модулю, используемому в качестве структурного компонента в SAB, должно сопутствовать объявление прототипа. Синтаксис декларации прототипа компонента имеет вид:

```
COMPONENT <имя ENTITY компонента> IS
    [ <образ настройки> ]
     <образ порта>
END COMPONENT  [  <имя ENTITY компонента>];
```

Здесь <образ настройки> и <образ порта> — прямая копия высказываний GENERIC и PORT из текста объявления ENTITY соответствующего компонента.

Если в устройстве есть несколько одинаковых модулей, то прототип декларируется только один раз. Например, если в проект входит несколько однотипных регистров, представленных в библиотеке одним и тем же ENTITY, то в структурном теле размещается единственная декларация прототипа регистра, даже если конкретные экземпляры имеют различные параметры. Тем не менее каждому экземпляру, включаемому в проект, называемому также вхождением модуля (INSTANCE), при структурном описании присваивается собственное имя. Это собственное имя предъявляется в разделе операторов архитектурного тела в виде метки оператора вхождения компонента.

В подразделе объявления сигналов обязательно объявление всех соединений между блоками. Каждый сигнал относят к типу, определенному в разделе PORT соответствующего модуля.

Иногда в соединяемых модулях однотипные, в сущности, сигналы портов могут быть объявлены как принадлежащие к данным несовместимых типов. Допустим, что разработчик одного модуля определил группу входных линий как битовый вектор, а разработчик другого модуля определил подобную группу выходных линий как сигнал целого типа, а эти порты надо соединить. Тогда необходимо вводить два сигнала для одной линии связи, условно включая между ними модуль преобразования типа данных, который фактически в аппаратуре не реализуется. Сигнал, подключаемый к приемнику, определяется как соответствующая функция преобразования типа данных от сигнала источника.

Основными операторами SAB являются операторы вхождения компонентов (Component Instantiation Statement), которые определяют порядок соединения включаемых в проект модулей и, возможно, их параметры.

```
<оператор вхождения компонента> ::=
    <метка вхождения> : ([ COMPONENT  ] <имя компонента>
     GENERIC MAP (<Список соответствий параметров настройки>)
     PORT MAP  (<список соответствий порта>);
<список соответствий> ::=
     ([ <Формальный параметр>] =><фактический параметр>
    «,[ <Формальный параметр> ] => <фактический параметр>»)
```

По форме списки соответствий настроек и порта совпадают со списком соответствий вызова подпрограмм. Но для параметров настройки фактическим параметром может быть только константное выражение, а для порта — только имя сигнала либо имя порта "главного" модуля. Нельзя объявлять в качестве фактического параметра в списке соответствий порта имя входа или выхода другого компонента. Все связи осуществляются только через объекты, объявленные в текущем архитектурном теле как сигналы. (В принципе, объявление сигналов может осуществляться в общедоступном модуле PACKAGE.)

Подобно вызову подпрограмм возможны полная и сокращенная формы записи списка соответствий. В сокращенной записи (без формальных параметров) все элементы списков соответствий записываются в том же порядке, как в списках параметров и сигналов порта данного компонента. Если используется полная форма списка соответствий, то порядок записи элементов в списке произволен.

Рассмотрим для примера структурное описание устройства, изображенного на рис. 3.19. Предположим, что ENTITY и архитектурные тела модулей *V*, *W* и *Z* скомпилированы в рабочую библиотеку проекта. Каждому вхождению компонента в проект присвоено собственное имя. Например, компонент *U*1 структурной схемы реализован как модуль типа *V*, т. е. модуль *V* является его прототипом.

Не вдаваясь в детали описания функционирования этих компонентов, определим, что узел *U*1 принимает байты данных по линии *A*, в зависимости от управляющих сигналов преобразует их и выдает результат на четырехбайтный выход. Модуль *W* выполняет некоторые преобразования входного слова данных. В конкретной структуре компонент *U*2 принимает старший байт данных от компонента *U*1 и преобразует его. Этот байт замещает старший байт выходных данных компонента *U*1, и сформированный таким образом код поступает на вход компонента *U*3, выход которого является выходом устройства.

В архитектурном теле программы описания этого устройства, представленном в листинге 3.40, после объявления прототипов компонентов определены

сигналы: каждой линии связи сопоставляется сигнал с собственным именем. Для большей наглядности на рисунке и в тексте программы порты модулей обозначены строчными буквами, а сигналы — прописными, хотя формально это не имеет значения. Строчные и прописные буквы в обозначениях имен равнозначны, а имена портов компонентов и имена сигналов независимы и, в частности, могут совпадать.

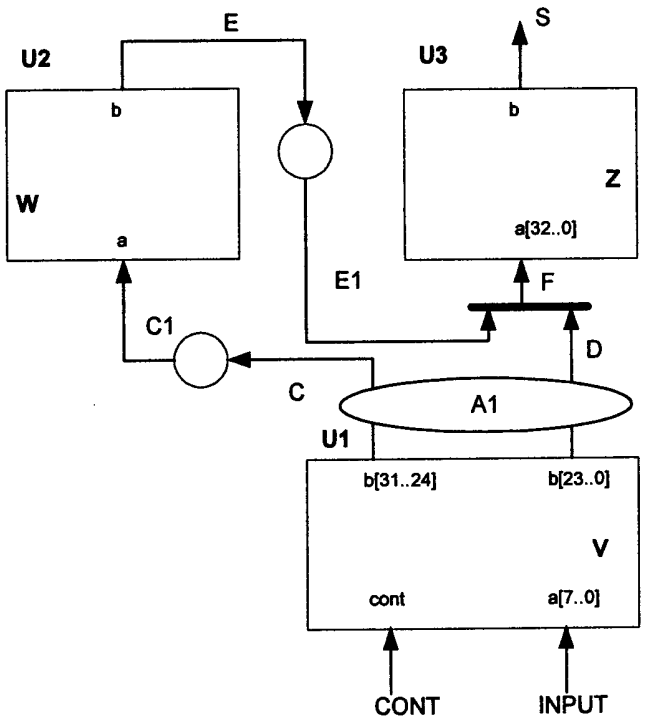

Рис. 3.19. Структурное представление проекта

Каждый компонент в архитектурном теле представлен своим оператором вхождения, метка которого определяет его имя в проекте, вслед за чем записано имя прототипа и списки соответствий.

Если типы данных соединяемых портов компонентов совпадают, то объявляется один общий сигнал, имя которого размещается на соответствующих местах в списках соответствий портов компонентов. В данной программе примером описания такого соединения является сигнал d. При несовпадении типов данных портов, а такое может произойти при переносе проектных модулей в различные проекты, следует вводить условные модули преобразования типов. На рисунке такие "виртуальные" модули, которым при фактической реализации не сопоставляется никаких элементов, представле-

ны кружочками. Так старшие разряды выходного кода компонента *U*1, представленные группой A(31 DOWNTO 0), переименованы оператором с меткой rename и подвергнуты преобразованию в целый тип для согласования с типом данных, определенным для входного порта модуля *W*.

Обратим внимание на форму записи списков соответствия. Для компонентов *U*1 и *U*2, прототипами которых являются модули *V* и *W* соответственно, используется сокращенная форма записи списков соответствия портов, причем для компонента *U*2 опущен список соответствия параметров, что означает, что принято значение по умолчанию — в данном случае width:= 8. Оператор вхождения для компонента *Z* показывает полные формы списков соответствия. Ключевое слово OPEN в качестве формального параметра означает неиспользуемый порт. Для входного порта это соответствует использованию постоянного значения сигнала, определенного по умолчанию.

Листинг 3.40

```
ENTITY structure_example IS
    PORT ( input: IN std_logic (7 DOWNTO 0);
           controle: IN std_logic (3 DOWNTO 0);
           s: OUT std_logic);
END structure_example;
ARCHITECTURE  structure OF primer IS
COMPONENT V
    PORT(  A:IN std_logic_vector (7 DOWNTO 0);
           CONTROLE:IN std_logic_vector (3 DOWNTO 0);
           B: OUT std_logic_vector (31 DOWNTO 0);
           C:OUT std_logic);
END COMPONENT;
COMPONENT W
       GENERIC (width:integer:=8);
       PORT(  A:IN integer RANGE  2**width-1 DOWNTO 0;
              B: OUT integer RANGE  2**width-1 DOWNTO 0);
END COMPONENT;
COMPONENT  Z
        GENERIC (width:integer:=16);
        PORT(     A: IN std_logic_vector (width-1 DOWNTO 0)
                  C:in std_logic;
                  B,D: OUT std_logic);
END COMPONENT;

SIGNAL  c, e: std_logic_vector (7 DOWNTO 0);
SIGNAL cI,eI:integer RANGE 0 TO 255;--цифровые эквиваленты сигналов C и E
SIGNAL a, f: std_logic_vector(32 DOWNTO 0);
SIGNAL d: std_logic;
```



```
BEGIN
U1: v
      PORT MAP (input, controle, a,d);
U2:W
      PORT MAP (cI,eI);
U3:z
     GENERIC MAP(width=>32)
     PORT MAP(A=>f,C=>d, B=>S,D=>OPEN);
-- согласование типов данных
rename: c<= a(31 DOWNTO 24);
   cI<= conv_integer ( c);
    e<= conv_std_logic_vector ( eI, 8);
    f<=e & a(23 DOWNTO 0);
END structure;
```

Оператор вхождения можно трактовать как вызов процедуры со специальным, наглядным синтаксисом. Но есть и более существенные отличия.

- Вызов подпрограммы инициирует исполнение тела подпрограмм, являющегося набором последовательных операторов. Оператор вхождения вызывает к исполнению архитектурное тело, которое содержит параллельные операторы, и сам является параллельным оператором, исполняемым при каждом изменении его входных сигналов.
- Внутренние переменные и сигналы встраиваемого модуля определяются как статические (в отличие от переменных подпрограмм), т. е. сохраняют свои значения между инициализациями.

3.2.12. Настройка и конфигурирование компонентов

Очень часто устройства проектируются не только как изделия с наперед заданными свойствами, но и для возможности их использования в различных приложениях, требующих однотипных преобразований. Соответственно, текстовое описание желательно создавать в формах, допускающих модификацию в определенных пределах свойств представляемых объектов проектирования. Есть два основных пути создания программ, описывающих множество модулей с идентичными функциями, иначе — перестраиваемых модулей:

- использование параметров настройки (GENERIC);
- разработка нескольких архитектурных тел, подчиненных общему ENTITY, иными словами, имеющих одинаковую алгоритмическую сущность при различии способа описания или способа реализации.

Модуль, содержащий декларацию параметров настройки (GENERIC), называют *параметризованным*. Фактическое значение задается в списке соответствий оператора вхождения.

Параметры, определяющие количественные свойства реализаций (например, разрядность данных, время задержки), используются в выражениях внутри подчиненных архитектурных тел как обычные константы. Но, кроме того, часто применяются параметры структурного характера, уточняющие функции, реализуемые конкретными вхождениями параметризованного модуля и/или его структуру. Чаще всего такие параметры используются в операторах генерации, синтаксис которых определен как:

<оператор генерации> ::=
    <метка оператора генерации>: <схема генерации> GENERATE
        <параллельный оператор> {<параллельный оператор>}
    END GENERATE [<метка оператора генерации>];
<схема генерации> ::=
    IF <булевское выражение> | FOR <имя> IN <диапазон>

Схема генерации с ключевым словом IF разрешает (или блокирует) создание компонентов, представленных вложенными параллельными операторами. Схема с ключевым словом FOR определяет число компонентов, создаваемых в структуре модуля в зависимости от значения параметра настройки.

Программа (листинг 3.41) представляет регистр с синхронной загрузкой параллельного кода, который, в зависимости от параметров настройки, может интерпретироваться как сдвигающий регистр со сдвигом в сторону старших разрядов (right), со сдвигом в сторону младших (left), или реверсивный регистр сдвига (bidir). Каждая версия представлена в программе своим оператором PROCESS. Операторы IF-GENERATE, в зависимости от значения параметра настройки dir, который может задаваться при включении модуля в другие проекты, определяют конфигурацию, используемую в конкретном проекте. В данном случае может быть выбрана одна из трех возможных версий. Оператор проверки прекращает компиляцию и выдает сообщение об ошибке, если при включении задано недопустимое значение параметра настройки.

Листинг 3.41

```
LIBRARY ieee;
USE IEEE.std_logic_1164.all;
ENTITY shIFt_register IS
  GENERIC( num_bits: integer:=8; -- число разрядов
        dir:string:="right"); -- тип регистра – "left" – со сдвигом влево
                              --              "right" – со сдвигом вправо
                               --             "bidir" – реверсивный
  PORT (clk:IN std_logic;
     oper:IN std_logic_vector (1 DOWNTO 0);
     -- 00 – пустая операция
     -- 10 – левый сдвиг для типа "bidir" и сдвиг для других типов
```



```
     -- 01 – правый сдвиг для типа "bidir" и сдвиг для других типов
     -- 11 – загрузка кода
        dl, dr:IN std_logic;
        input:IN std_logic_vector  (num_bits-1 DOWNTO 0);
        output:OUT std_logic_vector (num_bits-1 DOWNTO 0)
    );
      end shift_register;
 ARCHITECTURE behave OF shift_register IS
BEGIN
ASSERT (dir="left" or dir="right" or dir= "bidir") REPORT
      "mode must be 'left', 'right' or 'bidir' "
       SEVERITY error;
LEFTSHIFT: IF dir="left"   GENERATE
 PROCESS(clk)
 VARIABLE state:std_logic_vector(num_bits-1 DOWNTO 0);
BEGIN
 IF (clk='1'and  clk'event) THEN
         IF (oper="11") THEN   state:=input;
               ELSIF (oper="10" or oper="01") THEN
                  FOR i IN  num_bits-1 DOWNTO 1 LOOP
                        state(i):=state(i-1);
                  END LOOP;
                    state(0):=dl;
             END IF; --oper
       END IF; --clk='1'
         output<= state;
         END PROCESS;
      END GENERATE LEFTSHIFT;
RIGHTSHIFT:  IF dir="right"   GENERATE
 PROCESS(clk)
  VARIABLE  state:std_logic_vector(num_bits-1 DOWNTO 0);
BEGIN
IF (clk='1'and  clk'event) THEN
            IF (oper="11") THEN   state:=input;
               ELSIF (oper="01" or oper="10") THEN
                   FOR i IN  0 TO num_bits-2  LOOP
                        state(i):=state(i+1);
                     END LOOP;
                   state(NUM_BITS-1):=dR;
            END IF; --oper
      END IF;     --clk='1'
   output<= state;
   END PROCESS;
 END GENERATE RIGHTSHIFT;
```

```
bidir:  IF dir="bidir"    GENERATE
 PROCESS(clk)
 VARIABLE  state:std_logic_vector(num_bits-1 DOWNTO 0);
BEGIN
IF (clk='1'and  clk'event) THEN
            IF (oper="11") THEN   state:=input;
              ELSIF (oper="01") THEN
                  FOR i IN 0 TO num_bits-2  LOOP
                       state(i):=state(i+1);
                  END LOOP;
                  state(NUM_BITS-1):=dR;
               ELSIF (oper="10") THEN
                  FOR i IN  num_bits-1 DOWNTO 1 LOOP
                       state(i):=state(i-1);
                   END LOOP;
                   state(0):=dl;
            END IF; -- oper
       END IF; -- clk='1'
     output<= state;
  END PROCESS;
 END GENERATE bidir;
END behave;
```

В случаях, когда вариативность компонента достигается разработкой нескольких архитектурных тел, т. е. первичному проектному модулю (ENTITY) этого компонента в библиотеке проекта соответствует несколько различных архитектурных тел, проектный модуль высшего уровня иерархии должен содержать объявление конфигурации компонента. Объявление конфигурации определяет, какое именно архитектурное тело компонента используется в текущем проекте или сеансе отладки. Объявление конфигурации компонента подчиняется следующему синтаксическому правилу:

<конфигурация компонента> ::=
    FOR <спецификация компонента> <индикатор привязки>;
<индикатор привязки> ::=
    ENTITY <имя Entity>(<имя архитектурного тела>)
<спецификация компонента> ::= <список вхождений> : <имя компонента>
<список вхождений> ::=
    <метка вхождения> «, <метка вхождения> » | OTHERS | ALL

Пусть, например, в библиотеку проекта скомпилирован текст программы, приведенной ранее в листинге 3.1, т. е. проекту высшего уровня иерархии доступны его ENTITY и несколько архитектурных тел. Создадим проект, включающий три таких компонента, каждый из которых анализирует шестнадцать разрядов входного кода, а также блок суммирования результатов,

получаемых в каждом из компонентов. Предположим, что по некоторым соображениям оказалось целесообразным один из компонентов представить "чистым поведением", а два других — на уровне логических макроячеек. Тогда архитектурное тело такого проекта может иметь следующий вид (листинг 3.42):

Листинг 3.42

```
ENTITY bit_count_usage IS
  Port (real_input:IN std_logic_vector( 47 DOWNTO 0);
        Real output:OUT inreger RANGE 0 TO 48);
END bit_count_usage;
ARCHITECTURE selected_architectures OF bit_count_usage IS
COMPONENT bit_count
  PORT ( input : IN std_logic_vector (15 DOWNTO 0);
    output: OUT integer RANGE 0 TO 15);
END  COMPONENT;
FOR m1: bit_count USE ENTITY work.bit_count(pure_behave);
FOR others: bit_count USE ENTITY work.bit_count(comb_logic);
  SIGNAL first, second, third: integer RANGE 0 TO 15;
BEGIN
m1:  bit_count
  PORT MAP( input => real_input (15 DOWNTO 0),output=>first);
m2:  bit_count
  PORT MAP( input => real_input (31 DOWNTO 16), output=> second);
m3:  bit_count
  PORT MAP( input => real_input (47 DOWNTO 32),output=> third);
real_output<=first+second+third;
END selected_architectures;
```

В данном примере объявление конфигураций компонентов следует трактовать таким образом.

Для представления компонента m1 в проекте используется первичный проектный модуль ENTITY bit_count, реализованный в соответствии с подчиненным ему архитектурным телом pure_behave. Остальные компоненты bit_count реализуются в соответствии с архитектурным телом comb_logic.

Кроме записи конфигурации непосредственно в архитектурном теле, ссылающемся на конфигурируемый модуль, язык VHLD предусматривает возможность записи конфигураций в отдельно выделенном первичном проектном модуле, который называется *декларацией конфигурации* (Configuration Declaration). Иерархический уровень декларации конфигурации такой же, как у декларации ENTITY и декларации пакетов, и декларация конфигурации не может быть их составной частью. Конфигурации компонентов, содержащиеся в модуле декларации конфигурации, во-первых, доступны для раз-

личных архитектурных тел, а во-вторых, обеспечивают возможность сквозного конфигурирования иерархических структур. BNF-форма декларации конфигурации имеет вид:

```
<декларация конфигурации> ::=
    CONFIGURATION <имя конфигурации> OF <имя ENTITY> IS
      FOR <имя архитектурного тела>
         «<конфигурация компонента>  END FOR »
      END FOR:
    END [ CONFIGURATION ] [ <имя конфигурации> ];
```

Пусть, например, конфигурации компонентов исключены из архитектурного тела selected architectures, и для их объявления создан проектный модуль следующего содержания:

```
CONFIGURATION ttt OF bit_count_usage IS
  FOR  selected architectures
   FOR m1: bit_count USE ENTITY work.bit_count(pure_behave);
   END FOR;
   FOR   OTHERS: bit_count USE ENTITY work.bit_count(comb_logic);
   END FOR;
  END FOR;
END CONFIGURATION;
```

Тогда оператор включения модуля bit_count_usage в иерархический проект может выглядеть следующим образом:

```
<метка вхождения>: COMPONENT work.ttt
   PORT(input=> <аргумент>, output=> <приемник результата>);
```

Запись соответствует синтаксису VHDL'93. В VHDL'87 после метки указывается только имя ENTITY. Конфигурация (если объявленному ENTITY таковая сопоставлена) подключается по умолчанию.

В общем случае декларация конфигурации может также содержать объявления параметров и даже объявления соответствий портов.

3.2.13. Пакеты в VHDL. Концепция видимости описаний

Мы уже неоднократно обращались к понятию пакета (PACKAGE) в VHDL. Пакет — это программный модуль, содержащий описание объектов, доступных нескольким другим программным модулям. В пакете могут быть объявлены глобальные типы, константы, функции, сигналы и т. п. Фактически, общие для нескольких подпроектов типы и сигналы можно объявить только в пакете.

Рассмотрим более формально правила записи пакетов.

Пакет представляется двумя структурными единицами — обязательным первичным модулем "декларация пакета" (PACKAGE DECLARATION) и необязательным вторичным модулем "тело пакета" (PACKAGE BODY).

```
<декларация пакета> ::=
    PACKAGE <имя пакета> IS
        <раздел деклараций пакета>
    END  [ PACKAGE ]  [ <имя пакета> ];
```

Раздел деклараций пакета может содержать спецификации подпрограмм, декларации типов и подтипов, констант, сигналов, атрибутов, компонентов и ряда других объектов.

Тело пакета содержит конкретизацию способов вычисления функций и записывается в соответствии со следующим синтаксическим правилом:

```
<package_body> ::=
    PACKAGE BODY <имя пакета> IS
        <раздел деклараций пакета>
    END [ PACKAGE BODY ]  [ <имя пакета> ];
```

Раздел деклараций тела пакета содержит тела подпрограмм (спецификация этих подпрограмм обязательно присутствует в разделе деклараций пакета), а также дополнительные декларации объектов, используемых в представленных подпрограммах. Могут декларироваться типы, константы, вложенные подпрограммы, объекты некоторых иных классов. Эти объекты недоступны для других проектных модулей.

В качестве примера в листинге 3.43 приведен текст программы, содержащий декларацию и тело пакета std_logic_util, имеющегося в числе пакетов, сопровождающих систему отладки VHDL программ ModelSim. Декларация пакета содержит объявление функций, включенных в тело пакета, в данном случае, функций преобразования типов данных из std_logic_vector в положительное целое и наоборот.

Листинг 3.43

```
library IEEE;
use IEEE.std_logic_1164.all;
package std_logic_util is
    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
                                   return STD_LOGIC_VECTOR;
    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER;
end std_logic_util;

package body std_logic_util is
    type tbl_type is array (STD_ULOGIC) of STD_ULOGIC;
```

```
    constant tbl_BINARY : tbl_type :=
    ('0', '0', '0', '1', '0', '0', '0', '1', '0');

    function CONV_STD_LOGIC_VECTOR(ARG: INTEGER; SIZE: INTEGER)
                        return STD_LOGIC_VECTOR is
    VARIABLE result: STD_LOGIC_VECTOR(SIZE-1 DOWNTO 0);
    VARIABLE temp: integer;
    BEGIN
    temp := ARG;
    for i in 0 to SIZE-1 loop
          IF (temp mod 2) = 1 THENresult(i) := '1';
        else result(i) := '0';
   end IF;
       IF temp> 0 THEN temp := temp / 2;
        else temp := (temp - 1) / 2; -- simulate ASR
       end IF;
    end loop;
    return result;
    end;

    function CONV_INTEGER(ARG: STD_LOGIC_VECTOR) return INTEGER is
    VARIABLE result: INTEGER;
    BEGIN
    assert ARG'length <= 32
        report "ARG is too large in CONV_INTEGER"
        severity FAILURE;
    result := 0;
    for i in ARG'range loop
        IF i /= ARG'left THEN
        result := result * 2;
        IF tbl_BINARY(ARG(i)) = '1' THEN
            result := result + 1;
        end IF;
        end IF;
    end loop;
    return result;
    end;
end std_logic_util;
```

Любая декларация, содержащаяся в заголовке пакета, доступна любому проектному модулю, или, как говорят, становится видимой из этого модуля, если пакет предварительно скомпилирован в одну из библиотек системы проектирования, например рабочую библиотеку проекта `work`, а модуль содержит высказывание использования этого пакета (Use Clause). Декларация

использования пакета, приведенного в листинге 3.43, может выглядеть следующим образом:

```
USE work. Std_logic_util.all;
```

Концепция видимости объектов в VHDL

Рассмотрим в более широком аспекте принципы, заложенные в концепцию видимости языка.

Разделы деклараций могут присутствовать в различных программных блоках:

- в заголовке пакета;
- в теле пакета;
- в `ENTITY`;
- в архитектурном теле;
- в блоке или процессе;
- в теле подпрограммы.

Всякий проектный модуль может использовать объекты, объявленные в модулях высшей по отношению к этому модулю иерархии, используя их имена, если объект с тем же самым идентификатором не объявлен в этом модуле. Недопустимы ссылки "вниз" или на другие ветви иерархии.

Эти концепции иллюстрируются примером диаграммы видимости объявлений (рис. 3.20). Диаграмма представляет гипотетический проект с интерфейсом, заданным `ENTITY main`, которому подчинены два архитектурных тела `first` и `second`. Рассмотрим подробнее архитектурное тело `first`. В разделе операторов этого тела присутствуют два независимых блока `BLOCK1` и `BLOCK2`, причем в `BLOCK1` вложен еще один программный модуль `BLOCK3`. Соотношения видимости не изменятся, если на месте операторов блоков `BLOCK1` и `BLOCK2` записать операторы процесса.

Операторы в составе `BLOCK1` могут использовать объекты, определенные в разделах описаний этого блока, равно как в иерархически старших по отношению к нему программных конструкциях: разделе описаний и параметров настройки `ENTITY main`. Операторам блока `BLOCK3` доступны не только эти объекты, но и объекты, дополнительно определенные в разделе описаний `BLOCK3`. Однако описания, локализованные в блоке `BLOCK2`, не доступны остальным операторам блока `BLOCK1` и, соответственно, не разрешается обратная ссылка. Некоторые запрещенные варианты использования объявленных объектов отображены на рисунке стрелками, помеченными крестиками.

