Московский государственный технический университет
имени Н.Э. Баумана

Учебное пособие

Ю.В. Берчун

ЯЗЫК ОПИСАНИЯ ЭЛЕКТРОННОЙ АППАРАТУРЫ VHDL

Издательство МГТУ им. Н.Э. Баумана

Московский государственный технический университет
имени Н.Э. Баумана

Ю.В. Берчун

ЯЗЫК ОПИСАНИЯ ЭЛЕКТРОННОЙ АППАРАТУРЫ VHDL

*Рекомендовано Научно-методическим советом
МГТУ им. Н.Э. Баумана в качестве учебного пособия*

М о с к в а
Издательство МГТУ им. Н.Э. Баумана
2 0 1 0

УДК 681.326(075.8)
ББК 22.18
Б52

Рецензенты: *С.Р. Иванов*, *Р.Ш. Загидуллин*

Берчун Ю.В.

Б52 Язык описания электронной аппаратуры VHDL : учеб. пособие / Ю.В. Берчун. — М.: Изд-во МГТУ им. Н.Э. Баумана, 2010. — 61, [3] с.: ил.

Пособие посвящено проектированию узлов ЭВМ, периферийных устройств и прочих цифровых систем с помощью высокоуровневого языка описания аппаратуры VHDL. Этот язык является международным стандартом и используется в качестве основы лингвистического обеспечения как в системах анализа (моделирования), так и в системах синтеза цифровой аппаратуры. Ведущие мировые САПР БИС поддерживают описания на языке VHDL.

Учебное пособие посвящено основам языка VHDL и предназначено для быстрого ознакомления с концептуальными положениями этого языка. Рассмотрены вопросы и базовые принципы параллельного программирования, положенные в основу языка VHDL, а также принципы организации VHDL-проекта и взаимосвязь компонентов проекта с физическими процессами, протекающими в реальных цифровых устройствах.

Для студентов, изучающих язык VHDL в рамках учебных курсов «Элементная база ЭВМ», «Архитектура ЭВМ», «Организация вычислительных систем».

УДК 681.326(075.8)
ББК 22.18

ВВЕДЕНИЕ

Развитие *систем автоматизированного проектирования* (САПР) в области электроники привело к созданию *унифицированных языковых средств* исходного описания проектов *больших* и *сверхбольших интегральных схем* (БИС, СБИС). Языки такого рода необходимы для решения задач высокоуровневого проектирования цифровых СБИС. Под *высокоуровневым* понимается алгоритмическое и логическое проектирование СБИС, а основными задачами высокоуровневого проектирования являются моделирование проектов, синтез логических схем и верификация.

Унифицированными языками для описания СБИС стали VHDL и Verilog. Они фактически являются международными стандартами и входными языками промышленных САПР СБИС различных типов — заказных, полузаказных и *программируемых логических интегральных схем* (ПЛИС).

Например, VHDL служит базовым языком проектирования (сквозного моделирования и синтеза) в свободно распространяемой САПР WebPack ISE фирмы Xilinx, которая является одним из крупнейших производителей кристаллов ПЛИС. Поддержка языков VHDL и Verilog обязательна для всех крупных САПР ведущих мировых производителей, таких как Cadence, Mentor Graphics, Synopsys и др. Существуют и специализированные САПР, ориентированные на синтез решений на базе ПЛИС различных производителей, например Active-HDL фирмы ALDEC.

1. УРОВНИ ОПИСАНИЯ ЭЛЕКТРОННОЙ АППАРАТУРЫ

Современные цифровые электронные приборы являются, без сомнения, *сложными системами*, состоящими из миллионов транзисторов. Проектирование любой сложной системы предполагает применение *блочно-иерархического подхода*, позволяющего сократить вероятность ошибок, обусловленных большой размерностью задачи. Проектирование цифровой электронной аппаратуры не является исключением, более того, здесь уровни иерархии можно выделить гораздо более четко, чем во многих других прикладных областях. Для блоков, описываемых на каждом из уровней, характерны свой набор элементов, способ представления информации (язык описания), используемый математический аппарат.

Обычно выделяют пять уровней описания вычислительных систем:

1) системный;
2) вычислительных процессов;
3) функционально-логический;
4) схемотехнический;
5) компонентный.

Функционально-логический уровень является самым емким, в нем дополнительно выделяют ряд подуровней:

1) функционирования ЭВМ;
2) устройств (узлов) ЭВМ;
3) регистровый;
4) вентильный.

Языки описания аппаратуры (Hardware Description Language — HDL) позволяют описывать блоки в первую очередь на функционально-логическом уровне, наибольшее распространение имеют на регистровом и вентильном подуровнях и при описании узлов ЭВМ. Проекты на языках HDL могут быть использованы и для решения задач системного уровня проектирования. Кроме того,

специальные расширения языка (например, VHDL-AMS) могут рассматриваться как промежуточный подуровень описания между вентильным подуровнем функционально-логического уровня и схемотехническим уровнем, что позволяет моделировать не только цифровую, но и аналоговую аппаратуру.

2. ОБЗОР HDL

2.1. История развития HDL

С начала 1970-х годов стала актуальной проблема создания стандартного средства документации схем и алгоритмов дискретных систем переработки информации и цифровой аппаратуры, одинаково пригодной как для восприятия человеком, так и для обработки на ЭВМ.

Известно большое число предшественников современных HDL, как отечественных, так и зарубежных. Отечественные — «МОДИС», «МОДИС-В87» «Автокод-М», MPL, ООС-2, «Форос», «Алгоритм», «Пульс», «Симпатия» и др., зарубежные — CDL, DDL, ISPS, CONLAN, HILO и др.

В настоящее время известны два стандартных языка описания аппаратуры — VHDL и Verilog-HDL, называемый далее для краткости Verilog.

Язык *VHDL* (Very high speed integrated circuit Hardware Description Language — язык описания сверхскоростных БИС) был разработан международной группой по заданию Министерства обороны США в начале 1980-х годов в целях обеспечения единообразного понимания подсистем различными проектными группами. В 1987 г. спецификация языка VHDL была утверждена стандартом ANSI/IEEE STD 1076-1987. Удобство и относительная универсальность конструкций этого языка достаточно быстро привели к созданию программ моделирования систем на основании их описания в терминах VHDL.

С начала 1990-х годов разрабатываются прямые компиляторы VHDL-описаний в аппаратные реализации различных классов. Наряду с необходимостью более адекватного представления в языке современных тенденций в цифровой схемотехнике это привело к созданию расширенного стандарта ANSI/IEEE STD 1076-1993. В

1999 г. была утверждена последняя версия стандарта ANSI/IEEE STD 1076-1999, известная как VHDL-AMS (AMS — Analog and Mixed-Signal Extensions). Наиболее существенным нововведением этой версии языка, как понятно из названия, является появление конструкций, обеспечивающих эффективное описание аналоговых и смешанных цифро-аналоговых устройств.

Работу над усовершенствованием стандарта ведет группа VASG (VHDL Analysis and Standardization Group). Ведутся также работы по стандартизации внутренней формы представления VHDL-описаний в ЭВМ (группа VIFASG — VHDL Intermediate Form Analysis and Standardization Group), формы задания тестов для VHDL-моделей (группа WAVES — Waveform and Vector Exchange to Support Design and Test Verification), задания параметров задержек компонентов (группа VITAL — VHDL Initiative Towards Application-Specific Integrated Circuit Libraries), алфавита представления значений сигналов в моделях и операции в этом алфавите (стандарт IEEE std_logic_1164) и т. д.

