2. Язык описания цифровой аппаратуры VHDL

2.1. Введение

Развитие языков описания цифровой аппаратуры (**H**ardware **D**escription **L**anguage, HDL) насчитывает несколько десятилетий. К началу 80-х годов прошлого века таких языков было несколько, однако не существовало стандартного языка описания аппаратуры, сравнимого с такими стандартными языками программирования, как Си или Паскаль. Поэтому, начиная с 1983 г., Министерство обороны США выступает в роли спонсора в разработке языка описания аппаратуры на базе сверхскоростных интегральных схем (ССИС, VHSIC), языка VHDL (**V**ery high speed integrated circuits **H**ardware **D**escription **L**anguage). Первоначальное назначение этого языка заключалось в том, чтобы обеспечить возможность осуществлять обмен проектами между различными соисполнителями работ по программе VHSIC. Однако разработчики языка собрали и реализовали предложения и рекомендации многих известных специалистов в области вычислительной техники, так что язык VHDL отражает общее мнение о том, какими характеристиками должен обладать эффективный стандартный язык описания аппаратуры.

В августе 1985 г. была выпущена версия 7.2 языка VHDL, и этим завершился первый важный этап разработки языка. Версия 7.2 представляет вполне законченный язык в том смысле, что она содержит тщательно проработанные конструкции для структурного (статического) и поведенческого (функционального) представления, а также средства для документирования проектов. Однако после выпуска версии 7.2 под эгидой Министерства обороны США функции спонсора по дальнейшему развитию языка VHDL взял на себя ИИЭР, который поставил конечную цель: создание усовершенствованной, стандартной версии этого языка. Процесс пересмотра языка был закончен к маю 1987 г., когда было подготовлено и передано промышленным фирмам для рассмотрения справочное руководство по языку VHDL. В июне 1987 г. полномочные члены ИИЭР проголосовали за принятие этой версии языка VHDL в качестве стандартной и в декабре 1987 г. ИИЭР официально утвердил ее как стандарт. Его обычно называют VHDL’87. Затем, к 1993 г. он был усовершенствован, и его назвали стандарт VHDL’93.

2.2. Общие положения

Цифровая система представляет собой набор блоков (подсистем), выполняющих отдельные этапы цифровой обработки информации. Они объединены между собой связями (проводами), по которым блоки обмениваются информацией (сигналами). Цифровые системы работают во времени, преобразуя и передавая информацию. Физически эта информация отображается в электрических сигналах (напряжениях), присутствующих в системе.

Важной характеристикой цифровых систем является то, что сигналы в них передаются и обрабатываются параллельно. Причем в процессе обработки этих сигналов в каждый момент времени некоторые блоки системы активны, они обрабатывают сигналы, а другие – простаивают (они пассивны). Иллюстрацией этого может служить схема, приведенная рис. 1. Здесь показаны три логических блока. Пусть в момент времени t1 на входе блока 1 значение сигнала S1 изменяется. Блок активизируется и в момент времени $t2 > t1$, величина которого определяется задержкой в блоке 1, на его выходе появится новое значение сигнала S2. После этого блок 1 переходит в пассивное состояние, ожидая очередного изменения входного сигнала. Одновременно активизируется блок 2 и в момент времени $t3 > t2$ он передает на блок 3 результаты своей работы (сигнал S3), активизируя его. Если в этот момент сигнал S1 тоже примет новое значение, блок 1 вновь активизируется и, работая параллельно с блоком 3, начнет обработку нового значения сигнала S1.

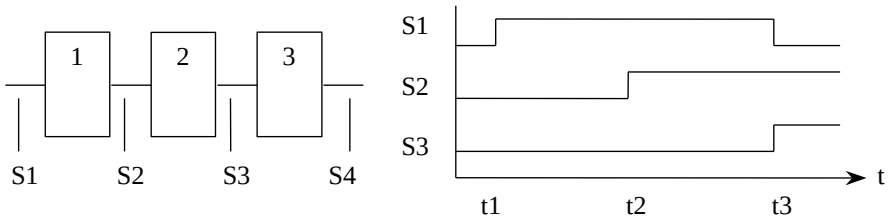

Рис.1

Здесь следует подчеркнуть, что пассивный блок не выключается вовсе. Он, как электрическая схема, состоящая из транзисторов, резисторов, диодов и т.д., постоянно включен и готов к приему и обработке новой информации. Но, до тех пор, пока на вход не будет подан новый набор сигналов, он простаивает и на его выходе значения сигналов не изменяются. Это означает, что в процессе моделирования цифровой системы нет необходимости в каждый момент времени анализировать работу всех блоков системы. Достаточно отслеживать прохождение изменений по схеме и моделировать работу только тех блоков, на входе которых происходят изменения. Такие изменения называются событиями, а метод моделирования, учитывающий только события, называется событийным.

Любой язык описания и моделирования цифровых систем должен отображать все эти их особенности. Поэтому в VHDL любая моделируемая система рассматривается как замкнутый объект, имеющий входы и выходы для связи с внешним миром. Как и любой язык программирования, VHDL имеет большой набор операторов, используемых для описания логики работы системы. Однако основными здесь являются параллельные операторы, которые используются для описания логики работы каждого блока системы. Их особенностью является то, что они выполняются не в том порядке, в котором они присутствуют в моделирующей программе, а в зависимости от изменения сигналов, передаваемых в эти операторы.