Декларации, содержащиеся в `ENTITY`, видимы для операторов всех подчиненных ему архитектурных тел независимо от уровня вложения. Объявления "внутри" архитектурного тела невидимы для других архитектурных тел, в том

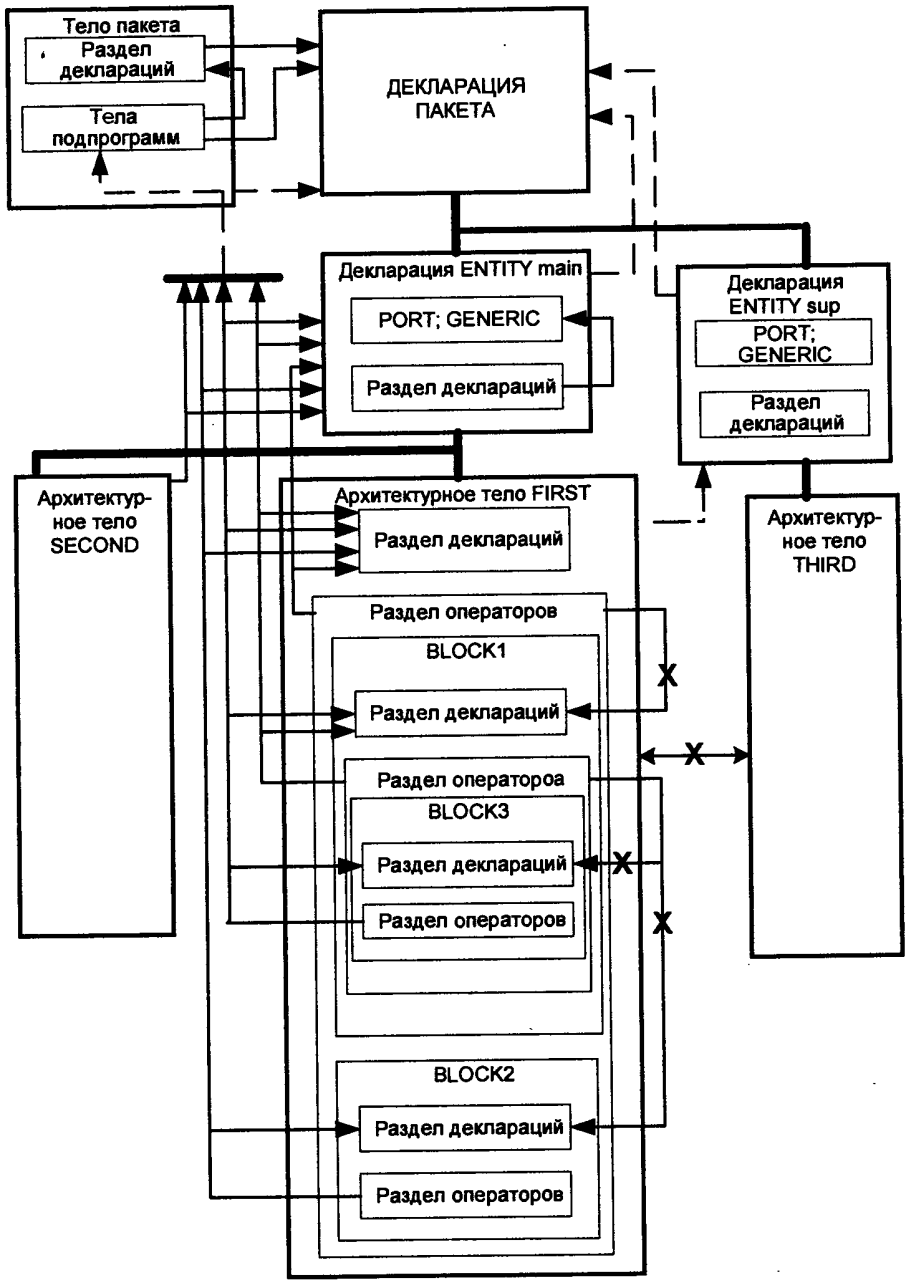

Рис. 3.20. Пример диаграммы видимости деклараций в совокупности программных модулей

числе относящихся к одному ENTITY. Декларации имен портов и параметров настройки, содержащиеся в ENTITY некоторого проекта, могут быть сделаны доступными для архитектурных тел, относящихся к ENTITY другого проекта, через объявление прототипа в подразделе декларации компонентов (пример допустимой ссылки отображен на рисунке штрихпунктирной линией).

Декларации, содержащиеся в пакете, доступны любому программному модулю, если его текст содержит высказывание использования этого пакета (Use Clause). В любом модуле, имеющемся в программе, содержащей такое высказывание, допустимо использовать объекты, объявления которых имеются в декларации пакета. В том числе, становятся доступны подпрограммы, содержащиеся в теле этого пакета. Такой принцип назовем обусловленной видимостью, а соответствующие ссылки на диаграмме видимости отражены штриховыми линиями. Операторы в телах подпрограмм могут использовать объекты, представленные разделе деклараций тела пакета, в декларации пакета, а также локальные объекты, объявленные в теле этой же подпрограммы.

3.3. Элементы языка Verilog HDL

3.3.1. Предварительные замечания

Язык Verilog HDL, сокращенно Verilog, был разработан фирмой Gateway Design Automaton как язык моделирования, ориентированный на внутреннее использование. Фирма Cadence приобрела Gateway в 1989 году и открыла Verilog для общественного пользования. В 1995 году был определен стандарт языка — Verilog LRM (Language Reference Manual), IEEE1364-1995 [55].

Несмотря на то, что VHDL был создан ранее и, в принципе, предоставляет более широкие возможности, Verilog HDL стал достаточно популярен среди разработчиков систем, и компиляторы с этого языка, наряду с VHDL-компиляторами, включены в подавляющее большинство САПР интегральных схем [51, 54]. Привлекательными свойствами языка Verilog можно считать близость к синтаксису и семантике языка С, меньшее по сравнению с VHDL число служебных слов, что упрощает его изучение. Кроме того, из-за своих расширенных возможностей VHDL проигрывает Verilog по эффективности, т. е. на описание одной и той же конструкции в Verilog потребуется в 3—4 раза меньше символов ASCII, чем в VHDL [13]. Вопрос, какое представление порождает лучшее техническое решение при реализации в аппаратуре средствами типовых САПР, остается дискуссионным. Далее часто будет приводиться сравнение конструкций языков Verilog и VHDL, предполагая, что читатель хотя бы кратко ознакомился с содержанием предыдущего раздела. Читателям, которые хотят изучить только Verilog, все же следует ознакомиться хотя бы с *разд. 3.2.1 и 3.2.4.*

Базовой единицей языка Verilog является проектный модуль, или просто модуль, интегрирующий, в отличие от соответствующих конструкций VHDL, как определение интерфейса, так и правил функционирования представляемого устройства или блока. Элементами проектного модуля являются декларации и операторы. Концепции языка позволяют описывать устройства с использованием различных уровней абстракции, в том числе в форме "чистого поведения", представления регистровых передач и вентильного уровня представления. Модуль может содержать операторы включения других модулей, что позволяет создавать иерархические проекты. Обеспечивается поддержка процессов как нисходящего, так и восходящего проектирования.

Поведенческое представление дискретных устройств задается в форме арифметических и логических преобразований над исходными и промежуточными данными. Состав и обозначения разрешенных операций соответствует составу операций языка C и его обозначениям. Обеспечивается возможность представления взаимодействующих подсистем, для чего в языке определены как параллельные, так и последовательные операторы и процедуры. Параллельные операторы отображают поведение цепей без памяти, а при моделировании исполняются при изменении любого операнда в правой части оператора. Порядок записи параллельных операторов не имеет значения.

Последовательные операторы заключаются в выделенные программные блоки и при моделировании выполняются последовательно друг за другом в порядке записи. Результаты преобразований доступны для других блоков программы только после выполнения всех вложенных в блок операторов. Для каждого такого блока могут быть явно определены функции блокировки или инициализации предписанных преобразований со стороны других блоков (сравните с понятием PROCESS языка VHDL). В число последовательных операторов, в частности, входят последовательные операторы присваивания, условный оператор IF, оператор выбора CASE, операторы повторения LOOP. Вводятся конструкции, которые представляют действия, исполняемые в течение некоторого ненулевого временного интервала.

Структурное представление проектов обеспечивается возможностью встраивания в проект других модулей, причем используются традиционные для алгоритмических языков конструкции, аналогичные вызову соответствующих подпрограмм. Кроме того, в языке предопределен широкий набор логических примитивов, примитивов для представления двунаправленных цепей передачи и резистивных цепей.

Замечание

В данной книге примитивы для представления двунаправленных цепей передачи и резистивных цепей не рассматриваются, т. к. эти конструкции очень редко используются при проектировании ПЛИС.

Упрощенно структура текстового файла в Verilog определяется следующим образом:

```
<исходный текст> ::=
    « <Директива компилятора>»
      <описание>  «<описание>»
<описание> ::= <описание модуля> | <описание примитива>
<описание модуля> ::=
    module <имя модуля> [ (<порт>  «,<порт>») ];
        « <Декларация>    |  <параллельный оператор>»
    endmodule
    | macromodule [ (<порт> «,<порт>») ];
    «<Декларация>    |  <параллельный оператор>»
endmodule
<порт> ::=
    <объявление соответствия порта>
    |  . <имя порта> ([ <объявление соответствия порта> ])
<декларация> ::=
    <декларация параметров>
    | <спецификация портов>
    | <декларация сигналов>
    | <декларация времени>
    | <декларация численных переменных>
    | <декларация событий>
    | <декларация логических ячеек>
<параллельные операторы> : ==
    <параллельное присваивание>
    | <оператор вхождения модуля>
    | <оператор блока>
    | <оператор инициализации>
    | <оператор постоянного повторения>
    | <вызов подпрограмм>
```

Из приведенных правил синтаксиса следует, что текст программы может содержать произвольный набор проектных модулей, каждый их которых представлен именем, объявлением портов и телом модуля.

Понятие макромодуля с точки зрения принципов описания функционирования не отличается от понятия модуля. Есть только незначительные ограничения по использованию некоторых конструкций. Разница, в основном, относится к организации процедуры моделирования иерархических проектов. Если "обычный" модуль компилируется таким образом, что при моделировании вложенные программные единицы (подпрограммы, операторы вхождения модулей) интерпретируются как вызовы соответствующих процедур, то компиляция макромодуля предусматривает прямое вложение операторов макромодуля в вызывающую программу. При этом ряд внутренних

переменных макромодуля могут просто исчезнуть, как говорят, стать ненаблюдаемыми, т. к. их определение войдет в другие выражения. Использование концепции макромодуля обеспечивает уменьшение времени моделирования.

В теле модуля параллельные операторы и декларации размещаются в произвольном порядке, хотя любой объект декларируется раньше его использования в операторах. Правила декларирования будут определены позднее. Сейчас отметим, что имена портов сначала объявляются в списке портов, затем вводится их спецификация, т. е. определяется направление — вход, выход или двунаправленная линия, а потом декларируется тип. Спецификация порта определяется следующими правилами:

```
<спецификация порта> ::=
    <направление> [ <диапазон индексов> ] <имя> «,<имя> »;
    <направление> ::=
        input | output | inout
```

В необязательном разделе <директивы компилятора> могут быть заданы некоторые параметры процедур компиляции и моделирования.

Наиболее употребительные директивы компилятора это `'timescale`, `'define`, `'include`, `'ifdef`, `'else`, `'endif`.

Директива `'timescale` определяет два параметра для управления процессом моделирования. Первый задает масштаб времени, т. е. физический временной интервал, соответствующий единице модельного времени в выражениях, задающих время останова и задержки (временные интервалы в программах задаются целым числом). Второй параметр задает точность представления временных интервалов при преобразовании в модели данных этого типа, например, при округлениях.

Так, директива

```
'timescale 1 ns/10 ps
```

определяет, что абстрактная единица в выражениях, задающих время, эквивалентна 1 наносекунде, а разрешение программы моделирования 10 пикосекунд. Впрочем, именно такое объявление не имеет смысла, т. к эти значения принимаются по умолчанию.

Директива `'define` задает имена константам в программном модуле (не путайте с параметрами). После такого объявления имя может использоваться в выражениях, но ему должен предшествовать апостроф. Например, если записана директива

```
'define  gate_delay=5     //  определение константы "задержка вентиля",
```

то в модуле допустимо выражение

```
y= # 'gate_delay  a & b  //  использование константы для задания
                         //  задержки схемы И
```


Отметим, что комментарии в Verilog выделяются так же, как в C:

- подстрока текста или целая строка, начинающаяся с двух знаков дроби до конца строки, является комментарием;
- любой фрагмент текста, начинающийся с пары знаков `/*` и заканчивающийся парой знаков `*/`, является комментарием.

Директива `'include` позволяет включить в создаваемый файл текст из другого файла. Директивы условной компиляции `'ifdef`, `'else`, `'endif` подобны аналогичным директивам C-компиляторов и используются, например, для выбора имен и констант в зависимости от параметров, определенных в проектах высшего уровня иерархии.

3.3.2. Типы данных

Основная форма представления информации, с которой оперирует Verilog HDL, — это сигнальные данные, отображающие состояние компонентов и цепей в описываемом устройстве. Кроме того, используются служебные данные, служащие для задания конфигурации, управления моделированием и компиляцией, а также отображения результатов в процессе моделирования. К служебным типам данных относятся время, действительные и целые числа, строки. Правда, числовые значения в некоторых случаях могут сопоставляться сигналам.

Язык Verilog не предоставляет возможности объявления пользователем собственных типов данных.

Цепи и регистры

Сигналы в Verilog HDL представляются в четырехзначном алфавите {0, 1, X, Z}. Однако для декларации линий связи, к которым подключаются источники, характеризующиеся специфическими электрическими параметрами, например проходные ключи, схемы с открытым коллектором и т. п., драйверам сигнала могут быть приписаны дополнительные атрибуты — уровни силы (Strength Level) — которые доопределяют способ вычисления истинного значения сигнала на линии.

Введено две группы типов данных для представления сигналов: цепи (`nets`) и регистры (`registers`). Каждой группе соответствуют специфические способы сохранения присвоенных значений и, в конечном счете, различные схемные решения для их реализации в аппаратуре.

Цепи представляют физические связи между структурными компонентами устройства. Сама по себе цепь не сохраняет состояние — она должна управляться драйвером, который соответствует оператору параллельного присваивания, причем имя драйвера совпадает с именем цепи. Это, собственно, и отражается в принятом в языке Verilog названии операторов этого типа — `"continious"`, т. е. непрерывные, постоянно воздействующие на цепь.

Регистр — это обобщенное понятие, отражающее элементы, способные сохранять состояние. Программная модель предусматривает сохранение состояния от каждого присвоения до следующего. Иными словами, оператор присваивания значения регистру действует как сигнал установки нового значения. В Verilog определен широкий набор конструкций для задания условий изменения состояний регистров и представления различных способов управления триггерными устройствами.

Синтаксис основных деклараций переменных задается следующими правилами:

```
<декларация цепи> ::=
    <тип цепи> [ signed [<право доступа>] <диапазон>] [<задержка>]
    <список переменных>;
        | <тип цепи> [ signed ]  [ <сила драйвера> ]
        [ [<право доступа>] <диапазон>] [ <задержка> ]
        <список присвоений>;
<декларация регистра> ::=
    reg [ signed ]  [ [<право доступа> ]  <диапазон> ]
    <список переменных>;
<список переменных> ::=
    <имя переменной> «,<имя переменной>»
<право доступа> ::=
    scalared  | vectored
<диапазон> :=
    [<константное выражение> : <константное выражение>]
<сила драйвера> ::=
    (<Уровень силы 0>, <Уровень силы 1>)
```

Тип цепи может задаваться из следующего множества значений:

```
<тип цепи> ::=
    wire | wand | wor | supply0 | supply1 | tri.| tri0.| tri1 |
    triand.| trior.| trireg
```

Замечание

В языке Verilog, в отличие от VHDL, не различаются понятия сигнала и переменной — все данные, которые могут изменяться, называют переменными (variables). Специфика выполнения операций назначения и сохранения состояния переменных определяется их типом (net или register), а в некоторых случаях способом записи оператора присваивания.

Типы цепей `wire` и `tri` с точки зрения программной интерпретации эквивалентны. Различные обозначения служат лишь для лучшей читаемости программы. Обычно тип `wire` используется для цепей, подключенных к единственному источнику, а тип `tri` — для представления связей, управляемых несколькими источниками (драйверами).

Прочие типы цепей используются для представления различных вариантов монтажной и коммутационной логики, которые не характерны для ПЛИС, и в данной книге не рассматриваются.

Если в декларации отсутствует объявление диапазона, то соответствующее имя присваивается скалярной переменной, представляющей состояние одновыходного логического элемента или одной линии соединений. В противном случае предполагается объявление битового вектора, т. е. кода, число разрядов которого и порядок их нумерации задается конструкцией <диапазон>. Право доступа определяет, возможен ли доступ к отдельным компонентам вектора, т. е. можно ли присваивать значения отдельным разрядам кода и использовать эти значения в других выражениях. Если перед указанием диапазона записано ключевое слово `vectored`, то доступ к отдельным компонентам вектора не допускается. Если присутствует ключевое слово `scalared` (а также по умолчанию), каждый компонент кода может устанавливаться и анализироваться индивидуально. Ключевое слово `signed` может сопровождать декларацию векторов и задавать правила преобразования битовых векторов в арифметических операциях. При его отсутствии представляемый вектор рассматривается как двоичное представление беззнакового целого (старший разряд значащий). Если слово `signed` присутствует, вектор рассматривается как представление целого в дополнительном двоичном коде (старший разряд знаковый).

Задержка определяет интервал модельного времени, после которого результат операции присваивания этой переменной фактически изменяет ее значение. Могут задаваться несколько значений задержки, проявляющихся в различных условиях *(см. разд. 3.3.5)*. Объявление переменной может содержать присвоение значения, причем в такой конструкции присвоение может содержать не только константные выражения, но и выражения, включающие другие переменные. Иными словами, декларация переменной может фактически совмещать функции объявления переменной и оператора присваивания.

Конструкция <сила драйвера> присутствует только в тех случаях, когда декларация переменной содержит присвоение, и если в тексте программы имеется несколько операторов присваивания значения этой переменной. Уровень силы задает условия взаимного подавления сигналов от нескольких источников, подключенных к общей линии. Индивидуально задаются уровни силы для состояния драйвера, соответствующего состоянию логического нуля (<уровень силы 0>) и логической единице (<уровень силы 1>). Подробнее об учете уровня силы в операторах присваивания см. в *разд. 3.3.5.*

Примеры:

```
reg a;                  // скалярная регистровая переменная
wire w1, w2;            // декларация двух цепей
wire #5 w3= w1 && w2; //  совмещение объявления цепи и присваивания:
            // w3 принимает значение логического И от значений w1 и w2
            // через 10 единиц модельного времени после изменения
            // любого из них
reg[3:0] v; //  4-разрядный векторный регистр,
            // включающий v[3], v[2], v[1]  и v[0],
            // причем v[3] — старший разряд, а v[0] — младший
tri [15:0] busa;              // 16-разрядная шина с тремя состояниями
reg signed [0:3] signed_reg; // 4-разрядный регистр, код в котором
                    // трактуется как число в диапазоне  от  -8 до  +7
```

Любой разряд группы, объявленной как `scalared`, может использоваться в выражении как скаляр, при этом используется имя группы и индекс, записываемый в квадратных скобках непосредственно за именем группы. Если в скобках указан диапазон, то в операции участвует подгруппа разрядов из указанного диапазона. Если же переменная, объявленная как группа, присутствует в выражении без указания диапазона индексов, то в операции участвуют все разряды группы.

Пример. Пусть определено

```
reg [15:0] regA, regb;
reg  scal;
reg [3:0] regC;
```

и выполнено присвоение

```
regA = 16 'b 1111 1011 1101 0000;
```

Тогда приведенные операторы дадут следующие результаты:

```
a= regA[5];      // a принимает значение 0
regB=regA;       // regB принимает значение 16'b 1111 1011 1101 0000;
regC=regA[7:3]; // regC принимает значение 4'b1101;
```

Правила записи векторных констант

В реальном устройстве векторные данные представлены двоичными кодами, а моделирующая программа трактует их как совокупность независимых элементов, каждый из которых может иметь четыре возможные состояния. Теоретически для представления элемента достаточно двух битов, но практическая программная интерпретация может быть различной. Для задания исходных значений векторных переменных и реализации вывода используются укороченные формы их представлений: в виде чисел. Используется десятичное, шестнадцатеричное, восьмеричное и двоичное представление.

Синтаксис представления числовых констант имеет вид:

```
:⌈<Разрядность представления>⌉⌈<базовый формат>⌉ <число>
<базовый формат> ::=
| 'd  // десятичный формат, цифры 0-9;
| 'h  // шестнадцатеричный формат, цифры 0-9, буквы a-f, x,z
| 'o  // восьмеричный формат, цифры 0-7, буквы x,z;
| 'b  // восьмеричный формат, цифры 0,1, буквы x,z;
```

При записи числовых констант прописные и строчные буквы эквивалентны.

По умолчанию принимается десятичный формат. Для всех форматов, кроме десятичного, используются не только традиционные символы, представляющие численный эквивалент кода соответствующей группы разрядов, но и символы высокоимпедансного состояния "z" и неопределенного состояния "x". Все разряды группы, специфицированной символами "z" и "x", принимают такое состояние.

Опция `<Разрядность представления>` точно определяет число *двоичных* разрядов, значения которых заданы соответствующей константой.

Примеры:

```
5 'd 3      // пятибитовый код в десятичном представлении,
            // эквивалентно  5 'b 00011.
12 'h 4x2   // двенадцатибитовый код в трехразрядном шестнадцатеричном
            // представлении, эквивалентно  12 'b 0100xxxx0010.
6 'o z1     // шестибитовый код в двухразрядном восьмеричном
            // представлении, эквивалентно  6 'b zzz001
```

Если значение константы ограниченной длины присвоено векторной переменной, имеющей больше разрядов, то старшие разряды этой переменной принимают нулевое значение, если крайний левый символ в записи константы имеет любое значение '0' или '1'. Если же крайний левый символ имеет значение x или z, то дополнительные старшие разряды приемника принимают такое же значение.

Память

Одномерный массив битов в Verilog определяется как вектор, т. е. переменная соответствующего типа с указанием границ индексов. Двумерный битовый массив представляется исключительно как пронумерованная совокупность битовых векторов и называется памятью (memory). Синтаксис декларации памяти имеет вид:

```
<Декларация памяти> ::=
    <декларация элемента памяти> <диапазон>;
<Декларация элемента памяти> ::=
     reg ⌈ <диапазон> ⌉ <имя>;
```

Примеры:

```
reg  [7:0] mem_data [255:0] // память на 256 восьмиразрядных слов
reg bit_slice [15:0]        // память на 16 одноразрядных слов
```

Замечание

Не следует путать объявление одноразрядной памяти и кода с соответствующим числом разрядов. Например, объект `word`, объявленный как `"reg [15:0] word;"` вовсе не эквивалентен объекту `bit_slice`, определенному выше. Для переменной `word`, определенной таким образом, мы можем записать присвоение

```
word=0;
```

но присвоение `bit_slice=0` недопустимо.

Многомерные массивы в языке не определены.

Целые и действительные типы данных. Время

Целые переменные могут быть использованы для программного вычисления конфигурационных параметров проекта, но чаще применяются в том же смысле, что и регистры: чтобы сделать более наглядными фрагменты программ, представляющих арифметические преобразования. Если целый тип присвоен сигнальной переменной, то предполагается, что она представлена в реализации тридцатидвухразрядным кодом, и ее поведение подобно данным регистрового типа. Существенное отличие целых данных от регистровых — невозможность присвоения им неопределенных значений. Кроме того, код, сохраняемый регистровой переменной, интерпретируется как положительное число (старший разряд знаковый), целые же представлены дополнительным двоичным кодом.

Данные действительного типа также имеют свойства, подобные регистровым данным, но в языке вводится ряд ограничений на набор допустимых операций над ними *(см. разд. 3.3.3)*. Фактическое внутреннее представление определяется используемым для их интерпретации компьютером. Исходные данные и результаты обработки данных действительного типа представляются обычно в десятичном естественном или экспоненциальном формате.

Время в Verilog — это специфический носитель информации, имеющий целочисленное значение. Над данными типа "время" можно выполнять любые операции, присваивать их значение регистровым данным и наоборот. Текущий отсчет модельного времени сохраняется в специальном регистре времени, и его значение может присваиваться переменной типа "время" с использованием системной функции `$time`, например, следующим образом:

```
time current_time_mark, next_event_time;  // объявление переменных типа
                                          // "время"
```



```
current_time_mark=$time;                  // присваивание переменной
                                          // значения текущего времени
next_event_time=current_time_mark+5;      // вычисление времени
                                          // будущего события
```

Время задается в безразмерных единицах, называемых квантами модельного времени. Фактический масштаб времени устанавливается директивой компилятора `'timescale`.

Строки

Строковые данные чаще всего используются для организации выдачи сообщений о возникновении тех или иных ситуаций в процессе моделирования, которые предусматривает разработчик. Строковая константа записывается как произвольная последовательность символов, заключенная в двойные кавычки. Кроме стандартных символов клавиатуры применяются специальные конструкции управления выводом на дисплей, повторяющие соответствующие конструкции языка C: `"\n"` — перевод строки, `"\t"` — табуляция и т. д.

Если строка при исполнении программы может подвергаться модификации, то вводится переменная регистрового типа, конкретно — битовый вектор, число элементов которого достаточно для сохранения любого возможного значения строковой переменной. Каждый символ представлен восьмибитовым кодом ASCII. Например, для представления текста `"Simulation started"` необходимо 8×17 разрядов, т. е. регистр на 136 бит. Тогда допустимы следующие объявления и операторы:

```
reg [8*17:1] stringvar;
initial
  begin stringvar= "Simulation started";
    $display(" \n Seance of %S\n", stringvar);
  end;
```

Системная функция `$display()` вызывает отображение строки на дисплее. Над строковыми переменными можно производить такие же действия, как и над любыми другими данными, при этом каждая позиция переменной трактуется как один разряд кода ASCII. Кроме того, отметим специфическую строковую операцию конкатенации, объединяющую подстроки. Операция представляется как запись в порядке вхождения подстрок, формирующих результирующую строку. Подстроки разделяются запятыми, а вся конструкция заключается в фигурные скобки. Например, предыдущий программный фрагмент можно записать и так:

```
reg [8*17:1] stringvar, [8*(17+6):1] result_string;
  initial
```

```
begin stringvar= "Simulation started";
    result_string=("Seance of", stringvar);
    $display(" %S", result_string);
end;
```

Параметры

Параметрами (parameter) в Verilog называют константные значения, используемые в программах через объявленное имя. Синтаксис декларации параметров определен следующим образом:

<декларация параметров> ::=

```
parameter <имя>=<константное выражение>
«,<имя>=<константное выражение> »;
```

Программный модуль может содержать сколько угодно деклараций параметров. Тип параметра совпадает с типом константного выражения.

Примеры:

```
parameter x=25, f=9;    // Два параметра (константы) целого типа;
parameter  pi=3.14157; // Константа действительного типа;
parameter word_size=8, last_bit_number= word_size-1;
                        // значение Last_bit_number
                        // задано арифметическим выражением,
                        // включающим ранее определенный параметр;
parameter delay=100;    // время задержки, выраженное в единицах
                        // модельного времени.
```

Хотя значения параметров являются константами внутри модуля, в котором они определены, при включении в иерархические проекты им может быть присвоено значение, отличное от предопределенного в декларации.

3.3.3. Операции и выражения

Выражение — это конструкция, которая объединяет операнды и знаки операции для формирования результата. В языке Verilog в качестве операнда могут использоваться числа, имена переменных (цепей, регистров, целых и временных переменных), выделенные биты и группы битов регистров и цепей, а также вызовы функций, которые возвращают значения любого из перечисленных типов.