Язык *Verilog* был разработан в 1985 г. фирмой Gateway Design Automation как язык моделирования, ориентированный на внутреннее применение. Позднее, в 1989 г., эта фирма была куплена корпорацией Cadence, которая открыла Verilog для общественного использования. После этого язык был стандартизован — IEEE 1364-1995. В отличие от VHDL, который строго типизирован и синтаксически напоминает языки ADA и Pascal, Verilog базируется на C, имеет меньше встроенных возможностей саморасширения, но зато более прост в реализации, имеет более развитый интерфейс с языком C и лаконичен. В настоящее время принят стандарт IEEE 1364-2001, который, в частности, включил в себя ряд стилистических средств, сближающих его с VHDL. Несмотря на то, что VHDL был создан раньше и предоставляет более широкие возможности, Verilog стал достаточно популярен и компиляторы с этого языка наряду с VHDL-компиляторами включены в подавляющее большинство САПР БИС. Более того, зачастую поддерживается компиляция смешанных проектов.

2.2. Варианты использования HDL

Проектировщик БИС может составить функциональное HDL-описание проектируемого кристалла и, используя систему модели-

рования САПР, проверить его соответствие спецификации (провести функциональную верификацию). После этого с помощью системы логического синтеза он может автоматически синтезировать схему (получить ее структурное HDL-описание) в заданном элементном базисе, затем путем моделирования полученной логической схемы оценить корректность результатов синтеза, после чего с помощью системы автоматизированного конструкторского проектирования провести трассировку соединений, а моделированием проверить правильность работы схемы с учетом задержек и наводок.

Возможен автоматический синтез схем с учетом контролепригодности, синтез контролирующих тестов, а также анализ тестов на полноту и корректность. Языки HDL используются не только для представления проектируемых схем, но и для описания тестирующих программ (testbench) и тестов.

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

Инженер-эксплуатационщик цифровой электронной аппаратуры при наличии документации в виде HDL-описания устройства и тестирующей программы на их базе может осуществить модернизацию схем, использовать HDL-модели при поиске неисправностей в схеме и доработке контрольных тестов.

Стандартизация входных языков и внутренних интерфейсов подсистем САПР, в том числе и на базе HDL, создает общую коммуникационную среду проектирования, позволяет упростить интеграцию решений различных производителей программного обеспечения, обмен библиотеками моделей компонентов и проектов, модернизацию отдельных подсистем САПР.

2.3. Преимущества HDL

К основным достоинствам HDL следует отнести следующие:

- стандартность;
- многоаспектность и иерархичность;
- пригодность для восприятия человеком и обработки на ЭВМ.

Действительно, языки VHDL и Verilog официально признаны стандартом описания цифровой аппаратуры, который поддерживается военно-промышленным комплексом и радиоэлектронной промышленностью стран – лидеров в области электроники и микроэлектроники. Наличие такого стандарта облегчает обмен данными и документацией между отдельными группами разработчиков и эксплуатационщиков аппаратуры, различными САПР и их подсистемами.

Языки HDL пригодны для описания как схем аппаратуры, так и функциональных тестов и алгоритмов функционирования. Они применяются в широком диапазоне уровней структурной детализации описаний цифровой аппаратуры: от описания архитектуры ЭВМ на уровне устройств типа процессор – память до описания узлов типа триггеров на уровне вентилей и МОП-ключей, от описаний алгоритмов ЭВМ на уровне команд до описаний алгоритмов устройств на уровне межрегистровых передач и булевских функций, от описаний функциональных тестов до тестов проверки схем.

На высших уровнях абстракции HDL-описания можно рассматривать как средство спецификации требований к проекту.

Описания на различных языках HDL легко доступны для понимания человеком и пригодны для обработки такими компонентами САПР БИС, как подсистемы моделирования, подсистемы формальной верификации, подсистемы логического синтеза, подсистемы синтеза с учетом тестопригодности, системы синтеза и анализа контрольных тестов, временные анализаторы, кремниевые компиляторы, подсистемы автоматизации конструкторского проектирования и т. п.

3. ОБЩИЕ ПОЛОЖЕНИЯ

3.1. HDL с точки зрения схемотехника

В HDL-описании электронных устройств, как и в любой модели, отражаются только некоторые аспекты реальной системы.

Цифровую аппаратуру характеризуют, например, следующими аспектами:

• функциональный (реализуемая функция, алгоритм);

• временной (задержки, время отклика);
• структурный (типы и связи компонент);
• ресурсный (число вентилей, площадь кристалла);
• надежностный (время наработки на отказ);
• конструктивный (масса, габариты);
• стоимостной и т. д.

Языки HDL содержат средства, позволяющие отразить в основном функциональный, временной и структурный аспекты (рис. 3.1).

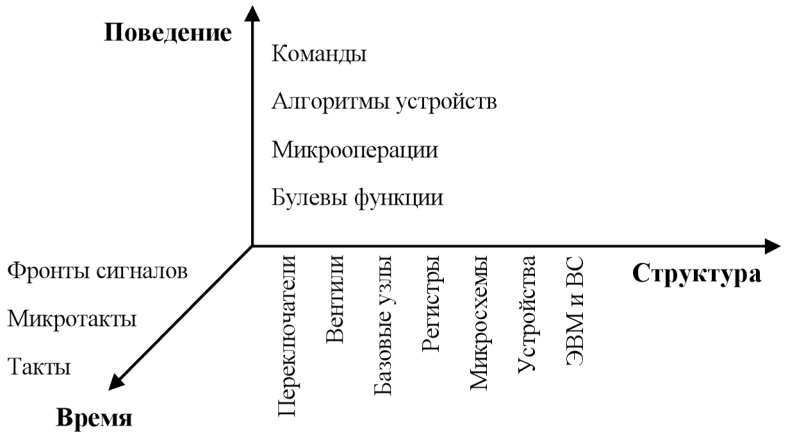

Рис. 3.1. Возможности языков HDL по отражению основных аспектов проектирования электронной аппаратуры

Как уже отмечалось выше, описание того или иного аспекта может быть детализировано в весьма широких пределах. Например, функциональное описание (от уровня системы команд и алгоритмов устройств до булевых функций); структурное описание (от уровня устройств типа процессор-память до уровня вентилей и переключателей); временные характеристики (от задержек фронтов сигналов до тактов и задержек электромеханических устройств).

Степень детализации того или иного аспекта определяется конкретными задачами. Например, описание некоторой микропроцессорной вычислительной системы может строиться как описание структуры, состоящей из микросхем БИС и СБИС, а описа-

ние самих микросхем строится как поведенческое, если их описание на вентильном уровне отсутствует либо слишком громоздко.

С точки зрения разработчика электронной аппаратуры HDL-описания должны давать возможность описывать как интерфейсы объектов проекта, так и их структуру (более того — многообразие структур). В зависимости от решаемой задачи при разработке описаний структур проекта могут быть использованы разные стили описания архитектур. Подробно эти понятия будут рассмотрены при изучении базовых понятий языка VHDL.

3.2. HDL с точки зрения программиста

Учитывая соображения, высказанные выше, средства HDL можно отнести к одному из двух разделов (рис. 3.2):

• *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 специальные типы данных, средства для описания процессов с учетом их протекания в реальном времени и средства для структурного представления проекта.