Для описания сложного блока может потребоваться целая подпрограмма или даже несколько п/п. Для решения подобной проблемы используется оператор создания параллельного блока. Фактически это операторные скобки. Открывающая скобка (оператор **process**) обозначает начало блока, закрывающая скобка (оператор **end process**) обозначает окончание блока. На вход такого блока поступают сигналы от других подсистем. Внутри блока они обрабатываются и передаются на выход блока в виде выходных сигналов. Для такой обработки используются последовательные операторы языка. Эти операторы аналогичны операторам обычных языков программирования, они выполняются в том порядке, в котором они присутствуют в программе. Отметим, что последовательные операторы могут присутствовать только внутри параллельного блока или подпрограммы.

В VHDL, кроме обычных для алгоритмических языков понятий константы и переменной, используется понятие сигнала, которое является базовым в языке описания аппаратуры. Сигналы являются представлением в модели на VHDL состояния проводников (физических сигналов) в моделируемом цифровом устройстве.

Сигналы, как и переменные, имеют некоторые значения, которые им присваиваются. Однако если переменную характеризует только значение, то сигнал характеризуется еще и моментом модельного времени, в который этот сигнал имеет данное значение. Можно сказать, что состояние сигнала – это пара: момент модельного времени/значение сигнала в этот момент. Моделирующая программа (симулятор) сохраняет значения всех (или только требуемых) сигналов для того, чтобы по окончании моделирования построить временную диаграмму.

Для того, чтобы особо выделить сигналы, в VHDL оператор присвоения значения сигналу записывается в виде (<=), тогда как оператор присвоения значения переменной – (:=). Например, если a и b – сигналы, а c и d – переменные, то правильной будет запись

a <= b; d := c;

Символ точка с запятой является признаком окончания оператора языка VHDL. Отметим, что оператор (<=) является параллельным.

Важной особенностью параллельных операторов является понятие задержки, что отражает реальные временные характеристики работы отдельных блоков системы. Например, запись

a <= b **after** 50 ns;

обозначает, что выходной для этого оператора сигнал a примет тоже самое значение, что и входной сигнал b, с задержкой 50 нс.

2.3. Как работает моделирующая программа

Для того, чтобы правильно использовать язык VHDL, следует представлять, как выполняется программа, написанная на этом языке. При описании моделируемой системы разработчик записывает параллельные операторы, которые отображают логику работы каждого блока моделируемой системы. Для объединения таких операторов он использует сигналы.

Подобную программу удобно анализировать, предположив [4], что она выполняется на параллельном компьютере, где каждый параллельный оператор выполняется на отдельном процессорном элементе (ПЭ). Процессорные элементы объединены линиями связи, по которым передаются сигналы. Операторы выполняются параллельно друг другу и независимо друг от друга. Все определяется только изменениями входных сигналов. Сигнал при этом используется одним ПЭ для сообщения другим ПЭ факта окончания обработки входных данных. Кроме того, сигнал используется для передачи исходных и промежуточных данных между ПЭ.

При запуске программы на исполнение вначале все переменные принимают заданное начальное значение. Затем во всех ПЭ начинают свое исполнение параллельные операторы. При окончании обработки ПЭ генерируют вычисленные сигналы и останавливаются. Эти сигналы, пройдя через соответствующие им линии связи до других ПЭ, запускают в них вычислительные процессы. Таким образом, по вычислительной системе проходят волны запусков. Такое функционирование вычислительной системы может продолжаться до тех пор, пока оно не будет остановлено с консоли, либо не окончится длительность этапа моделирования, заданная пользователем.

Реально симулятор (моделирующая программа), как правило, реализован на однопроцессорном компьютере. От разработчика он получает задание на моделирование, в котором указано время моделирования и временная диаграмма входных сигналов. Симулятор сначала устанавливает начальные значения всех внутренних и выходных сигналов и переменных. Далее он подает на все входы входные сигналы. Будем считать, что начальные установки схемы соответствуют этим значениям входных сигналов, в противном случае начинается моделирование процесса начальной установки схемы. Пусть в момент t1 (рис.2) первый входной сигнал изменит свое значение. Симулятор определит, на какие операторы поступает это изменение, и на оси модельного времени поставит отметки, соответствующие моментам, когда должна появиться реакция этих операторов. Эти отметки легко рассчитать, т.к. величина задержки явно указана в каждом операторе. Будем считать, что таких операторов 3, поэтому появляются отметки t2, t3, t4. Т.е. на эти моменты времени запланированы соответствующие изменения выходных сигналов.

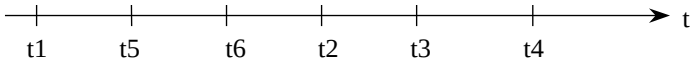

Рис.2

Если никаких изменений в схеме не будет, можно прямо перейти к моменту t2 и рассчитывать запланированные ранее изменения сигналов. Однако если в момент t5 изменит свое значение второй входной сигнал и это изменение приведет к реакции в момент t6, приходится вносить изменения в расположение отметок на оси модельного времени и после t1 следует перейти к моменту t5.

Таким образом, моделирование идет по дискретным приращениям модельного времени. Выполняя моделирование в момент t_i модельного времени, симулятор производит все

изменения сигналов, запланированные на текущий момент времени, на основании известных значений сигналов устройства, наблюдаемых в начале текущего цикла моделирования. Выполнив все заданные для это момента изменения сигналов, программа моделирования заканчивает текущий цикл моделирования и переходит к следующему моменту t_{i+1} модельно времени, который хотя бы на единицу больше текущего.

2.4. Дельта-задержка сигналов

Особая ситуация возникает в модели устройства, если оператор присваивания нового значения сигналу устанавливает нулевую задержку. Например, если в таком операторе отсутствует секция **after**, то считается, что задержка изменения сигнала составляет 0 нс, т.е. сигнал принимает новое значение в текущий момент времени моделирования.