За редкими исключениями данные разных типов совместимы в одном выражении, что отличает язык Verilog от VHDL. Даже если операнды имеют разную разрядность, их можно объединять в одном выражении, при этом по умолчанию используется расширение разрядности более короткого операнда. С точки зрения использования в выражениях данные целого типа и данные времени характеризуются теми же правилами, что и переменные регистрового типа.

Символы операций совпадают с принятыми в языке C. Их список приведен в табл. 3.7. В то же время интерпретация операций в Verilog имеет некоторые особенности.

Таблица 3.7. Символы операций Verilog

Группа операций	Символы операций	Наименование	Применимость к данным типа real
Конкатенация	{ }		Нет
Арифметические операции	+, –	Унарные арифметические	Да
	+, –, *, /	Бинарные арифметические	Да
	%	Модуль числа	Нет
Операции сдвига	>>, <<	Сдвиг кода вправо или влево на заданное число разрядов	Нет
Операции отношения	>, >=	Больше, больше или равно	Да
	<, <=	Меньше, меньше или равно	Да
Операции сравнения	==, !=	Равно — не равно (особенности вычисления см. далее)	Да
	===, !==		Нет
Операция свертки	&, ~&	Свертка по И или И-НЕ	Нет
	|, ~|	Свертка по ИЛИ или ИЛИ-НЕ	Нет
	^	Свертка по "исключающему ИЛИ"	Нет
	^~ или ~^	Свертка по "исключающему ИЛИ-НЕ"	Нет
Поразрядные операции	~	Инверсия	Нет
	&, ~&	Поразрядное И (И-НЕ)	Нет
	|, ~|	Поразрядное ИЛИ (ИЛИ-НЕ)	Нет
	^	Поразрядное "исключающее ИЛИ"	Нет
	^~ или ~^	Поразрядное "исключающее ИЛИ-НЕ"	Нет
Логические операции	!	Логическая инверсия	Да
	&&	Логическое И	Да
	||	Логическое ИЛИ	Да
Условная операция	?	Если <условие>?, то <значение 1> : иначе <значение 2>	

В отсутствии скобок операции групп, приведенных в таблице выше, имеют больший приоритет, т. е. являются старшими по сравнению с операциями групп, стоящих ниже их в таблице. Внутри каждой группы используются традиционные правила старшинства операций.

Рассмотрим особенности записи и интерпретации некоторых операций в языке Verilog.

Арифметические операции

Набор арифметических операций и правила старшинства достаточно традиционны для алгоритмических языков. Однако следует отметить, что результаты преобразования данных регистрового типа отличаются от результатов выполнения операций над данными целого типа. Данные регистрового типа рассматриваются как целое число в двоичном беззнаковом представлении. То есть, если мы присваиваем регистровой переменной константное значение

```
- N <базовый формат> X
```

то эта переменная будет представлена N-разрядным дополнительным кодом числа -X. Пусть, например, определено:

```
integer intA;
reg [15:0] regA;
```

Тогда

```
regA= -4'd12 // присвоен код "1111 1111 1111 0110",
             // являющийся дополнением до двух кода числа +12, двоичный
             // эквивалент которого равен "0000 0000 0000 1010"
intA= regA/3 // присвоено значение 21841, потому что код, находящийся
             // в regA, представляет число 65526
```

Операции отношения и сравнения

Результат операции отношения — это скалярная величина, принимающая значение 1, т. е. "истинно", если элементы справа и слева от символа операции удовлетворяют знаку операции в арифметическом смысле, и 0, т. е. "ложно", если не удовлетворяют. При этом учитываются отмеченные выше особенности представления данных, т. е. для регистровых данных большим считается код, у которого единица при просмотре слева направо появляется раньше. Если в коде любого из операндов присутствует хоть один разряд со значением 'x' или 'z', результат принимает значение 'x'. Подобно исполняются операции сравнения "==" и "!=". Несколько иначе формируется результат в операциях "===" и "!==". Здесь, если в сравниваемых кодах в одинаковых позициях находятся одинаковые значения, в том числе символы

неопределенности, то коды считаются равными, а при любых отличиях неравными. Результат операций "===" и "!==" может быть только 1 или 0.

Операции сдвига

В операциях сдвига операнд, находящийся слева от знака операции, сдвигается влево или вправо на число разрядов, записанное справа. Освободившиеся разряды заполняются нулями.

Логические и поразрядные операции

Результатом логической операции ИЛИ, И и инверсии, записываемых как ||, && и !, может быть единица, отображающая значение "истинно", нуль для представления значения "ложно" или значение 'x', что значит "неопределенно". В операции ИЛИ неопределенное значение поглощает нуль, но само поглощается единицей. В операции И неопределенное значение поглощает единицу, но само поглощается нулем. В практике чаще всего логические операции используются для скалярных операндов, в частности для объединения в одном условии результатов операций отношения и сравнения.

При применении к целым переменным результат операции && дает 0, только если один из операндов имеет значение 0, а в применении к векторным — если все разряды одного из операндов нулевые. Наоборот, операция || дает значение 1, если хоть один из операндов имеет определенное ненулевое значение. Унарная логическая инверсия истинна, если код операнда не нулевой.

В отличие от логических операций, поразрядные операции выполняются независимо над парами одноименных разрядов операндов. Эти частичные результаты собираются в том же порядке, что и в операндах, формируя результирующий код. Если операнды имеют различную разрядность, то более короткий операнд дополняется нулями со стороны старших разрядов. Длина кода результата равна длине большего операнда.

Операции свертки

Операции свертки имеют то же обозначение, что и поразрядные, но являются унарными и дают скалярный результат. Все разряды операнда рассматриваются как скалярные аргументы соответствующей многоместной операции. Например, при выполнении свертки по И разряды аргумента анализируются последовательно друг за другом, причем каждый раз выполняется операция И над значением нового разряда и результатом анализа предыдущего. Анализ может быть завершен, если на каком-то этапе получен нулевой результат. Фактически свертка по И даст нуль, если в коде операнда содержится хоть один нуль, а результат будут равен единице, если все разряды установлены в единицу. Если же в каком-либо разряде присутствует не-

определенное значение или символ высокоимпедансного состояния 'z' и нет разрядов, установленных в нуль, результат принимает неопределенное значение 'x'. Подобные рассуждения можно привести для остальных операций свертки.

Необходимо обращать внимание на правильное расположение пробелов в записях операций свертки, которое отличает их от логических операций. Так запись A && B представляет логическое преобразование "A, объединенное по логике И с B", а A & &B следует трактовать как "A, объединенное по логике И с результатом свертки B", т. е. A & (&B). Отметим, что если A — код, то аргумент &B будет дополняться нулями со стороны старших разрядов.

Условная операция

Условная операция является единственной тернарной операцией, синтаксис которой определяется следующим образом:

<условное выражение> ? <выражение 1> :<выражение2>

Если при вычислении условного выражения получено значение "истинно", то в качестве результата используется <выражение1>, а если "ложно", то <выражение2>. Если же получено значение "неопределенно", то выполняется поразрядная операция над результатами вычисления <выражения1> и <выражения2>. При этом разряду результата приписывается значение 0, если при вычислении обоих выражений в соответствующих разрядах получен 0, значение 1, если при вычислении обоих выражений в соответствующих разрядах получена 1, и неопределенное значение во всех остальных случаях.

3.3.4. Операторы *initial* и *always*

Как и в VHDL, в языке Verilog различают два вида операторов — параллельные (по терминологии разработчиков языка "continuous" — непрерывные) и последовательные (названные разработчиками языка "procedural" — процедурные). Последовательные операторы выполняются друг за другом в порядке записи, параллельные же при любом изменении сигнала, используемого в качестве аргумента. Последовательные операторы должны быть локализованы в теле составных операторов. Поэтому рассмотрение способов описания поведения устройств начнем с операторов этого типа.

Оператор инициализации initial запускается к исполнению единственный раз при начале моделирования и используется либо для задания начальных состояний в моделируемом устройстве, либо при записи программ моделирования (Test-Bench) для описания сигналов, поведение которых предопределено сценарием отладки.

Синтаксис оператора инициализации определен следующим образом:

<оператор инициализации> ::= initial <оператор>

Может создаться впечатление, что оператор инициализации задает единственное действие. Однако это не так. Вложенный оператор может быть составным. Не вдаваясь пока в детали определения составных операторов и блоков, отметим, что составной оператор представляет последовательность операторов, выполняемых друг за другом в порядке записи, ограниченную ключевыми словами begin и end. После исполнения всех действий, предписанных вложенными операторами, оператор "зависает" и не влияет на исполнение остальных программных конструкций.

Рассмотрим программу (листинг 3.44), представляющую тест для проверки некоторого устройства, в данном примере представленного оператором параллельного присваивания.

Листинг 3.44

```
module top;
reg [2:0] a;
wire [4:0] b;
assign #5 b ={c,&a,a};// параллельный оператор, изменение a и c вызывает
                      // его исполнение через 5 единиц модельного времени

initial c=1'b1;   // нет необходимости в конструкции begin-end, потому
                  // что вложен единственный оператор
initial
begin
a = 'b000;        // начальное значение
#100 a = 'b110;   // каждый оператор выполняется через 100 ед. модельного
                  // времени после исполнения предыдущего
#100 a = 'b111;
#100 a = 'b110;
end
initial
begin             // запуск системной функции отображения результатов
$monitor($time,, "a=%b, b=%b",a, b);
#1000 $finish;
end
endmodule
```

Операторы инициализации начинают исполняться "одновременно" (в том смысле, что их результаты недоступны для операторов, также имеющих нулевую отметку времени).

В результате моделирования будет получено:

```
0 a=000, b=xxxx
5 a=0000, b=10000
100 a=110, b=10000
```

```
105 a=110, b=10110
200 a=111, b=10110
205 a=111, b=11111
300 a=110, b=11111
305 a=110, b=10110
1000    $finish
```

Оператор постоянного повторения always также первый раз исполняется при начале моделирования, но затем повторяется каждый раз после завершения вложенного оператора. Синтаксис оператора имеет вид:

<оператор постоянного повторения> ::= always <оператор>

Если вложенный оператор составной, то действия повторяются после исполнения последнего оператора из числа вложенных в него. В связи с циклическим характером оператора always, он может иметь смысл только при наличии в его теле конструкций, предусматривающих его переход в состояние ожидания. Это может быть ожидание заданных моментов времени или иных событий в системе. Если не предусмотреть переход оператора в состояние ожидания, все другие операторы в программе будут заблокированы.

Например, конструкция

```
always a= ~a;
```

порождает бесконечный цикл с нулевой задержкой, в то время как

```
always #delay a= ~a;
```

представляет генератор импульсов единичной скважности и периодом 2*delay, причем в интервале ожидания могут исполняться другие операторы, например, вызываемые изменением переменной a.

Наиболее часто оператор always используется в сочетании с операторами, содержащими указание событий, инициирующих его исполнение *(см. разд. 3.3.8)*. В этом случае после исполнения действия, предписанного вложенным оператором, процесс приостанавливается до возникновения соответствующего события. Из изложенного видно, что оператор always подобен процессу в VHDL, содержащему список чувствительности или вложенный оператор ожидания.

3.3.5. Операция присваивания, операторы присваивания

Операция присваивания, или просто *присвоение*, может включаться как составной элемент деклараций переменных и параметров, входить в операторы параллельного присваивания, а иногда сама по себе трактоваться как законченный оператор. При выполнении присваивания приемник, который записывается слева от знака равенства, являющегося символом операции при-

сваивания, принимает значение, получаемое при вычислении выражения, записанного слева от знака равенства. Приемником может быть простая переменная, элемент вектора, группа разрядов вектора, а также объединение объектов указанных типов. Если приемник — совокупность объектов, то ее элементы записываются в фигурных скобках через запятые. Эти правила можно свести к следующим BNF-формулам:

```
<приемник> ::=
    <элемент приемника>
  | {<элемент приемника> «,<элемент приемника>» }
<элемент приемника> ::=
  <имя> [ [<индексное выражение> [ : <индексное выражение> ] ] ]
```

Применение правил к приемнику, содержащему единственный скалярный элемент, представляется очевидным. В случае группового приемника разряды результата присваиваются разрядам элементов приемника в порядке их записи в списке элементов приемника. Например, если определено

```
wire  [6:0] regA= 5'b0010010;
wire  [1:0] regB= 3'b1x;
wire c =1'bz;
reg [3:0] d,e;
```

то в результате присваивания

```
{d,e}= {c, regB, regA[4:0] }
```

получим d=4'bz1x1, e=4'b0010.

В Verilog определены два основных типа операторов присваивания — continuous, т. е. непрерывные операторы, и процедурные — procedural. Непрерывные присваивания всегда рассматриваются как параллельные, т. е. исполняются при изменении любой переменной, присутствующей в правой части операции присваивания. Процедурные выполняются только при определенных условиях, задаваемых специальными конструкциями. Они могут быть последовательными, т. е. выполняться друг за другом в порядке записи, если локализованы в так называемых последовательных блоках, или параллельными, если включены в параллельные блоки.

Непрерывное присваивание

Приемником в операторе непрерывного присваивания может быть только переменная типа "цепь", скалярная или векторная. Синтаксис оператора параллельного присваивания имеет вид:

```
<оператор параллельного присваивания> ::=
  assign [ <сила драйвера> ][ <задержка> ] <присваивание> «,<присваивание>»;
```

```
<задержка> ::=
  # <параметр задержки>
  | # ( <параметр задержки>, <параметр задержки>
    [,<параметр задержки> ] )
<параметр задержки> ::=
<Выражение времени>
| <Выражение времени> : <Выражение времени> : <Выражение времени>
```

Оператор исполняется при изменении любой переменной, присутствующей в правой части операции присваивания. Однако после вычисления результат заносится только в буфер (драйвер), а временная отметка предсказанного изменения заносится в календарь событий. Приемник получает новое значение только тогда, когда начнется отработка этого вновь предсказанного события, т. е. через интервал модельного времени, заданный выражением <задержка>. После этого результат сохраняется вплоть до следующего исполнения оператора, т. е. изменения любого аргумента.

Если опция <задержка> отсутствует, предполагается дельта-задержка, т. е. новое событие заносится в календарь событий с той же отметкой времени, что и инициирующее событие, но после всех событий, которые уже занесены в календарь и имеют такую же отметку времени. Таким образом, пока не исполнены все операторы, инициированные общим событием, аргументами преобразований являются состояния сигналов на момент начала отработки этого события.

Задержка может быть задана одним, двумя или тремя параметрами. Рассмотрим сначала константные параметры задержки для скалярных данных.

Если параметр единственный, то он задает значение модельного времени задержки для любых переходов. Если присутствует второй параметр, то первый определяет время задержки для перехода из нулевого состояния в единичное, а второй — из единичного в нулевое. Третий параметр, если он есть, специфицирует время задержки перехода в z-состояние. Например, рис. 3.21 отображает результат моделирования программы, представленной в листинге 3.45.

Листинг 3.45

```
module  delay_example;
 reg s1,s2,v;
 wire result;
   assign # ( 10,5,15)
                result=(s1==1) : 1'b1 : 1'bz,
                result=(s2==1) :  v : 1'bz;
   initial  s1=0; s2=0;v=0;
              #10 s1=1;
              #20 s1=0;
              #25 s2=1;
              #15 v=1;
               #10 v=0;
              #10 s2=0;
              #20  $finish;
endmodule
```


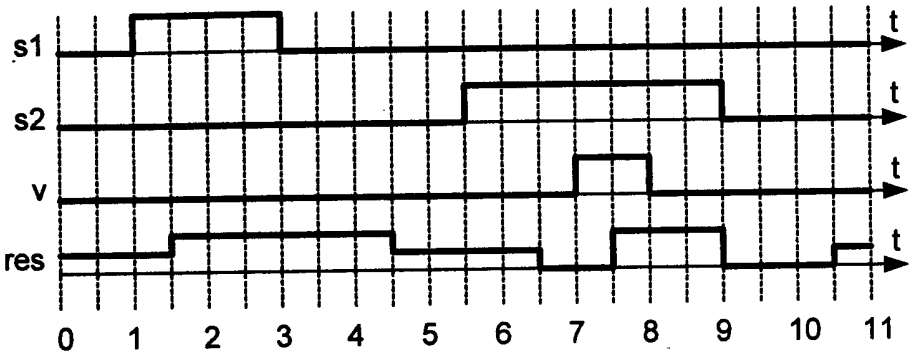

Рис. 3.21. Определение задержек в операторах параллельного присваивания

Некоторая специфика определена для присваивания значений векторам. В этом случае:

- если при вычислении выражения в правой части присвоения младший значащий бит (LSB) переходит в состояние единицы или сохраняется в состоянии единицы, то используется значение задержки, записанное в списке первым;
- если при вычислении выражения в правой части присвоения LSB переходит в состояние нуля или сохраняется в состоянии нуля, то используется второе значение, записанное в списке на второй позиции;
- если при вычислении выражения в правой части присвоения LSB переходит в состояние z или сохраняет состояние z, то используется третье значение;
- если при вычислении выражения в правой части присвоения LSB находился в неопределенном состоянии, или новое значение неопределенное, то используется наименьшее из значений задержки из списка.

В случаях, когда разработчика интересует время задержки для отдельных разрядов индивидуально, следует записывать не групповое присвоение, а индивидуальные присвоения для разрядов.

Эти правила иллюстрируются программой LSB_as_selector распечаткой результатов исполнения, приведенными в листинге 3.46.

Листинг 3.46

```
module LSB_as_selector;
reg [2:0] a;
wire [2:0] b,c;
assign #(10,20) b = a;
                c[0]=a[0];
                c[1]=a[1];
                c[2]=a[2];
initial
begin
a = 'b0000;
#100 a = 'b010;
#100 a = 'b101;
#100 a = 'b111;
end
initial
begin
$monitor($time,, "a=%b, b=%b",a, b);
#1000 $finish;
end
endmodule

Compiling source file     // распечатка результатов
Highest level modules:
LSB_as_selector
0 a=000, b=xxx, b=xxx
10 a=000, b=0000, c=0000  // предыдущее состояние было x, использовано
                          // минимальное значение
100 a=010, b=000, c=000
110 a=010, b=000, c=010   // c[1] перешло в 1 с минимальной задержкой
120 a=010, b=010, c=010   // b[0]сохранил 0, для b используется время
                          // перехода в нулевое состояние
200 a=101, b=000, c=010
 210 a=101, b=101, c=111  // b[0] перешел в 1, для b и c[0] использовано
                          // значение задержки для положительного фронта
220 a=101, b=101, c=101   // для c[1] задержка максимальна
300 a=111, b=101, c=101
 310 a=111, b=111, c=111  // b[1]  сохранил  единицу, для b  и c[1]
                          // использовано меньшее значение задержки
```

Параметр задержки может быть не только числовой константой, но и именем параметра или переменной и даже вычисляемым выражением. И наконец, задержка может быть объявлена тремя параметрами, представляющими, соответственно, минимальное, среднее и максимальное время задержки,

причем набор таких значений может быть задан индивидуально для каждого вида перехода.

Язык Verilog позволяет описывать устройства, в которых несколько источников работают на одну линию, при этом каждому источнику сопоставляется собственный оператор непрерывного присваивания переменной, представляющей сигнал на этой линии. Если при этом лишь один из таких источников может быть в активном состоянии, то ситуация описывается относительно просто: если хоть один источник находится в активном состоянии, переданное им значение, как более "сильное", по определению подавляет "слабые" сигналы, представляющие источники, находящиеся в отключенном состоянии (z-состоянии). Однако в практике возникают и иные ситуации. Во-первых, может быть необходимо обнаружение при моделировании сбойных ситуаций, связанных с некорректной активизацией нескольких драйверов, возможно подключение к шинам схем с открытым коллектором и подобных компонентов, выходное сопротивление которых, а значит влияние на результат, будет неодинаково в различных условиях. *Конструкция, определяющая степень такого влияния, называется "сила драйвера" и может включаться в оператор параллельного присваивания.*

Сила драйвера задается парой значений уровня, выбранных обязательно из разных столбцов приведенного списка:

```
supply1    supply0
strong1    strong0
pull1      pull0
weak1      weak0
highz1     highz0
```

Значения, заканчивающиеся символом 1, определяют уровень силы драйвера в состоянии логической единицы, а заканчивающиеся символом 0 — в состоянии логического нуля. Порядок записи безразличен. Если на сигнал действует несколько драйверов, имеющих различный уровень силы, то сигналу присваивается значение того, кто в данный момент представлен наиболее сильным сигналом (значения в приведенном списке упорядочены, причем выше в списке представлены более сильные значения). Если подключены равносильные драйверы, сигналу присваивается неопределенное значение. Например, моделирование фрагмента, представленного в листинге 3.47, породит временную диаграмму, отображенную на рис. 3.22.

Листинг 3.47

```
Wire a;
Reg b,c;
Assign  (strong0,weak1) a=b;
Assign  (strong1,weak0) a=c;
```

```
Initial  c=0; b=0;
         #10 c=1;
         #10 b=1;
         #10 c=0;
         #10 c=1;
         #10 b=0;
```

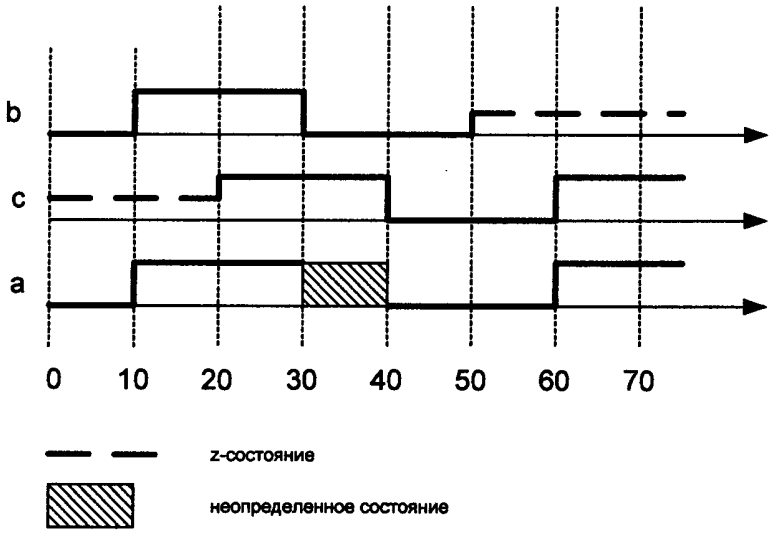

Рис. 3.22. Влияние уровня силы драйвера на значение сигнала

Последовательные присваивания

Процедурные последовательные операторы присваивания, которые мы будем называть просто последовательными присваиваниями, потому что последовательными могут быть только процедурные присваивания, локализуются в так называемых последовательных блоках. Подробнее концепция блока будет рассмотрена далее, пока же определим, что *последовательный блок — это последовательность операторов, заключенная между парой ключевых слов* `begin` *и* `end`. Последовательные операторы выполняются друг за другом в порядке записи, а приемником в них может быть только переменная регистрового типа. Но с точки зрения доступности результатов присвоения для последующих операторов имеются модификации. Язык Verilog не различает, в отличие от VHDL, категорий сигналов и переменных. *Но для отображения причинно-следственных связей наряду с параллельными операторами вводятся специфические, подобные VHDL, механизмы управления доступностью, задаваемые формой записи оператора: блокирующие и неблокирующие операторы присваивания, и соответствующая символика.*

Блокирующее присваивание запрещает исполнение других присваиваний до своего завершения. Это гарантирует строго последовательное исполнение операторов в блоке, даже если в записи результата присутствует опция задержки. Кроме того, такое определение делает результат присвоения непосредственно доступным любому последующему оператору в текущем программном блоке. Интересно, что если в блоке ни один оператор не содержит признаков приостанова исполнения, то с точки зрения окружения все операторы в блоке происходят как бы одновременно, и никакие промежуточные преобразования "невидимы" для других блоков.

Если присвоение содержит опцию задержки, то изменение предсказывается на момент модельного времени, отстоящий от момента исполнения оператора на объявленное число квантов модельного времени. Иными словами, если имеем несколько последовательных операторов со своими указаниями времени, то время задержки очередного присвоения от начала исполнения первого есть сумма задержек предшественников. Примеры блокирующих присвоений и соответствующие комментарии можно найти в листингах 3.44—3.46.

Неблокирующее присваивание отображается сочетанием символом <= и разрешает исполнение последующих операторов еще до собственного завершения.

Более того, если совокупность неблокирующих операторов инициирована общим событием, то эти операторы начинают исполняться одновременно с точки зрения правил, определенных дискретной событийной моделью, т. е. все они используют значения операндов, определенные *до* начала исполнения всей совокупности. Новые значения запоминаются в буфере до исполнения всех операторов, вызванных общим событием.

Неблокирующие присваивания предоставляют путь описания регистровых устройств с обратными связями. Рассмотрим следующий фрагмент:

```
always #10
      begin
            a_blocked=input;
            b_blocked=a_blocked;
            a_non_blocked<= input;
            b_non_blocked<= a_non_blocked;
      end
```

Каждые 10 единиц модельного времени последовательно выполняются все операторы присваивания, причем отметка времени для всех изменений одинакова. Но поведение переменных `b_blocked` и `b_non_blocked` существенно различно. Переменные `a_blocked` и `b_blocked` примут одинаковое значение `input`. В то же время присваивание переменной `b_non_blocked` выполняется параллельно с присваиванием `a_non_blocked`. То есть пара переменных `a_non_blocked` и `b_non_blocked` моделирует линию задержки на время 10 единиц модельного времени.

Можно видеть, что *блокирующее присваивание эквивалентно присваиванию значения переменной в VHDL, а неблокирующее — последовательному сигнальному присваиванию.*

3.3.6. Операторы принятия решений

Оператор условия и оператор варианта позволяют выбрать один из возможных путей исполнения алгоритма в зависимости от текущих условий. С точки зрения их интерпретации в процессе моделирования они относятся к классу последовательных (процедурных) операторов и исполняются вслед за оператором, предшествующим им в программном блоке. При реализации в аппаратуре воспроизводятся блоки, выполняющие все описываемые альтернативы, из которых либо инициализируется в каждый момент только один блок, либо выходные сигналы считываются только с одного из блоков.

```
<оператор условия> ::=
    if ( <выражение>) <оператор>
       [else <оператор>];
```

Если вычисление выражения дает ненулевой результат, выполняется первый оператор. Если выражение дает нуль, то в сокращенной версии (без ключевого слова `else`), не выполняется никаких действий, а в полной версии — второй оператор. Важно отметить, что в Verilog отсутствует логический или булевский тип данных, а в качестве выражения, определяющего выбор, можно использовать любые выражения, дающие целый результат. При этом оператор

```
if (expression)...
```

имеет тот же смысл, что оператор

```
if (expression !=0)...
```

Логика исполнения оператора в целом подобна исполнению условного оператора в языке C, но имеется важная особенность, связанная с тем, что в Verilog вычисление выражения может дать неопределенное значение. В этом случае исполняется ветвь `else`. Здесь уместно напомнить, что операции сравнения определены в двух версиях. Если используется обозначение "==" (два знака равенства), то результат сравнения считается неопределенным, если хоть один разряд операндов не определен. Если используется обозначение "===" (три знака равенства), то операнды считаются совпадающими и в тех случаях, когда у них в одноименных разрядах присутствуют неопределенные значения.

