МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ТАГАНРОГСКИЙ ГОСУДАРСТВЕННЫЙ РАДИОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

В.Г. Ивченко

ПРИМЕНЕНИЕ ЯЗЫКА VHDL ПРИ ПРОЕКТИРОВАНИИ СПЕЦИАЛИЗИРОВАННЫХ СБИС

УЧЕБНОЕ ПОСОБИЕ

Таганрог 1999

УДК 621.382.82(07)

Ивченко В.Г. Применение языка VHDL при проектировании специализированных СБИС: Учебное пособие. Таганрог: Изд-во ТРТУ, 1999. 80 с.

Рассматриваются понятия высокоуровневого языка описания аппаратуры VHDL, многоуровневое представление и моделирование специализированных сверхбольших интегральных схем (СБИС) на языке VHDL, применение различных стилей VHDL-описаний в проектировании СБИС. Изложенный материал иллюстрируется на примерах конкретных проектов.

Ил. 24. Библиогр.: 3 назв.

Печатается по решению ред.-изд. совета Таганрогского государственного радиотехнического университета

Р е ц е н з е н т ы :

Кафедра микроэлектроники Московского инженерно-физического института (Технического университета); И.И. Шагурин, д-р техн. наук, профессор, зам. заведующего кафедрой.

В.В.Беспятов, канд. техн. наук, заведующий отделом Научно-конструкторского бюро вычислительных систем, г.Таганрог.

ISBN 5-8327-0037-6

ВВЕДЕНИЕ

Быстрый рост степени интеграции и функциональной сложности современных электронных устройств приводит к необходимости совершенствования и развития методов проектирования больших и сверхбольших интегральных схем (БИС и СБИС). Эта задача наиболее актуальна для специализированных интегральных схем. Это объясняется тем, что они, как правило, характеризуются нерегулярной архитектурой, обуславливающей сложность проектов, в то время как рынок электронных изделий требует разработки в короткие сроки чрезвычайно широкого ассортимента подобных устройств. Наряду с требованиями ко времени разработки необходимо обеспечить бездефектность и высокое качество проектов.

Поскольку современные СБИС содержат миллионы полупроводниковых структур на кристалле, широко использовавшийся ранее метод поэлементного проектирования архитектуры вычислительных систем (ВС) по восходящей методологии не в состоянии обеспечить бездефектное проектирование сложных ВС в приемлемые сроки. Только на этапе графического или текстового ввода схемы потребуются недопустимо большие затраты времени. Кроме того, данная методология не обеспечивает эффективную адаптацию архитектуры устройств на конкретную задачу, а также возможность описания программных средств и учет этого описания при проектировании. Ошибки, выявленные на верхних уровнях представления, приводят к необходимости повторного выполнения этапов маршрута проектирования, начиная с уровня, на котором были допущены просчеты.

Более новым является подход с использованием методологии нисходящего проектирования. Он свободен от перечисленных недостатков. Проектирование сверху-вниз основывается на многоуровневом иерархическом представлении устройств, и разработка ведется в соответствии с системной иерархией от общего описания системы к детальному описанию составляющих ее компонент.

На первом этапе разработчик описывает ВС при помощи поведенческих или функциональных моделей, отображающих функцию и временные характеристики системы. Затем, на основании исходного описания, средствами САПР компилируется структурное представление проекта. Структурная схема является исходными данными для подсистем САПР, генерирующих топологическое описание проектируемого устройства.

Как правило, на каждом этапе рассматриваемого маршрута проектирования с помощью моделирования выполняется верификация описания или проектных решений на соответствие заданной функции ВС. Выявленные недостатки устраняются либо с помощью оптимизации, выполняемой на более высоких уровнях представления, либо исправлением ошибок, допущенных на предыдущих уровнях. При проведении верификации на самом нижнем уровне выполняется проверка соблюдения топологических норм, определяемых конкретной технологией.

Таким образом, нисходящее проектирование позволяет проектировщику сосредоточить внимание на поведенческом или функциональном представлении системы, не отвлекаясь на структуру. Объем работ, затрачиваемых при исправлении возможных ошибок, значительно меньше, чем при проектировании снизу-вверх, так как нет необходимости выполнять повторно все проектные операции, начиная от низшего уровня представления вплоть до системного.

Комплекс средств проектирования должен обладать развитым лингвистическим обеспечением (ЛО), характеризующимся достаточно гибкими описательными способностями для реализации преимуществ нисходящего проектирования. Для этого ЛО включает языки описания аппаратуры (ЯОА), позволяющие создавать адекватные модели проектируемых устройств и эффективно выполнять соответствующее имитационное моделирование, обладая при необходимости независимостью от конкретных аппаратных структур.

Языком описания называется набор синтаксических и семантических правил, определяющий формат представления устройств.

Описательным возможностям ЯОА, а также средствам для документирования проектов всегда уделялось особенное внимание. Дело в том, что на различных этапах проектирования используются различные алгоритмы и программы и, соответственно, различные виды исходных данных. Поэтому требуются языки описания, позволяющие представлять исходные данные для проектирования в форме, воспринимаемой существующими пакетами программ. В зависимости от типа программы и набора исходных данных используются самые разнообразные языки описания: язык описания логических связей, язык описания соединений транзисторов, язык описания электрических постоянных и др. Однако каждый из языков описания является входным форматом независимо разработанных программ, и поэтому имеет индивидуальные особенности. Поскольку такие ЯОА специализированы, то они не взаимозаменяемы. В результате, при проектировании интегральных схем, например, возникала необходимость подготовить входные данные с использованием до 10 различных языков описания.

Сложно осуществить и автоматическое преобразование между языками, так как различаются принципы построения моделей описания. Это объясняется тем, что описательные способности первоначального языка в результате преобразования будут ограничены. Поэтому одной из основных задач в развитии ЯОА явилась разработка концепции общего языка описания.

Стандартизация ЯОА позволяет избежать избыточности в описательных выражениях языков и обеспечить их унификацию для поддержки документирования проектов. Этот аспект очень важен, так как сопровождение проекта документацией способствует успешному выполнению разработки. Способность ЯОА к многоуровневому представлению устройств обеспечивает транспортабельность проекта, что позволяет различным проектировочным подразделениям эффективно взаимодействовать.

Разработанные и унифицированные к настоящему времени языки описания аппаратуры (VHDL, ISP, UDL/I, Verilog, ICL и др.) поддерживаются большинством существующих и широко используемых систем автоматизированного проектирования (САПР) ВС (например, Mentor Graphics, Compass, Cadence). Унификация ЯОА позволяет организовывать эффективное взаимодействие между различными САПР, способствуя созданию экономичных проектов. Например, разработка, выполненная с использованием свободно распространяемой САПР БИС Alliance, в формате Verilog передавалась в САПР Cadence. На основании описания ВС в Cadence выполнялась автоматическая генерация тестов и моделирование устройства с целью проверки его функционирования на соответствие заданной функции.

Наиболее широкое применение приобретает язык VHDL (VHSIC Hardware Description Language). Он разрабатывался как язык описания аппаратуры для высокоскоростных интегральных схем. Первоначальное назначение языка заключалось в обеспечении обмена проектами между различными соисполнителями работ по созданию сверхскоростных интегральных схем. Однако позже с учетом предложений и рекомендаций известных специалистов в области ВС язык был усовершенствован и стандартизован Институтом инженеров по электротехнике и радиоэлектронике (IEEE), в результате чего в 1987 году был утвержден стандарт IEEE Standard 1076 VHDL.

Язык VHDL обеспечивает высокоуровневую абстракцию описания аппаратных средств благодаря наличию как множества предопределенных типов данных, так и возможности создавать пользовательские иерархически организованные типы данных на основе базовых, заложенных в языке.

Благодаря этим возможностям, и, так как язык VHDL легко воспринимается как программными средствами, так и человеком, он может использоваться на этапах проектирования, верификации, синтеза и тестирования аппаратуры, для передачи проектных данных, модификации и сопровождения проекта. В настоящее время он используется для работы с ВС любого уровня сложности - микросхема, плата, блок, устройство, ЭВМ, комплекс.

В пособии изложены основы языка VHDL, а также особенности его применения при проектировании специализированных СБИС. Для эффективного использования VHDL проектировщику необходимо уверенно ориентироваться в предложениях этого языка. Поэтому с самого начала этим элементам уделяется большое внимание.

1. ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА VHDL

1.1. Первичная абстракция языка VHDL

VHDL является формальной записью, предназначенной для описания функции и логической организации цифровой системы. Функция системы определяется как преобразование значений на входах в значения на выходах. Причем время в этом преобразовании задается явно. Организация системы задается перечнем связанных компонентов.

Объект проекта (entity) представляет собой описание компонента проекта, имеющего четко заданные входы и выходы и выполняющей четко определенную функцию. Объект проекта может представлять всю проектируемую систему, некоторую подсистему, устройство, узел, стойку, плату, кристалл, макроячейку, логический элемент и т.п.

В описании объекта проекта можно использовать компоненты, которые, в свою очередь, могут быть описаны как самостоятельные объекты проекта более низкого уровня. Таким образом, каждый компонент объекта проекта может быть связан с объектом проекта более низкого уровня. В результате такой декомпозиции объекта проекта пользователь строит иерархию объектов проекта, представляющих весь проект в целом и состоящую из нескольких уровней абстракций. Такая совокупность объектов проекта называется иерархией проекта (design_hierarchy).

Каждый объект проекта состоит, как минимум, из двух различных типов описаний: описания интерфейса и одного или более архитектурных тел.

Интерфейс описывается в объявлении объекта проекта (entity_declaration) и определяет только входы и выходы объекта проекта.

Для описания поведения объекта или его структуры служит архитектурное тело (architecture_body).

Чтобы задать, какие объекты проекта использованы для создания полного проекта, используется объявление конфигурации (configuration_declaration).

В языке VHDL предусмотрен механизм пакетов для часто используемых описаний, констант, типов, сигналов. Эти описания помещаются в объявлении пакета (package_declaration).

Если пользователь использует нестандартные операции или функции, их интерфейсы описываются в объявлении пакета, а тела содержатся в теле пакета (package_body).

Таким образом, при описании цифровой системы на языке VHDL, пользователь может использовать пять различных типов описаний: объявление объекта проекта, архитектурное тело, объявление конфигурации, объявление пакета и тело пакета. Каждое из описаний является самостоятельной конструкцией языка VHDL, может быть независимо проанализировано анализатором и поэтому получило название "Модуль проекта" (design_unit). Модули проекта, в свою очередь, можно разбить на две категории: первичные и вторичные. К первичным модулям относятся различного типа объявления. Ко вторичным - отдельно анализируемые тела первичных модулей. Один или несколько модулей проекта могут быть помещены в один файл, называемый файлом проекта (design_file).

Каждый проанализированный модуль проекта помещается в библиотеку проекта (design_library) и становится библиотечным модулем (library_unit). Данная реализация позволяет создать любое число библиотек проекта. Каждая библиотека проекта в языке VHDL имеет логическое имя (идентификатор). Фактическое имя файла, содержащего эту библиотеку, может совпадать или не совпадать с логическим именем библиотеки проекта. Для ассоциации логического имени библиотеки с соответствующим ей фактическим именем предусмотрен специальный механизм установки внешних ссылок.

По отношению к сеансу работы VHDL существует два класса библиотек проекта: рабочие библиотеки и библиотеки ресурсов.

Рабочая библиотека - это библиотека, с которой в данном сеансе работает пользователь и в которую помещается библиотечный модуль, полученный в результате анализа модуля проекта.

Библиотека ресурсов - это библиотека, содержащая библиотечные модули, ссылка на которые имеется в анализируемом модуле проекта.

В каждый конкретный момент пользователь работает с одной рабочей библиотекой и произвольным числом библиотек ресурсов.

Возможность создания и использования многих библиотек ресурсов позволяет пользователю классифицировать библиотечные модули по различным признакам. Например, в одной библиотеке хранить описания микросхем одной серии, в другой - описания микросхем другой серии и т.д. Или в одной библиотеке хранить описания микросхем с одним типом задержки, в другой - описания микросхем с другим типом задержки и т.д.

1.2. Лексические элементы

Любой машинный язык характеризуется определенным множеством разрешенных *лексических элементов*. Текст описания на языке VHDL состоит из одного или более файлов проекта. Файл проекта представляет собой последовательность лексических элементов, каждый из которых составлен из символов строго определенного набора символов. Текст каждого модуля проекта является последовательностью отдельных лексических элементов. Каждый лексический элемент - это либо ограничитель, либо идентификатор (который может быть служебным словом), либо абстрактный литерал, либо символьный литерал, либо строковый литерал, либо битово-строковый литерал, либо комментарий.

1.2.1. Набор символов

В тексте VHDL-описания разрешено использовать только графические символы и символы управления форматом ASCII (Американский стандартный код обмена информацией). Базовый набор символов является достаточным для составления любого описания. Символы, входящие в каждую категорию базовых графических символов (basic graphic character), определены следующим образом:

1) заглавные буквы (upper case letters)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z ;

2) цифры (digits)

0 1 2 3 4 5 6 7 8 9 ;

3) специальные символы (special characters)

символ	имя	символ	имя
"	кавычки	.	точка, период
#	диез	/	слэш, разделитель
&	амперсант	:	двоеточие
'	апостроф	;	точка с запятой
(	левая круглая скобка	<	меньше
)	правая круглая скобка	=	равно
*	звезда, умножить	>	больше
+	плюс	_	подчеркивание
,	запятая	|	вертикальная черта
-	дефис, минус		

4) символ пробела (space character).

К символам управления форматом (format effector) относятся ISO (Международная Организация по Стандартизации) и ASCII символы, называемые: горизонтальная табуляция, вертикальная табуляция, возврат каретки, подача строки, подача страницы.

В остальные категории графических символов входят следующие символы:

5) строчные буквы (lower case letters)

a b c d e f g h i j k l m n o p q r s t u v w x y z ;

6) прочие специальные символы (other special characters)

символ	имя	символ	имя
!	восклицательный знак	]	правая квадратная скобка
$	доллар	^	орфографический знак
%	процент	`	тупое ударение
?	вопросительный знак	{	левая фигурная скобка
@	коммерческое at	}	правая фигурная скобка
\	обратный слэш	~	тильда
[	левая квадратная скобка		

1.2.2. Разделители и ограничители

Разделителем может быть либо символ пробела, либо символ управления форматом, либо конец строки. Символ пробела не является разделителем внутри комментария, строкового литерала или символьного литерала, включающего этот символ.

Между любыми двумя соседними лексическими элементами разрешено ставить один или более разделителей, а также перед первым лексическим элементом или после последнего лексического элемента в каждом модуле проекта. По крайней мере, один разделитель необходим между идентификатором или абстрактным литералом и соседним идентификатором, или абстрактным литералом.

Ограничителем может быть один из следующих символов (входящих в базовый набор):

& ' ( ) * + , - . / : < = > |

Ограничителем может быть также один из составных ограничителей, составленный из стоящих рядом специальных символов:

ограничитель имя

```
 =>        стрелка
 **        двойная звезда, возведение в степень
 :=        присваивание переменной
 /=        не равно
 >=        больше или равно
 <=        меньше или равно, а также назначение сигнала
 <>        блок
```

Каждый лексический элемент должен помещаться на одной строке, т.к. конец строки рассматривается как разделитель. Кавычки, диез и подчеркивание, а также два рядом стоящих дефиса не являются ограничителями, но могут являться частью других лексических элементов.

1.2.3. Идентификаторы

Идентификаторы используются в качестве имен, а также в качестве зарезервированных слов. Идентификаторы могут содержать латинские буквы в верхнем и нижнем регистре, цифры и символ подчеркивания. Заглавные и строчные буквы в идентификаторах считаются равнозначными.

Так как пробел является разделителем, то использование его недопустимо внутри идентификатора.