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

В модели можно попытаться этими временными задержками пренебречь (например, когда разработчика интересует только функциональное поведение модели, а ее детальное временное поведение не рассматривается). Часто, на начальных этапах проектирования системы, просто нет еще достоверной информации о задержках в компонентах схемы, тогда основной задачей является соблюдение функциональной корректности модели устройства.

Для решения этой проблемы, типовой для систем дискретного моделирования, в семантику языка VHDL введен специальный механизм. Суть его в следующем: предполагается, что симулятор, закончив текущий цикл моделирования для момента t_i модельного времени, не сразу переходит к следующему моменту модельного времени $t_{i+1} > t_i$. Он проверяет, имеются ли изменения сигналов, вновь запланированные на момент t_i.. Это может произойти при указании в программе на VHDL нулевых задержек. Если выявлены новые изменения сигналов на тот же момент t_i модельного времени, симулятор выполняет новый цикл моделирования, отрабатывая эти изменения. Это может продолжаться до тех пор, пока не будет определено, что по результатам текущего цикла моделирования не появилось новых сигналов, запланированных на момент t_i, после чего симулятор переходит к моделированию момента t_{i+1} модельного времени.

Если первый из рассмотренных циклов работы симулятора соответствовал моменту t_i модельного времени, то каким точкам на оси модельного времени соответствуют следующие циклы? С точки зрения модели на VHDL, которая указывает, что некоторый сигнал изменяется с нулевой задержкой, можно было бы сказать, что симулятор “крутится”, оставаясь в том же моменте t_i модельного времени.

Однако можно рассматривать ситуацию и по-иному. Временная последовательность таких дополнительных циклов отражает, в определенной степени, и зависимости сигналов (логику влияния одних сигналов на формированные других), и последовательность этих влияний. Выстроить эту последовательность циклов работы симулятора позволяет введенное в языке VHDL понятие дельта-задержки, которая обозначается символом Δ. Дельта-задержке не приписывается никакого числового значения. Эта абстракция используется только для упорядочивания последовательности событий в модели и отрабатывающих их циклов работы симулятора. На формируемых симулятором временных диаграммах, на оси модельного времени дельта-циклы не отражаются. Все соответствующие значения сигналов выстраиваются на диаграммах в одной вертикали, соответствующей моменту времени t_i.

2.5. Структура описания моделируемой системы

В языке VHDL моделируемая логическая схема рассматривается как объект проекта. Таким объектом может быть как часть цифровой системы, так и вся система в целом. Описание объекта моделирования состоит из декларативной части и описания архитектуры.

В декларативной части описываются связи объекта с внешним миром — входы и вы-

ходы объекта. Это — спецификация его интерфейса. Она имеет вид:

entity <имя объекта> **is**
<интерфейсные операторы>
end entity <имя объекта>;

Как уже отмечалось, символ ; (точка с запятой) является синтаксическим элементом конструкций языка VHDL, отмечающим окончание оператора языка.

В описании архитектуры определяется функция моделируемого объекта, т.е. дается описание того, как объект формирует выходные сигналы на основании входных сигналов и внутреннего состояния объекта. Его конструкция следующая:

architecture <имя описания> **of** <имя объекта> **is**
<объявления объекта>
begin
<тело объекта>
end architecture <имя описания>;

Описание архитектуры разделено на две части. До ключевого слова **begin** выполняются все объявления, после него — исполнимые параллельные операторы. В первой части могут быть объявлены тип и подтип, константа, глобальная переменная, сигналы, объявление и тело процедуры и функции и т.д. Однако все они видимы только в пределах этой архитектуры. Для простых объектов секция <объявления объекта> может отсутствовать.

Последовательность записи параллельных операторов в теле архитектурного описания значения не имеет. Порядок выполнения или, как иногда говорят, порядок срабатывания параллельных операторов определяется не последовательностью их записи в тексте программы между **begin** и **end**, а другими правилами. Основным принципом здесь является управление от потока изменений сигналов. Событие изменения значения сигнала, являющегося входным для параллельного оператора, запускает срабатывание данного оператора, т. е. его выполнение. Для параллельного оператора типа **process** имеется возможность явно указывать сигналы, изменение которых запускает процесс. Для параллельных операторов присваивания значений сигналам изменение любого входного сигнала инициирует их срабатывание.

Отметим, что в соответствии со стандартом VHDL’87 применение ключевых слов **entity** и **architecture** вслед за **end** не является обязательным. Но в программах на VHDL оператор **end** используется достаточно часто. Поэтому хорошим стилем программирования является явное указание того, какой “открывающей скобке” соответствует текущий оператор **end**. В стандарте VHDL’93 это обязательно.

Разделение описания моделируемой системы на декларативную и архитектурную части объясняется тем, что на разных этапах проектирования архитектурная часть может иметь разную степень детализации и вообще их может быть несколько. Выбор конкретного вида архитектурного описания осуществляется при задании конфигурации проекта.

2.6. Комментарии

Комментарий начинается двумя знаками минус и продолжается до конца строки. Он может присутствовать в любом месте VHDL-описания. Комментарий не учитывается при моделировании схемы.

2.7. Идентификаторы

Для того, чтобы можно было работать с константой, переменной или сигналом, им необходимо присвоить идентификатор (имя). Имя может состоять из нескольких букв, цифр и знаков подчеркивания. Первый символ имени — буква. Два подряд подчеркивания недопустимы. Недопустимо подчеркивание и в конце имени. В VHDL нет различия между пропис-

ными и строчными буквами. Идентификатор AbC7 эквивалентен aBC7, но A_3 не эквивалентен A3.