Замечание

Ни в коем случае нельзя в операции сравнения использовать обозначение "=" (один знак равенства), ибо один знак равенства представляет присваивание.

На месте первого оператора может использоваться пустой оператор, отображаемый знаком "точка с запятой". В качестве оператора может использоваться составной оператор, т. е. последовательность операторов, ограниченная операторными скобками `begin` и `end`, а также другой условный оператор. Конструкция с многовариантным выбором может выглядеть, например, следующим образом:

```
if (<выражение1>)
   if (<выражение2>) <оператор1>
   else              <оператор2>
else  if (<выражение 3>) <оператор3>
      else               <оператор4>
```

В этом примере `<оператор1>` выполняется при ненулевых результатах вычисления `<выражения1>` и `<выражения2>`, `<оператор3>` выполняется при нулевых результатах вычисления `<выражения1>` и ненулевых для `<выражения3>` и т. п.

При большом количестве альтернатив и при возможности свести признаки выбора в единственную переменную более компактное описание процедуры принятия решений дает оператор варианта.

```
<оператор варианта> ::=
    <определитель оператора варианта>( <ключевое выражение>)
      « <Вариант>  «,<Вариант> »  : <оператор>»
    endcase
<определитель  оператора варианта> ::= case| casez | casex
<вариант> :=  <константное выражение>  | default
```

Трактовка версии оператора выбора с ключевым словом `case` достаточно традиционна: исполняется лишь тот оператор, значение варианта которого совпадает со значением ключевого выражения. Если ни один из вариантов не совпадает с вычисленным значением ключевого выражения, то выполняется ветвь `default` или не выполняется ни один оператор. *В отличие от C и подобно VHDL, выполняется только один вариант.* Например, следующий оператор описывает дешифратор с трехразрядным адресным входом, входом разрешения и восемью выходами при представлении выбранного выхода низким уровнем сигнала.

```
case (en,in_code)
4'd8: result = 8'b011111111;
4'd9: result = 8'b101111111;
4'd10: result = 8'b11011111;
4'd11: result = 8'b11101111;
4'd12: result = 8'b11110111;
4'd13: result = 8'b11111011;
4'd14: result = 8'b11111101;
4'd15: result = 8'b11111110;
```

```
4'd0, 4'd1, 4'd2, 4'd3, 4'd4, 4'd5, 4'd6, 4'd7: result = 8'b11111111;
default result = 8'bx;
endcase
```

Отметим, что в версии с ключевым словом case при сравнении учитываются все разряды, в том числе должны совпадать значения разрядов, обозначенных как неопределенные и находящихся в z-состоянии. Рассмотрим пример, в котором предусмотрена возможность определения вариантов с неопределенным значением аргумента:

```
case (select[1:2])
2'b00: out = 0;
2'b01: out = a;
2'b0x,
2'b0z: out = a ? 'bx : 0;
2'b10: out = b;
2'bx0,
2'bz0:  out = b ? 'bx : 0;
default:  out = 'bx;
endcase
```

Если бы в этом примере отсутствовали варианты, представляющие неопределенные значения, то при неопределенных сигналах на входе select выход также принял бы неопределенное значение. Здесь же нули на входах a или b "подавляют" неопределенные значения на входах select. Отметим, что при интерпретации этой программы при нулевом сигнале на входе select[1] состояние входа select[2] не имеет значения.

Версии оператора варианта с ключевыми словами casex и casez позволяют задавать совокупность вариантов, которые выполняются при совпадении хотя бы некоторых разрядов, а остальные разряды при этом не важны (так называемые "don't care"). В версии casez не учитываются при проверке на совпадение разряды, имеющие значение z как в записи варианта, так и в результате вычисления ключевого выражения. Так, коды 'b00zz101 и 'bzz1010z с точки зрения выбора в операторе casez совпадают. Оператор с ключевым словом casex рассматривает разряды, установленные в состояние высокого импеданса или неопределенное состояние, как несущественные. При этом символы неопределенности могут содержаться как в результате вычисления выражения, так и в записи варианта, причем в записи варианта вместо символов z и x можно использовать вопросительный знак. Например, приоритетный декодер может быть представлен следующим оператором:

```
casez (ir)
5'b1????: request_code=5;
5'b01???: request_code=4;
5'b001??: request_code=3;
```



```
5'b0001?: request_code=2;
5'b00001: request_code=1;
5'b00000: request_code=0;
endcase
```

3.3.7. Операторы повторения

В языке Verilog определены четыре формы операторов повторения:

```
<оператор повторения> ::=
    forever <оператор>
    | repeat ( <выражение>) <оператор>
    | while ( <выражение>)<оператор>
    | for ( <присвоение>; <выражение>; <присвоение>) <оператор>
```

Во всех случаях вложенный оператор может быть простым или составным, т. е. содержать совокупность последовательных операторов, заключенную между ключевыми словами begin и end.

Оператор с ключевым словом forever повторяется бесконечно каждый раз после исполнения вложенного оператора. При моделировании исполнение может быть прекращено вызовом системной функции $finish или системными средствами. Очевидно, что такой оператор может быть полезен, только если вложенная в него конструкция предусматривает прерывания выполнения, например блокирующие присваивания с объявлением времени, оператор ожидания wait.

Остальные формы операторов повторения по синтаксису, да и по интерпретации записанных действий, практически не отличаются от таких же операторов языка C, поэтому ограничимся несколькими простыми примерами. Отметим только, что реализуемые подмножества языка требуют использования "логически статических" выражений для операторов повторения.

Листинг 3.48 представляет описание последовательного умножителя. Оператор always начинает исполняться после положительного фронта сигнала start. После приема кодов сомножителей в сдвигающие регистры и обнуления регистра результата составной оператор, вложенный в оператор repeat, повторяется, причем число повторений задается выражением в скобках, в нашем случае, параметром size, значение по умолчанию для которого определено как 8. Исполнение вложенного оператора инициировано сигналом clock и предусматривает стандартные для последовательного умножения действия: прибавление содержимого регистра множимого к результату, если младший разряд регистра множителя нулевой, с последующим сдвигом кодов в регистрах множимого и множителя.

Листинг 3.48

```
module  multiply (clock,start, a,b,result,ready);
input clock,start;
input a,b;
output result,ready;
parameter size = 8, longsize = 16;
wire [size:1] a,b;
reg [size:1] opb;
reg ready;
reg [longsize:1] result;
reg [longsizые:1] opa;
always @ ( posedge start)
  begin
    opa = a; // загрузка новых исходных данных
    opb = b;
    result = 0;
    ready= 0;
    repeat (size)         // повторить для всех разрядов
      @ ( posedge clock) // блок инициируется фронтом clock
       begin
          if (opb[1])  result = result + opa;
 opa = opa <<1;
 opb = opb>> 1;
 end
 ready=1;
 end
endmodule
```

Фрагмент, представленный в листинге 3.49, описывает процесс подсчета числа единиц в байте входных данных `in_data`. На каждом шаге алгоритма выполняется сдвиг кода аргумента, и, если в младшем разряде нуль, к результату прибавляется единица. Цикл прекращается, когда сдвигающий регистр очищен.

Листинг 3.49

```
. . .
begin
  reg [7:0] shift_reg;
  result = 0;
  shift_reg = rega;
  while(shift_reg)
    begin
```



```
      if (shift_reg [0]) result = result + 1;
      shift_reg  = shift_reg>> 1;
    end
end
```

Этот же алгоритм может быть записан с использованием оператора `for`:

```
begin :countls
  reg [7:0] shift_reg;
  result = 0;
  for (shift_reg = rega; shift_reg!=0; shift_reg = shift_reg>> 1)
      if (tempreg[0]) result = result + 1;
end
```

В операторе `for` первое присвоение задает начальное значение переменной цикла, следующее за этим выражение — условие окончания (цикл прекращается, когда выражение дает нуль или не определено), а последнее (необязательное) — любые дополнительные присвоения, чаще всего преобразование переменной цикла.

3.3.8. Инициализация процедурных операторов

Выше уже неоднократно отмечалось, в языке Verilog порядок исполнения операторов определяется не только и не столько порядком их записи. Предусмотрен широкий набор средств, определяющих условия, при которых оператор будет исполнен. Эти условия оформляются в виде префиксов операторов и выражений языка, в том числе логических и арифметических выражений, а также и выражений присваивания. Среди этих средств наиболее важное значение имеют *префиксы управления временем* и *префиксы событийного управления*.

В *разд. 3.3.5* приведен ряд примеров использования префикса времени перед операцией присваивания. Префикс начинается с символа #, после которого записывается число, имя переменной времени или выражение. Однако действие этого префикса в различных ситуациях неодинаково.

Префикс времени перед оператором непрерывного присваивания означает вычисление нового значения на основе текущих значений аргументов и сохранение этого значения в буфере на время, заданное параметром префикса. Префикс перед словом `begin`, открывающим последовательный блок, означает задержку начала исполнения всего блока.

Если префикс предшествует последовательному оператору, то оператор исполняется после предыдущего в последовательности через временной интервал, заданный префиксом. Фактически, это означает приостанов исполнения программного блока. В качестве исходных данных для оператора используется значение переменных на момент начала его исполнения. Заме-

тим, что изменения аргументов могут быть произведены другими операторами, исполняемыми в "параллельно" инициированных блоках.

Если префикс времени предшествует блокирующему оператору присваивания, то вычисленное значение присваивается приемнику сразу после исполнения оператора, а если оператору неблокирующего присваивания — то после исполнения всех операторов, инициированных общим событием.

И наконец, префикс, расположенный перед записью операции в правой части оператора присваивания (такую запись называют "intra-assignment control", что можно перевести как управление изнутри присвоения), означает задержку присвоения результата приемнику, хотя при вычислении выражения также используются значения аргументов, существовавшие на момент инициализации оператора.

Перечисленные правила иллюстрируются программой, представленной в листинге 3.50 и на рис. 3.23, показывающем результаты ее моделирования.

```
module  delay_example;
wire d;
reg a,b,c; assign  #10 d=a;
initial
 begin a=0;
          #20 a=1;
           #20 a=0; end
initial #5
     begin b=a;
          #20 b=a;
          #20 b=a;
       end
 initial
     begin #5 c= #5 a;
           #20 c= #5 a;
           #20 c= #5 a;
       end
endmodule
```

На рисунке кружками отмечены моменты вычисления новых значений в операторах присваивания, а стрелки указывают моменты изменения сигналов, вызванные соответствующим оператором.

Если представление инициализации операторов с указанием времени характерно скорее для моделирования, то другой вид управления инициализацией, а именно событийное управление, непосредственно влияет на реализацию проекта.

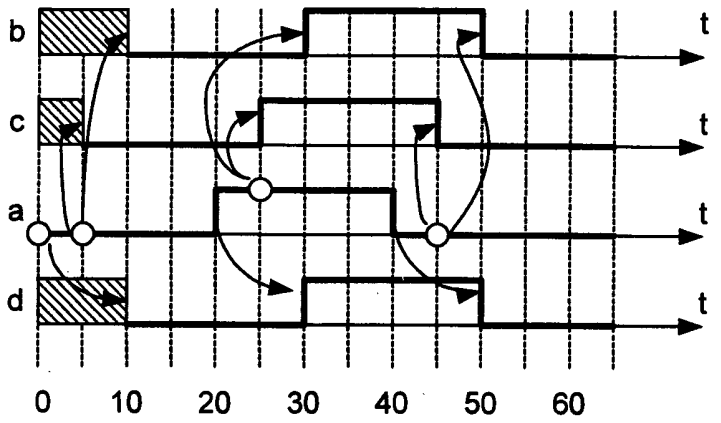

Рис. 3.23. Управление изнутри присвоения

Событийное управление отображается в программе префиксом события, который предшествует оператору, блоку или выражению.

```
<префикс события> ::=
    @(<событийное выражение> « or <событийное выражение> »)
    |  @<идентификатор события>
<событийное выражение> ::=
   <выражение>
     | posedge <скалярное выражение>
     |  negedge <скалярное выражение>
```

Событийное выражение вычисляется "непрерывно", т. е. его значение переопределяется при любом изменении входящих в него операндов.

Простейший случай — использование простого выражения в качестве событийного — означает, что оператор будет исполняться при любом изменении значения выражения. Например, оператор

```
@ (clock) q=d;
```

будет исполняться при любом изменении сигнала `clock`, т. е. как после положительного, так и после отрицательного фронта управляющего сигнала. Но такая конструкция не вполне соответствует поведению большинства реальных триггерных схем, которые реагируют только на один из фронтов. В принципе, конечно, можно использовать в качестве управляемого оператора не простое присвоение, а условный оператор с целью блокирования действий на одном из фронтов, но Verilog предоставляет более компактную запись такого поведения. Префикс, содержащий слово `posedge`, обеспечивает исполнение оператора только при изменении результата вычисления выражения, записанного в скобках, из нулевого состояния в единичное.

Таким образом, оператор

```
always  @ (posedge (clock & enable))  q=d;
```

определяет D-триггер с динамическим управлением, срабатывающий при разрешающем единичном сигнале на входе `enable` по нарастающему фронту сигнала `clock`.

Аналогично, ключевое слово `negedge` задает чувствительность только к спаду значения выражения. В префиксах со словами `posedge` и `negedge` выражение должно быть скалярным, т. е. таким, результатом которого является одноразрядный результат.

Если оператор может быть инициализирован при различных ситуациях, используется объединение инициирующих событий с использованием слова `or`. Тогда любое из перечисленных событий приводит к исполнению оператора. Так оператор

```
always @(posedge clock or d)     if(clock) q=d;
```

интерпретирует D-триггер с потенциальным управлением положительным уровнем сигнала `clock`.

Следующий пример описывает J-K-триггер с динамическим управлением и асинхронным сбросом:

```
always @(posedge clock or posedge reset)
       begin
          if( reset) q<=0;
              else  case {j,k}
                       0:;
                       1:q<=1'b0;
                       2:q<=1'bl;
                       3:q<=~q;
                    endcase
       end
```

Префикс событийного управления, подобно префиксу времени, может записываться перед выражением в правой части присвоения. Это означает, что выражение вычисляется в момент исполнения оператора, но фактическое присвоение выполняется при наступлении события. Включение дополнительного выражения повторения позволяет задать поведение, при котором присвоение будет выполняться после заданного числа инициирующих событий. Например, оператор

```
a_delayed= repeat (5) @(posedge clock) a_origin;
```

определяет, что значение `a_origin`, имевшееся на момент исполнения оператора, будет присвоено регистровой переменной `a_delayed` только по прошествии пяти нарастающих фронтов тактирующего сигнала.

Если какая-то комбинация условий инициализации операторов применяется достаточно часто, то такому событию можно присвоить идентификатор и далее использовать это имя. Например, событию "фронт тактового сигнала" можно было бы присвоить имя и в дальнейшем использовать это имя следующим образом:

```
event  specific_situation;                    // присвоение имени событию
always @(posedge clock)
       posedge reset -> specific_situation; // определение события
...
always @(specific_situation)...               // событийное управление
```

3.3.9. Блоки

В предыдущих разделах мы неоднократно использовали блочное представление программы. Составные операторы, вложенные в операторы инициализации и в операторы постоянного повторения, являются простыми примерами блоков. Блок объединяет операторы, связанные общими правилами инициализации. В данном разделе дается более подробное изложение концепции блока, принятой в языке Verilog.

Различают последовательные и параллельные блоки. Формальный синтаксис блока определен следующим образом:

```
<блок> ::=
    <открывающее слово>
    [ : <имя блока> [ <раздел деклараций блока>] ]
    « <оператор> »
    <закрывающее слово>
```

Раздел деклараций может содержать декларации локальных имен (констант, регистровых и численных переменных, а также событий), определенных только внутри данного блока.

Последовательный блок ограничивается закрывающим и открывающим словами `begin` и `end`. Операторы в последовательном блоке исполняются друг за другом в порядке записи. При наличии префиксов задержки время в процессе исполнения такого блока накапливается *(см. ранее приведенные примеры в листингах 3.44—3.47 и др.)*. Исполнение завершается после реализации последнего оператора в блоке.

Параллельный блок заключен между словами `fork` (распараллелить) и `join` (объединить). Порядок записи операторов в таком блоке не имеет значения. Операторы исполняются параллельно в смысле, определенном дискретной событийной моделью, т. е. по мере возникновения в системе инициирующих событий (изменения состояний определенных переменных или достижения системой моделирования предопределенных отметок времени), при-

чем задержки вложенных операторов исчисляются от момента инициализации параллельного блока. Изменение значений сигналов, которые производятся в других блоках перед исполнением каждого оператора, учитываются в результатах вычисления выражений.

Рассмотрим фрагмент, представленный в листинге 3.51, результаты моделирования которого приведены на рис. 3.24. Здесь время исполнения всех операторов, вложенных в параллельный, отсчитывается от фронта сигнала a.

Листинг 3.51

```
initial
 begin a=0;
    #10 a= 1;
    #20 a= 0;
    #70 a=1;
    #20 a=0;
  end
always @(posedge a)
         begin
            fork
               #20 b=0;  //
               #10 b=a;  // Это присвоение выполнено раньше предыдущего
               # 30 b=1'bz; // время отсчитывается от фронта сигнала a
            join
            #5 b=1'bz;  // время отсчитывается от выполнения последнего
                        // оператора в параллельном блоке
         end
endmodule
```

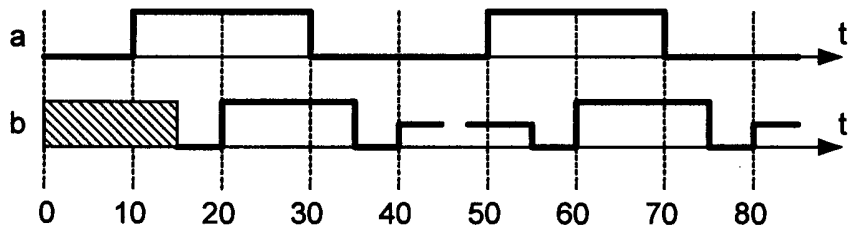

Рис. 3.24. Результаты моделирования программы (листинг 3.51)

В целом, однако, представляется, что подобное поведение удобнее описывать с использованием непрерывных присвоений или блокирующих присвоений.

3.3.10. Подпрограммы

Как и в традиционных языках программирования, использование подпрограмм позволяет обеспечить структуризацию, следовательно, лучшую понимаемость программ, а также экономит время проектировщика, позволяя однократно описывать однотипные фрагменты устройств и/или алгоритмы их функционирования. И так же, как и в других языках, в Verlog различают два вида подпрограмм, отличающихся по способу возвращения результата в вызывающую программу: *задачи* (task) и *функции* (functions). При вызове всем объектам подпрограммы, которые представлены локальными именами и которые мы будем называть формальными объектами, сопоставляются фактические объекты — значения или имена переменных, используемые в конкретном вызове вместо соответствующих формальных объектов.

Замечание

В большинстве публикаций по программированию, в том числе при представлении языка VHDL в данной книге, для определения информации, передаваемой между вызывающей программой и подпрограммой, используются термины "формальный параметр" и "фактический параметр". Но в Verilog понятие "параметр" имеет несколько иной смысл, а параметр может сам быть передаваемым объектом. Поэтому во избежание путаницы здесь использовано относительно нетрадиционное название для обозначения данных, сопоставляемых при вызове.

Задача возвращает результаты через сопоставление формальных и действительных объектов, функция возвращает единственное значение через имя подпрограммы на месте ее вызова в выражении. Подобный аппарат определен почти во всех языках программирования. Однако в Verilog есть специфические отличия по сравнению с традиционными языками как с точки зрения синтаксиса деклараций, так и с точки зрения правил использования.

Правила декларирования подпрограмм могут быть сведены к следующему:

```
<декларация подпрограммы> ::=
    <заголовок подпрограммы>
    « <декларация> »
    « <оператор> »
    <закрывающее  слово подпрограммы>
<заголовок подпрограммы>  ::=     task <имя>;  |  function  <имя>;
<закрывающее ключевое слово подпрограммы> ::=   endtask |  endfunction
```

Можно обратить внимание на то, что в отличие от большинства традиционных языков и VHDL, объявление имен объектов, принимаемых подпрограммой, не выделяется в специальном списке. Их объявление размещается в разделе деклараций подпрограммы вместе с объявлением внутренних данных подпрограммы. Набор допустимых деклараций такой же, как и в программном модуле *(см. разд. 3.3.1)*. Ограничения сводятся к тому, что в дек-

ларации функции не может быть выходных формальных объектов и должен быть хоть один входной формальный объект.

Кроме отмеченных отличий существуют следующие:

- Программный модуль "Задача" может содержать опции событийного и временного управления, т. е. для задачи могут предусматриваться условия инициализации и предполагаться ее выполнение в течение некоторого времени. Функция выполняется с "нулевой задержкой", конструкции, специфицирующие событийное или временное управление, в функциях запрещены.
- В задаче допускается вызов других задач и функций, функция может вызывать другую функцию, но не задачу.
- В разделе операторов функции должно содержаться присвоение значения имени функции.

Вызов задачи, как и вызов функции, предусматривает запись имени, вслед за тем в скобках через запятые записываются фактические объекты, причем *порядок записи фактических объектов строго соответствует порядку появления имен отображающих формальных объектов в декларации подпрограммы.*

Вызов задачи рассматривается как оператор, и его инициализация подчиняется общим правилам исполнения операторов.

- В последовательном блоке вызов задачи инициируется после исполнения предыдущего оператора. Возможно дополнительное управление инициализацией с использованием префиксов событийного или временного управления. Естественно, внутри себя задача может моделировать процесс, протекающий во времени.
- В параллельных блоках исполнение задачи инициируется при возникновении событий, предусмотренных префиксами операторов вызова, в том числе после достижения указанного префиксом времени.

Вызов функции традиционно включается в выражение, и предусмотренный алгоритм исполняется мгновенно (с точки зрения дискретной событийной модели), как только инициирован оператор, использующий это выражение. По умолчанию при реализации в аппаратуре каждому вызову функции, имеющемуся в программе, сопоставляется регистр для сохранения результата, состояние которого может измениться в момент исполнения соответствующего оператора.

В качестве примера в листинге 3.52 приведено описание устройства, реализующего преобразование

```
result = a × b − c × d;
```

Умножители представлены вызовами задачи `mul`. Алгоритм умножения, использованный в описании задачи `mul`, эквивалентен алгоритму, представленному модулем `multiply` в листинге 3.47. Здесь только немного модифицирована запись: использовано автоматное описание алгоритма и вызов функции для суммирования. Для реализации операций сложения и вычитания используется вызов функции `abd_subb`, причем из примера видно, что вызовы функции входят в правые части операторов присваивания, в том числе могут содержаться внутри задачи. Результат вычитания произведений присваивается порту `result` при исполнении непрерывного оператора, который инициируется положительным перепадом сигнала готовности одного из умножителей.

Листинг 3.52

```
module  mult_and_subb (clock,start, a,b,c,d,result);
input clock,start,input a,b,c,d;
output result;
wire [8:1] a,b,c,d;
reg ready;
reg [17:1] result;
reg [16:1]prod1,prod2;
function  add_subb;
input  opa,opb;
input direction; // 1 - сложение, 0 - вычитание
  if (direction) add_subb=opa+opb;
    else add_subb=opa-opb;
 endfunction

task mul;
parameter size=8;
input clk,start;
input [size:1] opa,opb;
output [2*size:1] product;
output ready;
reg ready, state;
reg [size:1]  shift_opb;
reg [2*size:1] shift_opa,product;
reg [3:0]count;
begin
@ ( posedge clock or posedge start)
   if (start) state=0;
    else  case (state)
         0: begin if (start)
                   shift_opa = opa;
                   shift_opb = opb;
                   product = 0;
```

```
                count=size;
                 ready= 0;
                 state=1;
            end
        1: begin if (count==0) begin
                                state=0;
                                ready=1;
                            end
                 else if (shift_opb[1])
                            product = add_subb(product,shift_opa,1);
                shift_opa = shift_opa <<1;
                shift_opb = shift_opb>> 1;
                count= add_subb(count,1,0);
            end
    endcase
end
endtask
mul (8,clk,start,a,b,prod1,ready);
mul  (8,clk,start,c,d,prod2,);
always @(posedge ready) result=add_subb({1'b0,prod1},{1'b0,prod2},0);
endmodule
```

3.3.11. Структурное описание проекта

Большинство современных языков проектирования аппаратуры поддерживают возможность описания проекта в виде совокупности заранее описанных компонентов и их связей, и язык Verilog не является исключением. Всякая иерархия представлена главным проектным модулем, который называют вершиной проекта, и совокупности подчиненных проектных модулей. Вершина проекта содержит операторы вхождения компонентов. Подчиненный проектный модуль может быть, в свою очередь, вершиной следующей иерархии. В общем случае, проектные модули независимы в том смысле, что каждый может использоваться самостоятельно в различных конструкциях и быть вершиной проекта. Проектные модули, относящиеся к одному проекту, могут находиться в одном файле или представляться несколькими файлами. В последнем случае проектные файлы должны компилироваться в библиотеку проекта в порядке их вхождения в иерархию снизу вверх.

Вхождение проектного модуля в проект высшего иерархического уровня отображается *оператором вхождения*, формальный синтаксис которого определяется следующим образом:

<оператор вхождения> ::=
    <имя модуля> [<присвоение значений параметрам>]
    <объявление вхождения> «,<объявление вхождения> »;

<присвоение значений параметрам> ::=
    # (<выражение> «,<выражение> »)
<объявление вхождения> ::=
    <имя вхождения> (список соединений)
<список соединений> ::=
    <соединение порта модуля> «,<соединение порта модуля> »;
    | <именованное соединение> «,<именованное соединение> »
<соединение порта модуля> ::= <выражение> | <пробел>
<именованное соединение> ::= .<имя порта> ([<выражение>])

При описании вхождения модуля в иерархический проект используются два имени: имя вхождения и имя модуля. Первое присваивает собственное имя включаемой подсхеме, а второе определяет способ ее функционирования и реализации в форме указания имени программного модуля, описывающего его функционирование или структуру. Несколько вхождений могут быть представлены однотипными подсхемами, т. е. ссылаться на одинаковый программный модуль, но каждый экземпляр имеет собственное имя. Как видно из синтаксических правил, допустимо объявление нескольких однотипных встраиваемых модулей в одной декларации.