Пример:

```
COUNT   X   C_OUT      FFT   Decoder
VHSIC   X1  PageCount  STORE_NEXT_ITEM
```

1.2.4. Комментарии

Комментарий начинается с двух рядом стоящих символов дефиса и ограничен концом строки. Комментарий может появиться в любой строке VHDL-описания. Наличие или отсутствие комментариев не оказывает влияние на правильность описания. Более того, комментарии не оказывают влияния на выполнение модуля моделирования; единственным назначением комментариев является повышение читабельности описания.

Горизонтальная табуляция может быть использована в комментарии после двойного символа дефиса и это эквивалентно одному или более пробелов.

Пример:

```
-- Комментарий
end;  -- processig of line is complete
-- длинные комментарии могут быть разделены
-- на несколько строк
---------- первые два дефиса начинают комментарий
```

1.2.5. Литералы

Литералы делятся на абстрактные, символьные, строковые, битово-строковые.

1.2.5.1. Абстрактные литералы

Имеются два класса абстрактных литералов: действительные литералы и целые литералы. Действительным литералом является абстрактный литерал, содержащий точку; целым литералом является абстрактный литерал без точки. Согласно другой классификации абстрактные литералы бывают десятичными и базированными.

Десятичные литералы

Десятичным литералом (decimal literal) является абстрактный литерал, выраженный в десятичной системе счисления (т.е. база счисления - это точно 10).

Символ подчеркивания, стоящий между двумя соседними цифрами десятичного литерала, не оказывает влияния на значение этого абстрактного литерала. Буква E в выражении экспоненты (если оно используется) может быть написана в любой форме (заглавной или строчной) - значение выражения от этого не меняется.

Выражение экспоненты в целом литерале не должно содержать знак минус. В абстрактных литералах допускаются незначащие нули. Так как пробел является разделителем, то он не разрешен в абстрактных литералах, даже между составными частями экспоненты. Значение 0 экспоненты разрешено только в целых литералах.

Пример:

```
11          0       1E6       123_456    -- целые литералы
11.0        0.331   3.141_592   -- действительные литералы
2.64E-12    1.0E+6  6.023E+24   -- действительные литералы
                                 -- с экспонентой
```

Базированные литералы

Базированный литерал (based literal) - это абстрактный литерал, выраженный в форме, которая явно содержит базу счисления. База счисления может находиться в пределах от 2 до 16.

Символ подчеркивания, стоящий между двумя соседними цифрами в базированном литерале, не влияет на значение этого абстрактного литерала. База счисления (base) и экспонента должны быть выражены десятичным числом. Буквой, используемой в качестве расширенной цифры (extended digit), могут быть буквы от A до F, соответственно представляющие цифры от 10 до 15. Буква в базированном литерале (будь то буква E в экспоненте или расширенная цифра) может писаться как в заглавной, так и в строчной форме - значение литерала от этого не меняется.

Предполагается обычный смысл базированного написания; в частности, значение каждой расширенной цифры не должно превышать значения базы. Экспонента обозначает степень базы, на которую умножается значение базированного литерала, взятого без выражения экспоненты, для получения значения этого литерала с экспонентой. Выражение экспоненты в целом литерале не должно содержать знак минус.

Пример:

```
-- целые литералы значения 255
2#1111_1111#  16#FF#       016#0FF#

-- действительные литералы значения 4095.0
16#F.FF#E+2                2#1.1111_1111_111#E11
|    |    |__ЭКСПОНЕНТА
|    |__ЧИСЛО
|__БАЗА
```

1.2.5.2. Символьные литералы

Символьный литерал формируется включением одного из 95 графических символов (включая символ пробела) между двумя символами апострофа. Символьный литерал содержит значение символьного типа.

Пример:

```
'A'   '*'   '''   ' '
```

1.2.5.3. Строковые литералы

Строковые литералы формируются из последовательности графических символов (возможно и пустой), заключенной между двумя символами кавычки, используемых как строковые скобки.

Строковый литерал имеет значение, представляющее собой последовательность символьных значений, соответствующих графическим символам этого строкового литерала, исключая сами кавычки. Если необходимо включить символ кавычки в состав строкового литерала, то этот символ надо повторить подряд два раза на соответствующем месте. Длина строкового литерала равна количеству символьных значений в представленной последовательности. (Каждый удвоенный символ кавычки учитывается как один символ). Строковый литерал должен писаться на одной строчке, т.к. он является лексическим элементом.

Пример:

```
"Setup time is too short"  -- сообщение об ошибке
""                         -- пустой строковый литерал
" "  "A"  """"            -- три строковых литерала длиной 1.
```

1.2.5.4. Битово-строковые литералы

Битово-строковые литералы (bit-string literal) формируются из последовательности расширенных цифр, заключаемой между двумя символами кавычки, используемых как скобки строки битов, и которой предшествует спецификатор базы (base specifier).

Спецификатор базы может принимать значения B, O, X. Если спецификатором базы счисления является B, то в качестве расширенных цифр могут выступать только цифры 0 и 1; если O, то цифры от 0 до 7; и если X, то цифры от 0 до 9 и от A до F.

Символ подчеркивания, стоящий между двумя соседними цифрами в битово-строковом литерале, не влияет на значение этого литерала. Буквой, используемой в качестве расширенной цифры, могут быть буквы от A до F, представляющие, соответственно, цифры от 10 до 15. Буква в битово-строковом литерале (расширенная цифра или спецификатор базы) может писаться как в заглавной, так и в строчной форме - значение литерала от этого не меняется.

Битово-строковый литерал имеет значение, представляющее собой последовательность значений предопределенного типа BIT (т.е. последовательность из '0' и '1'). Если указатель базы счисления есть B, то значение битово-строкового литерала есть сам литерал. Если указатель базы есть O (или X), то значением литерала является последовательность, полученная замещением каждой расширенной цифры последовательностью из трех (или соответственно четырех) значений предопределенного типа BIT.

Длина битово-строкового литерала равна числу значений типа BIT в представленной последовательности.

Пример:

```
X"FFF"    -- эквивалентно B"1111_1111_1111"
0"777"    -- эквивалентно B"111_111_111"
X"777"    -- эквивалентно B"0111_0111_0111"
```

1.2.6. Зарезервированные слова

Идентификаторы, перечисленные ниже, называются зарезервированными словами, они зарезервированы в языке для специального назначения.

ABS	**ELSE**	**MOD**	**RETURN**
ACCESS	**ELSEIF**	**NAND**	**SELECT**
AFTER	**END**	**NEW**	**SEVERITY**
ALIAS	**ENTITY**	**NEXT**	**SIGNAL**
ALL	**EXIT**	**NOR**	**SUBTYPE**
AND	**FILE**	**NOT**	**THEN**
ARCHITECTURE	**FOR**	**NULL**	**TO**
ARRAY	**FUNCTION**	**OF**	**TRANSPORT**
ASSERT	**GENERATE**	**ON**	**TYPE**
ATTRIBUTE	**GENERIC**	**OPEN**	**UNITS**
BEGIN	**GUARDED**	**OR**	**UNTIL**
BLOCK	**IF**	**OTHERS**	**USE**
BODY	**IMPORT**	**OUT**	**VARIABLE**
BUFFER	**IN**	**PACKAGE**	**WAIT**
BUS	**INITIALIZE**	**PORT**	**WHEN**
CASE	**INOUT**	**PROCEDURE**	**WHILE**
COMPONENT	**IS**	**PROCESS**	**WITH**
CONFIGURATION	**LABEL**	**RANGE**	**XOR**
CONSTANT	**LIBRARY**	**RECORD**	
DISCONNECT	**LINKAGE**	**REGISTER**	
DOWNTO	**LOOP**	**REM**	
	MAP	**REPORT**	

Зарезервированное слово не должно использоваться как объявленный идентификатор. Зарезервированные слова, различающиеся только в использовании соответствующих заглавных и строчных букв, рассматриваются как идентичные.

1.2.7. Допустимые замены символов

Для базовых символов "вертикальная черта", "диез", "кавычка" разрешены следующие замены:

1) вертикальная черта (|) может быть заменена восклицательным знаком (!) при использовании в качестве ограничителя;

2) диез (#) в базированном литерале может быть заменен двоеточием (:), при этом заменить надо оба знака в этом литерале;

3) кавычки ("), используемые как ограничители в строковом литерале с обеих сторон, могут быть заменены на проценты (%). При этом необходимо заменить обе строковые скобки, а из последовательности символов исключить все символы кавычки. Каждый символ процента внутри последовательности символов должен быть удвоен. Удвоенный символ процента интерпретируется в этом случае как один символ.

Такие замены не изменяют смысл описания.

Правила использования идентификаторов и абстрактных литералов таковы, что строчные и заглавные буквы могут быть использованы без различия. Эти лексические элементы, таким образом, могут быть написаны с использованием только базового набора символов.

1.3. Модели данных в VHDL

VHDL создан для обеспечения описания аппаратуры на высоком уровне абстракции. При этом такие понятия как "целое" (integer), "адрес", "регистр" или "вектор" могут быть более предпочтительными, чем "bit" для описания информации, проходящей через пути данных. В общем случае типы данных, требующиеся при построении модели, не могут быть предсказаны раньше времени. Таким образом, язык проектирования должен обеспечить разработчика средствами для конструирования произвольных сложных типов. Система типов в VHDL обеспечивает такую возможность.

Модели данных позволяют создавать различные типы и объекты данных на основе базовых предопределенных типов. Любой объект данных характеризуется определенным классом и типом.

Объекты данных (data_object) являются хранилищами для значений определенного типа. Все типы в VHDL конструируются из элементов, представляющих собой скалярные типы.

Существуют объекты данных четырех классов: константы, переменные, сигналы и файлы.

Константы и *переменные* содержат одно значение данного типа. Значения переменных могут быть изменены назначением нового значения в предложении назначения переменной. Значение константы устанавливается до начала моделирования и не может после этого изменяться.

Сигнал имеет текущее значение подобно переменной. Кроме этого, он имеет историю прошедших значений, на которые можно ссылаться, а также множество будущих значений, которые будут получены от формирователей сигналов. Новые значения для сигналов создаются предложениями назначения сигналов.

Файлы также являются хранилищами значений и формально определяются как объекты.

Каждый объект в описании должен ассоциироваться только с одним типом.

Тип объекта управления определяется множеством *возможных значений* и множеством *разрешенных операций*. Тип создается выражением его объявления.

Имеются операции двух видов. Операции первого вида являются предопределенными (например, операторы '+', '-' для значений целого типа integer и real). Другие операции явно кодируются в VHDL с помощью функций и подпрограмм. Их можно отнести к пользовательским операциям.

Значение объекта данных определяется выражением в правой части предложения назначения. В состав выражения могут входить константы, переменные, сигналы, операторы и указатели функций. Когда имя объекта используется в выражении, то при расчете значения выражения учитывается его текущее значение. Значение выражения имеет только один тип.

1.3.1. Скалярные типы

Скалярные (scalar) типы - это элементы, из которых конструируются все типы в VHDL. Базовое множество скалярных типов является предопределенным. По мере необходимости разработчик может создать дополнительные скалярные типы. В VHDL имеется четыре вида скалярных типов: *целый* тип, тип *с плавающей точкой*, *перечислительный* тип (enumeration) и *физический* тип. Разработчик имеет возможность задавать подтипы скалярных типов.

1.3.1.1. Целый тип

Объекты целого типа используются для представления абстрактных числовых значений. Тип integer является предопределенным. Он охватывает все целые числа, ограниченные разрядностью слова компьютера.

Добавочные целые типы могут быть объявлены явно заданием диапазона значений, допустимых для объектов данного типа. Рассмотрим несколько примеров объявлений целых типов.

```
type Apples is range 0 to 75;
type Oranges is range 0 to 75;
type Word_index is range 31 downto 0;
```

Диапазоны задаются либо убывающей, либо возрастающей последовательностью значений. Границы диапазона могут быть произвольными выражениями. Рассмотрим, как объявляются объекты, использующие эти типы.

```
variable Macintosh: Apples;
variable Seville,Valencia,av_oranges:Oranges := 10;
signal   control_selector:Word_Index;
```

Константе должно быть присвоено значение в момент объявления. Аналогично начальное значение может быть присвоено переменной.

Целые литералы (такие как -31523, 1, 212) могут быть использованы для представления значений любого целого типа. Для формирования выражения может быть использована комбинация имен объектов, литералов и операторов, как это сделано в следующих предложениях назначения:

```
av_oranges<=(Seville+Valencia)/2;
beta_level:=1-Alpha_level;
```

Все обычные арифметические операторы и операторы отношения являются предопределенными для целого типа. Тем не менее, оба аргумента оператора должны быть одного типа.

```
if Seville > Macintosh ... -- НЕВЕРНО
```

Нельзя сравнивать объекты типов Apples и Oranges. Предопределенный оператор ">" не работает с операндами различных типов, даже если они имеют одинаковый диапазон.

Тем не менее, все целые типы и типы с плавающей точкой являются *тесно связанными* типами (closely related types) и VHDL обеспечивает для этих типов преобразование (conversion) между любыми парами. Допускается любое из следующих сравнений:

```
if Apples(Seville)>Macintosh...
if Seville>Oranges(Macintosh)...
```

Значение выражения одного типа преобразуется к значению тесно связанного типа указанием перед выражением, заключенным в скобках, имени типа, к которому преобразуется выражение. При преобразовании между типом с плавающей точкой и целым типом будет выполняться округление до ближайшего целого.

1.3.1.2. Тип с плавающей точкой

Объекты типа с плавающей точкой используются для представления абстрактных числовых значений. Тип real является предопределенным. Он включает вещественные числа.

Добавочные типы с плавающей точкой могут быть объявлены явно заданием диапазона значений, допустимых для объектов данного типа. Рассмотрим пример объявлений типа с плавающей точкой.

```
type Probability is range 0.0 to 1.0;
```

Диапазоны задаются либо убывающей, либо возрастающей последовательностью значений. Границы диапазона могут быть произвольными выражениями. Рассмотрим, как объявляются объекты, использующие эти типы.

```
constant alpha_level:   Probability:=0.75;
variable  beta_level:   Probability;
```

Константе должно быть присвоено значение в момент объявления. Аналогично начальное значение может быть присвоено переменной.

Литералы с плавающей точкой представляют значения любого типа с плавающей точкой и всегда содержат десятичную точку или отрицательную экспоненту: например, 3.14159, -23.0, 1E-2. Запись с экспонентой может быть использована для любого вида числовых литералов: 9E – целое, а 0.324E-3 – с плавающей точкой.

Для формирования выражения может быть использована комбинация имен объектов, литералов и операторов.

Все обычные арифметические операторы и операторы отношения являются предопределенными для типа с плавающей точкой. Тем не менее, оба аргумента оператора должны быть одного типа. Предопределенный оператор ">" не работает с операндами различных типов, даже если они имеют одинаковый диапазон.

Все целые типы и типы с плавающей точкой являются тесно связанными типами (closely related types) и VHDL обеспечивает для этих типов преобразование (conversion) между любыми парами.

1.3.1.3. Перечислительные типы

Тип состоит из множества возможных значений, которые могут принимать объекты этого типа, вместе с множеством операций над типом. В объявлении перечислительного типа явно перечисляются идентификаторы и графические символы, которые означают значения типа. Идентификаторы и символы являются литералами для типа точно так же, как 3 и 245 являются литералами целого типа. Значения упорядочены и отношение упорядочения определяется последовательностью их появления в списке. Рассмотрим несколько примеров:

```
type severity is(OKAY,NOTE,WARNING,ERROR,FAILURE);
type color is(red,orange,yellow,green,blue,indigo,violet);
type bit6 is('U','0','1','F','R','X');
type fuzzy_logic is('0', may be,'1');
```

Нет необходимости писать объявления для следующих перечислительных типов, поскольку они являются предопределенными:

```
type character is (NUL,...,'A','B','C',...DEL);
type boolean is (False,True);
type bit is ('0','1');
```

Тип character включает символы для всех печатаемых и непечатаемых элементов кода ASCII, а также для их графических представлений.