Пример:
правильные идентификаторы: carry_OUT, Dim_Sum, Count_SUB_2goX, aBcDe,
неправильные идентификаторы: 7AB (начинается с цифры), A$B (специальный символ $), SUM_ (оканчивается подчеркиванием), PI_ _A (два подчеркивания подряд).

2.8. Типы данных

В языке VHDL реализована строгая типизация. Это означает, что смешение различных типов в одной операции будет восприниматься как ошибка. Средства строгого контроля типов играют ответственную роль, поскольку позволяют уточнять намерения разработчика.

Описание типа имеет следующий синтаксис:

type <имя> **is** <определение типа>.

В языке используются следующие типы данных:
- простые (скалярные) типы;
- составные типы;
- указательные типы;
- файловые типы.

Далее будем рассматривать только первые два типа. В группу скалярных типов включены числовые, перечислимые и физические типы, группу составных образуют строки, массивы и записи.

2.8.1. Числовые типы

К этим типам относятся целый, подтипами которого являются натуральный и положительный, и действительный.

2.8.1.1. Целые типы

Этот тип объявляется с помощью ключевого слова **integer**. Он позволяет использовать как положительные, так и отрицательные целые числа. Тип **natural** представляет неотрицательные, а тип **positive** — положительные целые числа. При этом, как правило, с помощью синтаксической конструкции

range <целое число> **to/downto** <целое число>

указывается диапазон изменения чисел. В этом случае параметр **integer**, **natural** или **positive** можно не использовать. Например:

tape t1 **is range** 1 **to** 200;
tape t2 **is range** 50 **downto** 10;

2.8.1.2. Действительный типы

Для представления действительных чисел используется тип **real**. Он имеет диапазон изменения от –1.0e+38 до +1.0e+38. Может показаться странным, что в языке описания аппаратуры имеется тип **real**. Однако этот тип очень полезен для высокоуровневых поведенческих описаний, например аналого-цифрового интерфейса или алгоритма обработки сигналов [2].

При использовании этого типа следует иметь в виду, что не все системы его поддер-

живают. Это относится, например, к среде разработки OrCAD Express [3].

2.8.2. Перечислимые типы

Это логические типы **boolean** и **bit**, символьный **character** и тип, определяемый списком значений. Сюда же можно отнести и целые типы, задаваемые диапазоном.

2.8.2.1. Логические типы

Тип **boolean** состоит из значений **true** (истина) и **false** (ложь). Все операторы **if** в языке должны проверять объекты или выражения этого типа. Тип **bit** состоит из значений 0 и 1.

2.8.2.2. Символьный тип

Тип **character** по сути представляет набор символов кода ASCII.

2.8.2.3. Тип, определяемый списком значений

При моделировании на абстрактном уровне значениям сигналов удобно сопоставлять имена, отражающие их смысл. В этом случае используется задание перечислимого типа списком значений с помощью конструкции вида:

type <имя> **is** (<значение1>, <значение2>,…, <значение_n>)

Например:

tape states **is** (on,off);

2.8.3. Физические типы

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

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

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

type <имя> **is range** <число> (**to**/**downto**) <число>
units
ed;
ked = X ed;
…
end units <имя>;

Здесь:
ed — имя первичного модуля (первичной единицы);
ked — имя вторичного модуля.
Например:

type length **is range** 0 **to** 1e9

```
units
        um;   -- микрон
        mm=1000 um;
        m=1000 mm;
        mil=254 um; -- 0,001 дюйма
        inch=1000 mil; -- дюйм
end units length;
```

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

Величина, описанная типом length, может принимать значения от 0 um до 1e9 um. Для задания значения величины может использоваться и действительное число. Оно будет округлено до ближайшего целого в основных единицах измерения. Так, для типа length запись 0.1 inch, 2.54 mm и 2 540528 mm будут интерпретированы как одно и тоже число 2540 urn.

2.8.3.1. Операции над физическими типами

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

1) операции сложения и вычитания могут производиться только над данными одного и того же типа; при этом получается результат того же типа, что и операнды;

2) значение физического типа может быть умножено на целое или действительное число; результату будет присвоен физический тип операнда;

3) значение физического типа можно делить на целое или действительное число, результат будет того же физического типа;

4) для операции деления оба операнда должны быть одного и того же физического типа, в этом случае будет получен результат целого типа.

2.8.3.2. Описание времени

VHDL содержит встроенный физический тип — время. Он описывается следующим образом:

```
type time is range 0 to 1e20
units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        s = 1000 ms;
        min = 60 s;
        hr = 60 min;
end units time;
```

В качестве базовой выбрана фемтосекунда (10^{-15} с). Диапазон (до 10^{20}) при такой базовой единице обеспечивает представление максимального периода времени до 100 000 с (27,7 ч). И базовую единицу времени, и диапазон может выбрать пользователь, однако диапазон ограничивается длиной слова инструментальной машины.

2.8.4. Строковый тип

Тип **string** — это строка символов, например **string** (0 **to** 9).

2.8.5. Массив

Массив представляет собой набор элементов одного и того же типа. Позиция каждого элемента задается скалярным значением — индексом. Для задания массива используется

конструкция:

type <имя> **is array** (<число> **to**/**downto** <число>) **of** <тип>

Например:

type word **is array** (0 **to** 31) **of bit;**

Разновидностью массива является битовый вектор. Для его определения используется стандартное имя **bit_vector**. Им, однако, удобней пользоваться при определении переменной, а не типа.