Оператор вхождения может рассматриваться как вызов подпрограммы с проблемно-ориентированным интерфейсом вызова. Кроме того, отличие состоит в том, что внутренние переменные включаемого модуля определены как статические, сохраняющие значения между инициализациями. Формальные объекты включаемого модуля (параметры и сигналы) при инициализации оператора замещаются фактическими объектами вызывающей программы. Оператор вхождения модуля обязательно параллельный, это означает, что преобразования, определенные в операторной части включаемого модуля, выполняются всякий раз, когда изменяются фактические объекты оператора вхождения.

Список присваиваемых при вызове фактических значений параметров предшествует имени вхождения. Порядок записи фактических значений параметров строго соответствует порядку объявления формальных имен параметров в описании включаемого модуля.

Сопоставление формальных и фактических переменных при вызове осуществляется либо в соответствии с позицией в соответствующем списке, либо по имени (именованное соединение). Недопустимо использовать разные способы сопоставления в одном операторе. При позиционном сопоставлении порядок записи фактических объектов в списке соединений порта точно соответствует порядку записи имен портов в тексте описания модуля. Неиспользуемые порты, а также входные порты, для которых значение сигналов выбирается заданным по умолчанию, отмечаются символом <пробел>. При сопоставлении по имени описание подключения порта начинается со знака "точка", после чего записано имя порта, а затем, в скоб-

ках, имя сигнала или выражение, вычисляющее значение сигнала, подаваемого на порт.

Рассмотрим в качестве примера модуль умножения восьмиразрядных операндов на базе четырехразрядных умножителей, интерфейс которых соответствует модулю, представленному листингом 3.47 (операторная часть модуля `multiply` может быть и другой, например представлять параллельную схему — в данном случае это не существенно). Программа (листинг 3.53) представляет устройство с четырьмя параллельно работающими блоками, каждый из которых выполняет попарное умножение тетрад кодов операндов с последующим взвешенным суммированием частичных произведений по формуле:

```
Product = mult * fact =
= (16 * mult[7:4] + mult[3:0]) * (16 * fact[7:4] + fact[3:0]) =
= 256(mult[7:4] * fact[7:4]) +
+ 16(mult[7:4] * fact[3:0] + mult[3:0] * fact[7:4]) + fact[3:0] * mult[3:0]=
= 256 * higher + 16 * (middle1 + middle2) + lower;
```

Для трех встроенных умножителей использовано сопоставление портов по имени, а для одного последнего — позиционное сопоставление.

Листинг 3.53

```
module  mult8x8 (clk,strt, mult,factor,product);
input clk,strt;
input mult,factor;
output product;
wire [8:1] mult,factor;
reg [8:1] v1, v2, v3,v4,elder_part;
reg [9:1] middle_part;
wire [16:1]  product;
wire fin;
multiply #(4,8)
   lower (.clock(clk),.start(strt),.a(mult[4:1]),.b(factor[4:1]),
            .result(v1),.ready(fin)),
   middle1(.clock(clk),.start(strt),.a(mult[8:4]),
            .b(factor[4:1]),.result(v2)),
   middle2(.clock(clk),.start(strt),
              .a(mult[4:1]),.b(factor[8:4]),.result(v3)),
   elder  (clk,strt,mult[8:4],factor[8:4],);
always @(posedge fin)
  begin
    middle_part={1'b0,v2)+ {1'b0,v3}+ {5'd0,v1[8:4]};
    elder_part= {3'd0,middle_part[9:4]}+v4;
  end
assign product={elder_part,middle_part[4:1],v1[4:1]};
endmodule
```


Отметим, что в программе, во избежание потери бита переполнения, разрядность переменной `middle_part` и кодов, используемых для ее вычисления, расширена на один разряд в сторону старших.

Иногда удобно объединить совокупность объявлений параметров нескольких различных компонентов в общую область программы. Это можно сделать с помощью специального программного блока декларации параметров `defparam`. В примере, представленном программой (листинг 3.54), модуль верхнего уровня иерархии `top` содержит два экземпляра регистра-защелки `latch`, но один из них (`m1`) — пятиразрядный, а второй (`m2`) — десятиразрядный. Эти параметры, а также времена задержки регистров, определены в выделенном программном модуле `parameter_definition`.

Листинг 3.54

```
module  top( in1,in2,en1,en2,adr_write,out);
  input [0:4] in1;
  input [0:9]in2;
  input  en1,en2,adr_write;
  output[0:9] out;
  wire [0:4] o1;
  wire [0:9] o2;
  wire [0:9] out;
  latch m1 (dat1, in1, en1); // оператор вхождения
  latch m2 (o2, in2, en2);
  assign #5 out={5'd0,o1}& adr_write | o2 &  !adr_write; // переключатель
endmodule
module latch (out, in, clk,en); // встраиваемые модули
  parameter size = 1, delay = 1;
  input [0:size-1] in;
  input clk,en;
  output [0:size-1] out;
  reg [0:size-1] out;
  always @(posedge clk or posedge clk)
  # delay out = in;
endmodule
module parameter_definition; // модуль декларации параметров
  defparam
  top.m1.size = 5,
  top.m1.delay = 10,
  top.m2.size = 10,
  top.m2.delay = 20;
endmodule
```

3.3.12. Примитивы

Подклассом встраиваемых модулей являются примитивы. Общее их свойство — они имеют единственный выходной порт. Различают предопределенные примитивы и примитивы, определяемые пользователем (User Defined Primitives, UDP).

Предопределенные примитивы

Образы предопределенных примитивов по умолчанию присутствуют в библиотеке, и дополнительных деклараций не требуется, а вызов подчиняется общим правилам объявления вхождений.

Приведем список наиболее употребительных предопределенных примитивов Verilog:

```
<имя предопределенного примитива> ::=
    and | nand |or |nor | xor | xnor | buf |bufif0 | bufif1 | not
```

Имя предопределенного примитива определяет реализуемую логическую функцию всех входных объектов в соответствии с общеупотребительной нотацией. Предопределенным параметром примитивов является время задержки, объявление которого предшествует имени вхождения, как и принято в декларации параметров вхождения, и выделяется символом #. Список соответствий портов начинается в примитивах с выходного порта, после чего в произвольном порядке записываются информационные входы. Для буферных схем (buf, bufif0, bufif1) список заканчивается указанием соответствия входа разрешения.

Включение примитивов, по сравнению с включением других модулей, дает дополнительные возможности. Во-первых, имеется прямая возможность объявления силы драйвера, соответствующего логической ячейке (это указание размещается непосредственно за именем модуля перед именем вхождения и параметром задержки, если таковой определяется). Указание силы драйвера определяет состояние линий связи, к которой подключается несколько источников по тем же правилам, что и для параллельных присвоений. Во-вторых, вводится возможность определения совокупности примитивов в форме вектора. Например, схема сравнения кодов может быть представлена следующим образом (листинг 3.55):

Листинг 3.55

```
module  code_compare (inp1,inp2,result);
parameter length=8;
input [lenlth-1:0] inp1,inp2;
 output result;
wire [lenlth-1:0] inp1, inp2,bit_wise _comp;
wire result;
```



```
xor #(5) comp [lenlth-1:0] (bit_wise_comp, inp1, inp2,);
assign result= | bit_wise_comp;
endmodule
```

Здесь конструкция

```
xor comp[lenlth-1:0](inp1, inp2, bit_wise_comp)
```

представляет набор из lenlth элементов "исключающее ИЛИ", имеющих задержку 5 единиц модельного времени каждый. Выходы этих элементов объединяются по логике ИЛИ, формируя единичный сигнал, если хоть в одном из разрядов аргументов имеется несовпадение.

Все же нам представляется, что описание комбинационных логических схем с использованием обычных логических выражений и присвоений выглядит более естественно и наглядно. Правда, разработчики языка утверждают, что представление логических преобразований с использованием примитивов позволяет уменьшить время моделирования.

Примитивы, определяемые пользователем

Пользователю предоставлена возможность создания собственных примитивов (User Defined Primitive, UDP), а именно моделей устройств, имеющих произвольное число входов, но единственный выход. Может быть задана произвольная логическая функция. Допускается создавать как UDP комбинационного типа, так и примитивы, обладающие памятью и фактически представляющие триггеры со специфическими функциями переходов.

Для определения UDP вводится специальная программная единица primitive. По своему иерархическому уровню primitive эквивалентен модулю и может находиться в тексте программы как *до*, так и *после* модуля, который содержит ссылку на этот примитив. Определение UDP разрешено представлять также в другой программе, заранее скомпилированной в библиотеку проекта. Определение UDP не должно входить в другой модуль (помещаться в программе между ключевыми словами module и endmodule).

Определение UDP подчиняется следующим синтаксическим правилам:

```
<UDP> ::=
    primitive <имя UDP> (<имя выхода>,
      <имя входа> «,<имя входа> »);
      input <имя входа> «,<имя входа> »;
      output <имя выхода>;
      [ <спецификация выхода> ] // только для последовательностных UDP
      [ <определение исходного состояния> ] // только для
                                            // последовательностных UDP
      <таблица истинности>
    endprimitive
```

Обратите внимание на то, что смысл имен входов и выходов определен дважды. Во-первых, он однозначно задается порядком записи в заголовке (т. е. в скобках после имени примитива), а во-вторых, специфицируется высказываниями `input` и `output`. Несовпадение этих двух определений является ошибкой.

Синтаксис определения UDP комбинационного типа отличается от синтаксиса определения последовательностных UDP, поэтому рассмотрим их раздельно. *В комбинационных UDP не предусматривается определение исходного состояния и спецификации выхода.*

Синтаксис таблицы для комбинационного UDP определен следующим образом:

```
<таблица для комбинационного UDP> ::=
  table
  « <список значений входов> : <значение выхода комбинационного UDP>;»
  endtable
<список значений входов> ::= <значение входа>  « <значение входа> »
<значение входа> ::= 0 | 1 | x |? | b
<значение выхода комбинационного UDP> ::= 0 | 1 | x
```

Слева от двоеточия в каждой строке таблицы приводится возможная комбинация значений аргументов, а справа — значение выхода при такой комбинации на входе.

Порядок записи значений входов должен строго соответствовать порядку следования их имен в заголовке определения примитива.

Значения входов и выхода `0,1` и `x` имеют традиционный для Verilog смысл. Символ `?` означает "не важно", т. е. отмеченный таким образом вход не влияет на результат. Символом `b` обозначают совокупность комбинаций входов, для которых не важно, находится ли соответствующий вход в нулевом или единичном состоянии, но неопределенное состояние этого входа будет вызывать другие результаты и требует дополнительного описания.

В листинге 3.56 в качестве примера приведена программа, содержащая определение примитива `vote` (голосование), реализующего распространенную функцию большинства

```
major =x&y | y&z | x&z;
```

и модуль двухразрядного сумматора, включающий этот примитив.

Листинг 3.56

```
primitive vote (major,x,y,z);
input x,y,z;
output major;
```



```
table
//  x    y    z    :    major
    1    1    ?    :    1;
    ?    1    1    :    1;
    1    ?    1    :    1;
    0    0    ?    :    0;
    0    ?    0    :    0;
    ?    0    0    :    0;
endtable
endprimitive
module add_2bit ( in1,in2, cin, result,cout);
input  in1,in2, cin;
output result, cout;
wire [1:0] in1,in2,result;
wire [1:0] carry;
vote  #4 c1 (carry[0],in1[0],in2[0],cin);
vote  #4 c2 (carry[1],in1[1],in2[1],carry[0]);
assign #4 cout=carry[1];
  result[0]=(in1[0] && in2[0] && cin) || carry[0] && (in1[0] || in2[0] ||
                                                      cin),

  result[1]= in1[1] && in2[1] && carry[0] ||
             carry[1] && (in1[1] || in2[1] || carry[0]);
endmodule
```

Определение UDP последовательностного типа (фактически автомата с двумя устойчивыми состояниями, т. е. триггера) отличается, прежде всего, тем, что для выхода вводится дополнительная спецификация, определяющая его как регистровую переменную в соответствии с общими правилами языка Verilog:

```
reg <имя выхода>;
```

Возможно определение начального состояния автомата за счет включения оператора вида:

```
initial <имя выхода> = <значение>;
```

Синтаксис представления таблицы для последовательностного UDP определен следующим образом:

```
<таблица для последовательностного UDP> ::=
  table
 « <список значений входов>:<исходное состояние>:<состояние перехода>; »
  endtable
<состояние перехода> ::= 0 | 1 | x | -
```

Состояние и выход имеют одно и то же значение.

Для последовательностных UDP, управляемых уровнем синхронизирующего сигнала, набор допустимых значений входов и исходных состояний такой же, как набор допустимых значений входов комбинационных UDP. Для оп-

ределения состояния перехода введен дополнительный символ -, означающий сохранение состояния.

Для примера в листинге 3.57 приведено определение примитива, соответствующего синхронному RS-триггеру с потенциальным управлением (единичный уровень разрешающий). При нулевом сигнале на входе clk, а также если на обоих информационных входах нулевые сигналы, состояние триггера не изменяется. Если clk=1, а на информационных входах присутствует запрещенная комбинация сигналов или неопределенные сигналы, то состояние элемента считается неопределенным.

Листинг 3.57

```
primitive  rsff_sync(q,clk,r,s);
input clk,r,s;
output q;  reg q;
table
//  clk   r    s    :    q(t)   :    q(t+1)
     0    ?    ?    :     ?     :      -;
     ?    0    0    :     ?     :      -;
     1    0    1    :     ?     :      1;
     1    1    0    :     ?     :      0;
     1    1    1    :     ?     :      x;
     1    ?    x    :     ?     :      x;
     1    x    ?    :     ?     :      x;
     1    ?    x    :     ?     :      x;
endtable
endprimitive
```

Для последовательностных UDP с динамическим управлением для одного из входов, определенного как синхронизирующий, записывается два значения сигнала в форме (v,w), где v≠w и v, w ∈ {0,1,x,b,?}. Это означает, что изменение выхода происходит после изменения соответствующего сигнала из состояния v в состояние w.

Программа (листинг 3.58) иллюстрирует такую запись. Представлен J-K-триггер с асинхронным сбросом по входу r (активный низкий уровень) и переключением нарастающим фронтом сигнала clk.

Листинг 3.58

```
primitive  jkff_r(q, r, clk, j, k);
input clk,r,j,k;
output q;  reg q;
initial q=1'b0;
table
```



```
//  r    clk     j    k    :    q(t)   :    q(t+1)
    0     ?      ?    ?    :     ?     :     0;
    1    (1?)    ?    ?    :     ?     :     -;
    1    (01)    0    0    :     ?     :     -;
    1    (01)    1    0    :     ?     :     1;
    1    (01)    0    1    :     ?     :     0;
    1    (01)    1    1    :     1     :     0;
    1    (01)    1    1    :     0     :     1;
endtable
endprimitive
```

3.4. Язык AHDL

3.4.1. Общая характеристика языка и структура программы

Язык описания аппаратуры Altera HDL (AHDL) был создан фирмой Altera в 1983 году. Компилятор с этого языка интегрирован во все пакеты проектирования фирмы (MAX+PLUS II, Quartus), а также разработки других фирм, и обеспечивает прямую компиляцию описания в файл конфигурации любых БИС разработки фирмы, равно как передачу описания в подсистему моделирования и преобразование в описания во входные языки других пакетов [4].

Весьма важным свойством языка, упрощающим создание сложных проектов, является наличие средств, позволяющих легко модифицировать отдельные фрагменты и использовать их в процессе иерархического проектирования устройства. Следует учитывать, что существенное влияние на конечный результат проектирования, т. е. на скомпилированный проект, оказывают не только используемые конструкции языка, но и установленные опции режимов компиляции. Эти опции сохраняются в файле конфигурации, имя которого совпадает с именем проекта, а расширение обозначается как acf. Такой файл создается автоматически при открытии проекта, и исходно в нем установлены опции, принятые в системе по умолчанию. Изменение опций проекта в САПР MAX+PLUS II можно осуществлять, используя пункт системного меню ASSIGN, либо редактор топологии FLOORPLAN EDITOR, а также путем прямой корректировки конфигурационного файла с помощью текстового редактора.

Язык AHDL относится к классу приборно-ориентированных языков. Описание устройств в языках этого класса обеспечивает больший по сравнению с языками высокого уровня контроль разработчика над создаваемой реализацией. Ряд конструкций языка служат для эффективного использования

архитектурных особенностей микросхем программируемой логики фирмы Altera. В сущности, язык AHDL является языком структурного описания, т. е. способом представления набора типовых компонентов и их настроек, а также связей между ними. Хотя в языке имеются конструкции, которые "выглядят" как описания поведения, например оператор условия IF-THEN, оператор выбора CASE, оператор повторения FOR-GENERATE, надо иметь в виду, что фактически подобные синтаксические конструкции являются описанием определенных структур. Так оператор IF-THEN-ELSE представляет переключатель, который в зависимости от управляющего сигнала, задаваемого условием, подключает к своему выходу (выходам) выходы одной из подсхем, описанных в альтернативных вариантах оператора. Поведенческий аспект скрыт от проектировщика в моделях компонентов, используемых внутри системы проектирования на этапе симуляции собранного проекта. При мысленном сопоставлении AHDL-программе некоторого поведения можно считать, что операторы языка относятся к классу параллельных операторов, т. е. выполнение действия, заданного оператором, происходит при любом изменении операндов другими операторами. При функциональном моделировании предполагается дельта-задержка, а при временном — задержки, значения которых близки к реальным задержкам в выбранных микросхемах.

Проект в AHDL может быть представлен одним текстовым файлом, содержащим описание проекта (AHDL-программу) и называемым *проектным* или *логическим модулем* (Design File). Иерархические проекты, кроме файла описания верхнего иерархического уровня (Top-Level Design File), содержат совокупность файлов, представляющих нижние уровни иерархии (Low-Level Design Files). В этом случае, программы каждого уровня иерархии содержат ссылки на модули следующего низшего уровня. Допускаются ссылки не только на проектные модули, описанные в языке AHDL, но и на модули, представленные в других формах, принятых в САПР MAX+PLUS II и Quartus (графической форме и других языках). Имя проекта должно совпадать с именем модуля верхнего уровня иерархии.

Программа на языке AHDL должна обязательно содержать две секции — *предпроектную* секцию (Subdesign Section), описывающую интерфейс проекта, и *логическую* секцию (Logic Section), описывающую структуру и функционирование схемы, составляющей собственно проект. Кроме обязательных секций в начало программы допускается помещать необязательную секцию *заголовка* (Title Section). Для легкой идентификации проекта целесообразно всегда включать в эту секцию оператор заголовка проекта, а остальные подсекции использовать в тех случаях, когда этого требует логика формирования проекта.

Рекомендуемая последовательность секций логического модуля и их назначение представлены в табл. 3.8.

Таблица 3.8. *Рекомендуемая последовательность секций логического модуля и их назначение*

№	Наименование	Обязат./ не обязат.	Назначение
1	Секция заголовка (Title Section)	Не обяз.	
	Оператор заголовка проекта (Title Statement)		Задает собственное имя проекту для связывания с другими проектами
	Оператор(ы) включения (Include Statement)		Включает фрагменты заголовка, находящиеся в других файлах
	Оператор(ы) объявления констант (Constant Statement)		Объявляет имена и фактические значения констант
	Оператор(ы) вычисляемых функций (Define Statement)		Объявляет идентификатор встраиваемой функции и способ вычисления ее значения
	Оператор объявления параметров (Parameter Statement)		Объявляет параметры настройки при создании параметризированных модулей
	Оператор(ы) объявления прототипов (Function Prototype Statement)		Предъявляет образы используемых в данном проекте подпроектов
	Оператор опции проекта (Options Statement)		Определяет трактовку двоичного кода в арифметических операторах
	Оператор(ы) проверки (Assert Statement)		Проверяет корректность конфигурации и выдает сообщение об ошибке
2	Секция предпроекта (Subdesign section)		
	Заголовок предпроекта	Обяз.	Определяет порты проекта
	Подсекция переменных (Variable Section)	Не обяз.	Присваивает имена внутренним узлам проекта, включая связи, комбинационные схемы, регистры, автоматы, встраиваемые модули
	Оператор(ы) проверки (Assert Statement)	Не обяз.	Проверяет корректность конфигурации и выдает сообщение об ошибке

Таблица 3.8 (окончание)

№	Наименование	Обязат./ не обязат.	Назначение
3	Логическая секция (Logic Section)	Обяз.	Секция включает нижеперечисленные операторы в произвольном порядке
	Оператор умолчания (Defaults Statement)	Не обяз.	Объявляет значение переменной, которое она примет по умолчанию (при отсутствии явного присвоения)
	Логическое уравнение (Boolean Equation)	Не обяз.	Присваивает значение переменной
	Оператор выбора (Case Statement)	Не обяз.	*См. разд. 3.4*
	Условный оператор (If-then Statement)	Не обяз.	*См. разд. 3.4*
	Оператор повторения (For Generate Statement)	Не обяз.	Генерирует совокупность однотипных модулей
	Оператор условной генерации (If — Generate Statement)	Не обяз.	Воспроизводит одну из альтернативных реализаций проекта в зависимости от параметров настройки
	Прямой вызов функции (In-Line Logic Function Reference)	Не обяз.	Определяет способ подключения встраиваемых модулей
	Оператор таблицы (Truth Table Statement)	Не обяз.	Задает логические функции и функции переходов автоматов в табличной форме
	Оператор проверки (Assert Statement)	Не обяз.	Проверяет корректность конфигурации и выдает сообщения об ошибке

3.4.2. Типы данных и выражения. Оператор присваивания

Как и в других языках, в AHDL определены скалярные и групповые типы данных. Группой называется совокупность объектов, которые подвергаются совместному или одинаковому преобразованию.

Определены следующие типы данных: *логический, целочисленный, символьный и перечислимый.* Специальных средств декларации типа не предусмотрено —

тип устанавливается по контексту. Отметим, что перечислимый тип применяется только для задания состояния цифровых автоматов. Скалярному данному соответствует *простое имя*, которое, как и в других языках, записывается в виде произвольной последовательности цифр и букв, начинающейся с буквы.

Данные символьного и целочисленного типов не могут изменяться в реализованном проекте, и их значения учитываются только на этапе компиляции. Символы (чаще объединяемые в строки) используются для определения варианта конфигурации настраиваемых (параметризируемых) модулей, а также задают содержание выводимых сообщений в операторах проверки. Данные целочисленного типа служат для задания параметров конфигурации, границ групповых данных, индексов элементов в группах. Целые числа можно использовать для представления групповых логических констант в виде целочисленного эквивалента соответствующего двоичного кода, однако и этот случай мы определим как данные логического типа.

Целые числа можно представлять в различных форматах: десятичном, двоичном, восьмеричном и шестнадцатеричном. Десятичный формат записывается как простая последовательность цифр 0—9. В других форматах последовательность разрешенных цифр заключается в двойные кавычки, перед которыми записывается символ формата. Способы записи чисел сведены в табл. 3.9.

Таблица 3.9. Способы записи чисел

Формат числа	Разрешенные цифры	Символ формата	Пример
Десятичный	0 — 9		1248
Двоичный	0, 1, x	B; b	B"1x001"
Восьмеричный	0 — 7	O; o; Q; q	O"705"
Шестнадцатеричный	0 — 9, A, B, C, D, E, F	H; h; X; x	H"1AE F"

Арифметические данные могут входить в *арифметические выражения*. Арифметические выражения, кроме типовых операций: сложение '+', вычитание '–', умножение '*', деление нацело 'DIV' — могут включать специальные функции, список которых приведен в табл. 3.10. Значения арифметических выражений вычисляются только на этапе компиляции проекта, а при реализации соответствующие значения рассматриваются как константы. Результат вычисления арифметического выражения — положительное целое. По умолчанию, если при логарифмировании и делении получено дробное значение, выполняется округление до ближайшего большего целого. Отрицательный результат заменяется нулевым.

Таблица 3.10. *Специальные функции*

Название	Обозначение	Пример	Комментарий
Возведение в степень	^	a^2	
Модуль	MOD	a MOD b	Остаток от деления a на b
Двоичный логарифм	LOG2	LOG2(12-4)	
Выбор значения	? :	(a<5) ? 3:8	Если выражение в скобках истинно, то выбирается первое значение, иначе второе
Округление до ближайшего большего целого	CEIL	CEIL(log2(156)) = 7	
Округление до ближайшего меньшего целого	FLOOR	FLOOR(15 DIV 4) = 3	

Логический тип является единственным способом представления сигналов в проектируемой системе. Данные этого типа (переменные, порты и константы) могут принимать три значения: GND, соответствующее логическому нулю, VCC, которое соответствует логической единице, и Z — высокоимпедансное состояние. Состояние Z могут принимать только логические переменные, представляющие выходы буферов с тремя состояниями или двунаправленные порты. Переменным и портам в соответствующих разделах программы должны быть сопоставлены имена и спецификации. Спецификация поясняет специфические свойства носителя данных и способ их использования.

Приведем формальное определение синтаксиса разделов, связанных с объявлением констант, портов и переменных.

```
<оператор объявления константы> ::=
    CONSTANT <имя> = <константное_выражение>;
<константное_выражение> ::=
    <арифметическое константное выражение>|
    <логическое константное выражение>|
<арифметическое константное выражение> ::=
    <целое>|<имя_константы>|<арифметическое_выражение>
<логическое константное выражение> ::=<целое> | VCC | GND
```



```
<предпроектная секция> ::=
    SUBDESIGN <имя_модуля>
      (<имя_порта> «,<имя_порта>»:<спецификация порта>
       «;<имя_порта> «,<имя_порта>»:<спецификация порта>»
     )
<имя_порта> ::=
    <имя>|
    <имя>[<граница>..<граница>] [ [<граница>..<граница>] ]
<спецификация порта> ::=
    INPUT [ = <значение по умолчанию> ]
    | OUTPUT
    | BIDIR [ = <значение по умолчанию> ]
    | MASHINE INPUT | MASHINE OUTPUT
<значение по умолчанию> ::= [ =GND | =VCC ]
<граница> ::=<арифметическое константное выражение>
<подсекция переменных> ::=
  VARIABLE <имя_переменной> «,<имя_переменной»:
                          <спецификация_переменной>;
  «<имя_переменной> «,<имя_переменной»:<спецификация_переменной>;»
  «<декларация вхождения подпроекта> »
<имя_переменной> ::=
    <имя>
  | <имя> <индексное выражение> [ <индексное выражение>]
<индексное выражение> :=
  [<граница>..<граница>] | []
<спецификация переменной> ::=
  NODE | TRI_STATE_NODE | <имя_примитива>
  |<декларация цифрового автомата>
```

Поясним некоторые введенные конструкции.

Константа, в конечном счете, представляет число, которое может использоваться как параметр конфигурации или как двоичный код для представления групп двоичных констант. При объявлении константы допускается использование в выражении другой, ранее объявленной, константы.