Общеалгоритмический компонент по составу, смыслу и принципам использования мало отличается от традиционных языков программирования, а форма записи (синтаксис и семантика языковых конструкций) весьма близка к традиционным языкам программирования высокого уровня.

На примере VHDL предварительно остановимся на некоторых составляющих проблемно-ориентированного компонента.

В числе проблемно-ориентированных типов данных следует отметить физический тип, в первую очередь — время. Пользователь также может вводить дополнительные типы данных, отражающих электрические или механические свойства моделируемых объектов.

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

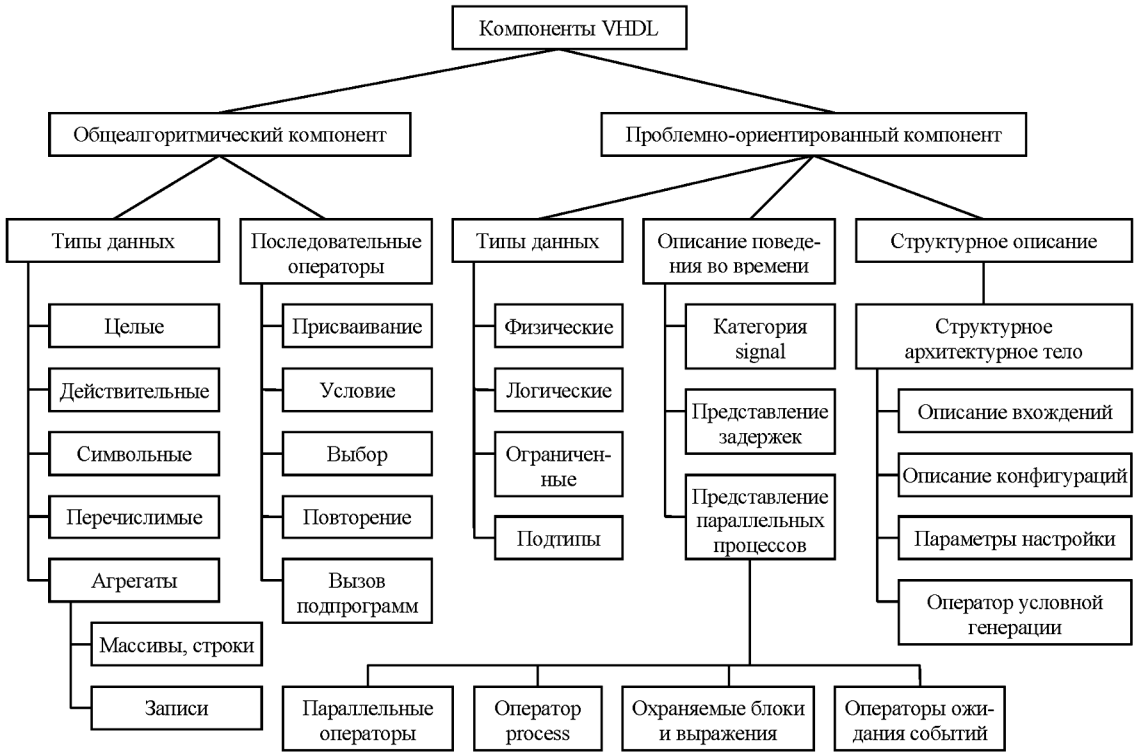

Рис. 3.2. VHDL как программная система

Среди средств представления поведения системы в реальном времени следует отметить средства представления параллелизма в реальной системе:

• понятие *сигнала* как единицы передаваемой информации между 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 обработке 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 включаемых в устройство 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 обзора можно видеть, что VHDL (и другие языки HDL) представляет собой развитую алгоритмическую систему, позволяющую описывать разнообразные структуры и явления в информационных системах.

4. ОСНОВЫ ЯЗЫКА VHDL

4.1. Структура проекта

Проект в системе проектирования на основе VHDL представляется совокупностью иерархически связанных текстовых фрагментов, называемых *проектными модулями*. Различают первичные и вторичные проектные модули:

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

Декларация *сущности* (entity) определяет имя проекта и его интерфейс, т. е. порты и параметры настройки. *Архитектурное тело* сущности описывает тем или иным образом функционирование устройства и (или) его структуру.

Каждой сущности сопоставляется одно или несколько архитектурных тел. Несколько вторичных модулей, соответствующих одному первичному, составляют набор возможных альтернативных реализаций объекта, представленного первичным модулем. Например, одной сущности может соответствовать несколько архитектурных тел, отличающихся степенью детализации описания и даже алгоритмом преобразования данных.

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 модулей 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>;
<объявление использования>::=
use <имя библиотеки>.<имя пакета>.<имя модуля>;

Иными словами, текст представляет собой произвольный набор первичных и вторичных модулей, причем каждому первично-

му модулю может предшествовать указание на библиотеки и пакеты, информация из которых требуется для построения этого модуля. После этого необходимо указать, что они будут использованы при помощи соответствующего объявления. В качестве имени модуля часто употребляют all — в этом случае будут использованы все имеющиеся модули. Отметим, что даже если несколько первичных модулей ссылаются на одни и те же библиотеки и модули, декларация использования должна предшествовать каждому первичному модулю в отдельности.

4.2. Сущности и архитектурные тела

Синтаксис декларации сущности имеет вид

<декларация сущности>::=
entity <имя проекта> is
[<объявление параметров настройки>]
[<объявление портов>]
[<раздел деклараций>]
[begin <раздел операторов>]
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 | buffer

Объявление *параметров настройки* включается в декларацию сущности для создания проектов, которые предполагается использовать как фрагменты в различных других проектах, причем возможна модификация некоторых свойств данного компонента, точнее, выбор параметра из множества значений, определенного типом.

При определении портов задают имена входных (in), выходных (out, buffer) и двунаправленных (inout) линий передачи информации и тип данных, передаваемых через порты. Разница между режимами out и buffer состоит в том, что порты, специфицированные как buffer, могут использоваться в цепях обратной связи внутри самой сущности, в то время как использование режима out приведет к ошибке моделирования. Объявление портов, как следует из

представленных правил синтаксиса, не обязательно. 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.

Для параметров и входных портов можно задавать значения по умолчанию. Они принимаются, если соответствующим единицам информации не присвоены другие значения в модулях высшего уровня иерархии.

Раздел деклараций сущности имеет такое же содержание и такой же смысл, что и раздел деклараций архитектурного тела (см. ниже) — объявляются локальные типы данных, подпрограммы, сигналы и т. п. При этом объявленные объекты доступны во всех архитектурных телах, подчиненных этой сущности.

Раздел операторов сущности может содержать только весьма ограниченный набор служебных операторов, которые на практике применяются редко. Поэтому обычно этот раздел остается пустым.

Архитектурные тела представляют содержательное описание проекта. Архитектурное тело в некотором смысле подчинено соответствующей сущности. 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 тело можно было бы отнести к тому или иному типу, не существует — речь идет скорее не о четкой классификации, а о стилях описания, для каждого из которых характерными являются определенные операторы и которые лучше отражают разные аспекты одного и того же объекта (структурный и поведенческий соответственно).