Логические операторы and, or, nand, nor и xor определяются для операндов типа bit или типа boolean и дают результат того же самого типа, что и операнды (но нельзя задать один операнд типа boolean, а другой типа bit). Операторы отношения "=", ">", ">=" и другие дают результат типа boolean независимо от того, какие типы операндов. Рассмотрим несколько примеров:

```
signal chip_select,data_rdy,inhibit: bit;
variable level:severity;
variable test_result:probability;
signal proceed:boolean;
          .
          .
          .
chip_select<=data_rdy and not inhibit; --Логические
        -- операции над битами при назначении сигналов
if level <= red then ... -- Выражение с отношением,
                         -- использующее
                         -- перечислительный тип
proceed <= test_result > Alpha_level; -- Результат
                            -- операции отношения над
                            -- значениями с плавающей
                            -- точкой булевого типа
```

Не нужно путать два значения символа "<=". Этот символ используется как при назначении сигналу значения некоторого выражения, так и представляет отношение "меньше или равно".

1.3.1.4. Подтипы скалярных типов

Если желательно, чтобы скалярный объект принимал значения некоторого типа из ограниченного диапазона, то это может быть отражено в тексте проекта при помощи объявления и использования подтипа. Предположим, к примеру, что разработчик желает создать сигнал A типа severity и что A может принимать только значения OKAY, NOTE и WARNING.

```
type severity is (OKAY,NOTE,WARNING,ERROR,FAILURE);
subtype go_status is severity range OKAY to WARNING;
signal A: go_status;
```

Объявление подтипа определяет базовый тип (base type) и ограничение диапазона (range constraint). Любое значение, назначенное A, должно быть типа severity, который является базовым типом для A. Программа моделирования будет проверять, попадает ли значение в диапазон от OKAY до WARNING, в момент выполнения назначения. Если это не выполняется, то моделирование будет остановлено и будет выдано сообщение, описывающее это нарушение.

Базовый тип и ограничение диапазона могут быть включены прямо в объявление объекта, если имеется немного объектов, которые должны быть объявлены с некоторым подтипом. Рассмотрим объявление, эквивалентное объявлениям подтипа и сигнала, приведенным выше:

```
signal A: severity range OKAY to WARNING;
```

Выбор из двух методов для определения подтипов объектов зависит от удобства.

Так как операторы определяются для типов, а не для подтипов, то объекты с общим базовым типом могут свободно использоваться в одном выражении.

```
type Counter is range 0 to 100;

subtype low_range is Counter range 0 to 50;
subtype mid_range is Counter range 25 to 75;
subtype hi_range is Counter range 50 to 100;

variable low_count: low_range;
variable mid_count: mid_range;
variable hi_count:  hi_range;
          .
          .
          .
mid_count:=(hi_count + low_count)/2;
```

Значения hi_count и low_count оба имеют тип Counter. Сначала вычисляется значение выражения. Затем до выполнения назначения это значение проверяется с использованием ограничений диапазона для mid_count.

1.3.1.5. Физические типы

Физические типы позволяют разработчику непосредственно выразить величины в физических единицах измерения. В VHDL используется один физический тип - предопределенный физический тип TIME (время). Объявление физического типа задает множество единиц, определенных в терминах некоторой базовой единицы. В случае типа TIME базовой единицей является fs (фемтосекунда), а производными единицами являются ps, ns, us и так далее. Рассмотрим определение типа TIME.

```
type TIME is range -(2**31-1) to 2**31-1
     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;
```

Диапазон типа TIME определяет диапазон базовых единиц, который может быть точно представлен объектами типа. Физические литералы, использующие любые из определенных имен для физических единиц, будут автоматически преобразовываться к фемтосекундам.

Рассмотрим два примера физических типов. Они не являются предопределенными типами, но эти и другие физические типы могут быть получены из библиотечного пакета, поставляемого продавцами матобеспечения САПР.

```
type resistance is range 0 to 2**31-1
     units
           nOhm;
```

```
            uOhm   =1000 nOhm;
            mOhm   =1000 uOhm;
            Ohm    =1000 mOhm;
            kOhm   =1000 Ohm;
            megOhm =1000 kOhm;
      end units;

type voltage is range -(2**31-1) to 2**31-1
      units
            nV;
            uV     =1000 nV;
            mV     =1000 uV;
            V      =1000 mV;
            kV     =1000 V;
            megaV  =1000 kV;
      end units;
```

Когда значение физического типа делится на другое значение того же самого типа, то единицы измерения исчезают и результат становится совместимым с любым целым типом.

Допускается умножение физического типа на число с плавающей точкой, в этом случае получается результат физического типа. Эти идеи иллюстрируются в следующих примерах.

```
total_time := 1 ns + .039 s - min_time;
output_volts<=supply_volts-500mV after reset_duration+5 ms;
nom := .75* max;

function "*" (I:current; R:resistance) return voltage is
begin
     return nV* ( real (I/nA)* real(R/nOhm)*1.0E-9);
end;

function "*" (R:resistance; I:current) return voltage is
begin
     return nV * ( real ( I/ nA )* real(R/nOhm)* 1.0E-9);
end;
```

Каждое объявление функции перезагружает оператор умножения таким образом, что он будет выполняться, когда перемножаются значения физических типов current и resistance. Результат должен быть физического типа voltage.

1.3.1.6. Предопределенные атрибуты скалярных типов

На некоторые характеристики скалярных типов и подтипов могут быть ссылки в выражениях в удобной и компактной форме, называемой *записью атрибута* (attribute notation). Если T является именем скалярного типа, то T'high является наибольшим возможным значением типа, a T'low - наименьшим возможным значением. Например, color'high = violet, a color'low = red . T'right - самое правое значение типа, a T'left - самое левое значение типа. T'left отличается от T'low, a T'high от T'right, только если направление для типа - downto.

Рассмотрим значения этих четырех атрибутов для типа Word_index, введенного ранее.

```
type Word_index is range 31 downto 0;
     Word_index'left   = 31
     Word_index'right  = 0
     Word_index'low    = 0
     Word_index'high   = 31
```

Следующие два предопределенных подтипа целого типа используют атрибут 'high в их выражениях для диапазонов:

```
subtype natural is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Значение атрибута может быть использовано в объявлении подтипа, в задании параметров цикла, в назначении:

```
subtype longwave is color range color'left to yellow;
for J in Word_Index'left downto 9 loop...
status<=severity'high when emergency else severity'low;
```

1.3.1.7. Предопределенные функционально-значные атрибуты

Целый, перечислительный и физический типы имеют *позиционный номер*, связанный с каждым значением типа.

Позиционный номер первого значения в перечислительном типе - нуль; каждый следующий знак имеет позиционный номер на единицу больше, чем предшествующий. Например, позиционный номер значения true в

типе boolean равен 1; позиционный номер '1' в типе character равен 49 (в соответствии с его ASCII кодом); позиционный номер элемента '1' в типе bit равен 1.

Позиционные номера определяют упорядоченность значений типа. Выражение отношения 'A'<'Z' (со значением true) имеет смысл, потому что 65 (позиционный номер элемента 'A' в типе character меньше, чем позиционный номер 'Z', равный 90). Имеет также смысл говорить о *последователях* и *предшественниках* элемента; '1' следует за '0' в типе bit, а yellow предшествует green в типе color.

Функционально-значные атрибуты 'pos и 'val используются для преобразования значения типа в соответствующий позиционный номер и наоборот. Ниже приведены выражения, результат которых булевого типа и равен true.

```
...severity'pos(ERROR)=3...
...color'val(3)=green...
...color'val(severity'pos(ERROR))=green...
```

Функции 'pred и 'succ возвращают элементы, чьи позиционные номера на единицу больше или на единицу меньше, чем у аргумента.

```
...color'succ(orange) = yellow ...
...color'pred(indigo) = blue ...
...bit'succ('1')... --
      -- ОШИБКА: 'succ не определен для bit'high
```

Рассмотрим подтипы типа, использующие общие позиционные номера, а также функции 'succ и 'pred с общим родителем:

```
subtype reverse_caps is character range 'Z' downto 'A';
variable backward:reverse_caps;
variable forward: character;
             .
             .
             .
...character'pos('H')=72...
...reverse_caps'pos('H')=72...
             .
             .
             .
forward:=backward;
...character'succ(forward)=reverse_caps'succ(backward) ...
```

Кроме того, определяются две добавочных функции, учитывающих направления подтипа. Функция 'rightof отличается от 'succ, а 'leftof от 'pred в случае, когда подтип имеет убывающий диапазон:

```
...character'rigtof('B')='C'...
...reverse_caps'rightof('B')='A'...
```

Все эти функции действительны также для целых и физических типов. Значение целого совпадает с его позиционным номером; позиционный номер значения физического типа является целым числом, представляющим соответствующее количество базовых единиц.

```
if Oranges'pos(Seville) > Apples'pos(McIntosh)...
function"*"(I:current;R:resistance) return voltage is
begin
return
nv*(real(current'pos(I)*real(resistance'pos(R))*1.0-0); end;
```

1.3.2. Массивы и записи

Каждому индивидуальному скалярному объекту может быть присвоено собственное имя. Эта запись становится громоздкой, если мы имеем дело с объектами, которые представляются таблицами или на которые чаще всего ссылаются, как на связанную группу объектов. VHDL обеспечивает два вида составных (composite) типов, поддерживающих связывание объектов: массивы и записи.

1.3.2.1. Массивы

Массив представляется набором одного или более идентичных элементов, рассматриваемых как одномерный вектор, двумерная матрица или как произвольная прямоугольная структура более высокой размерности. Объявление массивного типа может быть объявлением *неограниченного* или *ограниченного* массива, или объявлением *массивного подтипа*. Любому скаляру из массива может быть присвоено значение, он может использоваться в выражении или связываться с портом компонента, или быть параметром подпрограммы. Элемент массива может быть выбран путем задания значения индекса вместе с именем массива. Непрерывная часть или секция (slice) одномерного массива может быть выбрана целиком путем использования диапазона вместо индекса. Выражение или возвращаемое значение функции может иметь значение, являющееся массивом.

Объявления массивов

В объявлении типа для *неограниченного массива* (unconstrained) задается *число индексов*, *тип* и *позиция* каждого индекса, а также *тип элементов* массива. В нем не определяется число элементов в каждом измерении массива. Неограниченные массивные типы в следующем списке являются предопределенными:

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

Каждый из этих типов имеет одно измерение. Тип bit_vector индексирован значениями предопределенного типа natural и имеет элементы типа bit. Тип string индексируется предопределенным типом positive и имеет элементы типа character. Запись `range<>` (читается как "*ящик диапазона*" - range box) означает, что определение границ индекса было отложено. Границы подставляются в момент, *когда создается* объект данного типа.

Замечание. Как видно, в отличие от большинства языков программирования (Си, Паскаль, Фортран и др.), в VHDL можно задавать тип индексов массива.

Индексы массива могут быть целого или перечислительного типа. Элементы могут быть любого типа. Например:

```
type matrix is array(integer range <>,integer range<>)of real;
type color_accumulator is array(color range <>) of natural;
type color_match is array(natural range<>) of color;
type bit6_data is array(positive range<>) of bit6;
type bit6_address is array(positive range<>) of bit6;
type transition_delay is array(bit6 range<>,bit6 range<>)of time;
type conversion_vector is array(bit6 range<>) of bit;
```

Объявление объекта типа массив определяет *имя типа* и *ограничения на индекс* (index constraint):

```
variable square: matrix(1 to 10,1 to 10);
signal A_register,B_register: bit6_data(63 downto 0);
signal parts_per_color:color_accumulator(green to indigo);
constrant part_id:string := "M00368";
variable bit_equivalence: conversion_vector(bit6);
```

Каждая индексная позиция в объявлении объекта типа массив, который использует неограниченный тип, должна быть ограничена. Диапазон может быть ограничен:

1) при помощи to или downto (первые три примера);
2) он может также заменяться диапазоном начального значения (как в part_id);
3) диапазон может быть назван именем индекса соответствующего перечислительного типа (как в bit_equivalence).

В последнем случае диапазон является полным диапазоном перечислительного типа.

В случае, когда необходимо большое число объектов некоторого типа с одинаковыми ограничениями на индексы, может быть удобным объявить для этой цели *подтип*. Имя подтипа может быть использовано в качестве сокращения для полного обозначения подтипа:

```
subtype data_store is bit6_data(63 downto 0);
signal A_reg,B_reg: data_store;

subtype transform is matrix (1 to 4, 1 to 4);
variable X,Y:transform;
signal unit: transform;
```

Имеется также другая сокращенная запись, которая часто полезна при создании массивов. Рассмотрим следующую пару объявлений:

```
type transition_delay is array(bit6 range<>,bit6 range<>)of time;
subtype cmos_transition is transition_delay(bit6,bit6);
```

Эти объявления можно записать в следующем виде:

```
type cmos_transition is array(bit6,bit6) of time;
```

Отличие заключается в том, что неограниченный массивный тип transition_delay никогда явно не определяется. Это объявление создает *анонимный тип* (anonymous type) с объявлением, которое похоже на объявление для transition_delay, за которым непосредственно следует объявление подтипа для cmos_transition с данным ограничением на индекс.

Строки, битовые строки и агрегаты

Строки, *битовые строки*, *агрегаты* (strings, bit strings, aggregates) используются для конструирования значений для объектов массивных типов. Они могут использоваться в любом месте, где допускается значение типа массив, например, как начальное значение константы или операнд в выражении.

Строковая запись может быть использована для массивных значений любого одномерного массива, элементы которого имеют тип character; например:

```
signal data_bus:bit6_data(15 downto 0);
                .
                .
                .
data_bus<="UUUUUUUUFFFFFFFF";
```

Замечание. Не следует путать запись символа (например, 'A') с записью строки, представляющей массив длиной 1 (например, "A").

VHDL позволяет компактно описывать *битовые строки* (значение типа bit vector) в базисе 2, 8 и 16.

```
constant clear:bit_vector :=B"00_101_010";
constant empty:bit_vector :=O"052";
constant null:bit_vector  :=X"2A";
```

Все три константы имеют одно и то же значение. Символы подчеркивания могут использоваться в любом месте в битовой строке для облегчения чтения. Расширенными цифрами (extended digits) для шестнадцатеричного представления являются буквы от A до F, причем могут использоваться как большие, так и маленькие буквы.

Тип элемента массива, созданного *агрегатом*, может быть любого предопределенного типа или иметь тип, определенный пользователем. Массивные агрегаты формируются при помощи *позиционной* (positional) записи, *поименованной* (named) записи или комбинации этих двух форм. Рассмотрим несколько примеров.

Позиционная форма записи:

```
constant clear:bit_vector:=('0','0','1','0','1','0','1','0');
constant mos_delay:transition_delay :=
--  'U'    '0'    '1'    'F'     'R'     'X' --второй индекс
--                                          --первый индекс
  ( (0ns,  4ns,   5ns,   3ns,    5ns,     0ns), --'U'
    (5ns,  0ns,   5ns,   3ns,    5ns,     0ns), --'0'
    (5ns,  4ns,   0ns,   3ns,    5ns,     0ns), --'1'
    (5ns,  4ns,   5ns,   0ns,    5ns,     0ns), --'F'
    (5ns,  4ns,   5ns,   3ns,    0ns,     0ns), --'R'
    (5ns,  4ns,   5ns,   3ns,    5ns,     0ns)) --'X'
```

Поименованная форма записи:

```
parts_per_color <= (green => 3,indigo => 10,blue => 5);
square := (1 to 10 => (1 to 10 => 0.0));
```

Комбинированная форма записи:

```
unit <= ( (1 => 1.0, others => 0.0),(2 => 1.0 others =>0.0),
(3  => 1.0, others => 0.0),(4 => 1.0 others =>  0.0));
```

Это новое объявление для clear имеет точно тот же смысл, что и предыдущее. Агрегат записывается, как список значений элементов, разделенных запятыми. Первое значение элемента назначается элементу с самым левым значением индекса и затем эта операция выполняется в порядке слева-направо. Значением элемента может быть произвольное выражение.

Константе mos_delay в процессе инициализации присваивается значение массива времен перехода между состояниями. Агрегат состоит из списка строковых значений, разделенных запятыми. Каждое строковое значение само представляется в форме агрегата. Заметим, что последнее предписание на изменение массива наиболее быстрое.

Сигналу parts_per_color присваивается значение с использованием поименованной, а не позиционной формы записи.

В этом случае агрегат также является списком, элементы которого разделены запятыми, но в то же время каждый элемент списка состоит из выбора значения индекса, правой стрелки и значения, которое должно быть назначено выбранному элементу массива. Значения индекса нет необходимости представлять в порядке, соответствующем типу, так как каждое значение индекса явно поименовано.

Все элементы в square будут иметь значения 0.0 после выполнения назначения. Вложенные агрегаты порождают сложное значение, имеющее значение 0.0 во всех строковых элементах с индексами от1 до 10. Затем это сложное значение назначается каждой строке массива.

Сигналу unit будет назначено значение 1.0 для всех элементов, находящихся на главной диагонали, нуль для всех других элементов. Поименованная запись используется для выбора одной индексной позиции и присвоения значения элементу в каждой строке. Для остальной части строки используется операция выбора others. Во втором измерении массива unit назначение производится при помощи позиционной записи.

Операции над массивами

Массив может появляться в выражении во множестве случаев. Элемент массива выбирается путем задания значений индексов для каждой индексной позиции. Два массива так же, как два значения одного типа, можно *сравнивать*, используя операторы равенства и неравенства. Результат будет булевого типа.

Для одномерных массивов определяются две дополнительных операции: *вырезка* (slicing) и *конкатенация* (знак &). Вырезка позволяет выбрать непрерывное подмножество массива. Конкатенация создает большой массив из двух массивов или из массива и одного значения элементного типа. Все приведенные ниже булевы выражения имеют значения true.