Если после имени порта или переменной размещается выражение в квадратных скобках, имеется в виду объявление группы (фактически, битового массива) или обращение к элементу группы. Такие группы называются постоянными. (Кроме того, язык предусматривает объединение скалярных данных в так называемые временные группы, определенные только в пределах одного оператора и рассмотренные далее.) При отсутствии индексного выражения при объявлении переменной предполагается объявление скалярного данного. Запись двух индексных выражений подряд вслед за именем объявляемого данного определяет двумерную группу. Группы (массивы) большей размерности не определены.

Спецификации портов INPUT и OUTPUT определяют входы и выходы проекта, соответственно, а BIDIR — порт ввода/вывода. К двунаправленному порту можно подключать только элементы, имеющие высокоимпедансное состояние выхода, а точнее, сопоставить в программе имя порта проекта и имя порта встраиваемого модуля, имеющего такие свойства. *(Подробнее о встраивании модулей в проект см. в разд. 3.4.7 и 3.4.9).* Для скалярных входных и двунаправленных портов можно определять значение по умолчанию. Это значение будет использоваться, если проект включается в проект высшего уровня иерархии, а соответствующий контакт не задействован. Порты MASHINE INPUT и MASHINE OUTPUT обеспечивают передачу сигналов цифровых автоматов между модулями одного иерархического проекта. Проект высшего уровня иерархии таких портов иметь не может.

Переменная, специфицированная как NODE, обычно является выходом комбинационной логической схемы (например одной или нескольких макроячеек), функция которой задается оператором присваивания значения этой переменной в логической секции. Переменная, специфицированная как TRI_STATE_NODE, представляет шину, к которой можно подключать компоненты, допускающие высокоимпедансное состояние на выходе. Переменные, специфицированные как примитивы или вхождения подпроектов, отражают встраиваемые модули как поставляемые с системой проектирования, так и создаваемые проектировщиком, причем способ их соединения определяется в логической секции.

Скалярные логические переменные могут входить в булевские выражения. Табл. 3.11 содержит список определенных в AHDL двуместных и одноместных булевских операций. Символическое и буквенное обозначение операций равнодопустимы (эквивалентны). Например, a & !b эквивалентно a AND NOT b. Важно иметь в виду, что знак !, стоящий перед именем операнда, инвертирует только операнд, а этот же знак, стоящий перед другим знаком операции (#, & или $), задает инверсию результата соответствующей операции. Так a !# b эквивалентно !(a # b). В сложных логических выражениях сохраняются общепринятые правила старшинства (см. правую колонку табл. 3.11) и раскрытия скобок. Логическое выражение может быть правой частью оператора присваивания (в AHDL оператор присваивания обозначается обычным символом равенства =), а также входить как условие в операторах условия.

***Таблица 3.11.** Список определенных в AHDL двуместных и одноместных булевских операций*

Название	Обозначение		Уровень приоритета
	Буквенное	Символьное	
Инверсия	NOT	!	1
И (конъюнкция)	AND	&	2

***Таблица 3.11** (окончание)*

Название	Обозначение		Уровень приоритета
	Буквенное	Символьное	
И-НЕ	NAND	!&	2
Исключающее ИЛИ	XOR	$	4
Равнозначность	XNOR	!$	4
ИЛИ (дизъюнкция)	OR	#	5
ИЛИ-НЕ	NOR	!#	5

Для примера рассмотрим программу (листинг 3.59), описывающую модуль, названный simple. Проект содержит три логических уравнения, одно из которых формирует внутреннюю переменную c на основе входных данных, а два других — выходные данные. При реализации в универсальном базисе (И, ИЛИ, НЕ) потребуется инвертор, два элемента И и один элемент ИЛИ. Фактически же, компилятор (по возможности) автоматически игнорирует факторизацию, и, в данном случае, устройство будет реализовано на двух ячейках типа "четырехвходовая LUT" для семейств класса FLEX, либо двух макроячейках SUM_OF_PRODUCT для микросхем семейств класса MAX. Если модуль simple включается в качестве компонента в другой проект, то порт b может и не подключаться, при этом используется значение по умолчанию, и в таком случае выход out2 будет генерировать константную логическую единицу.

Листинг 3.59

```
SUBDESIGN simple
(
   a0, a1: INPUT;
   b: INPUT=VCC;
   out1, out2 : OUTPUT;
)
VARIABLE c :  NODE;
BEGIN
   c= a1 & !a0
   out1 =  c & b;
   out2 =  c # b;
END;
```

Элементы постоянных групп могут участвовать в логических выражениях как скаляры. При этом после имени группы в квадратных скобках указыва-

ется индекс элемента. Например, `elem[5-1]` представляет пятый справа элемент группы `elem[7..0]`. Индекс может быть представлен числом, целочисленной константой или переменной, а также арифметическим выражением. Кроме того, в логическое выражение может входить целая группа или любая его часть. Это допускается, если разрядности групп, участвующих в логическом преобразовании, совпадают, а также если в преобразовании участвуют группа и скалярная переменная. При этом операции выполняются над парами, имеющими одинаковые относительные позиции в группе, безотносительно к их индексам. В квадратных скобках после имени группы указывается диапазон битов, участвующих в преобразовании. Если диапазон явно не указан, в преобразовании участвует вся группа.

Пусть, например, в разделе переменных определено:

```
a[3..0],b[7..2],c[4..1], x,y,u,v :node;
```

Тогда допустимы следующие операторы:

- `a[3..0]=c [4..1];`

  эквивалентно `a[0]=c[1]; a[1]=c[2]; a[2]=c[3]; a[3]=c[4].`

- `a[2..0]=b[5..3] & c[4..2];`

  эквивалентно `a[0]=b[3] & c[2]; a[1]=b[4] & c[3], a[2]=c[4],` `a[3]` не изменяется.

- `a[ ]=c[ ] & !b[5..2];`

  эквивалентно `a[0]=b[2] & !c[1]; a[1]=b[3] & !c[2]`, и т. д.

- `a[ ]= c[ ] # x;`

  эквивалентно `a[0]=c[1] # x; a[1]=c[2] # x`, и т. д.

Как уже отмечалось, число может использоваться как сокращенная форма записи двоичной константной группы. Таким образом:

```
b[5..2]=c[ ] # 9; -- эквивалентно b[2]=VCC; b[3]=c[2]; b[4]=c[3]; b[5]=VCC;
```

Число разрядов группы в правой части оператора присваивания (приемника) не обязательно совпадает с разрядностью выражения в левой части (источника), но оно должно быть кратно числу разрядов источника. Если это так, то источник повторяется в поле приемника слева направо до заполнения поля приемника. Например, присвоение `b[7..2]= B"01"` дает результат `b[7..2]= B"010101"`.

В то же время выражение `a[] & c[1..0]` на представленном в примере наборе данных недопустимо, т. к. группы, участвующие в преобразовании, имеют некратную разрядность. Ошибкой является также отсутствие индексного выражения после имени, присвоенного группе. Так на представленном наборе данных недопустима запись `a=c`. Подобное присвоение следует записывать так: `a[]=c[]`.

Для логических групповых данных кроме "обычных" логических операций определены арифметические операции и операции сравнения. Операции сравнения возвращают логический нуль (GND), если условия сравнения не выполнены, и логическую единицу (VCC) — в противном случае. Арифметические операции над группами — инверсия, инкремент, сложение и вычитание — предполагают реализацию арифметических операций над кодами аргументов по правилам беззнаковой двоичной арифметики, причем по умолчанию (если не установлена опция `OPTIONS BIT0=MSB`) разряд с наименьшим номером считается младшим.

Примеры. Пусть `res`, `arg1`, `arg2` — группы одинаковой разрядности, `a` и `c` — скаляры. Тогда допустимы такие операторы:

```
a=(arg[ ]>arg[ ]) !&c;
res[5..0]=arg1[ ] + arg2[ ];  res[]=in[ ]-H"A7";
increm_value[ ]=init_value[]+1.
```

Кроме одномерных и двумерных групп, определяемых в предпроектной секции и подсекции переменных и называемых *постоянными*, программист может объединять скалярные переменные и другие постоянные группы во *временные* группы (в материалах фирмы Altera — `Sequential`, что переводится как последовательные). Временная группа не требует декларации и непосредственно входит в выражения или является приемником в операторе присваивания. BNF-форма временной группы имеет вид:

<временная_группа> ::=(<имя_переменной> «,<имя_переменной>»)

Временная группа определена только в пределах текущего оператора и объединяет с целью описания одинаковых преобразований несколько логических переменных или других групп. Фактически, выполняется операция конкатенации компонентов. Например, при реализации присвоения

```
(d,a[2..0])=b[4..1] # (x,y,u,v);
```

будет получено `a[0]=b[1] # v; a[1]=b[2] # u; a[2]=b[3] # y; d=b[4] # x.`

3.4.3. Оператор выбора и оператор условия

Оператор условия (`IF-THEN STATEMENT`) и оператор выбора (`CASE STATEMENT`) по смыслу и даже синтаксически не отличаются от подобных конструкций языка VHDL *(см. разд. 3.2.6)*. Тем не менее, AHDL предоставляет некоторые дополнительные возможности.

В операторе `IF-THEN` условием может быть не только выражение сравнения, но и любое логическое выражение, дающее скалярный логический результат. Так конструкция `IF a & b THEN` эквивалентна `IF a&b==VCC THEN`, и даже

IF a&b==1 THEN. В последнем случае единица — это, в сущности, константная группа из одного разряда. Впрочем, на наш взгляд, вторая версия из приведенных представляется предпочтительной, т. к. обеспечивает "ясность" прочтения программы.

Другая особенность AHDL — возможность не записывать все альтернативные варианты. Если какая-либо переменная не определяется ни в одном из альтернативных вариантов операторов, то при возникновении соответствующих условий эта переменная принимает так называемое *базовое значение*. По умолчанию базовое значение — это логический нуль GND для скалярных переменных и строка нулей для групп. Однако базовое значение может быть определено также в подсекции умолчания. Указанные особенности иллюстрируются программой, представленной листингом 3.60. Если a и b примут значение нуля, выход получит значение, заданное в разделе умолчаний. Если бы раздел DEFAULTS отсутствовал, то при тех же условиях на выход выдавался бы нулевой код.

Листинг 3.60

```
SUBDESIGN defalt_example
( a,b:INPUT;
    e[3..0]:INPUT
   c[3..0]:OUTPUT)
BEGIN
  DEFAULTS c[]=B"1111";
  END DEFAULTS;
   if a then c[]= e[];
     elsif b then c[]=e[]+ B"1110";
   end if;
```

Использование DEFAULTS позволяет подобным образом сократить запись оператора CASE, правда, в операторе CASE такое же сокращение дает конструкция "WHEN OTHERS".

Еще один способ сокращения длины записи — использование логических покрытий. *Логическое покрытие* определяет совокупность комбинаций аргументов, на которых логические выражения принимают одинаковое значение. Для записи логического покрытия используют символ 'x' — неважное значение (do not care). Запись B"11XX" покрывает комбинации аргументов, имеющих две единицы в старших разрядах. Использование логического покрытия в операторе CASE иллюстрируется описанием приоритетного шифратора, представленным листингом 3.61. Здесь присвоение значения порту request_exist явно присутствует только в одном из вариантов, который выбирается, если все входы — логические нули. В остальных случаях принимается значение по умолчанию. Присвоение переменной неопределенного значения (строка priority_level=B"XX" в рассматриваемом примере) предоставляет компилятору возможность выбрать произвольное значение с целью оптимизации схемной реализации.

Листинг 3.61

```
SUBDESIGN priority
( ena:input= VCC;-- разрешение запроса
  interupt_request[3..0]: INPUT; -- линии входа запроса
  priority_level[1..0]    : OUTPUT; -- код запроса
  request_exist: OUTPUT  -- есть запрос
)
BEGIN
 DEFAULTS request_exist=VCC;
 END DEFAULTS;
 CASE (ena,interupt_request[]) IS
WHEN B"0XXXX",B"10000" =>  request_exist[]=GND
                           priority_level[]=B"XX";
    WHEN B"11XXX" =>priority_level=B"11";
    WHEN B"101XX" =>priority_level=B"10";
    WHEN B"1001X" =>priority_level=B"01";
    WHEN B"10001" =>priority_level=B"00";
    END CASE;
END;
```

3.4.4. Таблицы в AHDL

Таблицы истинности часто позволяют компактно представлять логику функционирования как комбинационных, так и регистровых схем, в том числе цифровых автоматов. В языке AHDL введен специальный оператор таблицы (TRUTH TABLE STATEMENT), обеспечивающий возможность описания нескольких логических функций многих переменных в табличной форме.

```
<оператор таблицы> ::=
  TABLE
    <список аргументов> => <список выходов>;
    <список значений аргументов>=><список результирующих значений>;
    «<список значений аргументов> => <список результирующих значений>;»
  END TABLE;
<список аргументов> ::=<аргумент>| «, <аргумент>»
<аргумент> ::=
  <имя_порта>|<имя_переменной>
  |<состояние автомата>
  |<имя встроенного модуля>.<имя порта модуля>
<список выходов> ::=<выход> «, <выход> »
```

```
<выход> ::= <имя_порта>|<имя_переменной>|<состояние автомата>
  | <имя встроенного модуля>.<имя порта модуля>
<список результирующих значений> ::=
<константное выражение> «, <константное выражение> »
```

Каждая строка оператора таблицы, кроме первой, сопоставляет набору значений аргументов набор результатов. Значения в списках значений записываются в том же порядке, что и имена данных в списках аргументов и выходов, в первой строке таблицы, причем значения записываются в форматах, соответствующих типам переменных в списках. Вопросы включения в таблицы состояний автоматов и портов встроенных модулей рассматриваются далее. Логический нуль и логическая единица в таблицах записываются в укороченной форме — как цифра 0 и 1, соответственно. Число строк не ограничено, хотя, очевидно, не превышает общего числа комбинаций аргументов. Сокращение длины таблицы обеспечивается использованием базовых значений, а также логических покрытий комбинаций аргументов. Применение символов 'X' в разделе результирующих значений разрешает компилятору для реализации соответствующего условия использовать любое значение с целью минимизации. В качестве примера запишем оператор таблицы, описывающий такую же логическую функцию, что и оператор CASE в листинге 3.61 в предыдущем разделе.

```
TABLE
    ena, interupt_request[]     => request_exist, priority_level[];
    0,      B"XXXX"         =>      0,              B"XX";
    1,      B"0000"         =>      0,              B"XX";
    1,      B"1XXX"         =>      1,              B"11";
    1,      B"01XX"         =>      1,              B"10";
    1,      B"001X"         =>      1,              B"01";
    1,      B"0001"         =>      1,              B"00";
END TABLE;
```

3.4.5. Оператор повторения

Оператор повторения (FOR-GENERATE STATEMENT) в AHDL присутствует только в одной модификации — с априорно заданным числом повторений. В фактической реализации производится не просто повторение действия как такового, а реализация совокупности единообразно описанных компонентов.

Синтаксис оператора повторения имеет вид:

```
<оператор повторения>::=
  FOR <переменная цикла> IN <диапазон> GENERATE
                <оператор> «<оператор>»
END GENERATE;
<диапазон> ::=<граница> TO <граница>
```


Переменная цикла — любое имя, не совпадающее с ключевыми словами или другими идентификаторами в программе. Переменная цикла не требует специального объявления и всегда является целой.

Во многих случаях повторяющиеся действия (фрагменты) представимы простыми операторами над группами. Поэтому в AHDL-программах оператор FOR-GENERATE следует использовать при описании рекуррентных алгоритмов, когда результаты исполнения некоторых циклов (фактически, выход структуры, сгенерированной на одном из проходов цикла компилятором) используются в качестве исходных данных для последующих циклов.

В листинге 3.62 представлен пример описания узла контроля на четность входного байта. Для каждого *i*-го разряда формируется сигнал odd_carry[i+1], являющийся признаком четности совокупности этого разряда и всех предыдущих.

Листинг 3.62

```
TITLE "odd_checker";
CONSTANT length=8; -- число разрядов
(  odd_in:INPUT; -- вход расширения
   data[length-1..0]:INPUT;
   odd_result:output
)
VARIABLE odd_carry[length..0]:node;
BEGIN odd_carry[0]=odd_in;
      FOR i IN 0 TO length-1 GENERATE
            odd_carry[i+1]= odd_carry[i] $ data[i];
      END GENERATE;
  odd_result=odd_carry[length];
end;
```

Завершая обзор основных операторов языка AHDL, укажем, что допустимы вложения операторов. Например, оператор повторения может включать вложенный оператор повторения, оператор выбора и т. д.

3.4.6. Описание регистровых схем

Включение в проект триггерных устройств (как отдельных триггеров, так и регистров) выполняется подобно вызову других библиотечных модулей, что в AHDL трактуется как вызов функции. Пока мы ограничимся только одним из способов описания включения триггеров и регистров в проект — *декларацией вхождения*. В *разд. 3.4.9* будет представлен еще один способ, называемый *прямым вызовом функции*. Там же будет подробно описан вызов функций, в том числе формальные определения синтаксиса соответствующих выражений.

Вхождение регистра или отдельного триггера должно быть объявлено в разделе деклараций переменных следующим образом:

```
<Вхождение регистра> ::=
    <имя> [ [<граница>..<граница>] ] : <тип триггера>;
<тип триггера>    ::=
   LATCH | DFF | DFFE | JKFF | JKFFE | SRFF | SRFFE | TFF | TFFE
```

Регистровая схема определяется как группа триггеров, т. е. с указанием диапазона индексов.

В табл. 3.12 сведены триггеры, содержащиеся в библиотеке примитивов пакета MAX+PLUS II и доступные к использованию в AHDL-программах.

***Таблица 3.12.** Триггеры, содержащиеся в библиотеке примитивов пакета MAX+PLUS II*

Тип регистра	Список контактов
LATCH	d, ena, q
DFF	d, clk, clrn, prn, q
DFFE	d, clk, clrn, prn, ena, q
JKFF	j,k, clk, clrn, prn, q
JKFFE	j,k, clk, clrn, prn, ena, q
JKFF	s,r, clk, clrn, prn, q
JKFFE	s,r, clk, clrn, prn, ena, q
TFF	t, clk, clrn, prn, q
TFFE	t, clk, clrn, prn, ena, q

В таблице, кроме имен триггеров, приведены списки предопределенных портов, где:

- clk — синхросигнал (определено динамическое управление, причем положительный фронт активный);
- clrn — асинхронный сброс (активный логический ноль);
- prn — асинхронная установка (активный логический ноль);
- ena — разрешение работы (ena=VCC разрешает запись данных в триггерах LATCH и разрешает прохождение сигнала clk в остальных типах триггеров);
- d,j,k,r,s — информационные входы в соответствии с общепринятым обозначением информационных входов триггеров [3, 15, 27];
- q — выход триггера.

Функция переходов однозначно задается типом триггера. В логической секции сигналы на входах триггеров определяются логическими выражениями. При этом вход представляется как последовательность имени регистра или триггера и имени контакта (в соответствии с табл. 3.12), разделенных точкой. Если в программе отсутствует присвоение какому-либо из управляющих входов, то по умолчанию считается, что на этот вход подана логическая единица.

Выходы триггеров, представляемые в форме <имя>.q, могут использоваться как аргументы в любых логических выражениях. Если выход триггера или регистра является также и выходом схемы, можно применить два варианта: или ввести порт со своим именем, а в логической секции выполнить присвоение порту значения выхода триггера, или объявить одно и то же имя триггера дважды: сначала в предпроектной секции со спецификацией OUTPUT и, кроме того, в секции переменных со спецификацией типа триггера.

Указанные правила иллюстрируются программами, приведенными в листингах 3.63 и 3.64. Модуль, описанный листингом 3.63, представляет восьмиразрядный регистр с асинхронным сбросом и синхронной загрузкой, причем загрузка выполняется при единичном сигнале на входе load. Отметим, что отсутствие присвоения значения входу prn означает, что этот вход пассивен.

Программа из листинга 3.64 описывает счетчик с тремя состояниями с асинхронным сбросом, построенный на J-K-триггерах (JKFF). Аналог этой схемы на дискретных компонентах представлен на рис. 3.25. Отметим, что практическая реализация в микросхемах фирмы Altera отличается от представленной на рисунке. Триггеры типа J-K компилятор преобразует в соответствующую реализацию на D-триггерах, гарантируя при этом функциональную совместимость.

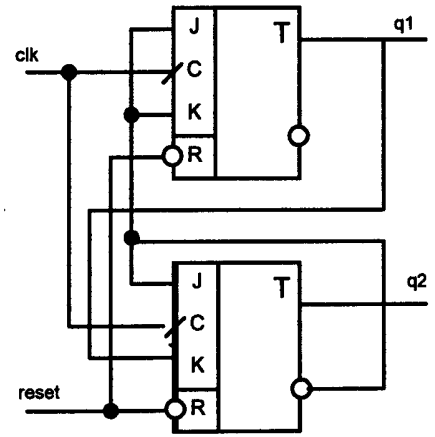

Рис. 3.25. Счетчик с тремя состояниями (аналог программы в листинге 3.64)

Листинг 3.63

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

Листинг 3.64

```
SUBDESIGN count3
( clk, reset:INPUT;
  q1,q2: OUTPUT)
VARIABLE
  q1,q2:jkff;

BEGIN
  q1.clk=clk;
q1.clrn=reset;
  q2.clk=clk;
  q2.clrn=reset;
  q1.j=!q2.q;
  q1.k=!q2.q;
  q2.j= q1.q;
  q2.k= q2.q;
END;
```

Нижеприведенные программы описывают регистровые устройства, ориентированные на реализацию арифметических действий с сохранением результатов. Программа, представленная листингом 3.65, описывает реверсивный счетчик. По умолчанию его разрядность равна 8 и задана в разделе параметров. Однако при включении этого модуля в качестве компонента в иерархические структуры могут задаваться иные значения *(см. разд. 3.4.9)*. Изменение состояния счетчика выполняется по фронту сигнала `clk`, если при этом присутствует входной перенос (`carry_in:=VCC`), или контакт `carry_in` вообще не используется. После переполнения счетчик переходит в нулевое состояние. Выходной перенос не предусмотрен; в принципе, его роль может исполнять старший разряд счетчика, естественно, с потерей эффективной разрядности.

Листинг 3.65

```
PARAMETERS
(n=8 — разрядность
);
SUBDESIGN revers_count
( clk:INPUT,
reset,up_down,carry_in:input=VCC;
  q[n..0]: OUTPUT
)
VARIABLE
  q[n..0]:dff;
```



```
BEGIN
  q[].clk=clk & carry_in;
  q[].clrn=!reset;
  IF up_down==vcc THEN
    q[].d=q[].q+1;
  ELSE q[].d=q[].q-1;
  END IF;
END;
```

Программа, представленная в листинге 3.66, описывает накапливающий сумматор и, на наш взгляд, не требует специальных комментариев.

Листинг 3.66

```
PARAMETERS (n=8);
SUBDESIGN accum_add
( clk,.reset:INPUT;
  data[n-1..0]:INPUT;
  sum[n-1..0]: OUTPUT;
  overflow : OUTPUT -- переполнение
)
VARIABLE
  q[n..0]:dff;
BEGIN
  q[].clk=clk;
  q[].clrn=reset;
    q[].d=q[].q+(GND,data[]);
    sum[]=q[n-1..0].q;
    overflow=q[n].q;
END;
```

3.4.7. Монтажная логика и буферные примитивы

Специфический класс встроенных функций языка AHDL — это так называемые *примитивы буферов*. Кроме достаточно универсального модуля TRI, используемого для представления схем с тремя состояниями на выходе, имеется ряд "приборно-ориентированных" примитивов, позволяющих включать в проект специфические ресурсы микросхем программируемой логики фирмы Altera.

В общем случае, *монтажная логика* описывается путем записи в программе нескольких операторов, присваивающих значение одной и той же переменной или группе. Такое присвоение интерпретируется как объединение по логике ИЛИ логических выражений, соответствующих правым частям этих операторов.

Так, совокупность присвоений

```
Wired_or= select[1] & a1;
Wired_or= select[2] & a2:
Wired_or= select[3] & a3;
```

воспроизводит функции монтажного ИЛИ для трех сигналов a1, a2, a3, выбираемых кодом select[], и эквивалентна

```
Wired_or=select[1] & a1 # Wired_or # select[2] & a2 # select[3] & a3;
```

Заметим, что если ни один из сигналов не выбран, принимается базовое значение, т. е. GND.

Если какой-либо драйвер дает высокоимпедансное состояние, то соответствующее присвоение игнорируется, за исключением случая, когда все драйверы в высокоимпедансном состоянии — тогда выход принимает значение 'z' (это правило подобно правилам преобразования, установленном для данных подтипа STD_LOGIC в языке VHDL). Высокоимпедансное состояние не может быть присвоено сигналу через "обычное" логическое выражение или таблицу. Это состояние могут принимать только выходы примитивов TRI, OPDRN или порт, специфицированный как BIDIR.

Примитив TRI имеет два входных порта и один выходной. Первый вход IN — вход данных, второй OE — управляющий. При логической единице на входе OE — выход повторяет вход данных, в противном случае находится в высокоимпедансном состоянии. *Примитив OPDRN* имеет один вход и один выход. Если на вход поступает логический нуль, то на выходе присутствует логический нуль, в противном случае выход находится в высокоимпедансном состоянии.

Программа (листинг 3.67) иллюстрирует подключение двух источников (регистры v[0][] и v[1][]) на общий выход через буферы с тремя состояниями выхода под управлением кода select[]. Обратите внимание на то, что фактически присвоение значения переменной io в программе выполняется дважды, и, кроме того, допустимы присвоения значения соответствующему сигналу в модуле высшего уровня иерархии.

Листинг 3.67

```
SUBDESIGN Double_bus_reg
(
    clk     : INPUT;
    load[1..0],select[1..0]     : INPUT;
    io[7..0]          : BIDIR;
)
  VARIABLE v[1..0][7..0]:NODE; -- два восьмиразрядных регистра
     W[1..0][7..0]:TRI;    -- два трехстабильных восьмиразрядных буфера
```



```
BEGIN
 FOR j IN 0 TO 1 GENERATE
 FOR i IN 0 TO 7 GENERATE
  v[j][i]=DFF(io[i], clk, load[j],);
  w[j][i].oe=select[j];-- каждый буфер управляется своим сигналом
                       -- разрешения
  w[j][i].in=v[j][i];  -- и имеет свой вход

  io[i] = w[j][i].out;-- здесь представлено по два присвоения значения
              -- каждому биту выходного порта от одного из двух буферов.
 END GENERATE;
 END GENERATE;
END;
```

Замечание

Здесь для описания подключения регистров использована конструкция "In-Line-Function Reference" (вызов функции в строке) — подробнее об этом *см. разд. 3.4.9.*

Примитивы CARRY и *CASCADE* (в языке — соответствующие функции) явно указывают компилятору на необходимость использования специфических ресурсов микросхем семейств FLEX и APEX — цепочечного логического переноса и логики прямой связи смежных элементов [4, 31]. Применение этих ресурсов может обеспечить уменьшение времени задержки в логических схемах, реализованных на основе последовательной декомпозиции логических выражений, а также уменьшить затраты коммутационных ресурсов микросхемы. Хотя синтаксически оба примитива имеют один формальный входной параметр и возвращают один результат, в качестве фактического параметра этих функций можно использовать логические выражения и таким образом определять сравнительно сложную логику связи смежных макроячеек.