Определены следующие правила записи архитектурных тел:

<архитектурное тело>::=
architecture <имя архитектуры> of <имя сущности> is
<раздел деклараций>
begin
<раздел операторов>
end [architecture] <имя архитектуры>;

Здесь имя архитектуры — это любое индивидуальное имя проектного модуля. Имя сущности задает первичный модуль, которо-

му 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.

4.3. Типы данных

Язык VHDL основан на *концепции строгой типизации данных*, т. е. любой единице информации в программе обязательно присваивается 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.

Язык VHDL предоставляет некоторый базовый набор типов данных (их классификация представлена на рис. 4.1), которые не требуют объявления в программе пользователя. Кроме того, пользователь может определить свои типы данных и подключить существующие из библиотек. Различают *скалярные* типы данных и *агрегатные* типы. Объект, отнесенный к скалярному типу, рассматривается как законченная единица информации. Агрегат представляет упорядоченную совокупность скалярных единиц, объединенных единым именем.

Данные, используемые в программах, относятся к одной из категорий:

- константы;
- переменные;
- сигналы.

Различие между переменными и сигналами будет рассмотрено в подразд. 4.4.

Декларация объектов имеет следующий вид:

<декларация объекта> ::=
<категория> <имя>{,<имя>}:<тип> [:=<выражение>];
<категория> ::= constant | variable | signal

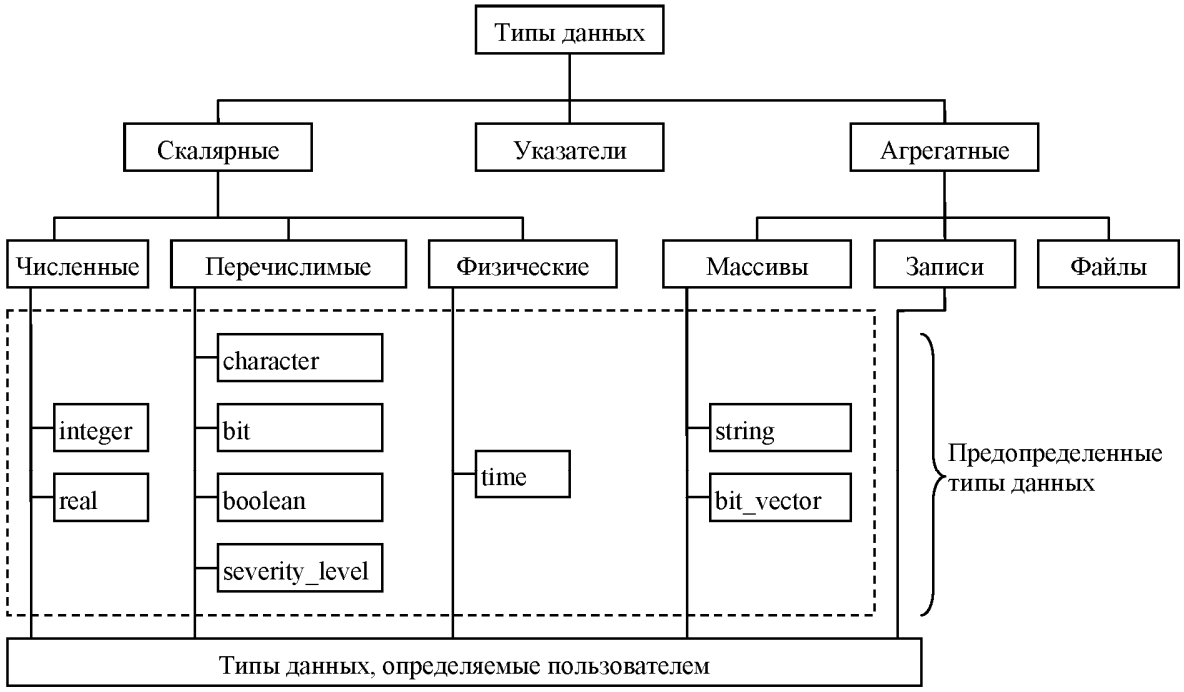

Рис. 4.1. Классификация типов данных в VHDL

Одна декларация может определить несколько объектов. Выражение в декларации должно совпадать по типу с декларируемым объектом. Оно задает значения константы либо начальные значения сигналов и переменных.

4.3.1. *Предопределенные типы данных*

В языке предварительно определены следующие типы данных:

<предопределенные типы>::=
integer | real | bit | boolean | character | string | time | but_vector
| severity_level | file_open_status | file_open_kind

Типы integer и real определяют численные данные — целые и действительные соответственно. Количество байт, отводимых для хранения чисел может зависеть от реализации, но стандартом считается 4 байта, что соответствует типам данных int и float для стандарта ANSI C. Для них определены следующие бинарные арифметические операции (табл. 4.1).

Таблица 4.1

Бинарные арифметические операции

Операция	Пример	
	Выражение	Результат
Сложение	5+6	11
Вычитание	8-3	5
Умножение	3*6	18
Деление (для целых)	9/4	2
Деление (для вещественных)	9/4	2.25
Деление по модулю (для целых)	9 mod 4 -14 mod 5	1 1
Остаток от деления (для целых)	9 rem 4 -14 rem 5	1 -4
Абсолютное значение (модуль)	abs (-5)	5
Возведение в степень	2**4	16

Также определены унарные арифметические операции сохранения и смены знака. В арифметических выражениях применяются

традиционные способы определения приоритетов операций, включая заключение в скобки. Поддерживаются также операторы отношения (сравнения): =, /=, <, <=, >, >=. Результатом сравнения является значение типа boolean.

Данные типа bit могут принимать значения из множества {'0', ' 1'}. Для данных типа bit определены следующие логические операции:

- not — инверсия;
- or — операция ИЛИ;
- nor — операция ИЛИ-НЕ;
- and — операция И;
- nand — операция И-НЕ;
- xor — неравнозначность;
- xnor — равнозначность.

Данные типа boolean также могут принимать два значения: {true, false}, и для них определены те же операции, что и над данными типа bit. Разница между типами bit и boolean состоит в том, что первые применяются для представления уровней логических сигналов в аппаратуре, а вторые для представления обобщенных условий, например результатов сравнения. Данные разных типов несовместимы.

Тип character объединяет символы, определенные в используемой операционной системе — буквы, цифры, специальные символы. Язык 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. Отметим, что символы '0' и '1' имеют двойное значение — и символьное, и логическое. В каждом конкретном случае тип данных определяется по контексту.

Тип time служит для задания задержек элементов и времени приостанова процессов при моделировании. Запись временной константы имеет вид

<целое> <единица измерения времени>

Определены следующие единицы измерения времени:

- fs — фемтосекунда;
- ps — пикосекунда;

• ns — наносекунда;
• us — микросекунда;
• ms — миллисекунда;
• s — секунда.

Над данными типа «время» определены операции отношения, сложения и вычитания, а также умножения и деления на целое.

Данные этого типа применяются для задания задержек элементов и длительности интервалов останова. Важную роль при моделировании имеет определенная на этом типе функция now, возвращающая значение текущего времени на момент ее вызова. Например, выражение now > 1 us примет значение true через 1 мкс модельного времени от начала моделирования и может обеспечить формирование последовательности входных воздействий или контрольных точек при моделировании.

Тип severity_level задает 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 выполнять при обнаружении некоторых условий. Фактическая трактовка действий в стандарте не оговорена и оставлена на усмотрение разработчиков системы моделирования.