```
constant A:bit_vector := "01010";
constant B:bit_vector := "010";
variable S:string (1 to 5);
          .
          .
          .
...A(0)='0' ... -- Элемент выбирается при помощи индекса
...A(3)='1' ...
...A(1 to 3) = "101"...   -- Вырез выбирается диапазоном
...B(1 to 2) = A(3 to 4) ...
...A = '0'&"101"&'0'... -- Массив создается конкатенацией
...A = B & "10" ...
S:="AbCdE";   -- Переменной S назначается значение
...S(2) = 'b' ...
...S(4) = 'd' ...
...S(3 to 5) = "CdE"...

S(2 to 4) := "XYZ"; -- Вырезу из S назначается новое значение
...S = "AXYZE"...
```

Операторы *упорядочивания* (<,<=,>,>=) являются предопределенными для одномерных массивов с элементами целого или перечислительного типа. Сравнение производится поэлементно слева-направо до тех пор, пока не будет обнаружено различие или массивы не будут просмотрены. Необязательно, чтобы массивы были одинаковой длины; если элементы короткого массива совпадают с начальными элементами длинного массива, то длинный массив будет считаться больше, чем короткий. Приведенные ниже булевы выражения являются истинными.

```
name1 := "Jones";
name2 := "Smith";
...name1 =  name1 ...
...name2 < "Smithson"...

count1 :=(2,3,6);
count2 :=(2,3,7);
...count1 <= count2...
...count2>(1,3,7,9)...
```

Логические операторы, определенные для скаляров типа bit и boolean, могут также использоваться для одномерных массивов с элементами этих типов (и тесно связанных с ними). Операция выполняется последовательно над каждым элементом и результатом является массив с элементами, имеющими тот же тип, что и аргументы. Оба массива должны иметь одну и ту же длину и тип.

```
signal Areg,Breg:bit_vector(32 downto 0);
type bool_vector is array (natural range <>) of boolean;
signal stage_full,required:bool_vector(1 to 24);
                    .
                    .
                    .
Areg<=Areg xor Breg;
stage_full<=(stage_full(13 to 24) & stage_full(1 to 12))
                                        and required;
```

Если массивное значение назначается объекту, то число элементов в подтипе должно проверяться в процессе моделирования. Если размер неправильный, то моделирование будет остановлено и будет выдано соответствующее сообщение об ошибке.

Например, следующее назначение будет вызывать ошибку во время моделирования.

```
Areg <= Areg&Breg; -- ОШИБКА: значение выражения не будет
                   -- переслано в Areg
```

Для того, чтобы быть назначенным массиву данного подтипа или быть связанным с ним, массивное значение должно быть корректного базового типа, то есть недостаточно только того, чтобы совпадали типы входящих в массив элементов.

```
signal data_reg:bit6_data(0 to 7);
signal address_reg: bit6_address(0 to 7);
          .
```

```
          .
          .
data_reg<=address_reg;  -- НЕВЕРНО !
```

Несмотря на то, что оба сигнала являются массивами элементов, имеющими тип bit6, назначение не допускается. Тем не менее преобразование типов (type conversion) допускается между двумя массивными типами если:

1) массивные типы имеют одинаковое число измерений;

2) элементы имеют одинаковые типы и либо индексы целого типа, либо индексы одинакового перечислительного типа.

О двух таких типах говорят, что они тесно связанные (closely related). Таким образом, допустимо следующее преобразование типов и назначение массивов:

```
data_reg <= bit6_data(address_reg);
```

1.3.2.2. Записи

Запись (record) - совокупность объектов, принадлежащих *одному* классу (константы, переменные или сигналы), но, возможно, имеющих различные типы и сгруппированных вместе под одним именем. Записи позволяют рассматривать группы связанных объектов либо как единое целое (unit), либо как отдельные объекты (entities) в зависимости от конкретной ситуации. Элементы записи могут иметь любой предопределенный или определенный пользователем тип, включая ограниченные массивы, а также другие вложенные записи. Тип record должен всегда объявляться до того, как создаются объекты этого типа; недопустимо включать определение записи в объявление объекта, как это делается в некоторых других языках.

Рассмотрим несколько примеров.

```
type coordinate is
     record
          X,Y: length;
     end record;

type index_string is
     record
          str: string (1 to string_len);
          pos: natural range 1 to string_len;
     end record;

type component_id is
     record
          name: string (1 to 20);
          num: natural;
          pos: location;
     end record;

signal current_position, next_position: coordinate;
variable S1, S2, S3: index_string;
variable id: component_id;
```

В объявлении записи приводится ее имя и список имен и типов каждого поля (field) записи. Например, сигнал current_position имеет поля с именами X и Y физического типа length (который должен быть объявлен в другом месте).

Можно ссылаться на всю запись, используя ее простое имя, а также можно ссылаться на отдельное поле, которое может быть получено при помощи выбранного имени (selected name), которое включает символ ".".

```
S1:=S2;
current_position.X <= 37.3 um;
next_position.Y <=current_position.Y+current_height;
S3.str(S3.pos):='A';
S1.str:=S2.str(1 то 3) & S3.str(4 to string_len);
```

Выбранное имя поля записи может быть использовано везде, где может быть использован объект, имеющий тип, совпадающий с типом данного поля. Первый пример - пример назначения записи. Следующие два назначения иллюстрируют выбор поля как в источнике, так и в цели назначения. В четвертом примере S3.str индексируется целым значением S3.pos для выбора одного элемента строки. В следующем примере производится конкатенация частей строковых полей S2 и S3 для формирования значений для поля S1.str.

Значение записи может быть обработано при помощи *агрегатной* формы записи, подобной той, что используется для массивов.

```
constant base_id: component_id :=
     (name => "multiport latch",
      num => 39427,
      pos => (23 um, 123.2 um));
```

```
S1:=((1 to 3 => S2.str(6), others => " "),3);
```

Начальное значение base_id устанавливается при помощи поименованного агрегата. Значение поля pos, самого являющегося записью, записывается при помощи позиционного агрегата. Каждой из первых трех индексных позиций первого поля в S1 назначается шестой символ из S2, а остальные позиции этого поля заполняются пробелами. Полю pos назначается значение 3.

1.3.3. Ссылочные типы и динамические объекты

Все объекты, которые мы рассматривали до сих пор, начинают существовать в результате объявления, появившегося в тексте. Если объявление появляется в процессе, блоке или пакете, то объект создается в начале моделирования и существует все время. Если объявление находится в подпрограмме, то объект создается каждый раз при вызове этой подпрограммы и прекращает существование, когда происходит выход из подпрограммы.

VHDL также обеспечивает возможность для явного создания и удаления объектов под управлением программы. Переменная *ссылочного* типа (access type) похожа на переменную типа *указатель* в других языках (например, в Си), она обеспечивает способ ссылки на *динамически созданный объект* при помощи *генератора* (allocator). Со ссылочным типом могут объявляться только переменные, но не сигналы. Генератор возвращает значение соответствующего ссылочного типа, это значение назначается переменной, и имя переменной может затем использоваться для получения доступа к новому объекту.

```
type coordinate is
    record
      X,Y: length;
    end record;
type locator is access coordinate;
variable A,B,C,D: locator;
constant origin: coordinate:= (0 cm, 0 cm);
                            .
                            .
                            .
A:= new coordinate; -- новый объект создан
B:= new coordinate '(1 um, 2 um);
-- здесь устанавливаются начальные значения
A.X:= B.X+1 um; -- ссылка к полям нового объекта
C:= B; -- C и B сейчас ссылаются к одному объекту
D:= new coordinate; -- создается другой объект
D.all:= origin; -- назначается значение новому объекту
deallocate(A); -- освобождается объект, на который ссылается A.
```

Переменные A, B, C, D имеют ссылочный тип, пригодный для ссылки на объекты типа coordinate. Первое предложение назначения распределяет объект типа coordinate и устанавливает ссылочное значение переменной A, указывающее на новый объект. Назначение для B не только создает новый объект, но также дает его *начальное значение*. Начальное значение задано *квалифицированным выражением*, представленным при помощи агрегата.

В следующем предложении назначается значение полю X объекта, на который указывает A. Затем ссылочной переменной C назначается значение, хранящееся в B. С этого момента B и C указывают на один и тот же объект. Следующее назначение присваивает всем полям D значения из соответствующих полей константы origin. Это изменяет значение объекта, на который указывает D, но не значение самого D. В последнем примере показано динамическое освобождение объекта, созданного в первом примере. Первоначально распределенная память освобождается для повторного использования. Переменная A не будет сбрасываться автоматически.

Проиллюстрируем использование динамического распределения и некоторых других идей, представленных в этом разделе, на подробном примере.

Для начала рассмотрим следующий пакет.

```
package mem_helper is
subtype mem_word is bit_vector (0 to 31);
type page is array (0 to 16#FFF#) of mem_word;
type page_pointer is access page;
type sparse_memory is array (0 to 16#FFF#) of page_pointer;
end mem_helper;
```

Тип page является массивом из 4096 элементов (16^3=4096) , каждый из которых является 32 битовой переменной типа bit_vector (тип word объявлен в другом месте). Каждый элемент типа page_pointer объекта типа sparse_memory содержит ссылочное значение, указывающее на объект типа page. Если все страницы были распределены в памяти, то объект sparse_memory может хранить содержимое 16-ти мегасловной памяти (4096x4096=16777216) - полное 24 битовое (2^{24}=16777216) адресное пространство (с 12 битовым адресом страницы (2^{12}=4096) и 12 битовым адресом внутри страницы).

Тем не менее, скорее всего, при любом моделировании будет задействована только малая часть полного адресного пространства. Поэтому было бы лучше распределять память под страницы только тогда, когда они необходимы. Ниже приведен пакет типов и подпрограмм для управления таким объектом.

```
use mem_helper.all;
package mem_type is
subtype mem is sparse_memory;
subtype word is mem_word;
procedure store (VM: inout mem; loc:address; contents: word);
procedure retrieve
  (VM: inout mem; loc: address; signal value: out word);
end mem_type;
```

В объявлении пакета mem_type содержатся подтипы типов sparse_memory и mem_word, а также процедура для сохранения слова данных в переменной типа mem и функция, восстанавливающая ранее запомненное значение. Заметим, что подтипы не накладывают дополнительных ограничений. Пакет Defs используется вследствие того, что он содержит тип address и функцию IntVal. Рассмотрим, каким образом этот пакет может быть использован.

```
use mem_type.all;
variable sys_mem:mem;
       .
       .
       .
wait until mem_request='1';
if read_write='1' then
-- Читать запрос. Передать на шину данных
-- содержимое памяти
   retrieve (sys_mem,address_bus,data_bus);
else
-- Запись. Сохранить текущее значение на data_bus
-- в памяти
   store (sys_mem, address_bus, data_bus);
end if;
```

Объявление пакета должно иметь связанное с ним тело пакета в случае, если в объявлении пакета объявлены какие-либо подпрограммы. Спецификация каждой из этих подпрограмм должна появиться в соответствующем теле пакета.

Функции store и retrieve определены в теле пакета mem_type.

```
package body mem_type is
procedure store
    (VM: inout mem; loc: address; contents: word) is
constant page_no:natural:= IntVal (loc (0 to 11));
constant page_addr: natural:= IntVal (loc (12 to 24));
begin
  if VM (page_no)= null then
     VM (page_no)= new page'(0 to 16#FFF# => X"00000000");
  end if;
      VM (page_no)(page_addr):= contents;
end store;
procedure retrieve
   (VM:inout  mem; loc: address; signal value:out word) is
   constant page_no: natural:= IntVal (loc (0 to 11));
   constant page_addr: natural:= IntVal (loc (12 to 24));
begin
   if VM(page_no)= null then
        value<= X"00000000";
     else
         value<= VM (page_no)(page_addr);
   end if;
   return;
 end retrieve;
 end  mem_type;
```

VM является массивом памяти, в который записывается contents по адресу loc. И contents и loc рассматриваются, как константы внутри процедуры, а VM и изменяется, и проверяется и поэтому должен иметь вид inout. В процедуре store используется функция IntVal для присвоения начального значения константе page_no. Это начальное значение является целым числом, совпадающим со старшими битами адреса. Константе

page_addr присваивается начальное значение, эквивалентное младшим битам. Инициализация выполняется при каждом вызове функции.

Значение null присваивается по умолчанию тем ссылочным переменным, которым еще не присвоено ссылочное значение, указывающее на распределенный объект. Если выбранная страница не существует (то есть, если соответствующая ссылочная переменная имеет значение null , то распределяется новая страница. При этом производится ее инициализация путем занесения значения '0' в каждый бит каждого элемента.

В любом случае word запоминается в соответствующей странице и по соответствующему адресу. Индекс (page_no) предназначен для выбора элемента из VM; но так как элемент сам является массивом, то необходим второй индекс (page_addr), записываемый сразу за первым, для выбора элемента на странице. Очень заманчиво записать это так (page_no, page_addr), будто бы мы ссылаемся на двумерный массив. Но это было бы некорректно, так как в VHDL, однако, имеется разница между двумерным массивом и массивом, чьи элементы сами являются массивами.

После разбора способа функционирования процедуры store легко понять работу функции retrieve. Заметим, что возвращаемый параметр value должен быть объявлен как сигнал вида out, так что он может служить источником сигнала, связанного с ним в данном вызове. Функция возвращает 32 бита, равных '0', в случае если запрошенная страница не была распределена, и содержимое соответствующей страницы и адреса в противном случае.

1.4. Операции в VHDL

В языке VHDL определен набор основных операторов для предопределенных типов. Для создаваемых пользовательских типов существует возможность определить аналогичные операции.

Аддиативные операторы сложения "+" и вычитания "-", *мультипликативные* операторы умножения "*" и деления "/" определены для типов integer, real и физических типов. Мультипликативные операторы взятия модуля "mod" и остатка "rem" определены для типа integer.

Логические операции not (НЕ), and (И), or (ИЛИ), nand (И-НЕ), nor (ИЛИ-НЕ), xor (исключающее ИЛИ) определены для типов bit и boolean.

Операции *отношения* "=" (равно), "/=" (не равно), ">" (больше), "<" (меньше), ">=" (больше или равно), "<=" (меньше или равно) определены для типов integer, real, а также всех перечислимых и физических типов. Эти операторы дают результат типа boolean.

Для одномерных массивов определены две дополнительные операции - *вырезка* и *конкатенация*. Вырезка позволяет выбрать непрерывное подмножество массива:

```
constant S: bit_vector :="01010";
...S(2)='1' ... -- элемент выбирается индексом
...S(1 to 4)='0101' ... -- вырез выбирается диапазоном
```

Конкатенация создает массив из двух массивов, или из массива и одного значения элементного типа:

```
constant D: bit_vector :="011";
variable F: string (1 to 5);
F(1 to 5) := S(1 to 3) & D(1 to 2);
```

Для пользовательских типов данных можно определелить любые операции с помощью функций. Например, функцию сложения для объекта данных типа bit_vector можно определить как:

```
package functions is
function "+" (a,b: bit_vector) return bit_vector;
```

Далее в теле пакета должна быть описана реализация этой функции. Подобным образом можно создать новые функции и для предопределенных типов данных.

1.5. Операторы управления

Подобно многим машинным языкам высокого уровня в VHDL к *управляющим* операторам относятся условный оператор if, операторы цикла for, loop, оператор выбора case, оператор ожидания wait, оператор возврата return.

Условный оператор *if* имеет следующий формат:

```
if Условие1 then
            оператор1;
            оператор2;
              ...
elsif Условие2 then
            оператор1;
            оператор2;
              ...
else
            оператор1;
            оператор2;
```

```
            ...
end if;
```

Условия должны быть типа boolean. Допускается наличие любого числа конструкций elsif, причем конструкции elsif и else не являются обязательными.

Операторы цикла организуют циклический порядок выполнения операторов:

```
for i in 5 downto 0 loop
 Sum(i):=S*i;
end loop
```

```
S_count: while i<=N loop
 SumT:=SumT+i;
 i:=i+1;
end loop S_count;
```

Оператор выбора *case* выполняет оценку условия и, в зависимости от его значения, выполняет соответствующие группы операторов. Он удобен при небольшом количестве возможных значений условия:

```
case Par is
  when '1' => out:=1;
  when '0' => out:=0;
  when 'Z' => out:=3;
end case;
```

Управляющее выражение может иметь любой скалярный тип, либо быть одномерным массивом.

Оператор *wait* применяется для приостановки процесса на определенный период времени или до момента наступления определенного события. Оператор wait может содержать любую комбинацию из трех дополнительных операторов: on, until, for. Указанный оператор приостанавливает процесс до момента, пока не изменится некоторый сигнал в списке чувствительности, расположенном после on. В это время будет произведено вычисление условия, расположенного после until. Условие - выражение типа boolean. Если получается истинное значение, то выполнение процесса возобновляется. Таймаут, следующий после for, устанавливает интервал времени, через который процесс возобновится. Оператор wait может применяться как со всеми тремя дополнительными условиями, так и с любыми их сочетаниями. При этом условия, проверяемые с помощью on, until и for, проверяются последовательно. Например:

```
 WAIT on X,Y until (Z=0) for 100ns;
 WAIT on X,Y until (Z=0);
 WAIT on X,Y;
 WAIT for 100ns;
```

Оператор возврата *return* (return_statement) используется для завершения выполнения самого внутреннего объемлющего тела функции или процедуры.

Оператор возврата разрешен только внутри тела функции или процедуры и применяется к самой внутренней объемлющей функции или процедуре.

Оператор возврата, стоящий в теле процедуры, не должен содержать выражение. Оператор возврата, стоящий в теле функции, должен содержать выражение.

Значение этого выражения определяет результат, возвращаемый этой функцией. Типом этого выражения должен быть базовый тип обозначения типа, заданного после зарезервированного слова return в спецификации этой функции. Считается ошибкой, если выполнение функции завершается средством, отличным от оператора возврата.

При выполнении оператора возврата сначала вычисляется выражение (если оно задано) и делается проверка на принадлежность значения выражения подтипу результата. Если проверка имеет успех, то на этом выполнение оператора возврата завершается; также заканчивается выполнение объемлющей подпрограммы. Если проверка не имеет успеха, возникает ошибка.

1.6. Пакеты

В языке VHDL предусмотрен механизм пакетов для часто используемых описаний, констант, типов, сигналов. Эти описания помещаются в объявлении пакета (package declaration). Если пользователь использует нестандартные процедуры или функции, их интерфейсы описываются в объявлении пакета, а тела содержатся в теле пакета (package body). Ссылку на описания, содержащиеся в пакете, можно сделать, указав имя пакета, и через точку ту его часть, которая используется. Для использования всего пакета применяют полную ссылку с помощью расширителя .all :

```
package full_p is  -- объявление пакета
 subtype mem is sparse_memory;
 procedure pow (S: inout mem);
end full_p;
package body full_p is  -- тело пакета
 ...  -- тело пакета
end full_p;
```

```
 ...
use full_p.all -- подключаются все описания, введенные
               -- в пакете full_p
 ...
```

1.7. Процессы

Оператор *процесса* определяет независимый последовательный процесс, представляющий поведение некоторой части проекта. Выполнение оператора процесса состоит из повторяющегося выполнения последовательности операторов. После того, как последний оператор в этой последовательности будет выполнен, выполнение оператора процесса продолжается, начиная с первого оператора в этой последовательности.

Процесс состоит из объявлений и операторной части, следующей за словом begin. В объявлениях процесса можно создавать переменные, в то время как объявлять сигналы в этой части не допускается. Внутренние переменные имеют область видимости только внутри процесса, в котором они объявлены.

Предложения внутри процесса называются последовательными предложениями (sequential statements). В отличие от выражений конкретизации компонентов в структурном стиле описания и параллельного назначения в потоковом стиле, выполняемых одновременно, предложения в процессе (потоковое описание) выполняются одно за другим подобно тому, как это происходит в языках программирования. Предложение выполняется только тогда, когда процесс выполнения достигает этого предложения. Типы предложений, допустимые в процессах, включают все условные предложения и предложения циклов, которые встречаются в языках программирования высокого уровня.

В языке VHDL существует два варианта оператора-процесса

```
process(X,Y,Z)                process
_______________               ________
_______________               ________

end process;                  end process;

Вариант A                     Вариант B
```

Вариант A - это процесс, который активизируется, когда меняет свое значение некоторый сигнал в его списке чувствительности (сигналы X, Y, Z). Вариант B не имеет списка сигналов запуска и предполагает, что процесс всегда активен. Вариант A фактически эквивалентен следующему:

```
process
_________
________
WAIT on X,Y,Z
end process;
```

После начала моделирования процесс выполняется только один раз, а затем переходит в состояние ожидания (в конце выполнения), пока не изменятся сигналы, которые его перезапустят. При использовании процесса варианта A нельзя использовать операторы WAIT, поскольку они всегда выполнялись бы до подразумеваемого оператора WAIT в конце процесса, т.е. был бы потерян смысл процесса варианта A. В то же время процессы варианта B могут иметь любое необходимое число операторов WAIT.

Оператор процесса называется *пассивным процессом*, если ни сам процесс, ни любая процедура, для которой этот процесс является родительским, не содержат оператор назначения сигнала. Такой процесс или любой параллельный оператор, эквивалентный такому процессу, может использоваться в разделе операторов объявления объекта.

1.8. Функции и процедуры

В VHDL, как и в других языках, *функции* используются для выполнения часто используемых операций. Для создания функции необходимо описать ее объявление и тело:

```
package conv is
function IntVal(loc: adress) return integer;
package body conv is
function IntVal(loc: adress) return integer is
 variable adr_tmp:integer;
begin
 for i in 15 downto 0 loop
     adr_tmp:=adr_tmp*2;
     if loc(i)='1' then
        adr_tmp:=adr_tmp+1;
     end if;
 end loop;
 return adr_tmp;
end IntVal;
```

```
end conv;
```

Объявление функции содержит имя функции, входные параметры (если они есть), тип возвращаемого значения. Описание тела функции выполняется по правилам, общим для VHDL: в области объявлений описывается заголовок функции, объявляются, если необходимо, внутренние объекты данных, затем, после слова begin описывается реализация функции.

Описание *процедуры* подобно описанию функции, за исключением возвращаемого значения и его типа. Пример описания нескольких процедур приведен в разделе, посвященном динамическим типам данных.

1.9. Оператор блока

Оператор блока определяет внутренний блок, представляющий часть проекта. Блоки могут быть иерархически вложенными для обеспечения декомпозиции проекта.

Если после зарезервированного слова block стоит какое-либо выражение (выражение защиты), то предполагается, что в начале области объявлений (block_declarative_part) этого блока неявно описан сигнал с именем GUARD предопределенного типа BOOLEAN, и это выражение определяет значение этого сигнала. Типом выражения защиты должен быть тип BOOLEAN. Сигнал GUARD может быть использован для управления выполнением ряда операторов внутри блока.

Неявный сигнал GUARD не должен иметь источника. Если в конце оператора блока используется метка, то она должна повторять метку этого блока. Ниже приведен пример оператора блока:

```
lab_0: block
-- выражение защиты
   (S1 = '1' and NCLAR = '1')
-- заголовок блока
   port ( P1,P2 : inout Bit );
   port map (P1 =>w25, P2 =>leng);
-- область объявлений оператора блока
       -- объявление компонент  :
       component Full_Adder
          port (X, Y, Cin: Bit; Cout, Sum: out Bit);
       end component;
       -- объявление сигналов :
       signal A,B,C: Bit;
       signal OK: Boolean;
begin
  -- операторная область оператора блока
  lab_1:  block (Set = '1')
     begin
         lab_2:   process (guard)
              begin
                   if guard then
                      x <= '1';
                   else
                      x <= null;
                   end if;
         end process lab_2;
  end block lab_1;
lab_2: CheckTiming ( tPLH, tPHL, Clk, D, Q );
lab_3 :Name_34 <= guarded
             null,null after h             when des>hu
             else  (a+(j > h)**k)          when j=l
             else    -l and (r or g)       when s
             else    rith ;
       ...
end block lab_0;
```

2. ВИДЫ ПРЕДСТАВЛЕНИЯ ОПИСАНИЙ ПРОЕКТОВ В VHDL

VHDL поддерживает три различных стиля для описания аппаратных архитектур: структурное описание, потоковое описание, поведенческое описание. Все три стиля могут самостоятельно или совместно использоваться для проектирования архитектуры ВС.

2.1. Структурное описание

При *структурном* описании (structural description) объекта проекта архитектура представляется в виде иерархии связанных компонентов.

Каждый экземпляр компонента представляет часть проекта, которая, с другой стороны, может быть описана объектом проекта низшего уровня, также состоящим из связанных компонентов. Таким способом может быть построена иерархия объектов проекта, которая представляет весь проект.

Компонентом может быть один вентиль, микросхема, плата или целая подсистема; иерархия может представлять структурное разбиение проекта или функциональную декомпозицию.

Основные методы и приемы структурного описания представлены далее на примере простой платы памяти.

Структурная схема платы памяти показана на рис. 1 и 2. На этой схеме на уровне блоков показан массив памяти, состоящий из восьми 64Кх1 блоков ПЗУ (ROM), присоединенный к гипотетической шине, а также компонент шинного интерфейса, который обеспечивает временной режим и управление для микросхем памяти и реализует логику управления шиной.

Соответствующее описание платы памяти на VHDL имеет две части: определение интерфейса между проектом и внешним миром (объявление проекта) и проект самой платы (тело архитектуры). На рис. 1 показано объявление объекта для платы памяти. В строке после комментариев разрешается доступ ко всем элементам в пакете определений, называемом Defs, который создан разработчиком для этого проекта. Этот пакет располагается в библиотеке проекта с именем Work.

```
-- интерфейс для платы памяти
use Work.Defs.all; -- Объявления для этого проекта
entity MemoryBoard is
     port ( ABus:     in   address;
            DBus:     out  byte;
            MemReq:   in   wbit;
            BusReq:   out  wbit;
            BusAck:   in   wbit;
            DataRdy:  out  wbit );
     constant Board_id: tribit_vector:="110";
end MemoryBoard;
```

Рис. 1. Объявление интерфейса для платы памяти

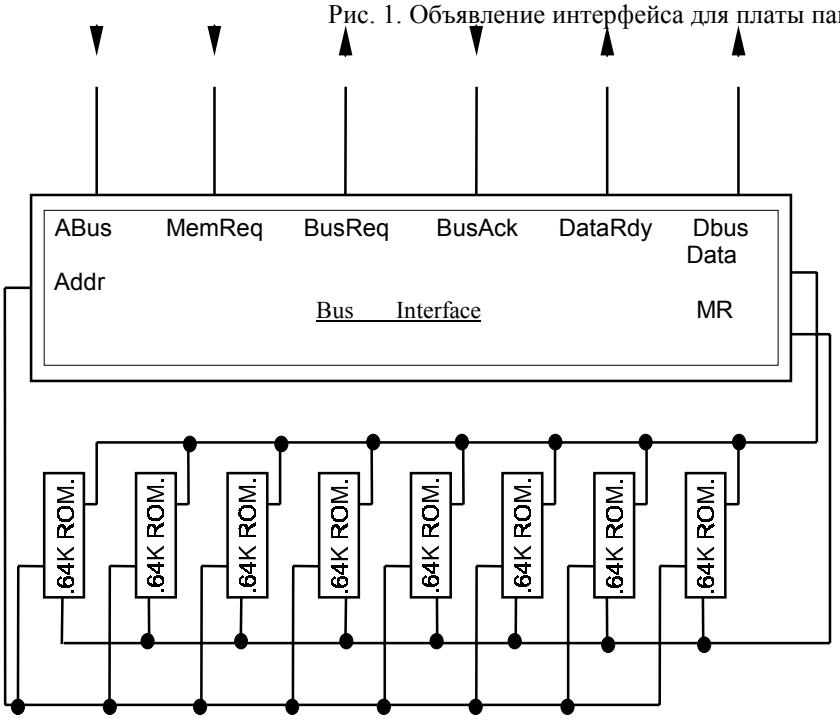

Рис. 2. Простая плата памяти

С ключевого слова entity (объект) начинается описание интерфейса между платой памяти и ее окружением. Разработчик дает этому описанию имя MemoryBoard. После обработки описание будет записано в библиотеку проекта под этим именем.

В списке портов, следующим за ключевым словом port, приводится описание каждой связи с внешним миром посредством ее имени (name), вида (mode) и типа (type). Имя создается разработчиком и, подобно всем идентификаторам, состоит из букв, подчеркиваний и цифр. Идентификатор всегда начинается с буквы. Вид описывает направление передачи данных через порт - либо входное (in), либо выходное (out). Тип описывает способ представления информации, проходящей через порт.

В VHDL имеется предопределенный логический тип - тип bit, принимающий значения '0' и '1'. Тем не менее, разработчик свободен в объявлении дополнительных типов, представляющихся ему подходящими. В данном случае он объявил wbit как четырехзначный тип, который принимает значения '0', '1' ,'Z', 'E'. Значение 'Z' представляет высокий импеданс или отключенное состояние выхода с тремя состояниями, а значение 'E' представляет недопустимое или ошибочное условие. Вводимые типы должны быть объявлены до того, как они могут быть использованы. Объявления типов wbit, address, byte и tribit_vector находятся в пакете Defs.

За списком портов следует объявление константы с именем Board_id, которая будет использована в архитектуре как часть логики декодирования адреса. На любой элемент, объявленный в объявлении объекта, может быть ссылка в архитектуре.

Для архитектуры платы памяти Memory_Board, изображенной на рис. 2, выбрано имя Strucrture. Она приведена на рис. 3. Эта архитектура должна быть записана в ту же самую библиотеку проекта, что и объект Memory_Board. Архитектура Structure разбивается на две части ключевым словом begin. Объявления должны быть расположены до begin, а предложения языка, реализующие проект, после begin.