Фактический параметр примитива CASCADE обязательно записывается в форме логической операции И или ИЛИ, один из операндов которой это константа или выход другого примитива CASCADE, а второй операнд — произвольная логическая функция. Возвращаемое значение функции CASCADE присваивается переменной типа NODE, которая может стать аргументом следующего примитива CASCADE или использоваться в одном (и только одном) логическом выражении в операторе присваивания. Во втором случае эта переменная должна объединяться по логике И либо ИЛИ с произвольной логической функцией для формирования сигнала. В БИС семейств FLEX такая запись интерпретируется структурой, показанной на рис. 3.26. Здесь представлены две макроячейки, объединенные через каскадирующий буфер. Узел каскадирования, обозначенный C, в зависимости от записи фактического параметра примитива CASCADE выполняет операцию И либо ИЛИ, а

LUT реализует произвольную функцию четырех аргументов. Принцип использования примитива CASCADE иллюстрируется программой (листинг 3.68), которая представляет узел проверки двух шестнадцатиразрядных кодов на равенство. Несовпадение любой пары одноименных битов дает логическую единицу, распространяющуюся по цепочке каскадированных макроячеек.

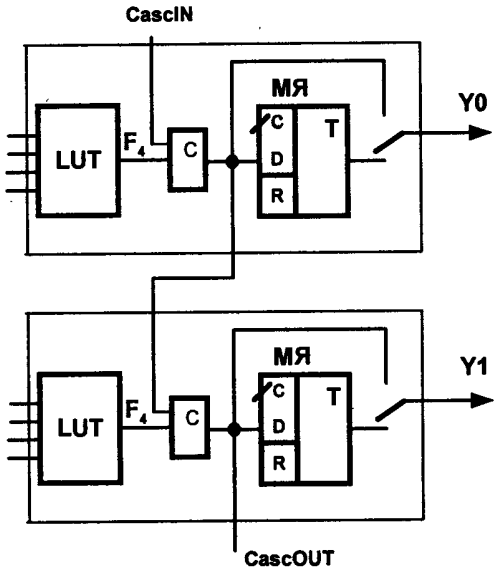

Рис. 3.26. Пример каскадирования макроячеек

Листинг 3.68

```
SUBDESIGN comp_casc
( a[15..0],b[15..0]:INPUT;
  result:OUTPUT;
)
VARIABLE temp[7..0]: NODE;
BEGIN
 temp[0]=GND;
 FOR j IN 0 TO 6 GENERATE
  temp[j+1]= cascade( (a[2*j] $ b[2*j])
       # (a[2*j+1] $ b[2*j+1]) # temp[j]);
 END GENERATE;
 result= temp[7]#(a[14] $ b[14])
       # (a[15] $ b[15]);
END;
```


Логическое выражение, используемое как формальный параметр примитива CARRY, может содержать произвольную логическую функцию трех переменных, одна из которых является выходом другого, следующего в цепочке, примитива CARRY. Другая функция тех же трех переменных может использоваться как независимый выход. Наиболее часто логика CARRY служит для создания арифметических устройств. Любой i-й ячейке такой цепочки следует сопоставить оператор:

$$\mathrm{Out}_i.d = F_{1i}(a_i, b_i, c_i);\ \ c_{i+1} = \mathrm{CARRY}\,(F_{2i}(a_i, b_i, c_i));\ \ i = (0, n-1);$$
$$c_0 = \mathrm{const},$$

где F_{1i} и F_{2i} — логические функции, реализуемые первой и второй настраиваемыми логическими схемами LUT соответствующей макроячейки.

Примитивы LCELL *и* SOFT служат для указания компилятору на необходимость или возможность включения макроячеек в структуру реализованного устройства.

Вызов функции LCELL гарантирует, что выражение, определяющее фактический параметр, будет воспроизводиться отдельно выделенной макроячейкой. Например, выражение

```
x= a & b # c & d;
```

и выражение

```
x= LCELL( a & b) # LCELL( c & d);
```

реализуют одну и ту же функцию. Но первая воспроизводит ее на одной макроячейке, а вторая на трех (два элемента И и один ИЛИ). Явное выделение части выражения примитивом LCELL позволяет наблюдать результат реализации соответствующей подфункции в сеансе моделирования. Дело в том, что если даже объявить вспомогательную переменную для представления части логического выражения, но записать ее формирование с помощью "обычного" присвоения, а не с использованием примитива LCELL, то в проекте такая переменная может "исчезнуть" в результате поглощения в другом выражении. Явная декомпозиция и использование LCELL могут быть полезны также для выравнивания задержек распространения сигналов по разным путям в схеме. Правда, этой возможностью следует пользоваться весьма осторожно, т. к. время распространения зависит не только и не столько от числа ячеек в цепи передачи, сколько от расположения логических ячеек и распространения сигналов по цепям связи.

Примитив SOFT подобен LCELL, но предоставляет компилятору возможность использовать или не использовать выделенную ячейку для реализации логического выражения, являющегося фактическим параметром функции LCELL. Это в некоторой степени рекомендует компилятору способ декомпозиции

сложных выражений, если такая декомпозиция потребуется. Кроме того, присвоение вида

```
<переменная>= SOFT(<выражение>)
```

делает переменную в правой части оператора наблюдаемой в сеансах моделирования, несмотря на то, что в реализации эта переменная может быть поглощена.

В целом, как отмечалось, использование приборно-зависимых буферных примитивов обеспечивает разработчику возможность частичного управления процедурой синтеза.

3.4.8. Цифровые автоматы

В AHDL цифровой автомат определен как переменная перечислимого типа. Имя автомата и список допустимых значений этой переменной (состояний автомата) определяются в разделе переменных:

```
<декларация цифрового автомата> ::=
<Имя_автомата>: MACHINE [ OF BITS (<бит состояния> «,<бит состояния>») ]
    WITH STATES (<список состояний>);
<список состояний> ::=
        <Имя состояния> [ = <код состояния> ]
        «,<имя состояния> [ =<код состояния> ] »
<Имя состояния> ::=<идентификатор>
```

У любого автомата по умолчанию определены три управляющих входа:

- <Имя_автомата>.reset — подача логической единицы на этот вход безусловно переводит автомат в исходное состояние, т. е. первое из перечисленных в списке состояний;
- <Имя_автомата>.clk — автомат может изменять свое состояние по положительному фронту этого сигнала в соответствии с определенной в логической секции функции переходов;
- <Имя_автомата>.ena — сигнал разрешения работы; изменение состояния (кроме сброса) может происходить только при единичном сигнале на входе ena. Если вход не используется, переключение автомата разрешено.

Число состояний определяется алгоритмом работы и выполняется на этапе содержательного представления проекта. Каждому состоянию присваивается уникальное имя. Задание кода состояния не обязательно. Отметим, что методические материалы фирмы Altera [31] рекомендуют использовать сокращенную форму (без явного присвоения состояниям кодов в списке состояний и использования ключевого слова OF BITS), предоставляя назначение кодов состояния компилятору пакета MAX+PLUS II. Полагают, что автоматическое назначение состояний обеспечит лучшую оптимизацию проекта

в соответствии выбранным разработчиком критериями качества. В таком случае влияние разработчика на выбор способа кодирования реализуется путем задания опций проекта в системе MAX+PLUS II. Может быть задано либо двоичное кодирование состояний, либо кодирование по принципу "одно состояние — один бит". Двоичное кодирование обеспечивает уменьшение числа триггеров в регистровом блоке автомата, а кодирование "одно состояние — один бит" — упрощение логической части схемы. По умолчанию для микросхем семейств MAX используется первый способ (двоичное кодирование), а для семейств FLEX — второй.

Функция переходов задается набором операторов, присваивающих переменной, представляющей автомат, значения (идентификаторов состояний) из числа объявленных в списке состояний. При этом могут использоваться и логические выражения, операторы IF-THEN, CASE, а также оператор таблицы и их сочетания.

В языке AHDL входы и выходы — это обязательно логические переменные (AHDL, в отличие от VHDL, не допускает объявления входов и выходов как данных перечислимого типа, но, в отличие от Verilog HDL, в нем возможно и даже рекомендуется задавать набор состояний их именами, а не кодами).

Описание автоматов Мура и автоматов Мили с асинхронными выходами совпадают по форме и отличаются только логическими соотношениями, задающими функции выходов. В автоматах Мура выход зависит только от состояния, а в автоматах Мили и от входа. Для представления автомата Мили с синхронизируемыми выходами следует его выходные переменные специфицировать как триггеры, а на входы синхронизации и установки этого "дополнительного" регистра подать те же сигналы начальной установки и синхронизации, что и на остальные управляющие входы автомата. В листинге 3.69 приведено описание автомата Мили, заданного графом переходов (рис. 3.27), с использованием оператора таблицы.

Замечание

В ряде работ, в том числе [3] и материалах фирмы Altera, автомат Мили с асинхронными выходами назван просто "автоматом Мили", каноническая версия автомата Мили не рассматривается, да и автомат Мура определен иначе, чем принято здесь. Авторы пользуются классификацией автоматов, введенной в работах [15, 24], и не вправе обсуждать терминологию в пределах данной книги, но обращают внимание читателя на возможные разночтения.

Листинг 3.69

```
SUBDESIGN my_meal
( clock, reset, xin: INPUT;
 y[1..0]: OUTPUT)
```

```
VARIABLE
    meal: MACHINE WITH STATES
                  (s0,s1,s2);
    y[1..0]: dff; --***
 BEGIN
   meal.reset=!reset;
   meal.clk=clock;
   y[].clrn= !reset;
   y[].clk=clock;
TABLE
meal,    xin    =>    meal,    y[].d;
s0,    0    =>    s0,    B"00";
s0,    1    =>    s1,    B"01";
s1,    0    =>    s0,    B"01";
s1,    1    =>    s2,    B"10";
s2,    0    =>    s2,    B"01";
s2,    1    =>    s1,    B"10";
END TABLE;
END;
```

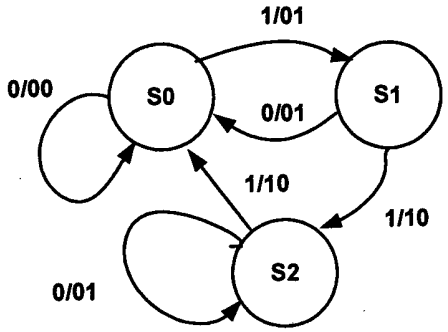

Рис. 3.27. Граф переходов и выходов автомата Мили (пример)

Рис. 3.28 иллюстрирует способ реализации этого автомата при компиляции в микросхемы семейства FLEX (принято унитарное кодирование состояний) и является расшифровкой файла отчета результата компиляции в системе MAX+PLUS II. Компилятор назначил такие коды состояний: `s0 = B"000"`, `s1 = B"110"`, `s2 = B"101"`. Функции логики синтезированы в форме:

```
F1 = xin & m2 # !xin & m1;
F2 = xin & m1 # !xin & m3;
F3 = xin & m2 # !m2 & m3;
F4 = xin & m2 # !xin &m1;
F5 = xin & m3 # !xin &(m1 # m2);
```


Если из программы исключить строку, отмеченную комментарием "***", то получим автомат Мили с асинхронными выходами, в котором выходы реализуются комбинационной логикой.

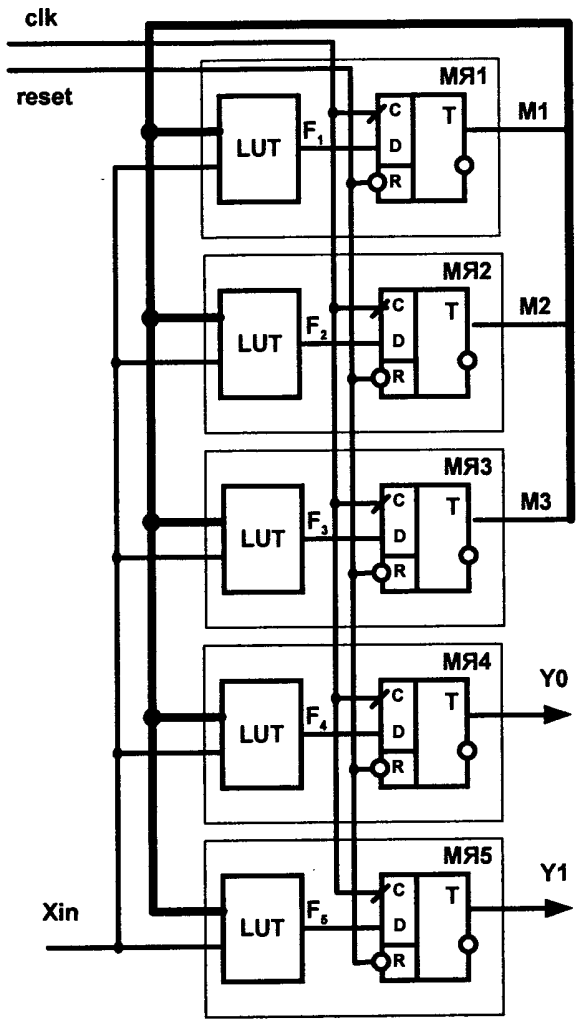

Рис. 3.28. Реализация автомата по листингу 3.69 в микросхемах типа FLEX10K

В листинге 3.70 представлена программа, интерпретирующая автомат Мура, заданный отмеченной таблицей переходов (см. табл. 3.6). Здесь вход и выход закодированы одним битом каждый, причем X0 и Y0 представлены логическим нулем, а X1 и Y1 — логической единицей.

Листинг 3.70

```
SUBDESIGN my_moor3
( clock, reset, v:INPUT;
 y: OUTPUT)
VARIABLE
moor: machine with
    states(S0,S1,S2);
 BEGIN
moor.reset=reset;
   moor.clk=clock;
CASE moor IS
  WHEN s0=>  IF v==gnd THEN moor= s0;
              ELSE moor= s1;
             END IF;
  WHEN s1=> IF v==gnd THEN moor= s1;
             ELSE moor= s2;
            END IF;
  WHEN s2=> IF v==gnd THEN moor= s0;
            ELSE moor= s1;
            END IF;
 END CASE;
IF moor=s1 THEN y=GND;
           ELSE y=VCC;
          END IF;
END;
```

Явное кодирование состояний разработчиком, как отмечено, рассматривается в рекомендациях фирмы Altera как вспомогательная опция. В частности, это может быть полезно, если выходы регистра состояний автомата непосредственно используются как выходные сигналы, т. е. реализуется автомат Мура без выходной логики. Биты кода состояний именуются и могут использоваться в качестве аргументов логических выражений логической секции. Кроме того, допустимо бит кода состояния использовать как выход проекта. В этом случае имя бита объявляется дважды — при декларации автомата и декларации порта. Интересно, что возможно определить число битов состояния, меньше необходимого для присвоения кода всем состояниям. В этом случае объявленные биты будут "привязаны" к коду состояния, остальные же произвольно доопределятся компилятором.

Программа (листинг 3.71) интерпретирует тот же автомат, что и программа (листинг 3.70), но здесь использовано явное задание одного из двух битов кода состояния, а также табличное представление функции переходов.

Листинг 3.71

```
SUBDESIGN my_moor3a
( clock, reset, v:INPUT;
 z: OUTPUT)
VARIABLE
moor: machine of bits (z)
     with states(s0= B"1",
           S1= B"0",
           S2= B"1");
 BEGIN
 moor.reset=reset;
   moor.clk=clock;
TABLE
moor, v     =>     moor;
  s0, 0     =>      s0;
  s0, 1     =>      s1;
  s1, 0     =>      s1;
  s1,  1     =>      s2;
  s2,  0     =>      s0;
  s2,  1     =>      s1;
END TABLE;
END;
```

В приведенных примерах подключение автомата в сложные проекты предполагается путем передачи через порты сигналов. Используются спецификации портов INPUT и OUTPUT. Кроме того, AHDL позволяет непосредственно экспортировать и импортировать состояния автоматов. Тогда один из портов модуля-источника, представляющего управляющий автомат, специфицируется как MACHINE OUTPUT, а порт модуля-приемника — как MACHINE INPUT. Пример иерархического проекта, один из модулей которого экспортирует, а другой импортирует состояние автомата, рассмотрен далее.

3.4.9. Иерархическое проектирование в AHDL

Любой проектный модуль может использоваться в качестве компонента в другом проекте. Для этого текст описания модуля верхнего уровня иерархии должен содержать ссылку на прототип включаемого модуля (подпроекта) и операторы, описывающие соединения включаемого модуля с прочими компонентами проекта. Ссылка в форме записи оператора объявления прототипа (Function Prototype Statement) может содержаться либо непосредственно в секции заголовка программы, либо в специальном файле. Такой "подключаемый" файл обычно имеет расширение inc и создается в тех случаях, когда несколько проектов содержат сходные разделы в секции заголовка (Tytle Section).

Подключаемый файл может создаваться с помощью обычных текстовых редакторов. Кроме того, в системе MAX+PLUS II включаемый файл с декларацией прототипа любого проектного модуля может создаваться автоматически. Достаточно, имея в окне редактора системы текст описания этого модуля, вызвать к исполнению команду **Create Default Include File** меню **File**. Имя создаваемого в этом случае подключаемого файла совпадает с именем SUBDESIGN включаемого проектного модуля и, соответственно, с именем текстового файла, содержащего описание этого модуля.

Для доступа к информации, заключенной в подключаемом файле, модуль высшего уровня иерархии должен содержать оператор включения (Include Statement) файла, записываемый в форме:

```
<оператор включения> ::=
   INCLUDE "[ <путь к файлу> ] <имя подключаемого файла>.inc";
```

Текст подключаемых файлов может содержать любой набор операторов, допустимых в секции заголовка проекта, в том числе объявление прототипа. Этот текст как бы замещает при компиляции оператор включения. Путь к файлу — традиционная для DOS запись локализации файла в системе.

В общем случае синтаксис оператора объявления прототипа имеет вид:

```
<оператор объявления прототипа> ::=
    FUNCTION <имя подпроекта> ( <список входов>)
       [  WITH ( <список параметров>)]
       RETURNS ( <список выходов>);
```

Список входов содержит имена портов, содержащихся в предпроектной секции прототипа и специфицированных как INPUT или MACHINE INPUT, список выходов содержит имена портов, специфицированных в описании прототипа как OUTPUT, BIDIR или MACHINE OUTPUT. Список параметров содержит имена, содержащиеся в разделе PARAMETERS текстового файла, описывающего прототип. Имена входов, выходов и параметров в декларации прототипа точно совпадают с именами соответствующих данных в программе, описывающей модуль-прототип, и заносятся в списки через запятые, причем порядок записи в списках не имеет значения. Например, если в проект включается модуль accum_add, представленный программой (листинг 3.66), следует включить в текст описания проекта декларацию прототипа (или оператор включения файла, содержащего такую декларацию) следующего вида:

```
FUNCTION accum_add (clk, reset, data[(n - 1)..0])
  WITH (n)
  RETURNS (sum[(n - 1)..0], overflow);
```

Для модулей, поставляемых со стандартной библиотекой системы MAX+PLUS II, в системе присутствуют соответствующие подключаемые

файлы, доступные к использованию в любых проектах, выполняемых в этой среде. Оператор включения такого файла в программе, использующей библиотечный модуль, обязателен, хотя путь можно не указывать. Прототипы примитивов (триггеров, буферных схем) автоматически подключаются к любому проекту и дополнительной декларации не требуют.

Каждое конкретное использование некоторого компонента называют вхождением. Конкретизация соединений вхождения, а при необходимости также и параметров настройки, возможно двумя способами: через использование *декларации вхождения* (Instance Declaration) или через *прямой вызов функции* (In-Line-Function Reference).

Декларация вхождения размещается в подсекции переменных, т. е. здесь каждому встроенному модулю присваивается собственное имя и объявляется способ его реализации — имя функции прототипа и значения параметров настройки. BNF-форма декларации вхождения имеет вид:

```
<декларации вхождения подпроекта (AHDL)> ::=
    <Имя вхождения> «,<имя вхождения>»:<имя функции>
          [ WITH (<имя параметра> = <значение>
          «,<имя параметра> = <значение>») ]
```

Если для параметризованного модуля в декларации вхождения отсутствует указание значения какого-либо параметра, этот параметр принимает значение, определяемое в тексте описания соответствующего модуля как "значение по умолчанию".

Синтаксис декларации вхождения допускает объявление нескольких однотипных модулей, каждый их которых в проекте получает собственное имя. Отметим, что имя вхождения может содержать индексное выражение, т. е. быть именем группы. Такая запись соответствует декларации группы однотипных модулей, которые имеют одинаковое имя, но различные индексы.

Переменная, представляющая состояние порта включенного модуля, записывается следующим образом:

```
<имя порта вхождения> ::= <имя вхождения>.<имя порта прототипа>
```

и может использоваться в логической секции подобно другим переменным. Ограничение лишь в том, что входной порт вхождения может быть только приемником, т. е. записываться лишь в правых частях операторов присваивания, а выходной, наоборот, не может входить в правые части операторов присваивания, а используется в логических выражениях, включая левые части операторов присваивания.

Как имя вхождения, так и имя порта прототипа в записи такой переменной могут содержать индексное выражение. Индексное выражение в имени вхождения должно присутствовать, если в декларации вхождения объявлена группа встраиваемых модулей. Индексное выражение в этом случае опреде-

ляет конкретный модуль или подгруппу из группы модулей, соединения которых представлены данным оператором. Индексное выражение в имени порта прототипа должно присутствовать, если порт прототипа является группой. Такое индексное выражение определяет, какие разряды или какой из разрядов группы определяются данным оператором.

Использования Instance Declaration иллюстрируется программой, озаглавленной "two register example" (листинг 3.72). Проект содержит два регистра-защелки (lpm_latch), объединенных в группу reg[1..0] и вызываемых из библиотеки параметризованных модулей системы MAX+PLUS II. Каждый регистр загружается с общего входа din по своему синхросигналу. Выход формируется как результат побитового сравнения содержимого регистров в обратном порядке, т. е. старший разряд одного регистра сравнивается с младшим разрядом другого.

Листинг 3.72

```
TITLE "two register example";
FUNCTION lpm_latch (data[LPM_WIDTH-1..0],-- входной код
                    gate,  -- сигнал разрешения записи
                    aclr, aset, aconst) - вспомогательные входы
  WITH (LPM_WIDTH, LPM_AVALUE)
  RETURNS (q[LPM_WIDTH-1..0]); -- выходные данные
PARAMETERS (n=8);
Subdesign double_reg
(load0,load1,reset:input;
          din[n-1..0]: input;
       eq [n-1..0]:output
)
VARIABLE reg[1..0] : lpm_latch WITH (LPM_WIDTH=n);
BEGIN
reg[0].data[]=din[];
reg[0].gate=load0;
reg[1].data[]=din[];
reg[1].gate=load1;
reg[].aclr=reset;
  FOR i IN 0 to n-1 GENERATE
    Eq[i]= reg[1].q[i] !# reg[0].q[n-1-1];
  END GENERATE;
END;
```

Другой способ включения модулей в иерархический проект называют *прямым вызовом функции* (In-Line-Function Reference). Такой вызов не требует декларации вхождений в секции переменных и во многих случаях обеспечивает более компактную запись. Вызов записывается индивидуально для каждого вхождения и размещается в логической секции. Оператор вызова имеет синтаксис:


```
<прямой вызов функции> ::=
    <выходные цепи> = <имя функции> (<список соответствий входов>)
    [ WITH (<Список соответствий параметров>)]
    [ RETURNS (<список соответствий выходов>]
<входные цепи> ::= <имя переменной>
    | (<имя переменной> «,<имя переменной>»
<соответствие входа> ::= [<имя входного порта>=] <имя переменной>
<соответствие параметра> ::= [ .<имя параметра>= ] <константа>
<соответствие выхода> ::= .<имя выходного порта>
```

Уточним, что в случаях, когда функция возвращает несколько значений, фактические переменные в левой части оператора присваивания записываются как временная группа — в круглых скобках, разделенные запятыми. Все списки в рассматриваемом контексте записываются как последовательность элементов списка, разделенных запятыми. Укороченные формы записи списков соответствий предполагают позиционное сопоставление фактических и формальных параметров функций, а полные — сопоставление по имени.

Позиционное сопоставление требует точного соответствия порядка следования имен и параметров в списках порядку их следования в декларации прототипа. Если какой-либо вход или параметр не используется или используется значение по умолчанию, соответствующая позиция в списке отмечается как пустая. То же касается описания входных цепей. При сопоставлении по имени порядок записи не имеет значения, важно лишь совпадение имени формального параметра с именем, указанным в декларации прототипа.

Разные способы прямого вызова непараметризированных модулей иллюстрируются программой, представляющей проект MACRO (листинг 3.73). Проект содержит два четырехразрядных счетчика, первый запускается входным сигналом clk, а второй — сигналом переполнения первого. К выходу второго счетчика подключен дешифратор. Списки входов и выходов первого счетчика и дешифратора представлены с использованием сопоставления по имени, а второго счетчика — через позиционное сопоставление. Отметим, что в этом примере для декларации прототипа использованы библиотечные подключаемые файлы, но в тексте программы декларация прототипа 4count воспроизведена для лучшего ее понимания как комментарий.