Типы FILE_OPEN_STATUS и FILE_OPEN_KIND обеспечивают возможность контроля процедур обмена между программой моделирования и 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 работы с массивами и их элементами будут рассмотрены далее. В тексте программы строковая константа заключается в двойные кавычки.

4.3.2. *Скалярные типы, вводимые пользователем*

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

<декларация типа данных>::= 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>

Начнем рассмотрение пользовательских типов данных со скалярных типов.

Определение *перечислимого типа* имеет вид

<определение перечислимого типа>::=
(перечислимое значение {,перечислимое значение});
<перечислимое значение>::=
<идентификатор> | <символьная константа>

Например, определение TYPE state IS (S0,S1,S2,S3); может представлять набор допустимых состояний системы, для каждого из которых определяются выполняемые действия и правила перехода в другое состояние.

Важнейшим приложением перечислимого типа данных являются расширенные алфавиты моделирования, в частности тип std_logic и соответствующий ему тип std_logic_vector, описанный в пакете std_logic_1164. Данный девятизначный алфавит моделирования рассмотрен в приложении настоящего методического указания.

Определение *численных типов пользователя* целесообразно, во-первых, для контроля совместимости данных в программах, а во-вторых, для точного задания разрядности слов, представляющих данные в проектируемом объекте. В общем случае определение ограниченного типа подчиняется синтаксическому правилу:

<определение ограниченного типа>::=
[<базовый тип>] <диапазон>;
<диапазон>::=
range <ограничение><направление><ограничение>
|range<>
<направление>::= downto | to

Направление (to — увеличение, downto — уменьшение) должно быть согласовано с соотношением ограничений.

Примеры:

```
type unsigned_short is integer range 0 to 255;
type my_data is integer range -2**(n-l)+l to 2**(n-l)-l;
type input__level is -10.0 to +10.0;
```

Тип unsigned_short объединяет целые положительные числа, которые могут быть представлены в байтовом формате.

Тип my_data объединяет целые в диапазоне, который объявляет пользователь через разрядность данных n. В этом случае пользователь точно указывает компилятору число разрядов, необходимое для представления данных, обеспечивая экономию ресурсов микросхемы по сравнению с неограниченным типом.

При объявлении типа input_level базовый тип явно не задан, тип ограничений устанавливается в соответствии с типом их фактических значений.

4.3.3. *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>];
Например:

```
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. В языке VHDL-AMS этот тип введен в пакет electrical_systems и фактически также может считаться предопределенным.

4.3.4. *Агрегатные типы*

Массивы

Массив, как и в других языках программирования высокого уровня, — это набор данных, объединенных общим именем и различаемых по порядковым номерам (индексам). Для того чтобы ввести объ-

ект 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. Диапазон, заданный конструкцией range<>, представляет собой объявление неограниченного массива. В этом случае определяется не диапазон значений индекса, а только тип индексной переменной. Такое определение предполагает задание диапазона при определении конкретного экземпляра объекта, относимого к такому типу, например, при вызове подпрограмм. В подобных случаях диапазон устанавливается динамически в соответствии с диапазоном подставляемого фактического параметра.

Примеры:

```
type ram1 is array (length-1 downto 0) of integer range 2**width-l downto 0;
type ram2 is array (length-1 downto 0, width-1 downto 0) of std_logic;
type ram3 is array (integer range<>, integer range<>) of std_logic;
```

Во всех приведенных декларациях объявляется матрица ячеек памяти емкостью length слов по width разрядов в каждом, причем предполагается, что эти параметры были ранее определены. Однако выполнено это разными способами, а значит, и ссылаться на эти типы следует по-разному; ram1 и ram2 определены как ограниченные типы массивов, ram1 — как одномерный массив целых, а ram2 — как двумерный массив битов; ram3 определен как неограниченный тип и требует задания границ индексов при декларации объектов выбранного типа.

Декларации объектов, принадлежащих приведенным типам, могут выглядеть следующим образом:

```
variable ram1_instance: ram1;
variable ram2_instance: ram2;
variable ram3_instance: ram3 (length-1 downto 0, width-1 downto 0);
```

При обращении к элементам массива в программе индексы помещаются в скобках за именем массива. Тип индексного выражения должен соответствовать типу индекса, объявленного при декларации типа массива. При обращении к элементу многомерного

массива индексные выражения записывают через запятые в порядке, определенном в декларации типа.

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

```
a := "1001";
b <= "1101" & a;
```

Здесь a и b — строки или битовые векторы, причем a — переменная, а 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 (в VHDL'87 их нет):

- 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.

В составе большинства современных САПР поставляются пакеты, определяющие арифметические операции над битовыми массивами (кодами). Как правило, они поставляются в виде пакета std_logic_arith.

Записи

Запись — это структура данных, каждая информационная единица которой, называемая полем записи, имеет индивидуальное имя и может быть индивидуального типа. Записи удобны 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;

Рассмотрим пример. Определим тип pixel, представляющий цветовые составляющие отображения точки на экране в формате цветопередачи, предусматривающей 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 (479 downto 0,   639 downto 0);
```

Этот экземпляр может сохранять информацию об изображении размером 480 строк по 640 элементов в строке. Выборка значения красной составляющей верхнего левого элемента изображения из такой памяти описывается оператором

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

В следующем примере определяется обобщенный тип для представления конечных автоматов. Автомат, как известно, задается множеством входов, множеством состояний и множеством выходов, а также соответствующими функциями на этих множествах. Значит, можно ввести универсальный тип:

```
type 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.

4.3.5. *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 и позволяет вводить дополнительные функции преобразования, определяемые только для данных подтипа.

Синтаксис декларации подтипа следующий:

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

Пример:

```
subtype 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.

Использование *функции разрешения* будет описано в подразд. 4.9.

4.4. Сигналы и переменные

Любой проект является описанием явлений в дискретных системах. Эти явления можно представить 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 сочетанием символов <=, а переменной — :=.

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 высо-

кого уровня и описывают набор действий, которые последовательно выполняются над исходными данными в целях 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 собой части алгоритма, которые в реальной системе могут исполняться одновременно. Эти части взаимодействуют между собой и с окружением проектируемой системы.

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* (см. подразд. 4.6) 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 специальный 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. подразд. 4.7.3).

Сформулируем наиболее существенные различия сигналов и переменных.

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

• Значение сигнала изменяется не сразу после выполнения присвоения. Оператору присваивания сопоставляется некий буфер, на-

зываемый контейнером или, чаще, драйвером сигнала. 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 сильнее задержана, если оператор присваивания содержит выражение задержки after.

• Переменная определена только внутри тела процесса, сигнал — во всем архитектурном теле.

• Переменной можно многократно присваивать значение в теле процесса. Сигнал внутри одного процесса может иметь только один драйвер, т. е. 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).

4.5. Атрибуты

Атрибуты — скаляры, отражающие некоторые свойства объектов, используемых в программных модулях (типов, переменных, агрегатов). Например, атрибуты типа предназначены для сжатого представления информации о множестве значений, объединенных типом, а атрибуты сигнала — для представления временных свойств сигнала. В разделах операторов нельзя присваивать значение атрибуту, способ его определения задается декларацией атрибута. Атрибуту присваиваются имя и тип; имя является обычной переменной в выражениях того типа, который присвоен атрибуту. Имя атрибута записывается следующим образом:

<имя атрибута>::=
<имя атрибутируемого объекта>'<определитель атрибута> [(<выражение>)]

Определитель атрибута (attribute designator) определяет свойство объекта, представляемое атрибутом. Необязательное выражение может задавать дополнительные данные для вычисления значения атрибута. Пользователь может создавать свои атрибуты, однако здесь мы ограничимся рассмотрением только наиболее употребительных предопределенных атрибутов.