2.8.5. Записи

Тип запись — это составной тип, состоящий из ряда полей. Описание типа записи имеет следующий синтаксис:

type <имя> **is record**
<описание типа 1>;
<описание типа 2>;
…
<описание типа n>;
end record <имя>;

Например, тип date (дата) можно описать как тип запись следующим образом:

type date **is record**
day: **integer range** 1 **to** 31;
month: month_name;
year: **integer range** 0 **to** 3000;
end record;

Здесь тип month_name (имя месяца) — перечислимый тип, содержащий имена месяцев.

2.9. Декларация объектов vhdl

Как уже отмечено, в VHDL есть три типа объектов — константы, переменные и сигналы. Для того, чтобы их можно было использовать, нужно сообщить моделирующей системе, что это будут за объекты, каковы их свойства. Для этого используются процедуры декларации (объявления, описания) объектов. Объект создается, когда дается его описание.

2.9.1. Объявление константы

Константа —это объект, значение которого не может изменяться. Для объявления констант используется оператор **constant**, за ним через пробел следует имя (идентификатор) константы, двоеточие, тип константы и оператор присвоения константе значения.

Примеры:

constant tieoff : mvl : = ‘1’; — это константа перечислимого типа (типа mvl), имеющая значение ‘1’;

constant ovfl_msg : **string** (1 **to** 25) := "Переполнение аккумулятора"; — это константа строкового типа длиной 25 символов;

constant int_vector : **bit_vector** (0 **to** 7) :=•00001000;

constant coeff : **real** := 4.217;

2.9.2. Объявление переменной

Переменные — это объекты, значения которых могут меняться. Изменение значений переменных осуществляется путем выполнения оператора присваивания для переменной; (например, a : = b + c;). Операторы присваивания для переменных не имеют временного параметра (т.е. их результат сказывается немедленно). В связи с этим переменные не имеют прямого отображения в аппаратных средствах, но полезны для реализации алгоритмов работы последних. Для объявления переменной используется оператор **variable**, за ним через пробел следует одно или через запятую несколько имен (идентификаторов) переменных, двоеточие, тип переменной и необязательный оператор присвоения начального значения.

Примеры:

variable run, fun : **boolean** := **false**;
variable count : **integer** := 0;
variable addr : **bit_vector** (0 **to** 11);

2.9.3. Объявление сигнала

Сигналы — это объекты, значения которых могут меняться и которые имеют временные параметры. Сигналы соответствуют физическим линиям (проводникам), которые соединяют элементы схемы. Для объявления сигнала используется оператор **signal**, за ним через пробел следует одно или через запятую несколько имен (идентификаторов) сигналов, двоеточие и тип сигнала.

Примеры объявления сигналов:

signal clk, resetn : **bit**;
signal counter : **integer range** 0 **to** 31;
signal instruction : **bit_vector** (15 **downto** 0);

Значения сигналов меняются операторами назначения сигналов, например
a <= в + с **after** 50 ns;
d <= е **nor** с;

В первом примере оператора назначения сигнала указано “через 50 нс”. Это означает, что сигнал А может в принципе принимать свое новое значение спустя 50 нс после текущего момента времени моделирования. Здесь сказано “в принципе” потому, что в модели может быть более одного оператора назначения, действующего для сигнала а.

Для сигналов, как и для переменных, указание начального значения не обязательно. В этом случае начальное значение будет равно левой границе диапазона изменения соответствующего типа. Так, если, например, объявлен сигнал в виде

signal a : **real**;

его начальное значение будет равно –1.0e+38

2.10. Операции

Язык VHDL имеет полный набор операций для работы с предусмотренными типами данных. Принято выделять пять групп операций.

1. Логические операции: **not** (НЕ), **and** (И), **or** (ИЛИ), **nand** (И-НЕ), **nor** (ИЛИ-НЕ), **xor** (исключающее ИЛИ).

2. Операции сравнения: = (равно), /= (не равно), < (меньше), <= (меньше либо равно), > (больше), >= (больше либо равно).

3. Операции сложения: + (сложение, присвоение знака +), - (вычитание, присвоение знака -), & (конкатенация, объединение).

4. Операции умножения: * (умножение), / (деление).

5. Смешанные операции: ** (возведение в степень), **abs** (абсолютное значение).

Логические операции определены для типа **bit**, типа **boolean** и для одномерных массивов, в которых каждый элемент имеет тип **bit** или тип **boolean**; таким образом, они определены и для типа **bit_vector**.

Все операции сравнения возвращают результат типа **boolean** (т. е. значение **true** или **false**). Все операции сравнения имеют левые и правые операнды (например, a /= b). Обе операции контроля эквивалентности (= и /=) в качестве левого и правого операндов могут иметь любой тип, при условии, что это один и тот же тип. Если тип составной (например, массив), проверка выполняется по его элементам. Оба операнда составного типа считаются равными, если и только если равны все их соответствующие элементы.

Операции сравнения (<, <=, >, >=) определены для любого скалярного типа; например, перечислимый тип

type mvl **is** (‘0’, ‘1’, ‘z’)

подразумевает упорядочение слева направо, так что имеет место отношение ‘1’ < ‘z’. Операции отношения определяются также для одномерных массивов, в которых каждый элемент — это дискретный тип. Предположим, например, что мы описали трехэлементный многозначный тип массив следующим образом:

type three_bit_mvl **is array** (0 **to** 2} **of** MVL;

В этом случае операции отношения могут выполняться над трехэлементными векторами. Сравнение начинается с минимального индексного значения и продолжается до максимального. Так, например, значение выражения (‘0’, ‘1’, ‘1’) <= (‘0’, ‘z’, ‘1’) есть истина.