```
architecture Structure of MemoryBoard is
component ROM64Kx1
       port ( Addr:  in  half_word;
              Data:  out wbit;
              Chip_select: in wbit );
end component;
component Businterface -- Bus handshake and
                       -- memory timing
       generic ( Board_id: in  wbit_vector(0 to 2));
       port( ABus: in address;
             DBus: out byte;
             MemReq: in wbit;
             BusReq: out wbit;
             BusAck: in  wbit;
             DataRdy: out wbit;
             Addr:  out half_word;
             Data:  in  byte;
             MR:    out wbit );
end component;
signal Addr:  half_word;
signal Data:  byte;
signal MR:    wbit;
            .
            .
begin
    INT: Businterface
           generic map (Board_id)
           port map( ABus,DBus,
                     MemReq,BusReq,BusAsk,DataRdy,
                     Addr,Data,MR);
    M0: ROM64Kx1 port map ( Addr,Data(0),MR);
    M1: ROM64Kx1 port map ( Addr,Data(1),MR);
    M2: ROM64Kx1 port map ( Addr,Data(2),MR);
    M3: ROM64Kx1 port map ( Addr,Data(3),MR);
    M4: ROM64Kx1 port map ( Addr,Data(4),MR);
    M5: ROM64Kx1 port map ( Addr,Data(5),MR);
    M6: ROM64Kx1 port map ( Addr,Data(6),MR);
    M7: ROM64Kx1 port map ( Addr,Data(7),MR);
end Structure;
```

Рис. 3. Архитектура платы памяти

Два первых объявления описывают компоненты, которые должны быть использованы. Здесь объявлены два вида компонентов ROM64Kx1 и Businterface. За именем каждого компонента следует список портов, определяющий интерфейс компонента. Список портов компонента похож на список портов в объявлении объекта, то есть каждому порту присвоено имя, вид (направление) и тип. Однако следует помнить, что, несмотря на одинаковые имена, эти порты являются различными объектами, с точки зрения языка.

С компонентом Businterface связан список классов (generic list). Использование параметров классов позволяет описать семейство компонентов одним описанием. Члены семейства различаются значениями, подставленными вместо каждого из параметров класса. Значения для параметров класса могут быть выбраны, когда создается экземпляр компонента.

Каждый экземпляр компонента должен быть ассоциирован с парой объект-архитектура из библиотеки. По умолчанию программа моделирования (simulator) будет осуществлять поиск объекта (entity) с совпадающим именем, списком портов и списком классов в библиотеке для использования в экземплярах компонента.

Три объявления сигналов (signal declaration) следуют за объявлениями компонентов. Сигнал специфицирует путь данных, который может соединять компоненты. Каждый сигнал имеет имя и тип. Если сигнал присоединяется к порту, то тип сигнала и тип порта должны *точно* совпадать. Например, сигнал с типом bit не может быть присоединен к порту типа wbit.

Экземпляры компонентов создаются и соединяются в разделе предложений (statement part) архитектуры, следующим за словом begin. Каждый экземпляр создается посредством предложения *конкретизации компонента* (component instantiation statement). Экземпляр всегда начинается с метки, за которой следует имя компонента. Допускается установка только предварительно объявленных компонентов.

В каждом экземпляре компонента ROM за предложением конкретизации следует предложение карты портов (port map). Каждый элемент в списке является либо именем одного из портов платы памяти, либо локально объявленным сигналом. Если одинаковые имена появляются в картах портов двух экземпляров, то соответствующие порты экземпляров соединены между собой.

Каждая последующая позиция в списке карты портов соответствует локальному порту в той же позиции объявления компонента. Например, адресные линии, выходящие из интерфейса шины (Addr), присоединяются к первому порту каждой микросхемы ROM и однобитовая линия данных от каждой микросхемы ROM присоединяется к одному из элементов массива сигналов с именем Data.

Интерфейс шины INT наряду с картой портов имеет также и карту класса (generic map). Константа Board_id, объявленная в объявлении объекта, используется в качестве значения для параметра класса. Значение константы Board_id будет использоваться внутри реализации шинного интерфейса для определения маски адреса платы. Из семейства плат, Businterface которых соответствует всем возможным адресам, выберется только одна, которая соответствует адресу, определенному константой Board_id.

Длинный список экземпляров ROM в предыдущем описании на структурную архитектуру для платы памяти изменяется только в соответствии с тем, какой индекс используется для выбора элемента массива Data. Альтернативное представление для массива компонентов ROM показано на рис. 4.

```
architecture Structure of MemoryBoard is
                  .
                  .
component ROM64Kx1
          port ( Addr: in   half_word;
                 Data: out  wbit;
                 Chip_select: in  wbit );
end component;
  signal Addr:  half_word;
  signal Data:  byte;
  signal MR:    wbit;
begin
    INT: Businterface  generic map (Board_id)
                       port map (Abus,DBus,MemReq,
                                 BusReq,BusAck,DataRdy,
                                 Addr,Data,MR);
    ROMarray:
    for J in 0 to 7 generate
           M: ROM64Kx1 port map (Addr,Data(J),MR);
    end generate;
end Structure;
```

Рис. 4. Генерация массива компонентов

В разделе предложений используется предложение *generate* для создания 8 экземпляров ROM компонентов. В каждом экземпляре символ J заменяется значением индекса.

2.2. Потоковое описание

При *потоковом* описании (data-flow description) объекта проекта его архитектура представляется в виде множества параллельных регистровых операций, каждая из которых управляется вентильными сигналами. Потоковое описание соответствует стилю описания, используемому в языках регистровых передач.

Структурный тип описания отражает декомпозицию на компоненты и делает акцент на соединениях, которые должны быть проведены между компонентами. Компоненты могут быть некоторыми абстрактными функциональными устройствами или они могут соответствовать физическим элементам, микросхемам или платам в большой системе.

В потоковом описании, наоборот, акцент делается на потоке информации между элементами с вентильным управлением. Этот поток информационного обмена регулируется и направляется управляющими элементами,

которые логически отделены от путей данных. Так как пути данных показаны явно, то потоковое описание не уделяет внимания структуре возможных реализаций.

Основные методы и приемы потокового описания объекта проекта представлены далее на примере блока шинного интерфейса простой платы памяти.

Схема блока шинного интерфейса платы памяти показана на рис. 5. Рассмотрим неформальный комментарий для описания шинного протокола и работы интерфейса, а затем рассмотрим, как этот комментарий может быть реализован в теле архитектуры VHDL, использующей потоковый тип описания.

Объявление объекта Businterface для шинного интерфейса показано на рис. 6. Объект Businterface разработан для того, чтобы иметь возможность разместить экземпляр компонента Businterface внутри архитектуры Structure of MemoryBoard, рассмотренной выше. Объект и компонент имеют одинаковые имена, список портов объекта совпадает со списком портов в объявлении компонента, и список классов объекта совпадает со списком классов в объявлении компонента. Объект Businterface использует типы, объявленные в пакете Defs.

Шинный интерфейс имеет три порта, которые присоединены к массиву ROM: ADDR, задающий адресные входы ROM; MR, присоединенный к линиям выбора микросхем; и DATA, который получает выходные данные ROM. Остальные порты интерфейса шины Businterface присоединены к внешней шине. ABus присоединена к адресным линиям внешней шины. Dbus присоединена к линиям данных, а оставшиеся сигналы задействованы в логике управления шиной. Все сигналы управления шиной имеют активный низкий уровень.

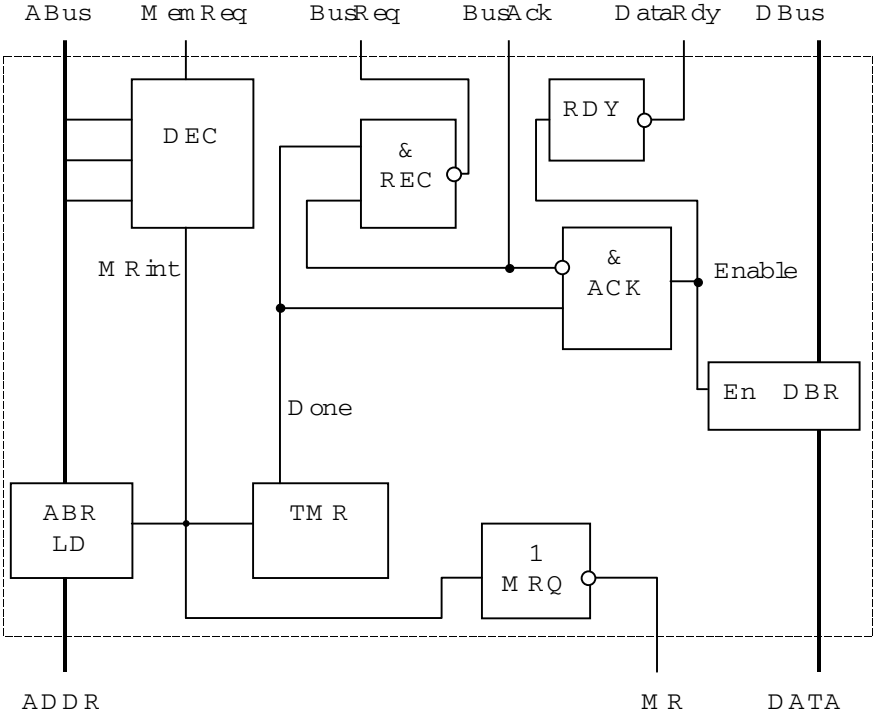

Рис. 5. Схемная диаграмма шинного интерфейса

```
use Work.Defs.all;
entity Businterface is
       generic
          (  Board_id:  in wbit_vector (0 to 2));
       port
          (  ABus:      in  address;
             DBus:      out byte;
             MemReq:    in  wbit;
             BusReq:    out wbit;
             BusAck:    in  wbit;
             DataRdy:   out wbit;
             Addr:      out half_word;
             Data:      in  byte;
             MR:        out wbit   );
 end Businterface;
```

Рис. 6. Объявление объекта для шинного интерфейса

Цикл доступа к памяти начинается, когда MemReq переходит в нуль и адрес, хранящийся в 16-18 битах ABus, равен номеру платы. Эта комбинация событий вызывает установление декодером DEC сигнала MRint на плате в единичное состояние. MRint управляет загрузочным входом триггера адреса ABR, запускает таймер TMR и устанавливает буфер MRQ в нуль.

MRQ управляет входами выбора микросхем ROM. Выход компонента TMR с именем Done переходит в состояние 1 через 175 нс после переключения MRint из нуля в единицу и вновь переключается в нуль, как только MRint отключается. Таким образом, запрос на шину задерживается на 175 нс после готовности линий адреса. Done сохраняет BusReq в нуле до тех пор, пока не будет получен ответ BusAck от арбитра шины. Шинный сигнал BusAck инвертируется и затем его конъюнкция с Done порождает сигнал Enable.

Это вызывает установку DataRdy в нуль и в то же самое время делает доступными тристабильные выходы элемента DBR. Выходы DBR будут доступны до тех пор, пока BusAck не переключится в единицу.

2.2.1. Объявление архитектуры для шинного интерфейса

Объявление архитектуры для описываемого шинного интерфейса показано на рис. 7. Архитектура DataFlow компонента Businterface объявляет три внутриплатных сигнала Done, Enable и MRint. Константа Zs объявляется и инициализируется, как массив высокоимпедансных логических состояний шириной в байт, Es соответствует массиву значений "ошибка".

Каждое устройство, отмеченное на схеме, представляется предложением *параллельного назначения сигналов* (concurrent signal assignment) в разделе предложений DataFlow, которое задает значения выхода после обработки значений выходов других устройств. В некоторых случаях для модификации поведения устройства используются управляющие сигналы. Символ "<=" служит для назначения сигналов.

```
architecture DataFlow of Businterface is
     signal Done, Enable, MRint: wbit;
     constant Zs:byte:="ZZZZZZZZ";
     constant Es:byte:="EEEEEEEE";
 begin
     RDY: DataRdy <= not Enable after 5ns;
     MRQ: MR <= not MRint after 5ns;
     ACK: Enable <= Done and not BusAck after 5ns;
     REQ: BusReq <= not(Done and BusAck) after 5ns;
     TMR: Done   <= MRint and MRint'delayed(175ns);
     DEC: Mrint  <=
            MemReq and
            not AnyOf(ABus(16 to 18) xor Board_id)
                      after 5ns;
     DBR: with Enable  select
               Dbus <= Data after 9ns  when '1',
                       Zs after 7ns    when '0',
                       Es              when others;
     ABR: block (MRint = '0')
          begin
              Addr <= guarded ABus(0 to 15) after 12ns;
          end block;
end DataFlow;
```

Рис. 7. Архитектура для шинного интерфейса

Все параллельные назначения действуют параллельно и порядок, в котором они записаны, не имеет значения. Для демонстрации различных форм параллельного назначения сигналов детально рассмотрим каждое из устройств. Первое предложение в архитектуре DataFlow для Businterface соответствует драйверу шины RDY. Целью присваивания является порт DataRdy. Сигнал, который появляется справа, определяет, что DataRdy будет все время равняться инверсии внутреннего сигнала Enable, но он будет задержан относительно изменений Enable на 5 нс с целью учета задержки прохождения через буфер. Аналогично драйвер MRQ представляется простым присвоением сигнала, переводящим порт MR в состояние, равное инвертированному значению MRint.

Вентиль ACK генерирует внутриплатный сигнал Enable. Вентиль REQ генерирует выход BusReq. В этих случаях сигнал справа отражает логику в схеме. Сигнал Done генерируется одновибратором TMR. Выражение для сигнала использует сигнал MRint'Delayed (175 ns), как точную варьируемую во времени копию MRint с задержкой 175 нс. 'Delayed является предопределенным атрибутом любого сигнала. Другие предопределенные атрибуты могут быть использованы для ссылки на предыдущее значение сигнала или для фиксации переднего или заднего фронта импульса.

DEC устанавливает на MRint значение сигнала, полученное после обработки шинного сигнала MemReq и части разрядов (с 16 по 18) адресной шины Abus. Board_id является константой класса, чье значение окончательно определяется конкретизацией компонента Businterface в архитектуре платы памяти. AnyOf - функция, объявленная в пакете Defs, которая возвращает значение типа MVL, равное '1', если в ее аргументе установлен любой бит, и возвращает '0' в противном случае.

Тристабильный драйвер DBR реализован при помощи другого вида предложения параллельного назначения сигналов, называемого *выборочным* (selected) *назначением сигналов*. В предложении имеется два *переключения* (waveforms), но только одно из них влияет на DBus в данное время. Выбор осуществляется при помощи значения сигнала Enable. Когда Enable равен '1', на выходах устанавливается значение сигнала Data,

приходящего с массива ROM. Когда Enable равен '0', то шинный интерфейс полностью отсоединяется от DBus путем установки высокоимпедансного состояния 'Z' на всех линиях. В случае, когда Enable равен 'Z' или 'E', на всех линиях устанавливается значение ошибки 'E'.

Триггер адреса ABR, являющийся триггером-защелкой, описывается при помощи *управляемого назначения* (guarded assignment). Выражение в скобках, в начале открывающего блок предложения, называется *управляющим выражением* (guard expression). Предложение внутри блока с ключевым словом guarded в правой части будет воздействовать на выход, только когда значение управляющего выражения истинно. В этом случае значение ADDR будет повторять значения с 0 по 15 разрядов ABus до тех пор, пока значение MRint равно 0. Когда MRint переключается в единицу в начале цикла доступа к памяти, то адрес фиксируется и последующие изменения на ABus не оказывают влияния на ADDR.

Архитектура использует множество логических операторов над типами MVL и MVL_vector. Эти операторы не являются предопределенными; они объявлены в пакете Defs.

2.2.2. Разрешенные сигналы

На рис. 8 приведен фрагмент архитектуры системы сбора данных, которая собирается из компонентов на уровне плат. Среди компонентов имеется MemoryBoard и плата AtoD, содержащая аналого-цифровые преобразователи и связанную с ними логику. Приведены объявления некоторых из сигналов на объединительной плате, которые должны быть использованы для соединения компонентов.

Во второй части рис. 8 после слова begin сигналы присоединяются к портам компонентов MemoryBoard и AtoD при помощи *именованного сопоставления* (named association). Каждый элемент списка портов состоит из имени порта (из объявления компонента), за которым следует символ =>, и затем сигнал, который должен быть связан с этим портом. (не следует путать символ <= , используемый при назначении сигнала, с символом =>, используемым в именованном сопоставлении).