Предопределенные атрибуты типов приведены в табл. 4.2. Здесь T — имя типа, N — целое, а X — вспомогательное выраже-

ние, тип которого совпадает с типом T. Тип перечисленных атрибутов, кроме T'pos и T'image, совпадает с атрибутируемым типом. Атрибут T'pos принимает целое значение, а тип данных атрибута T'image — строка.

Таблица 4.2

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
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

Предопределенные атрибуты массивов, перечисленные в табл. 4.3, упрощают запись подпрограмм и описаний настраиваемых модулей. Они, в частности, позволяют записывать границы обработки безотносительно к фактическому размеру массива.

В табл. 4.3 введены следующие обозначения: A — имя типа массива, а N — порядковый номер измерения многомерного массива. Для одномерного массива N=1, но выражение в скобках при записи атрибута можно вообще опускать. Тип результата всегда совпадает с типом индекса. Смысл определителей left, low, right, high такой же, как у определителей типов.

Таблица 4.3

Предопределенные атрибуты массивов

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

Атрибуты сигналов являются эффективным средством анализа поведения сигнала во времени (табл. 4.4, где символ S означает имя сигнала).

Таблица 4.4

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 закончен временной интервал, заданный выражением after)
S'quiet	boolean	S'quiet = not S'active
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)

4.6. Процессы

Как уже отмечалось выше, 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.

4.6.1. *Явно заданный оператор процесса*

Важнейшим составным оператором является *оператор процесса* process, синтаксис которого определен следующим образом:

<оператор процесса>::=
[<метка процесса>:] process[(<список чувствительности>)] [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* — изменения одного из сигналов, перечисленных в *списке чувствительности*, 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.

Общие правила интерпретации оператора process можно свести к следующим.

• Процесс «запускается» при изменении любого сигнала, перечисленного в списке чувствительности. Если список чувствительности 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 чувствительности обязательно должен содержать в своем теле оператор ожидания wait (см. подразд. 4.7.3). 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 действий 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.

4.6.2. *Неявно заданный оператор процесса*

Процесс может состоять из единственного оператора. В этом случае ключевые слова опускаются и оператор процесса называется *неявно заданным*. Такие процессы тоже имеют свой список чувствительности, который определяется всеми операндами. Все разновидности параллельного оператора присваивания (см. подразд. 4.8.1) являются примерами неявно заданных процессов.

4.7. Последовательные операторы

Последовательные операторы (Sequential Statement) по характеру исполнения подобны операторам традиционных языков программирования высокого уровня. 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 зрения окружения все операторы в теле процесса от его начала до оператора wait, а при отсутствии wait — до конца тела, исполняются одновременно. При использовании выражения задержки сигнала after изменение сигнала прогнозируется на еще дальше 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>

В данном разделе будут рассмотрены основные последовательные операторы VHDL.

4.7.1. *Операторы присваивания*

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 — *Waveform*, 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.

Если *временное выражение* опущено, задержка считается нулевой (так называемая *дельта-задержка*): изменение (если оно действительно предсказано при вычислении значащего выражения) заносится в календарь событий с 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 чувствительности вызывает исполнение тела процесса).

Транспортная модель задержки

Порядок исключения зависит от принимаемой *модели задержки*. Различают транспортную и инерционную модели задержки. Если в операторе присваивания присутствует ключевое слово transport, задержка считается транспортной.

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, более позднее, чем время первого из новых объявляемых переходов, и добавляются новые переходы.

Инерционная модель задержки

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

Инерционную модель применяют для описания устройств, не реагирующих на импульсы, длительность которых меньше некоторого наперед заданного значения. В этом случае, как и при 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 алгоритм моделирования

просматривает 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.

Иными словами, в инерционной модели задержки предполагается, что элемент не реагирует на сигналы, длительность которых меньше порога, равного времени задержки элемента.

Сравнение моделей задержки

Рассмотрим различные варианты модели задержки на следующем примере. Для этого будем считать объект с задержкой совокупностью двух компонентов: идеального элемента и элемента задержки. В частности, модель элемента 2И состоит из идеального вентиля и блока задержки (рис. 4.2).

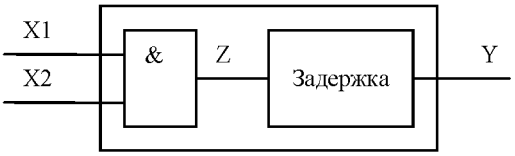

Рис. 4.2. Модель элемента с задержкой

Опишем данный элемент:

```
entity and2 is
   port (X1, X2 : in bit; Y : inout bit);
end and2;
```

Для сущности and2 создадим два архитектурных тела, в которых реализуем инерционную и транспортную задержки:

```
architecture and2_inertial of and2 is
begin
   Y <= X1 and X2 after 10 ns;
end;
architecture and2_transport of and2 is
```

```
begin
   Y <= transport X1 and X2 after 10 ns;
end;
```

Сравним результаты моделирования для данных архитектурных тел при подаче сигналов, длительность которых меньше времени задержки (рис. 4.3).

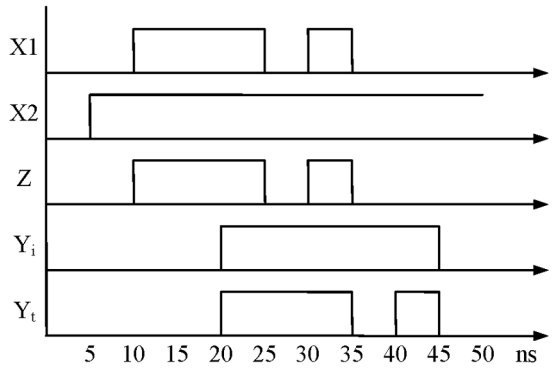

Рис. 4.3. Временная диаграмма для инерционной и транспортной моделей задержки

Для инерционной модели доступно использование дополнительной опции — можно игнорировать сигналы, длительность которых меньше определенного порога, отличного от времени задержки всего элемента. Такой вариант поведения называется резекцией.

Пример резекции сигнала длительностью менее 1 нс (более длительные импульсы проходят на выход):

```
Y <= reject 1 ns inertial X1 and X2 after 10 ns;
```

Фактически этот оператор эквивалентен следующим двум:

```
TMP <= X1 and X2 after 1 ns;
Y <= transport TMP after 9 ns;
```

4.7.2. *Оператор условия и оператор выбора*

Операторы условия if и 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.

Синтаксис *оператора условия* имеет вид

<оператор условия>::=
if <булевское выражение> then
  <оператор> {<оператор>}
{elsif <булевское выражение> then
  <оператор> {<оператор>}}
[else <оператор> {<оператор>}]
end if;

В роли 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 следует иметь в виду, что некоторые компиляторы могут оказаться неспособны выполнить прямую реализацию в аппаратуре синтаксических конструкций с большим числом уровней вложения.