Арифметические операции группы сложения (+ и -) определены для типов **integer** и **real**. Для типа **bit_vector** они не определены, так что пользователь должен сам предусматривать подпрограммы для этой цели. Операция конкатенации & выполняет свою обычную функцию, т. е. объединяет два одномерных массива с образованием единого одномерного массива, длина которого есть сумма длин обоих массивов-операндов. Например, (‘0’, ‘1’, ‘1’) & (‘0’, ‘z’, ‘1’) = (‘0’, ‘1’, ‘1’, ‘0’, ‘z’, ‘1’).

Операции присвоения знака (“+” и “-”) — это унарные операции, при помощи которых перед одиночными объектами типа **integer** или **real** ставится арифметический знак, обозначающий положительное или отрицательное значение. И опять для битовых векторов формирование дополнительного или обратного кода должно производиться специальными подпрограммами пользователя.

Для типов **integer** и **real** определены операции умножения и деления (* и /).

Операции и соответствующие операнды встречаются, естественно, в выражениях например, (a > b) или (c /= d). В сложных выражениях следует использовать требуемое количество пар круглых скобок.

2.11. Последовательные операторы языка vhdl

Еще раз отметим, что последовательные операторы присутствуют только внутри оператора **process** и используются для описания поведения сложного объекта моделирования. Они выполняются последовательно, друг за другом. При их выполнении модельное время останавливается.

2.11.1. Оператор присвоения значения переменной

Как уже отмечалось, это оператор “:=”. Приведем пример использования этого оператора для присвоения значений отдельным частям битового вектора. Объявим следующие переменные

variable a, b, c, d : **bit_vector** (1 **downto** 0);
variable z : **bit_vector** (0 **to** 7);

тогда значения первым и вторым четырем битам вектора z можно присвоить следующим образом

```
z(0 to 3) := c & d;
z(4 to 7) := (not a) & (a nor b);
```

2.11.2. Операторы управления

Такие операторы есть во всех языках программирования высокого уровня и принципы их использования в VHDL аналогичны общепринятым.

2.11.2.1. Оператор if

Полная форма оператора **if** (если) имеет следующий вид:

```
if <условие_1> then
        <последовательность операторов>
elsif <условие_2> then
        <последовательность операторов>
else
        <последовательность операторов>
end if;
```

Оба условия должны быть типа **boolean**. Возможно наличие любого числа фраз **elsif**. Фразы **elsif** u **else** являются необязательными.

2.11.2.2. Ореpатор case

Оператор **case** (выбор) позволяет выбрать одну из альтернатив, определяемых значением выражения, присутствующего в операторе:

```
case <выражение> is
        when <вариант_1> => <оператор_1>;
        when <вариант_2> => <оператор_2>;
        when <вариант_3> => <оператор_3>;
        when others => <оператор_4>;
end case;
```

Это выражение должно быть дискретного типа или типа одномерного массива символов, значения которых могут быть представлены как строки или строка битов. Выбор (вариант) должен быть такого же типа, как и выражение. Должны быть рассмотрены все возможные случаи. Вариант “**others**” (другие) облегчает выполнение этого условия.

Например

```
case n is
        when 0 => z <= 0;
        when 1 => z <= 1;
        when 2 => z <= not Z;
        when others => z <= z;
end case;
```

2.11.2.3. Оператор loop

Это оператор цикла. Проще всего организовать цикл использую фразу **for**. Например,

цикл

```
for i in 0 to 3 loop
a(i) := 2**i;
end loop;
```

рассчитывает степени 2 и заносит их в первые 4 элемента массива a. Фраза **while** позволяет выполнить выход из цикла по некоторому условию. Например, цикл

```
sum : = 0;
i := 1;
sum_int: while i <= n loop
sum := sum + i;
i := i + 1;
end loop sum_int;
```

вычисляет сумму первых n целых чисел. Здесь используется метка оператора sum_int.

2.11.2.4. Оператор next

Этот оператор используется для перехода на указанную в нем метку при выполнении некоторого условия. Например, в цикле

```
sum : = 0;
i := 1;
sum_int: while i <= n loop
next sum_int when i = 3;
sum := sum + i;
i := i + 1;
end loop sum_int;
```

выполняется вычисление той же самой суммы, что и в предыдущем примере, но без числа 3, поскольку эта итерация пропускается при помощи оператора **next**. Его можно использовать и для того, чтобы выйти из цикла.

2.11.2.5. Оператор exit

Оператор **exit** употребляется для того, чтобы завершить выполнение и закрыть оператор цикла. Пример

```
sum : = 0;
loop
val(x);
exit when x < 0;
sum := sum + x;
end loop;
```

Здесь показан потенциально бесконечный цикл, который прибавляет к накопленной сумме значения, поступающие из подпрограммы. Выход из этого цикла будет происходить в случае, когда подпрограмма val(x) выдаст отрицательное значение.

2.11.2.6. Оператор null

Оператор **null** (пустой оператор) не представляет действий. Он употребляется, чтобы точно указать, что нет действий. Типичное применение — в операторе **case**, чтобы опреде-

лить действия во всех случаях.

2.11.2.7. Подпрограммы

Как и в традиционных языках программирования, здесь вводится понятие подпрограммы, облегчающее многократное использование кода и параллельную разработку проекта многими специалистами. Если подпрограмма возвращает одно значение, ее оформляют как функцию, если много значений — это процедура. Большое число процедур и функций объединяются в пакеты.