```
architecture boards of acquisition_system is
   component MemoryBoard
     port ( ABus:    in  address;  DBus:     out  byte;
            MemReq:  in  wbit;     BusReq:   out  wbit;
            BusAck:  in  wbit;     DataRdy:  out  wbit);
   end component;
   component AtoD
     port ( ABus:    in  address;  DBus:     out  byte;
            BusReq:  out wbit;     BusAck:   in   wbit;
            DataRdy: out wbit );
   end component;
   signal ADBS:  address_bus;
   signal DBS:   byte;
   signal BRQ, BAC, DRY, MRQ:  wbit;
                  .
                  .
begin
   Slot_1:  MemoryBoard
      port map ( ABus=>   ADBS,   DBus=>   DBS,
                 MemReq=> MRQ,    BusReq=> BRQ,
                 BusAck=> BAC,    DataRdy=>DRY );
   Slot_2:AtoD
      port map ( ABus=>   ADBS,   DBus=>   DBS,
                 BusReq=> BRQ,    BusAck=> BAC,
                 DataRdy=>DRY);
   Slot_3: ...
                  .
                  .
```

Рис. 8. Архитектура, использующая плату памяти

Каждый из сигналов на соединительной плате присоединяется к нескольким портам. Например, ADBS присоединяется как к адресному порту платы памяти, так и к такому же порту на плате аналого-цифрового преобразователя. Каждая из подсистем использует значение ADBS для определения: выбрана ли она в текущий момент времени.

Смысл других соединений не так ясен. Сигнал BRQ устанавливается запросом на шину выходов обеих плат. Каждая плата устанавливает BRQ в '0', если ей требуется использовать шину данных, и '1' в противном случае. Говорят, что сигнал BRQ *имеет множество источников*. В каждый момент времени только одна из плат на соединительной плате устанавливает значение '0'. Какое же значение принимает BRQ?

Результат зависит от технологии реализации. При использовании ТТЛ логики с открытым коллектором результатом будет проводное "И" всех источников. При использовании ТТЛ с обычными выходами результатом может произойти выход из строя соответствующего логического элемента (представленный символом 'E' в

многозначной логике). *Функция разрешения* (resolution function) обеспечивает возможность моделирования этих зависящих от технологии особенностей.

Для того, чтобы понять, как функция разрешения связывается с сигналом, подобным BRQ, необходимо рассмотреть более подробно пакет Defs. Пакет содержит объявления типов и подпрограмм. Пакет делится на две части: объявление пакета и тело пакета. Тело пакета содержит реализации всех подпрограмм, имеющихся в объявлении пакета.

Объявление пакета для пакета Defs на рис. 9 демонстрирует механизм, необходимый для описания двух различных типов разрешения сигнала. Первый вид - разрешение типа логическое И. Второй - тристабильное разрешение. При тристабильном разрешении ожидается, что все источники сигнала, кроме одного, устанавливаются в отключенное состояние 'Z'. Оставшийся '0' или '1' в группе источников будет определять окончательное состояние сигнала.

```
package Defs is
-- многозначный логический тип
type MVL is ('1','0','Z','E');
type MVL_vector is array (natural  range<>) of MVL;
-- подтип проводной логики для MVL
function wired_and(X:MVL_vector) return MVL;
subtype wbit is wired_and MVL;
type wbit_vector is array (natural  range <>) of wbit;
-- подтип тристабильной логики для MVL
function tristate(X:MVL_vector) return MVL;
subtype tribit is tristate MVL;
type tribit_vector is array (positive range <> )
                                  of tribit;
-- подтипы используются для объявления сигналов,
-- переменных  и констант
subtype address is tribit_vector(0 to 18);
subtype byte is tribit_vector(0 to 7);
subtype half_word is tribit_vector(0 to 15);
-- некоторые служебные функции
function AnyOf(X:tribit_vector) return MVL;
function IntVal(X:tribit_vector) return integer;
-- Другие объявления функций, необходимые для
-- полноты Defs...
function "and" (X,Y:MVL) return MVL;
function "not" (X:MVL) return MVL;
                        .
                        .
end Defs;
```

Рис. 9. Объявление пакета Defs

Тип MVL представляется списком из четырех логических значений. MVL_vector является типом, который может быть использован для объявления одномерных массивов MVL-значений.

Один из способов, при помощи которого функция разрешения может быть связана с сигналом, является объявление подтипа. Разрешение “проводное И” связывается с сигналом подтипа wbit; тристабильное разрешение связывается с сигналом подтипа tribit.

Функция wired_and реализует разрешение сигнала типа проводное И. Она ассоциируется с сигналами подтипа wbit, так как это подразумевается в объявлении типа wbit. Как только сигнал типа wbit необходимо оценить, программа моделирования будет собирать значения всех источников в массив и вызывать функцию wired_and, которой в качестве аргумента будет передаваться этот массив. Функция wired_and будет просматривать собранные значения и возвращать одно *разрешенное значение* (resolved value), как результат. Это разрешенное значение становится новым значением сигнала. Тип wbit_vector может быть использован для объявления массивов wbit элементов. Каждый элемент будет разрешаться независимо.

Тип tribit, функция разрешения tristate и массив типа tribit_vector определяются аналогично. Программа моделирования будет передавать массив MVL-значений функции tristate, как только появится необходимость оценить сигнал типа tribit. Если больше, чем один элемент массива не равен 'Z', то tristate сигнализирует об ошибке, возвращая в качестве результирующего значения сигнала 'E'. В противном случае возвращаемым значением будет значение единственного не равного 'Z' сигнала. Значение каждого элемента массива tribit_vector будет разрешаться независимо.

Последние объявления в Defs создают подтипы и функции, которые используются в архитектуре плат и во всей этой главе. Заметим, что операторы and и not получили добавочный смысл - они в дальнейшем могут быть также хорошо использованы для MVL-значений.

2.2.3. Шины и регистры

Архитектура DataFlow для Businterface на рисунке 7 хорошо отражает схему шинного интерфейса. Ее соответствие словесному описанию шинного интерфейса гораздо хуже. Словесное описание дает разработчику последовательность состояний. Каждое состояние обусловлено некоторым множеством условий. В каждом состоянии сигналам присваиваются определенные значения. В DataFlow эта упорядоченная последовательность не совсем ясна. Управляющая логика и передача данных не разделены.

Ниже приведена модификация потокового стиля описания, в которой управляющая логика и передача данных строго разделены.

```
use Work.Defs.all;
architecture RT of Businterface is
   type state_value is (IDLE, NEED_DATA, NEED_SYS, DRIVE_SYS);
   signal RDY,REQ,MRQ: wbit  bus;
   signal DBR:         byte  bus;
   signal ABR:         half_word register;
   signal state:       state_value register;
begin
                  .
                  .
```

Рис. 10. Архитектура на уровне регистровых передач для шинного интерфейса

На рис. показана новая архитектура для Businterface. Объект типа state_value может принимать одно из четырех значений, представляющих состояния управляющей логики. Сигнал state будет хранить имя текущего состояния. Приведем значения состояний и их смысл:

IDLE - ожидание запроса на память, устройство отключено от шины данных;

NEED_DATA - запрос получен, ожидание готовности данных для ROM;

NEED_SYS - данные для ROM готовы, запрашивается использование шины данных;

DRIVE_SYS - подтверждение шины получено, управляемая шина данных.

Одно состояние будет следовать за другим в данном порядке, точно так же, как в словесном описании. Когда арбитр шины отменит разрешение на управление шиной, то управляющая логика вернется в состояние IDLE.

Большинство сигналов в архитектуре RT для Businterface являются *управляемыми сигналами* (guarded signals). Имеется два вида управляемых сигналов: *регистры* и *шины*. Не нужно путать это специализированное понятие "шина" с инженерным термином "шина", применяемым для любой связанной группы сигналов. Управляемые сигналы и обычные сигналы имеют следующие различия:

1) все управляемые сигналы должны быть *разрешенными*, то есть с каждым сигналом через его подтип должна быть связана функция разрешения;

2) если управляемому сигналу присваивается значение в предложении присваивания сигнала, то предложение должно быть управляемым, то есть его правая часть должна начинаться ключевым словом guarded; управляемое назначение почти всегда содержится внутри управляемого блока;

3) если управляющее выражение, контролирующее назначение - ложно, то сигнал становится *отсоединенным* (disconnected) и драйвер не может влиять на значение сигнала; массив, переданный функции разрешения, не должен включать значения, соответствующего отключенному драйверу.

Различия регистра и шины заключаются в следующем:

1) если ни один драйвер не подсоединен к шине, то функция разрешения должна вызываться с массивом нулевой длины. Функция должна тогда возвращать некоторое разрешенное значение для шины, которое представляет абсолютное отключение (например 'Z');

2) если ни один драйвер не подсоединен к регистру, то в регистре сохраняется состояние, которое было занесено в него при последнем подключении, а функция разрешения даже не вызывается.

Раздел предложений архитектуры RT состоит из трех секций. Первая секция (рис. 11) содержит управляемые назначения, которые определяют потоки данных, которые появятся в каждом состоянии шинного интерфейса.

Каждый блок с метками от F1 до F4 соответствует одному из управляющих состояний. Блок содержит множество назначений, которые являются активными, только когда контроллер находится в соответствующем состоянии. Это обеспечивается управляющим выражением для блока.

```
                              .
                              .
F1:block  (state  = IDLE) -- Управляющие выражения для
                          -- каждого блока определяют
                          -- состояние
 begin
      ABR<=guarded Abus (0 to 15) after 12 ns;
 end block;

 F2:block  (state  = NEED_DATA)
```

```
begin
     MRQ<=guarded '0' after 5 ns;
end block;

F3:block  (state  = NEED_SYS)
begin
     REQ<=guarded '0' after 5 ns;
end block;

F4:block  (state  = DRIVE_SYS)
begin
     RDY<=guarded '0' after 5 ns;
     DBR<=guarded Data after 9 ns;
end block;
                         .
                         .
```

Рис. 11. Архитектура регистровых передач для шинного интерфейса. Функциональные блоки

Объявления на рис. 10 включают четыре локальных шины и один локальный регистр, каждому выходному порту соответствует один сигнал. Только один или два из этих сигналов задействованы в каждом блоке. Когда контроллер находится в состоянии, которое не влияет на данный регистр, то этот регистр остается в предыдущем состоянии. Если контроллер должен ввести новое состояние, которое установит в регистре новое значение, то значение регистра должно измениться соответствующим образом. Когда контроллер находится в состоянии, которое не управляет данной шиной, тогда этой шине по умолчанию присваивается 'неуправляющее' (not driving) значение функцией разрешения сигнала, которая вызывалась с массивом аргументов нулевой длины.

Например, шина RDY является управляемой только в состоянии DRIVE_SYS. Как только система выходит из состояния DRIVE_SYS, функция разрешения типа wbit (функция wired_and) должна будет вызываться с массивом нулевой длины. Функция wired_and будет возвращать '1' в качестве значения сигнала. В результате RDY, когда отсоединяется, то переходит в состояние '1', подобно ТТЛ логике с открытым коллектором.

Каждый элемент DBR имеет тип tribit, которому соответствует функция разрешения tristate. Когда все драйверы элементов типа tribit отсоединены, вызывается функция tristate со входным массивом нулевой длины, которая возвращает высокоимпедансное состояние 'Z' и имитирует трехзначную ТТЛ логику.

Напротив, ABR является регистром. Он сохраняет значение, установленное на адресных входах в момент, когда контроллер покидает состояние IDLE. Он не изменится до тех пор, пока машина состояний опять не войдет в состояние, в котором ABR является управляемым. В примере имеется только одно такое состояние IDLE, но в общем случае любое число различных состояний может изменять значение ABR.

```
state <= NEED_DATA when
              state = IDLE and
              MemReq = '0' and ABus(16 to 18) = Board_id
            else NEED_SYS when
              state = NEED_DATA and
              state'stable(175 ns)
            else DRIVE_SYS when
              state = NEED_SYS and
              BusAck = '0'
            else IDLE when
              state = DRIVE_SYS and
              MemReq = '1'
            else state;
```

Рис.12. Переходы между состояниями

За потоковыми блоками следуют по одному условному предложению параллельного назначения сигналов, они составляют управляющую логику для RT (рис. 12). Имеется по одному условному предложению для каждого перехода в каждое новое состояние. Условием является конъюнкция текущего состояния и условия, сигнализирующего о переходе в следующее состояние.

Последний блок предложений параллельного назначения сигналов в RT (рис. 13) просто передает текущие значения сигналов в соответствующие порты.

```
                        .
                        .
DBus          <=DBR
   BusReq        <=REQ
   DataRdy       <=RDY
   Addr          <=ABR
   MR            <=MRQ
end RT;
```

Рис. 13. Присваивание значений портам устройства

2.3. Поведенческое описание

Поведенческий стиль описания определяет последовательно выполнимый код процедурного типа, подобный коду на языках программирования Фортран или Ада. Некоторые языки описания аппаратуры обеспечивают механизм для вызова кода, написанного на Фортране, Паскале или Си, для того чтобы описать сложное поведение. Другие, включая VHDL, интегрируют поведенческие предложения в язык.

Имеется много причин для использования поведенческого стиля в описании аппаратуры, но основным является то, что поведенческое описание определяет с любой желаемой степенью точности функционирование устройства без определения его структуры.

Например, разработчик может детально специфицировать поведение подсистемы, а проработку деталей реализации передать другим. Реализация может выполняться либо специализированным предприятием, либо может быть передана в другую группу в той же самой организации. При использовании поведенческого стиля описания разработчик избегает чрезмерной спецификации структуры или уклона разработки в сторону какой-либо одной технологии реализации.

С другой стороны, изготовитель может поставлять точное определение функционирования подсистемы без упоминания специфических деталей структуры или изготовления.

Используя поведенческое описание изготовителя, заказчик может построить соответствующее структурное описание для подсистемы.

Предложения поведенческого стиля в VHDL представляют современный язык структурного программирования, который ближе всего к Паскалю по области применения, мощности и легкости использования. На рис. 14 приведена реализация полной платы памяти в поведенческом стиле.