Проиллюстрируем применение условного оператора описанием поведения вентиля И2, рассмотренного в подразд. 4.7.1, для которого значения времени задержки для фронта и среза не совпадают:

```
architecture and2_if of and2 is
begin
    process (X1, X2)
    variable Z : bit;
    begin
        Z := X1 and X2;
        if Z='1' and Y='0' then
            Y <= '1' after 7 ns;   -- фронт
        elsif Z='0' and Y='1' then
            Y <= '0' after 3 ns;   -- срез
        end if;
    end process;
end;
```

Синтаксис *оператора выбора* имеет вид

<оператор выбора>::=
case <ключевое выражение> is
when <вариант> {|<вариант>} => <оператор> {<оператор>}
{when <вариант> {|<вариант>} => <оператор> {<оператор>}}
end case;
<вариант>::= <константное выражение> | <диапазон> | others

Разделителем в списках выбираемых вариантов является вертикальная черта, т. е. в определении оператора выбора вертикальная черта — это не метасимвол БНФ, в отличие от любых других определений, а синтаксический элемент 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 выполняется никаких действий, то в операторной части таких вариантов записывается пустой оператор.

Практически любой разветвленный фрагмент можно описать с помощью и оператора условия, и оператора выбора. Применение одного из способов записи — дело вкуса программиста. Тем не менее, оператор case предпочтительнее, когда выбор ветви алгоритма связан с одной переменной, принимающей дискретное множество значений. Иногда набор условий легко привести к одной такой переменной.

4.7.3. *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. Прекращение состояния приостанова процесса зависит от условий, заданных 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. Указанная версия пригодна для описания процедур инициализации систем, а также фрагментов, работа которых при некоторых условиях прекращается навсегда. Обычно такой оператор завершает программы генерации тестов, означая окончание тестовой последовательности.

Список сигналов в варианте wait on эквивалентен списку чувствительности процесса: исполнение будет продолжено после того, как один из сигналов списка изменит свое значение.

Приостанов, заданный конструкцией wait until, заканчивается, когда выполнено определенное оператором условие, т. е. соответствующее выражение принимает значение true.

Вариант ожидания по времени иллюстрируется процессом generator. Данный процесс выполняется бесконечно, приостанавливаясь каждые 50 нс модельного времени, причем перед приостановом уровень сигнала clock меняется на противоположный. В момент приостанова могут быть инициированы параллельные операторы программы, в том числе другие процессы:

```
generator: process
begin
clock <= '0';
wait for 50 ns;
clock <= not clock;
end process generator;
```

Процесс, содержащий оператор wait, не может иметь списка чувствительности. Это связано с трудностями описания системы, в которой может произойти повторная инициализация действий, в то время как реакция на предыдущее событие еще не реализована (например, произошло изменение одного из инициирующих сигналов, когда время ожидания еще не вышло). Еще раз напомним, что оператор wait, как и другие последова-

тельные операторы, может размещаться только в теле процесса или теле подпрограммы.

4.7.4. *Операторы повторения*

Операторы повторения loop позволяют сокращенно записывать совокупности однотипных действий:

<оператор 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 ключевыми 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.

Оператор с ключевым словом while обязательно должен содержать в теле цикла операторы, изменяющие описанное в итерационной схеме условие. Операторы цикла повторяются, пока при вычислении условия не получается значения false. Условие проверяется каждый раз перед исполнением тела цикла.

Оператор повторения с ключевым словом for повторяется для всех значений переменной из заданного итерационной схемой диапазона.

Отметим, что присвоенные в теле цикла значения переменных могут быть исходными данными для очередного цикла. Если же в цикле выполнено присвоение значения сигналу, то в следующих операторах тела и очередных повторениях того же цикла используются старые значения, если только тело цикла не содержит операторов ожидания.

Если при моделировании (и, в частности, при описании тестовых воздействий) смысл операторов повторения практически не отличается от смысла подобных конструкций в традиционных языках программирования высокого уровня, 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. Операторы повторения должны иметь логически постоянные границы, так как в противном случае не ясно, сколько повторяющихся блоков в устройстве реально потребуется.

Кроме описания в разделе итерационной схемы, на порядок реализации повторений можно влиять с помощью дополнительных операторов: оператора перехода к следующему циклу next и оператора 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 следующей итерации>::=
[<метка>:] next [<метка оператора повторения>] [when <условие>]

Оператор exit прекращает исполнение не только текущего цикла, но и всех последующих циклов, заданных итерационной схемой исполняемого оператора. Синтаксис оператора exit имеет вид

<оператор прекращения цикла>::=
[<метка>:] exit [<метка оператора повторения>] [when <условие>]

Необязательная метка в операторах next и exit используется при записи вложенных циклов. Такая метка указывает, что прерывается не только данный цикл, но и все иерархически предшествующие ему циклы, вплоть до цикла, оператор которого помечен этой меткой.

4.7.5. *Операторы 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>]
[severity <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 программы моделирования определяется значением уровня важности. Уровень важности — это выражение (обычно константа) типа severity_level. Напомним, что данные этого типа могут принимать четыре значения, причем действия, которые выполняются при значении булевского выражения false, задаются экспериментатором как опции симулятора. По умолчанию (т. е. при отсутствии в тексте указания важности) подразумевается уровень error.

4.8. 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 был рассмотрен 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 чувствительности (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.

4.8.1. *Параллельное присваивание*

Параллельное присваивание определено в трех различных формах:

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

Безусловное параллельное присваивание

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

введение ключевого слова guarded перед правой частью оператора присваивания сигналу, этот вопрос будет рассмотрен ниже.

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 параллельное присваивание и параллельное присваивание по выбору

Эти операторы во многом сходны с последовательным условным оператором и последовательным оператором выбора соответственно — описанные действия выполняются при определенных условиях. Отличие состоит в том, что условный оператор и оператор выбора являются составными, т. е. в них условие может задавать исполнение последовательности действий, а в операторах присваивания возможно только присваивание одного значения:

<условное присваивание>::=
<приемник> <= [guarded] [<модель задержки>]
{<прогноз поведения> when <условие> else} <прогноз поведения>;
<присваивание по выбору>::=
with <ключевое выражение> select
<приемник> <= [guarded] [<модель задержки>]
«<прогноз поведения> when <вариант>,»
<прогноз поведения> when <вариант>;

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

Важно отметить, что если условный оператор if и оператор выбора case не могут работать с данными, вырабатываемыми модулями, представленными различными операторами процесса, то условное присваивание и присваивание по выбору позволяют описывать такие ситуации.

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.

4.8.2. *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 по отдельности) только после исполнения всех операторов, инициированных одним событием. В этом смысле операторы, включенные в блок, не отличаются от обычных параллельных операторов.