2.11.2.7.1. Функции

Описание функции начинается оператором **function**, далее следует имя функции, затем в круглых скобках входные параметры, если они есть, оператор **return**, после которого указывается тип возвращаемого значения и слово **is**. Далее следует необязательный раздел описаний, оператор **begin** и выполняемые операторы функции. Завершает описание функции оператор **end** после которого вновь указывается имя функции. Среди выполняемых операторов обязательно должен быть хотя бы один оператор **return**, возвращающий рассчитанное значение. Пример функции

```
function maj3 (signal x,y,z : bit) return bit is
begin
        return (x and y) or (x and z) or (y and z);
end maj3;
```

2.11.2.7.2. Процедуры

Процедуры оформляются также как функции. Только в начале отсутствует оператор **return** и тип возвращаемого значения. Оператор **return** отсутствует и в теле процедуры.

2.12. Параллельные операторы языка vhdl

Как уже отмечалось, в исполнимой части описания архитектуры, т.е. после оператора **begin**, дается описание поведения моделируемого объекта. Оно содержит только параллельные операторы. Основными такими операторами являются оператор присвоения значения сигналу и оператор **process.**

2.12.1. Параллельные операторы присвоения значения сигналу

Это операторы вида

<имя сигнала> <= <выражение>;

В выражение, присутствующее в правой части оператора, входят сигналы, переменные и константы. Тип выражения должен соответствовать типу сигнала, имя которого указано в левой части оператора. Этот оператор будет выполняться при изменении любого сигнала, присутствующего в выражении. Совокупность этих сигналов называется списком чувствительности параллельного оператора.

Простейшим примером такого оператора является строка

```
a <= b;
```

Чтобы отразить реальные временные характеристики работы блока системы указывается величина задержки, например

```
a <= b after 50 ns;
```

2.12.2. Задержки в модели устройства

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

2.12.2.1. Инерционная задержка

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

Для описания этого вида задержек распространения сигналов в языке VHDL используется понятие инерционной задержки в операторе присваивания значения сигналу — ключевое слово **inertial**. По умолчанию, если в программе не указано это ключевое слов или что либо иное, используется модель инерционной задержки. Минимальная длительность сохранения установившегося значения входного сигнала (иными словам — минимальная длительность импульса на входе) по умолчанию считается равной времени распространения сигнала с этою входа до указанного выхода. Таким образом, операторы

```
z <= inertial (x or y) after 7 ns;
z <= (x or y) after 7 ns;
```

обозначают инерциальную задержку.

Однако в реальных схемах такая прямая зависимость между задержкой распространения сигнала и минимальной длительностью входного импульса выполняется не всегда, даже для простых схем. Поэтому, если разработчик схемы имеет более детальную информацию об инерционных задержках и минимальных длительностях импульсов на входах, он может явно указать минимальную длительность входных импульсов, отличную от указанной задержки формирования выходного сигнала, для чего используется ключевое слово **reject**. Например

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

В этом случае входной сигнал должен продержаться в новом значении не менее 3 нс, но соответствующее изменение на выходе наступит только через 7 нс после изменения сигнала на входе.

2.12.2.2. Транспортная задержка

При моделировании реальных цифровых систем иногда необходимо, чтобы изменения сигналов любой длительности не отбрасывались, а отрабатывались системой моделирования и влияли на формирование выходных сигналов. Во-первых, это учет задержек на распространение сигналов по проводам, соединяющим отдельные логические элементы. Задержки в линиях связи по своей природе существенно отличаются от задержек в логических элементах. Они имеют малую инерционность, могут передавать импульсы очень малой дли-

тельности.

Во-вторых, использование таких задержек удобно при моделировании устройств на верхнем уровне иерархии. Здесь задержка формирования сигналов на выходе сложного устройства может быть весьма значительна, однако это не повод отфильтровывать короткие импульсы входных сигналов. Если, например, мы моделируем устройство с памятью, время доступа к которой 50 нс, то это вряд ли будет основанием требовать минимальную длительность импульса для всех сигналов, используемых при чтении из памяти, равную 50 нс.

В этом случае используется ключевое слово **transport**. Например

z <= **transport** x **after** 7 ns;

2.12.3. Оператор process

Этот оператор позволяет параллельным образом описать логику работы блоков системы любой степени сложности. Оператор **process** — это ограниченный фрагмент текста, содержащий раздел описания и исполняемый раздел. Однако сигналы не могут быть декларированы внутри процессов.

имя_процесса: **process** (список чувствительности)
< Раздел описаний процесса >
begin
< Выполняемые операторы процесса >
end process имя_процесса;

Метка процесса (имя_процесса) не обязательна, но если она присутствует в операторе **process**, она обязательно должна повториться и в операторе **end process.**

Процесс активируется (выполняется) только в том случае, когда происходит изменение какого-либо сигнала в списке чувствительности этого процесса. При первом изменении одного из сигналов списка чувствительности, операторы процесса выполняются и результат выполнения поступает на выход блока, для чего внутри процесса должен присутствовать оператор присвоения значения сигналу. Новое значение сигнала может быть использовано другими блоками схемы, однако повторное выполнение операторов процесса будет выполнено только после очередного изменения в списке чувствительности. Отметим, что в списке чувствительности могут присутствовать сигналы, не используемые внутри оператора **process**.

2.13. Структурное описание объекта моделирования

Это описание позволяет описать систему как совокупность компонентов–подсистем, объединенных сигналами. Подсистемы также являются объектами моделирования, но находятся на более низком уровне иерархии описания проектируемого устройства. Каждая подсистема, в свою очередь, может быть представлена совокупностью подсистем более низкого уровня, и так далее, пока на каком-то уровне не будет задано поведенческое описание архитектуры компонента, или не будет использован предопределенный компонент. Компоненты могут проектироваться разными разработчиками и как правило организованы в виде библиотек проектирования.