Листинг 3.73

```
INCLUDE "4count";
INCLUDE "16dmux";
--FUNCTION 4count (clk, clrn, setn, ldn, cin, dnup, d, c, b, a)
    --RETURNS (qd, qc, qb, qa, cout);
```

```
--FUNCTION 16dmux (d, c, b, a)
   --RETURNS (qn[15..0]);
SUBDESIGN macro
(
    clk1              : INPUT;
    out[15..0]     : OUTPUT;
)
VARIABLE
    q1[3..0], q2[3..0]     : NODE;
    carry1:node;
BEGIN
(q1[3..0],carry)= 4count (.clk=clk,.dnup= GND)
                  returns (.qd,.qc,.qb,.qa,.cout);
 (q2[3..0],) = 4count (carry1,,,,, GND,,,,);
out[15..0]=16dmux(.(d,c,b,a)=q[3..0]);
END;
```

Для иллюстрации задания параметров модулей рассмотрим прямой вызов модуля lpm_add_sub (настраиваемый сумматор-вычитатель) из библиотеки параметризованных модулей системы MAX+PLUS II. Прототип функции для этого модуля имеет вид:

```
FUNCTION lpm_add_sub (cin, dataa[LPM_WIDTH-1..0], datab[LPM_WIDTH-1..0],
add_sub, clock, aclr)
 WITH (LPM_WIDTH, LPM_REPRESENTATION, LPM_DIRECTION, ONE_INPUT_IS_CONSTANT,
LPM_PIPELINE, MAXIMIZE_SPEED)
  RETURNS (result[LPM_WIDTH-1..0], cout, overflow);
```

Пусть на его основе строится восьмиразрядный сумматор в беззнаковом формате, причем его входы и выходы объявлены в проекте как переменные. Тогда прямой вызов с использованием списков с сопоставлением по имени будет иметь вид:

```
(sum[],cout)=lpm_add_sub (.cin=gnd, .dataa[]=a[], .datab[]=b)
  WITH (LPM_WIDTH=8, LPM_REPRESENTATION="UNSIGNED", LPM_DIRECTION="ADD")
RETURNS(.result[],.cout);
```

Вызов функции при тех же условиях, но с позиционным сопоставлением, запишется так:

```
(SUM[],cout,)=lpm_add_sub (gnd, a[], b[],,,)
              WITH (8, "UNSIGNED", "ADD",,,);
```

Пример описания иерархического проекта, в котором передача информации между модулями осуществляется через порты MACHINE OUTPUT и MACHINE INPUT представлен программами mult16_16 (листинг 3.74),

part_mult (листинг 3.75) и mult_contr (листинг 3.76). Проект реализует умножение шестнадцатиразрядных операндов с выдачей 32-разрядного результата. Используется последовательное умножение полуслов входных аргументов и накопление взвешенной суммы частичных произведений в соответствии с алгоритмом

$$OUT[31..0]= in\_a[7..0]\times in\_b[7..0] +(in\_a[7..0]\times in\_b[15..8]+ +in\_a[15..8]\times in\_b[7..0])\times 2^8+ +(in\_a[15..8]\times in\_b[15..8])\times 2^{16}.$$

Здесь используется традиционное при построении операционных устройств разделение на блоки — устройство управления, описанное программой mult_contr, и операционный блок part_mult. Устройство управления после сигнала start последовательно проходит четыре состояния: STEP1, STEP2, STEP3 и STEP4, каждому их которых соответствует определенная фаза вычислительного алгоритма, после чего возвращается в исходное состояние IDLE. Операционный блок принимает сигнал состояния, и в зависимости от фазы цикла вычислений, коммутирует входные полуслова на соответствующие входы встроенного параллельного комбинационного умножителя. В качестве умножителя использован библиотечный модуль LPM_MULT из библиотеки параметризованных модулей САПР MAX+PLUS II. Выходной коммутатор операционного блока позиционирует частичные произведения в разрядной сетке результата в зависимости от фазы цикла вычислений. В программе верхнего иерархического уровня mult16_16 декларируются включаемые модули и переменные, обеспечивающие связи между ними. В том числе вводится переменная, представляющая состояние автомата в устройстве управления и специфицируемая ключевым словом MACHINE. Кроме того, программа mult16_16 содержит описание накапливающего сумматора и триггера готовности ready. Накапливающий сумматор, выход которого является выходом устройства, представлен логическим оператором.

Отметим, что автомат меняет свое состояние по положительному фронту синхросигнала, результат на выходе блока part_mult появится с задержкой относительно этого фронта, складывающейся из задержки автомата, коммутаторов и параллельного умножителя. Чтобы код на входе накапливающего сумматора успел сформироваться к моменту фиксации в нем суммы, сумматор синхронизируется отрицательным фронтом тактового сигнала.

Листинг 3.74

```
INCLUDE "mult_contr.inc";
INCLUDE "part_mult.inc";
SUBDESIGN mult16_16
( clk,reset,start: INPUT;
 in_a[15..0],in_b[15..0]: INPUT; -- входные данные
```

```
 out[31..0]:     OUTPUT; -- произведение
 ready:     OUTPUT -- готовность результата
)
VARIABLE result[31..0]:dffe;
     ready: dff;
     busy: NODE;
     controle:MACHINE; --  декларация "внешнего" автомата
BEGIN
(busy,controle)=mult_contr (clk, reset, start); -- вхождение автомата
result[].d=result[].q+part_mult (in_b[],in_a[],controle);--накопление
result[].ena=busy; --  работа/ожидание
result[].clk=!clk;
out[]=result[];
ready.clk=busy; -- триггер готовности данных
ready.d= VCC;
ready.clrn=start;
END;
```

Листинг 3.75

```
INCLUDE "lpm_mult.inc";
CONSTANT Z_code= b"00000000";
SUBDESIGN part_mult
  (in1[15..0],in2[15..0]:INPUT;
   controle:MACHINE INPUT;
   p_m[31..0] : OUTPUT) -- частичное произведение
   VARIABLE a[7..0],b[7..0],c[15..0]:NODE;
  BEGIN
    c[]=lpm_mult (.dataa[]=a[],.datab[]=b[])
    WITH (LPM_WIDTHA=8, LPM_WIDTHB=8, LPM_WIDTHP=16,LPM_WIDTHS=16);
      CASE controle IS
        WHEN step1 => a[]=in1[7..0];b[]=in2[7..0]; -- умножение
                                                   -- младших байтов
          p_m[]=(z_code,z_code,c[]); -- размещение в младших разрядах
                                     -- результата
        WHEN step2 => a[]=in1[15..8];b[]=in2[7..0]; -- перекрестное
                                                    -- умножение
          p_m[]=(z_code,c[],z_code); -- размещение в средних разрядах
                                     -- результата
        WHEN step3 => a[]=in1[7..0];b[]=in2[15..8]; -- перекрестное
                                                    -- умножение
          p_m[]=(z_code,c[],z_code);
        WHEN step4 => a[]=in1[15..8];b[]=in2[15..8]; -- умножение
                                                     -- старших байтов
```



```
          p_m[]=(c[],z_code,z_code); -- размещение в старших разрядах
                                     -- результата
      END CASE;
  END;
```

Листинг 3.76

```
SUBDESIGN mult_contr
(clk,reset,start: INPUT;
  busy: OUTPUT;
 controle: MACHINE OUTPUT;
)
VARIABLE Int_state:MACHINE WITH STATES
          (idle,step11,step2,step3,step4);
 BEGIN
  int_state.clk=clk;
  int_state.reset=reset;
  controle=int_state;
  TABLE
  int_state, start=> int_state, busy;
   idle   ,    0 => idle,   0;
   idle ,    1 => STEP1,   0;
   step1,     x => step2,   1;
   step2,     x => step3,   1;
   step3,     x => step4,   1;
   step4,     x => idle,   1;
 END TABLE;
END;
```

В приведенных выше программах неоднократно использовались параметризованные модули, поставляемые с библиотеками САПР MAX+PLUS II. А можно ли создать собственный параметризованный модуль? В принципе, да. Но надо иметь в виду трудозатраты на создание параметризованного модуля и на возможные модификации исходного текста описания. Использование параметров количественного типа, например разрядности данных, не требует особых затрат. Намного сложнее предусмотреть и описать возможные модификации структуры и алгоритма функционирования, в том числе контроль корректности задания конфигурации при вызове описания из других модулей в иерархических проектах. Во многих случаях разработка параметризованной версии оказывается неоправданной. Тем не менее, рассмотрим некоторые особенности создания параметризованных модулей.

Параметризованный модуль, кроме обязательных предпроектной и логической секций, содержит оператор объявления параметров, синтаксис которого определен как:

```
<оператор объявления параметров> ::=
    , PARAMETERS (<имя> [ <значение по умолчанию> ]
     «,<имя> [ <значение по умолчанию> ] »);
<значение по умолчанию> ::=<строка> | <выражение>
```

Параметры внутри проектируемого модуля рассматриваются как константы, но их значение может быть установлено при включении модуля в качестве компонента в более сложный проект. Если модуль применяется автономно, или в проекте высшего уровня иерархии некоторые параметры не определены, используется значение по умолчанию.

Параметры, задающие функции, реализуемые конкретными вхождениями параметризованного модуля, чаще всего используются в операторах условной генерации, синтаксис которого определен как:

```
<оператор условной генерации> ::=
  IF <выражение проверки> GENERATE
    <оператор> «<оператор>»
    [  ELSE GENERATE <оператор> «<оператор>»]
  END GENERATE;
```

Оператор условной генерации может присутствовать в логической секции проектного модуля и в подсекции переменных. Аргументами выражения проверки являются параметры конфигурации. Выражение проверки может, например, предусматривать сравнение параметров с константами (равно, не равно и т. п.) или между собой. Если выражение истинно, реализуется первая группа операторов, в противном случае вторая, а в случае отсутствия конструкции ELSE GENERATE вообще никаких. Особый случай — проверка того, используется ли определенный порт при включении модуля в проект высшего уровня. Для такой проверки в AHDL предусмотрена функция USED (<имя>), которая возвращает логическую единицу, если порт с указанным именем определен в проекте высшего уровня, и ноль — в противном случае.

Главное отличие оператора IF-GENERATE от условного оператора IF-THEN состоит в том, что он выполняется только на этапе компиляции проекта, а не в физической реализации. Иными словами, в реализуемом устройстве будет воспроизведена только одна из возможных альтернатив в зависимости от задаваемых в модуле верхнего иерархического уровня значений параметров. Отметим, что, в отличие от VHDL, оператор IF-GENERATE в AHDL может представлять альтернативные реализации.

Полезным инструментом иерархического проектирования является оператор проверки, с помощью которого можно задать контроль корректности задания параметров при компоновке сложных проектов.


```
<оператор проверки> ::=
    ASSERT <выражение проверки> REPORT <строка сообщения>
      [ <переменная сообщения> «,<переменная сообщения> » ]
      [ SEVERITY <уровень важности> ]
<уровень важности> ::= INFO | WARNING | ERROR
```

Если выражение проверки ложно, то в процессе компиляции на терминал выдается строка сообщения, причем если в этой строке присутствует символ %, то на этом месте выводится значение переменной сообщения. Если в строке несколько символов %, то значения переменных сообщения выводятся в порядке их записи в списке. Уровень важности задает реакцию компилятора на обнаруженное несоответствие. Объявление уровня важности как ERROR или отсутствие конструкции SEVERITY является указанием прекратить компиляцию, в остальных случаях после выдачи сообщения компиляция продолжается.

Указанные возможности иллюстрируются на примере программы (листинг 3.77), которая может быть использована для представления различных по существу блоков — сумматора, вычитателя или управляемого сумматора-вычитателя. Кроме того, программу можно использовать как описание комбинационной схемы, так и схемы с памятью. Дополнительный параметр настройки — число разрядов.

Операторы IF-GENERATE разрешают или блокируют реализацию определенных последовательностей деклараций или операторов. Например, в зависимости от значения параметра type выходные данные программы определяются либо как узлы, либо как D-триггеры. Обратите внимание, что операторы проверки позволяют осуществлять контроль корректности задания параметров настройки.

Листинг 3.77

```
PARAMETERS (type="registered",
                direction="ADD",
                Width=8);
ASSERT ( width>0)
        REPORT "value of width must be greater then %" width
SEVERITY error;
ASSERT ((type=="registered")#(type=="unregistered"))
        REPORT "invalid design type declaration"
SEVERITY ERROR;
ASSERT( (Direction=="ADD")#(Direction=="SUBB")#(Direction=="ADD_SUBB"))
         REPORT "direction must be ADD,SUDD or ADD_SUBB"
SEVERITY ERROR;
```

```
SUBDESIGN add_s
   ( cin,clock, clear, control, a[width-1..0],b[width-1..0] :INPUT;
     result [width-1..0],cout: OUTPUT)
 VARIABLE
arg1[width..0], arg2[width..0], int_res[width..0],arg3[width..0]:NODE;
IF (type  =="registered")  GENERATE result[width-1..0],cout : dff;
 END GENERATE;
BEGIN
  ASSERT type=="registered" !$ used(clock)
          REPORT "for registered type use clock input"
         SEVERITY WARNING;
  Assert direction=="ADD_SUBB" !$ used(control)
          report "for registered type use clock input"
         SEVERITY ERROR;
 IF (type =="registered") GENERATE  result[].clk=clock; cout.clk=clock;
                                    result[].clrn=clear; cout.clrn=clear;
                                      (cout.d,result[].d)=int_res[];
                          ELSE GENERATE (cout,result[])=int_res[];
   END GENERATE;
IF cin==vcc  THEN arg3[]=1; ELSE arg3[]=-1;
END IF;
arg1[]=(a[width-1],a[]); -- расширение знакового разряда
arg2[]=(b[width-1],b[]);
IF (direction=="ADD") GENERATE int_res[]=arg1[]+arg2[]+arg3[];
END GENERATE;
  IF (direction=="SUBB") GENERATE int_res[]=arg1[]-arg2[]-arg3[];
       ELSE GENERATE
                 IF (control==vcc) THEN int_res[]=arg1[]+arg2[]+arg3[];
                                   ELSE int_res[]=arg1[]-arg2[]-arg3[];
                 END IF;
       END GENERATE;
END;
```

3.5. Что дальше?

Обзор языков проектирования дискретных устройств показывает, что все они имеют достаточно много общих черт. Во-первых, это преемственность по сравнению с традиционными языками с точки зрения состава средств алгоритмического описания, включая наборы базовых типов данных, базовые операции, операторы присваивания, условные выражения и операторы, операторы повторения и т. д. Во-вторых, это введение в состав языка средств описания параллельных процессов, прежде всего конструкций событийного и временного управления инициализацией операторов и блоков.

И в третьих, это наличие специфических типов данных и специальных средств для структурного представления проекта.

Замечание

В приборно-ориентированных языках конструкции событийного управления явно не вводятся, но, например в AHDL, по умолчанию все операторы инициируются по событийному принципу.

На вопрос, какой язык все-таки выбрать, однозначного ответа не существует. "Война языков", происходившая до середины 90-х годов, прежде всего между VHDL и Verilog, к сегодняшнему дню закончилась "вничью". По данным опросов [49], 60% разработчиков предпочитают Verilog, а 40% — VHDL. При этом в Америке шире используется Verilog, европейские компании отдают предпочтение VHDL. В России VHDL имеет явное предпочтение, хотя бы потому, что VHDL отражен в целом ряде книг [6, 8, 56, 57], но до сих пор нам не известна книга на русском языке, систематически излагающая Verilog.

Относительно языков ассемблерного уровня, следует ожидать, что они сохранят свою экологическую нишу. Наряду с графическими способами задания и схемным вводом в виде таблиц соединений они будут использоваться для представления критичных по времени исполнения фрагментов в проектах, ориентированных на конкретную элементную базу.

Наряду и параллельно с развитием языков происходит развитие средств их интерпретации. Естественно, новые опции и изменения синтаксических конструкций языка, вводимые в новых версиях стандартов, должны своевременно отслеживаться при разработках САПР, но это не является особой проблемой, т. к. свежие версии компиляторов появляются значительно чаще, чем происходит обновление стандартов. Более важным в разработках новых версий САПР и их оценке являются следующие моменты.

Одна из существенных проблем, связанных с использованием языков проектирования, — это различие между "эталонным языком" и "реализуемым подмножеством языка". Уже неоднократно отмечалось, что язык при проектировании используется для решения взаимодополняющих, но различных задач: спецификация проекта, моделирование и, наконец, синтез реализации. В языки включено достаточно много конструкций, которые предназначены исключительно для целей моделирования. Наиболее очевидные примеры — это операторы предупреждения языков VHDL и AHDL (Assert Statements), функции управления выводом в Verilog. Современные САПР не интерпретируют опции временного управления (операторы `WAIT FOR` в языке VHDL и префиксы вида `#<время>` в Verilog). Выражения задержки, установленные пользователем, тоже игнорируются синтезаторами. Обычно значения задержки назначаются в системе проектирования по результатам синте-

за, причем в ряде САПР предусмотрена генерация уточненного программного текста, в котором отражаются задержки реальных компонентов синтезированного устройства. Подобные уточненные описания предназначены для моделирования более сложных проектов, в том числе для экспорта в другие системы моделирования.

Кроме этого, пользователь вправе предусмотреть в программе фрагменты, ориентированные только на реализацию процедур моделирования. При записи таких фрагментов разработчик может не обращать внимание на эффективность и даже возможность их физической реализации — важна лишь простота и наглядность описания — и в этом случае допускается применить даже достаточно экзотические конструкции, вроде рекурсивных вызовов процедур или операторов повторения с необъявленными заранее признаками их завершения. Такие фрагменты обычно просто удаляются после предварительного тестирования и отладки перед передачей программы в подсистему синтеза. Существуют системы интерпретации, в которых программным блокам можно сопоставлять специфические признаки, определяющие, какие из них подлежат реализации, а какие используются только при моделировании. Например, синтезатор системы MAX+PLUS II игнорирует операторы INITIAL в программах на Verilog. Иногда разработчик намеренно использует несинтезируемые конструкции на начальных этапах проектирования для обеспечения большей наглядности спецификации или для сокращения времени предварительной отладки (как для экономии собственных трудозатрат, так и машинного времени для моделирования), предполагая дальнейшую декомпозицию проекта на реализуемые фрагменты (см. пример в *разд. 3.2.1*).

Но в то же время имеется ряд конструкций языков, которые в принципе синтезируемы, но не поддерживаются в конкретных САПР. Рассмотрим, например, VHDL-программу, представленную в листинге 3.78.

Листинг 3.78

```
ENTITY unrealis IS
  PORT( clock:IN bit;
        a,b: IN integer RANGE 0 TO 15;
         compare_out : OUT bit_vector (1 DOWNTO 0)
 );
END urealis;
ARCHITECTURE behave OF unrealis IS
BEGIN
 PROCESS (clock)
  VARIABLE more, less:bit;
    BEGIN
```



```
   IF (clock='0')THEN
      IF a>=b THEN more:='1'; less:='0';
      ELSE more:='0'; less:='1';
      END IF;
  ELSE compare_out<=more&less;
 END IF;
END PROCESS;
END behave;
```

Эта программа описывает устройство, которое по заднему фронту сигнала clock выполняет сравнение входных кодов с фиксацией результата на внутренних триггерах more и less. По переднему фронту clock происходит перезапись результата в выходной регистр compare_out. При попытке скомпилировать этот файл в системе MAX+PLUS II было получено сообщение об ошибке следующего содержания: "Else clause following clock edge must hold state of signal compare_out" (конструкция ELSE, следующая за фронтом тактирующего сигнала, должна сохранять состояние сигнала compare_out), хотя ни в стандарте, ни сопроводительных материалах фирмы ничего о таком ограничении не упомянуто. Эта же программа была успешно скомпилирована компилятором SimplifyPro, а результат компиляции в форме EDIF-файла был импортирован в систему MAX+PLUS II и без ошибок реализован в составе БИС семейства FLEX10K. Имеется достаточно много других примеров, когда компилятор "отказывался" интерпретировать регистровые схемы, синхронизируемые несколькими различными сигналами.

Еще хорошо, если неподдерживаемая опция перечислена в материалах, сопровождающих систему проектирования, или явно указывается компилятором, как в приведенном примере. Например, материалы по САПР MAX+PLUS II определяют, что не поддерживаются такие конструкции VHDL, как параметры настройки (кроме библиотечных параметризованных модулей), объявления конфигураций, данные действительного типа и многое другое. Компилятор языка Verilog этой САПР не поддерживает численные типы данных (кроме индексных выражений и оператора FOR), большинство операторов цикла, шинные типы данных, опции задержки и силы драйвера в операторах присваиваний и т. д.

К сожалению, многие неподдерживаемые опции не документируются. Возникали проблемы при интерпретации некоторых сложных конструкций, прежде всего фрагментов с большим числом вложений операторов. При этом компилятор MAX+PLUS II выдавал не слишком определенное сообщение "unknown problem" (неизвестная проблема). Поиск ошибки (впрочем, это даже и нельзя назвать ошибкой разработчика в традиционном смысле) в подобной ситуации становится чрезвычайно трудным.

В целом, несмотря на стремление разработчиков САПР обеспечить универсализацию средств синтеза и даже попытки стандартизации реализуемых

подмножеств языков, сохраняется значительное расхождение в наборе конструкций языков, входящих в реализуемое подмножество языка различных интерпретаторов. Разработчик, перед тем как приступить к описанию своего проекта, должен ознакомиться с ограничениями, определенными в той системе проектирования, которую он собирается использовать.

Сокращение объема нереализуемых подмножеств языков является одним из существенных направлений усовершенствования САПР больших интегральных схем.

Другим аспектом развития САПР дискретных устройств является повышение качества синтеза. Разрабатываются и внедряются новые эффективные алгоритмы оптимизации проектов по различным критериям. Обычно разработчик может задать критерий оптимизации — минимальное время или оптимальное использование ресурсов. Но наиболее совершенные средства последних поколений дают результаты, многократно превосходящие результаты, которые давали "старые" версии, одновременно по обоим параметрам. Возможности средств синтеза, представляемые различными фирмами, в этом смысле оказываются существенно различными, причем в настоящее время выигрывают фирмы, которые специализируются именно на разработках САПР. Их компиляторы на сегодня существенно превосходят по возможностям системы, предлагаемые непосредственно производителями ПЛИС. Фирма Altera даже рекомендует в критических случаях для предварительной компиляции VHDL- и Verilog-проектов, ориентированных на реализацию в ПЛИС этой фирмы, использовать средства третьих фирм, например Simplicity или Cadence, и импортировать результат синтеза в собственные САПР фирмы, несмотря на наличие в последних встроенных компиляторов. По опыту авторов такая процедура, в частности использование предварительной компиляции с применением компиляторов LeonardoSpectrum фирмы Mentor Graphics, позволяет в сложных проектах экономить до половины используемого ресурса. Кроме того, таким образом часто удается интерпретировать программы, не доступные для интерпретации встроенными компиляторами САПР фирмы Altera, как было показано выше на примере.

И наконец, значительные усилия разработчиков языков и средств их интерпретации направлены на повышение уровня абстракции описания. VHDL и Verilog показали себя весьма эффективными средствами проектирования "досистемных" уровней представления устройств. Эти уровни объединяют вентильный уровень представления, уровень регистровых передач и уровень операционных блоков. Однако современный этап развития средств обработки информации, в том числе разработка встраиваемых систем на кристалле, требует более обобщенного представления алгоритма, включая совместное представление и аппаратных, и программных частей проекта.

Под системным уровнем проектирования создатели и пользователи САПР информационных систем понимают "учет архитектурных особенностей систем SoC/ASIC до разделения задачи между программной и аппаратной частью... Это является важным поводом для использования языков системного уровня перед началом утомительной работы на уровне регистровых передач" [49]. Языки системного уровня рассматриваются не только как средства ускорения моделирования, но и ориентированы на решение задач сопряженного проектирования и сопряженной верификации.

Как VHDL, так и Verilog в их современной форме оказываются недостаточными для представления комплексных аппаратно-программных проектов. С одной стороны, эти языки не обладают достаточной мощностью для описания сложных процессов обработки данных (во всяком случае, программы при использовании только возможностей HDL были бы чрезвычайно громоздки), да и компиляторов HDL-программ в машинный код типовых компьютеров на сегодня не известно. Если даже предположить представление процессорного блока и программной части задачи с использованием типовых HDL, потребовались бы несоразмерно большие затраты машинного времени на процедуры моделирования, отладки и верификации. Традиционные же языки программирования, прежде всего C/C++, не содержат средств для представления многих аспектов поведения аппаратной подсистемы.

На сегодня можно выделить два основных подхода к созданию языка системного уровня будущего. Первый ориентируется на расширение языка C/C++ и введение в него конструкций для описания специфики аппаратных средств. Другой подход предусматривает расширение языков проектирования, чаще упоминается Verilog, за счет введения абстрактных понятий системного уровня. Как и следовало ожидать, первый подход приветствуется, прежде всего, системными программистами, а второй — разработчиками аппаратуры и, в некоторой степени, создателями прикладных управляющих программ.

В рамках первого подхода можно упомянуть SystemC, SpecC, CycleC, HandelC. Наиболее продвинутой является разработка языка SystemC, выполненная совместно фирмами Cadence, CoWare и Adelante Technologies в рамках проекта Open SystemC Initiative. В своей основе SystemC представляет версию C++, в которой введены классы для представления времени, параллелизма, специфических для аппаратуры типов данных и т. п. Возможности оперировать с такими объектами, собственно, и являются отличительной чертой именно языков проектирования. Введены, кроме того, специальные конструкции для моделирования взаимодействия подсистем. Интересно отметить, что эти классы построены на принципах описания, заложенных в VHDL, что по мнению авторов статьи [47] привлечет к этому языку не только системных программистов, но и сторонников VHDL. К сожалению, возможности и качество синтеза аппаратной части проектов на основе SystemC пока оставляют желать лучшего.

В направлении развития "от языков проектирования" наибольший интерес представляет Superlog, предложенный фирмой Co-design Automation. Superlog является расширением Verilog. Авторы проекта считают, что "проектировщики, работающие с языками проектирования, чрезвычайно легко адаптируются к новым возможностям"[49]. Среди привлекательных свойств языка (а точнее, системы моделирования Systemsim на его базе) авторы проекта также указывают на допустимость смешивания в одной программе фрагментов, записанных на C, Verilog и Superlog, мощные средства верификации.

Время покажет, какое из этих направлений будет более плодотворным, а может быть, как с VHDL и Verilog, возникнет ситуация их параллельного использования. В ближайшей же перспективе они, безусловно, будут развиваться параллельно.

Разумеется, развитие языков системного уровня не сможет привести к умалению роли Verilog и VHDL. Они по-прежнему остаются предпочтительными для детального моделирования аппаратуры и для создания проектов с преимущественно аппаратными средствами обработки. Более того, с повышением эффективности компиляторов они могут частично вытеснять языки ассемблерного уровня и схемотехнические приемы проектирования.

ГЛАВА 4

Примеры проектирования устройств с применением ПЛИС

4.1. Проектирование операционных устройств

4.1.1. Операционные устройства с микропрограммным управлением

Под операционным устройством понимают вычислительный узел, способный многократно выполнять любое преобразование из набора, предусмотренного для этого узла, каждый раз, когда на него поступает сигнал, инициирующий преобразование (команда). Часто реализация команды требует последовательного выполнения нескольких шагов. Это может быть связано с тем, что на некоторых шагах используются результаты, полученные на предыдущих шагах, или с тем, что данные поступают, а результаты должны выдаваться в определенной последовательности, или с наличием ограничений на затраты оборудования.

При высоких требованиях к производительности элементарные действия стараются распределять между несколькими параллельно работающими блоками, причем для алгоритмов, предусматривающих использование результатов некоторого шага на последующих шагах, применяют конвейерную реализацию (пример будет представлен далее). Однако при умеренных требованиях по производительности с целью уменьшения объема оборудования однотипные операции выполняют последовательно в одном и том же блоке, даже если операции функционально независимы (функционально независимыми называются операции, которые не используют результатов друг друга и, в принципе, могут исполняться параллельно).

Часто становится оправданной частичная перестройка функций операционных блоков в процессе исполнения команды. Элементарное действие в этом