Объединение операторов в блоки обеспечивает следующие возможности:

• структуризацию 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, инициирует вычисление значения этого выражения и присвоение полученного значения предопределенной переменной guard. Область действия этой переменной — 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='l' and read='l') is
begin
dbus <= reg(7 downto 0) when guard and byte_sel="00" else
    reg(15 downto 0) when guard and byte_sel="01" else
    reg{23 downto 16) when guard and byte_sel ="10" else
    reg(31 downto 24) when guard and byte_sel ="11" else "zzzzzzzz";
end block select_byte;
```

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

4.9. Разрешение сигналов и шины

В предыдущих разделах мы не рассматривали случаи, 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 множество ячеек памяти, а также буферы ввода-вывода. Этому соответствуют программные модели, в которых один сигнал назначается в нескольких параллельных операторах и процессах. В языке VHDL не все 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), используя декларацию 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. 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 любой из драйверов разрешаемого сигнала меняет состояние. 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 так называемое *монтажное ИЛИ*. В качестве базового типа данных выберем bit. Введем функцию разрешения wired_OR:

```
function wired_OR (inputs : in bit_vector) return bit is
variable X : bit := '0';
begin
    for i in inputs'range loop
        if inputs(i)='1' then
            X:='1';
            exit;
        end if;
    end loop;
    return X;
end;
```

Пусть эта функция определена в некотором архитектурном теле или пакете. Там же введем подтип данных:

```
subtype WO is wired_OR bit;
```

Объявим сигнал и в разделе операторов объединим выходы элементов (в данном случае — двух элементов И):

```
signal Y : WO;
…
Y <= X1 and X2;
Y <= X3 and X4;
```

Драйверы сигнала inputs неявно рассматриваются как битовый массив, границы которого определяются стандартным атрибутом range.

На практике обычно применяют девятизначный алфавит std_logic или std_ulogic. Первый отличается тем, что для него заранее предусмотрена функция разрешения и он фактически является подтипом для std_ulogic. Если предлагаемая функция разрешения не устраивает разработчика, он может определить ее самостоятельно и создать свой подтип на основе std_ulogic.

4.10. 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, экономят время проектировщика, позволяя заменить несколько описаний сходных фрагментов алгоритма одним объявлением подпрограммы и соответствующими ссылками на нее (вызовами) в основном тексте.

Каждая подпрограмма, встречающаяся в проектном модуле, должна быть представлена телом подпрограммы в разделе деклараций этого модуля или проектного модуля, иерархически старшего по отношению к данному.

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 как результат сопоставления объектов через список соответствий. Функция 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, — 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. значение, 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.

Предусмотрено две формы списка соответствий. Форма, в которой каждый элемент списка содержит формальный параметр и сопоставляемый ему фактический параметр, разделенные сочета-

нием символов =>, называется *сопоставлением по имени*. Альтернативная форма содержит только фактические параметры и называется *позиционным сопоставлением*. Для позиционного сопоставления требуется точное совпадение порядка записи фактических параметров в списках соответствия и порядка записи формальных параметров в интерфейсном списке подпрограммы. Если какой-либо параметр не используется или принимается значение входа, определенное 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.

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 чувствительности содержит входные фактические параметры оператора вызова.

Большое значение в подпрограммах имеет *оператор возврата* return. В процедуре этот оператор прекращает ее исполнение, передавая управление вызывающей программе. Если оператор return исполнен 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 функции обязателен. Он также прекращает исполнение подпрограммы и, кроме того, выполняет присвоение значения результату, который подставляется в вызывающей программе на месте вызова функции.

4.11. 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 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 соответствующие декларации сущности и однозначно заданное поведение, например поведенческое архитектурное тело, которое следует заранее скомпилировать в библиотеку проекта или иную библиотеку, объявленную перед декларацией сущности составного проекта.

Применение SAB представляется наиболее эффективным средством создания *иерархических проектов*. Концепция SAB облегчает 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 включенных операторов и деклараций.

В раздел объявлений SAB, кроме характерных для любых архитектурных тел деклараций, таких как декларации типов, сигна-

лов, 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. В раздел операторов могут входить и другие параллельные операторы, а если таких операторов относительно много, подобное архитектурное тело называют *смешанным*, или структурно-поведенческим.

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

<декларация прототипа компонента>::=
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 одной сущностью, 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.

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. В полной форме списка соответствий порядок записи элементов в списке произволен.

Каждый компонент в архитектурном теле представлен своим оператором вхождения, метка которого определяет его имя в проекте, после метки записаны имя прототипа и списки соответствий.

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

• вызов подпрограммы инициирует исполнение тела подпрограммы, 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.

4.12. Настройка и конфигурирование компонентов

Очень часто устройства проектируются не только как изделия с наперед заданными свойствами, но и как составляющие более крупных устройств, в которых выполняются однотипные преобразования. 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), в выражениях внутри подчиненных архитектурных тел являются обычными константами. Кроме того, часто применяются параметры структурного характера, уточняющие функции, реализуемые конкретными вхождениями параметризованного модуля, и (или) его структуру. Чаще всего такие параметры объявляются 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.

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> use <индикатор привязки>;
<список вхождений>::=
<метка вхождения> {,<метка вхождения>}
| others
| all
<индикатор привязки>::=
entity <имя сущности> (<имя архитектурного тела>)
<спецификация компонента> ::= <список вхождений> : <имя компонента>

4.13. Пакеты

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. Общие для нескольких подпроектов типы и сигналы можно объявить только в пакете.

Рассмотрим правила записи пакетов более формально.

Пакет представляется двумя структурными единицами — обязательным первичным модулем *декларации пакета* (package declaration) и необязательным вторичным модулем *тела 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:

<тело пакета>::=
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.

ЛИТЕРАТУРА

1. *Угрюмов Е.П.* Цифровая схемотехника. СПб.: БХВ-Петербург, 2005.

2. *Грушвицкий Р.И.*, *Мурсаев А.Х.*, *Угрюмов Е.П.* Проектирование систем на микросхемах с программируемой структурой. СПб.: БХВ-Петербург, 2006.

3. *Амосов В.В.* Схемотехника и средства проектирования цифровых устройств. СПб.: БХВ-Петербург, 2007.

4. VHDL: Справочное пособие по основам языка / В.П. Бабак, А.А. Корченко, Н.П. Тимошенко, С.Ф. Филоненко. М.: Додэка-XXI, 2008.

5. *Поляков А.К.* Языки VHDL и VERILOG в проектировании цифровой аппаратуры. М.: СОЛОН-Пресс, 2003.

6. *Бибило П.Н.* Основы языка VHDL. М.: ЛКИ, 2007.

7. *Бибило П.Н.* Системы проектирования интегральных схем на основе языка VHDL. М.: СОЛОН-Пресс, 2005.

8. *Бибило П.Н.*, *Авдеев Н.А.* VHDL. Эффективное использование при проектировании цифровых систем. М.: СОЛОН-Пресс, 2006.

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 в 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 его возможности 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. Например, неопределенное состояние принимает выход триггера после подачи активизирующего сигнала на синхронизирующий вход при запрещенной или неопределенной комбинации сигналов на его информационных входах. Символ 'Z' соответствует высокоимпедансному состоянию порта или отключенной линии.

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), имеющих повышенное выходное сопротивление по сравнению с активными источниками. В этом случае источник, генерирующий активный сигнал, подавляет слабый. 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_logic. 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.

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

ОГЛАВЛЕНИЕ

Учебное издание

Берчун Юрий Валерьевич

Язык описания электронной аппаратуры VHDL

Редактор *С.А. Серебрякова*
Корректор *М.А. Василевская*
Компьютерная верстка *С.А. Серебряковой*

Подписано в печать 16.06.2010. Формат 60×84/16. Бумага офсетная.
Усл. печ. л. 3,72. Изд. № 119. Тираж 100 экз. Заказ

Издательство МГТУ им. Н.Э. Баумана.
Типография МГТУ им. Н.Э. Баумана.
105005, Москва, 2-я Бауманская ул., 5.

ДЛЯ ЗАМЕТОК