2.13.1. Декларация компонента

Для того чтобы один объект моделирования мог быть включен в состав другого объекта, его необходимо декларировать (объявить) как компонент. Декларация компонента должна полностью совпадать с декларацией соответствующего ему объекта моделирования, но ключевое слово **entity** заменяется ключевым словом **component**:

component <имя компонента> **is**
<интерфейсные операторы>
end <имя компонента>;

Идентификатор <имя компонента> вводит имя компонента описываемого вида. Мы будем говорить о нем как о типе компонента. В структурном описании архитектурного тела может быть несколько компонентов этого типа. Назовем их компонентами–экземплярами или просто экземплярами.

Декларация компонентов может размещаться в декларативной части объекта моделирования (вместе с описаниями типов, подтипов), но может быть расположена и в тех файлах, в которых описаны объекты моделирования, используемые в качестве компонентов.

2.13.2. Включение компонента в модель объекта

Структурное описание объекта моделирования, находящегося на верхнем уровне иерархии, состоит из так называемых назначений компонентов. В рамках этих назначений определяются связи с другими объектами. В структурном описании объекта мы “собираем” его из других компонентов. Задав декларацию компонентов, мы как бы написали документ, спецификацию, проектируемого объекта, определив, какого типа компоненты в него входят.

Оператор назначения компонента имеет следующий синтаксис:

<индивидуальная метка>: **component** <имя компонента>
port map <список включения>;

Операторы назначения компонентов, наряду с процессами, являются параллельно выполняющимися.

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

Секция связей портов (входов и выходов) компонента **port map** определяет связи данного компонента с сигналами объекта и портами других компонентов. Они задаются списком включения, который содержит список фактических сигналов, связанных с портами компонента. Он имеет следующий синтаксис:

{<имя порта> => <имя сигнала> | <**open**>}, ….

либо просто

{<имя сигнала> | <**open**>}, ….

Каждый элемент в списке (сигнал) связывается с портом объекта моделирования, описанного на один уровень выше, или с его внутренними сигналами, или является независимым (“висящим в воздухе”), что помечается ключевым словом **open**. Этим словом следует помечать выходные порты компонента не соединяются ни с одним сигналом или входом другого компонента.

В первом случае мы имеем ключевое описание включения компонента. Здесь ключом является <имя порта>, это имя интерфейсного сигнала компонента. Поэтому в списке включения конструкции вида <имя порта> => <имя сигнала> могут следовать в произвольном порядке. Эта конструкция напрямую указывает к каким внешним сигналам (проводам) подключаются входы и выходы компонента.

Во втором случае имя порта компонента не указывается, здесь ключа нет. Это позиционное описание. Здесь порядок следования имен сигналов (внешних по отношению к ком-

поненту) должен строго соответствовать порядку, в котором описаны интерфейсные сигналы (входы и выходы) в описании интерфейса компонента.

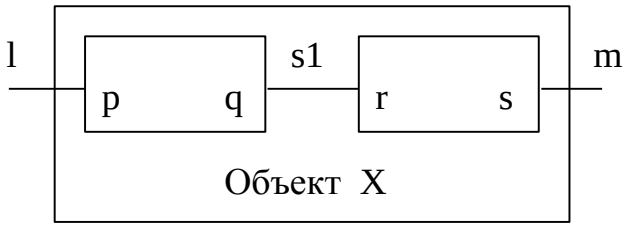

Рис.3

Рассмотрим пример. Предположим, что система X состоит из двух компонентов, a и b, и что они имеют следующие описания интерфейсов:

```
entity a is
  port (p: in bit; q: out bit);
end a;

entity b is
  port (r: in bit; s: out bit);
end b;
```

Предположим далее, что компоненты a и b соединяются и образуют систему X так, как показано на рис. 3. В этом случае систему X можно представить следующим описанием объекта и соответствующим архитектурным телом:

```
entity X is
  port (l: in bit; m: out bit);
end X;

architecture structural of X is
  signal s1: bit;  --  описание сигнала
  component a    --     описание компонента a
    port (p: in bit; q: out bit);
  end component;
  component b    --     описание компонента b
    port (r: in bit; s: out bit);
  end component;
begin
  a1: a port map (l,s1);  --  назначение компонента
  bl: в port map (s1,m):  --  назначение компонента
end structural;
```

Таким образом, соединение компонентов в систему моделируется при помощи сигналов, связывающих компоненты, которые вначале описываются, а затем назначаются. В простом примере, показанном на рис. 2, соединение компонентов системы выполнено с помощью единственного сигнала s1. Здесь используется позиционное описание, т.е. внешние по отношению к компоненту сигналы указываются в том порядке, в котором соответствующие входные и выходные сигналы компонента перечисляются в его описании.

То же самое можно сделать также при помощи ключевого соответствия. Для этого в языке VHDL для операторов назначения предусматривается такой вариант:

a1: a **port map** (q => s1, p => l);
b1: в **port map** (s => m, r => s1);

Здесь позиции не играют роли, соответствие обеспечивается сопоставлением имен.

2.12. ЛИТЕРАТУРА

1. Армстронг Дж. Р. Моделирование цифровых систем на языке VHDL. / Пер. с англ. – М.: МИР, 1992. – 175 с.
2. Бибило П. Н. Основы языка VHDL. Минск, 1999. – 201 с.
3. Е. Суворова, Ю. Шейнин. Проектирование цифровых систем на VHDL. Санкт-Петербург.: БХВ-Петербург, 2003. – 560 с.
4. Сергиенко А. М. VHDL для проектирование вычислительных устройств. Киев, 2003. – 203 с.