```
use Work.Defs.all; -- Объявления для этого проекта
entity MemoryBoard is
      port
        ( ABus:     in   address;
          DBus:     out  byte;
          MemReq:   in   wbit;
          BusReq:   out  wbit;
          BusAck:   in   wbit;
          DataRdy:  out  wbit );
      constant Board_id: tribit_vector:="110";
end MemoryBoard;
architecture Behavior of MemoryBoard is
    signal Addr:              integer;
    signal Data:              byte;
    signal MRint:             bit;
    constant Zs:byte:="ZZZZZZZZ";
begin
Memory:process
       begin
             wait until MRint='1';
             Data<=ROM_Contents(Addr) after 150ns;
       end process;
                        .
                        .
Control:process
             variable addr_temp:integer;
       begin
  wait until MemReq='0'and ABus(16 to 18) = Board_id;
                  addr_temp:=0;
                  for i in 15 downto 0 loop
                       addr_temp:=addr_temp*2;
                       if ABus(i)='1' then
                             addr_temp:=addr_temp+1;
```

```
                    end if;
                end loop;
                Addr<=addr_temp;
                MRint<='1';
        wait for 175 ns;
                BusReq<='0' after 5ns;
        wait until BusAck='0';
                BusReq<='1' after 5ns;
                DBus<=Data after 9 ns;
                DataRdy<='0' after 5 ns;
        wait until MemReq /='0' or ABus(16 to 18) /=
Board_id;
                MRint<='0';
                DBus<= Zs after 7ns;
                DataRdy<='1' after 5 ns;
      end process;
end Behavior;
```

Рис. 14. Поведенческое описание устройства

Объявление интерфейса для платы памяти MemoryBoard совпадает с объявлением, описанным с помощью структурного стиля, различаются только определения архитектуры.

В объявлениях архитектуры Behavior вводятся три локальных сигнала - скалярный сигнал целого типа Addr, сигнал Data массивного типа byte, и скалярный сигнал MRint перечислимого типа bit.

Операторная часть архитектуры содержит два процесса, начинающихся с ключевого слова process и заканчивающихся выражением end process.

Процесс с меткой Memory содержит описание выборки значения одного из элементов массива ROM_Contents и назначения этого значения сигналу массивного типа Data. Данное назначение происходит каждый раз, когда сигнал MRint принимает значение '1'. Массив ROM_Contents является одномерным массивом элементов типа byte и отображает данные, хранимые в ПЗУ.

Процесс Control последовательно описывает события, отражающие работу платы памяти. Когда моделирование начинается в момент времени t=0, процесс приостанавливается на первом операторе wait до тех пор, пока часть адреса Abus(16 to 18) не будет соответствовать константе Board_id и не поступит запрос на использование памяти MemReq. При достижении этого условия с помощью оператора цикла loop выполняется преобразование адреса на Abus в целое число, которое присваивается переменной addr_temp, а затем назначается сигналу Addr. Подобное преобразование выполнялось с помощью функции IntVal в разделе, посвященном динамическим объектам. После этого сигнал MRint принимает значение '1', что отслеживается процессом Memory, который выполняет выборку соответствующего элемента памяти. Следующий оператор wait вводит паузу длиной 175 нс, необходимую для ожидания готовности данных от ПЗУ. После паузы сигнал BusReq принимает значение '0'. Как видно, в данном примере можно обойтись без внутреннего сигнала Done, введенного в потоковом описании архитектуры DataFlow. Далее на следующем операторе wait ожидается подтверждение BusAck. Когда этот сигнал примет значение '0', сигнал BusReq сбрасывает активный низкий уровень, а порту Dbus назначается содержимое сигнала Data. При этом порту DataRdy назначается активный уровень '0'.

Следующий оператор wait приостанавливает процесс до момента, когда плата памяти станет неактивной. В этом состоянии сигнал MRint сбрасывается в неактивное состояние, порт Dbus переходит в третье состояние, а порт DataRdy устанавливается в '1'. Устройство находится в состоянии ожидания следующего цикла доступа.

3. МОДЕЛИРОВАНИЕ В VHDL

Одной из основных возможностей языка VHDL является наличие развитых средств моделирования ВС. Генерация тестов и возможность моделирования систем с выявлением и анализом неисправностей обеспечивает выполнение верификации разработок на различных этапах маршрута проектирования.

Два важных принципа определяют моделирование на VHDL. Во-первых, изменение значений сигнала всегда вызывает выполнение всех назначений сигнала в проекте, вследствие чего изменяются значения *целей*, участвующих в этих назначениях. Это, в свою очередь, вызывает дополнительные изменения. Таким образом, в результате получается *последовательность* событий. Одновременно может появляться много независимых последовательностей событий. Во-вторых, изменения могут осуществляться после некоторой задержки.

Сначала рассмотрим несколько понятий, необходимых для организации моделирования в VHDL.

3.1. Временные диаграммы

В правой части параллельного или последовательного назначения сигнала имеется одна или более *временных диаграмм* (waveforms), каждая из которых представляется списком элементов временных диаграмм, разделенных запятыми. В каждом из элементов диаграммы определяется значение сигнала и задержка, через

которую планируется появление этого значения. Например, сигнал step имеет текущее значение 0; предусмотрено следующее назначение сигнала:

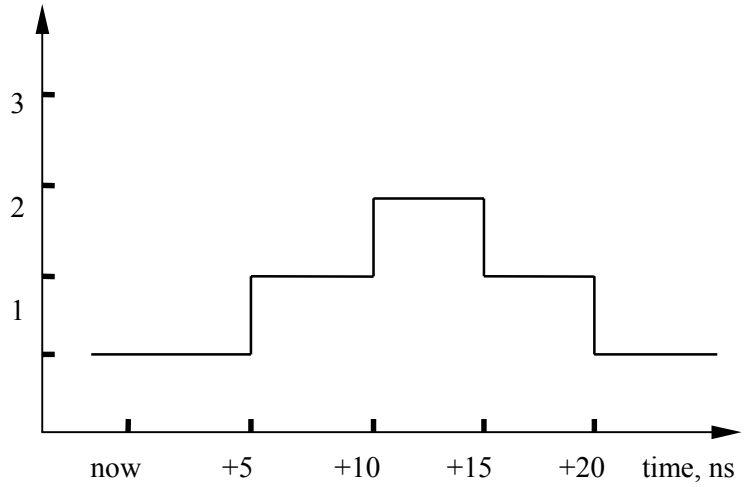

Рис. 15. Сигнал step, как функция времени

```
step <= 1 after 5ns, 2 after 10ns, 1 after 15ns,
        0 after 20ns.
```

Эта временная диаграмма содержит четыре элемента. Задержки задаются относительно момента, в который выполняется назначение. На рис. 15 приведены значения step как функции времени по отношению к моменту now - моменту выполнения назначения.

3.2. Формирователи

Формирователь (driver) сигнала - структура данных, которая содержит текущее значение и последовательность пар время-значение. Элементы временной диаграммы передаются формирователю, после того, как выполнится назначение сигнала. Каждое значение становится текущим значением формирователя, как только в процессе моделирования достигается соответствующий момент времени. Четыре элемента временной диаграммы, предназначенной для назначения сигналу step, добавляются к формирователю этого сигнала после того, как выполняется назначение. На рис. 16 показано представление формирователя.

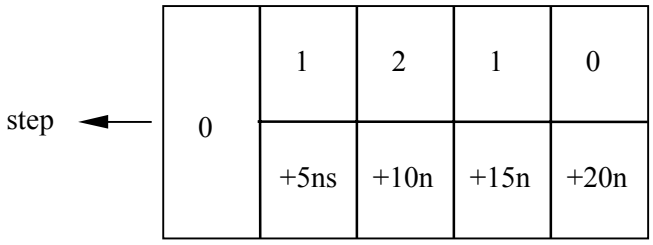

Рис. 16. Представление формирователя сигнала

Формирователь является *источником* (source) значения сигнала. Если сигнал связан с портом компонента, то этот порт также может быть источником сигнала. *Эффективное значение* (effective value) сигнала только с одним источником - это текущее значение формирователя или порта, связанного с сигналом. Однако, если у сигнала имеется более, чем один источник, тогда все источники учитываются при вычислении эффективного значения. Сигнал с множеством источников должен иметь связанную с его типом функцию разрешения. Функция разрешения вычисляет одно эффективное значение на основе массива значений источников, переданных ей программой моделирования.

Целью (target) назначения сигнала является сигнал или агрегат в левой части оператора назначения. Программа моделирования создает формирователь для каждого элемента цели каждого параллельного назначения сигнала. Внутри предложения process создается только по одному формирователю для каждого элемента-цели, даже если элемент появляется более, чем в одной цели. Рассмотрим некоторые примеры.

```
(min,max) <= limits (Q) after 10 ns;
reg (0 то 3) <= "0101" after gate_delay;
```

Цель первого назначения задана в форме агрегата. Программа моделирования создаст отдельные формирователи для сигналов min и max. Функция limits формирует запись с двумя полями; значение первого

поля размещается в формирователе для min, а значение второго - в формирователе для max. Задержка возникновения соответствующих значений обоих сигналов равна 10 нс относительно момента выполнения назначения.

Второй целью является вырез массива. Формирователи будут построены для каждого из четырех заданных элементов массива reg.

3.3. Модели временной задержки

В VHDL поддерживаются три различных модели временной задержки. Модель *инерционной задержки* (inertial delay) соответствует временному промежутку между стабильными значениями на входах и выходах логического элемента. Значения на выходах изменяются после того, как значения на входах будут неизменны в течение времени, равного задержке. Если же длительность изменений на входах меньше, чем время задержки, то значение выхода не изменится, то есть если длительность переходного процесса на входе меньше времени задержки, то он подавляется. Инерционная задержка является наиболее распространенной в реальном мире, поэтому принята в VHDL в качестве модели задержки по умолчанию.

Другой тип задержки, встречающийся в реальном мире, - задержка прохождения сигнала через металлическую проводящую трассу или через линию задержки. В этом случае любое изменение на входе вне зависимости от того, какой оно длительности, проявится на выходе через время, равное задержке прохождения. Задержка прохождения может быть во много раз больше, чем длительность входного импульса. Для описания этого предусмотрена модель *транспортной задержки* (transport delay), которая снимает требование, чтобы значения на входах были неизменны не менее времени задержки. *Переходные процессы* (transients) не подавляются, и любое изменение на входах вызывает аланалогичное изменение на выходе. Для использования модели транспортной задержки перед правой частью предложения назначения сигнала должно быть записано ключевое слово transport:

```
pad(i) <= transport input(i)after met_del*path_l(i);
```

Целью назначения является один элемент массива сигналов pad, присоединенный к элементу из массива input посредством металлического соединения, длина которого содержится в соответствующем элементе массива path_l. Значение met_del имеет физический тип time и представляет задержку, соответствующую задержке соединения единичной длины.

На рис. 17 приведены временные диаграммы на выходе буферного элемента с инерционной задержкой (A) и с транспортной задержкой (B), полученные в результате поступления на его вход временной диаграммы S.

VHDL имеет третью модель задержки, называемую *дельта задержкой* (delta delay). Это специальная модель задержки для моделирования последовательности событий. Дельта задержка может быть использована при моделировании с равными задержками, так как каждая дельта задержка между событиями может рассматриваться, как равная любой другой дельта задержке.

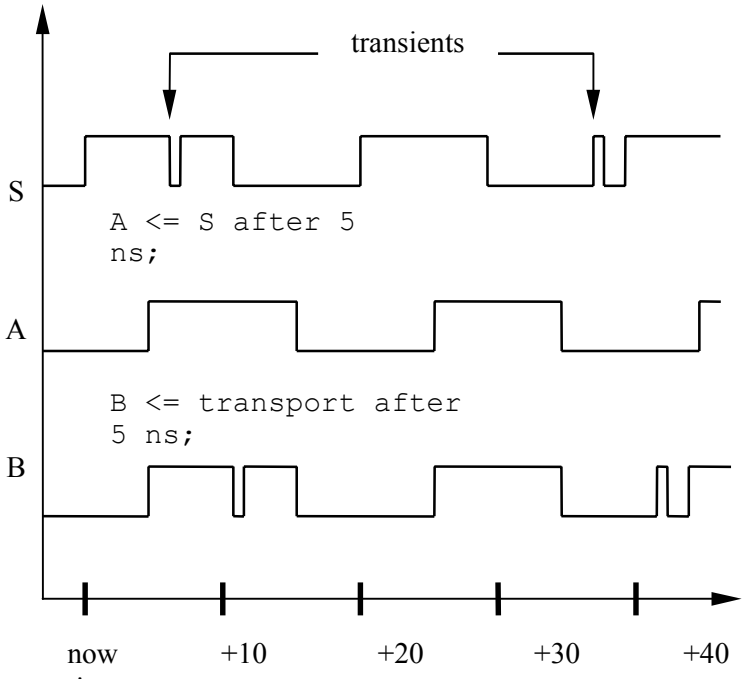

Рис. 17. Моделирование задержки прохождения через буфер

Дельта задержка представляется как бесконечно малая задержка, меньшая, чем любое измеримое время (т.е. фемтосекунда), но большая, чем нуль. Значение, назначенное с дельта задержкой, будет иметь эффект в будущем. Однако, поскольку задержка - бесконечно малая, то назначение значения произойдет до того, как

будут назначены значения с любой задержкой, имеющей реальное значение, вне зависимости от того, насколько оно мало. Необходимо отметить, что любое число последовательных дельта задержек не может быть приравнено никакой реальной величине времени.

Если первый (или только первый) элемент временной диаграммы имеет значение времени, равное нулю, или совсем не имеет предложения after, то этот первый элемент планируется с дельта задержкой. К примеру, следующие назначения используют дельта задержку.

```
x <= '0' after 0ns;
y <= a and b after 0 us;
z <= "00101010";
w <= i,j after 100 ns; -- 1-й элемент с дельта задержкой
                       -- 2-й элемент - с инерционной
```

Рассмотрим поведение простого предложения параллельного назначения сигнала, моделирующего инвертор и использующего дельта задержку:

```
d_out <= not d_in;
```

Предположим, что d_in переключается из нуля в единицу в некоторый произвольный момент now. Назначение выполняется в результате события на d_in. Планируется, что новое значение появится на d_out с дельта задержкой. Немедленно после назначения формирователь для d_out выглядит следующим образом (рис. 18).

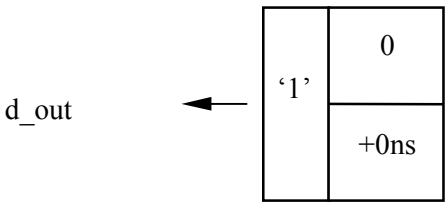

Рис. 18. Формирователь для d_out

Приращение времени выполняется только в случае, когда в модели еще запланированы события с задержкой, отличной от дельта задержки. На каждом реальном временном шаге все события с дельта задержкой обрабатываются первыми. Считается, что *последовательность циклов моделирования* (simulation cycles) может появиться в один момент времени моделирования. В этом случае формирователь для d_out будет получать свое новое значение на очередном шаге моделирования до приращения реального времени. На следующей временной диаграмме (рис.19) используется широкая
временная метка для того, чтобы графически проиллюстрировать то, что между входными и выходными изменениями сохраняется упорядоченная зависимость несмотря на то, что время моделирования не изменяется.

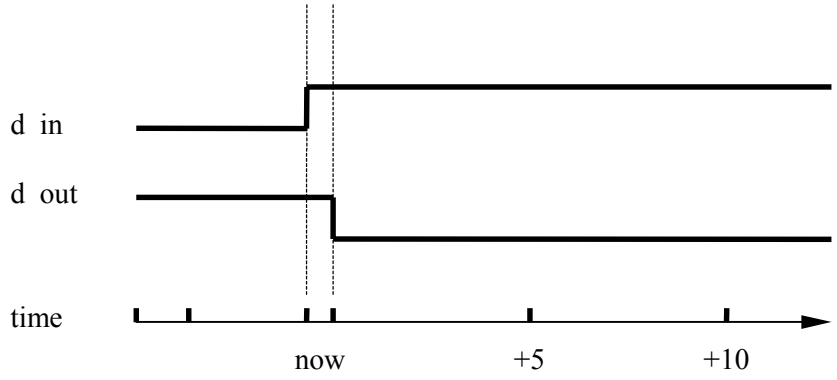

Рис. 19. Модель RS-триггера с дельта задержкой

3.4. Примеры моделирования в VHDL

Рассмотрим несколько практических примеров, поясняющих назначение рассмотренных понятий. Для формирования тестовых последовательностей широко используются модели разнообразных генераторов. Описание простого синхрогенератора приведено на рис. 20.

```
entity hilow_clock is
  generic(high_time,low_time:time:=0.5 sec);
```

```
  port(clk:out bit:='0'); --  clk вначале равен 0
end hilow_clock;
architecture sequential1 of hilow_clock is
 begin
  explicit_waits: process
  begin
     clk <= '1';  -- установить clk в '1'
     wait for high_time;
     clk <= '0';
     wait for low_time;
  end process;
 end sequential1;
```

Рис. 20. Генератор с управляемой длительностью логических нуля и единицы

Процесс explicit_waits устанавливает clk в '1', ожидает в течение времени, равного длительности единицы, затем устанавливает clk в '0', ожидает в течение времени, равного длительности '0', а затем повторяется (так как процесс является бесконечным циклом). В этих двух назначениях сигналов отсутствует ключевое слово after, так как назначения используют дельта задержку для установки значения clk. В данном случае каждым назначением создавалась одноэлементная временная диаграмма.

Многоэлементная временная диаграмма может быть использована в последовательном назначении сигнала для реализации синхрогенератора. Рассмотрим процесс period_wait, приведенный на рис. 21. В процессе планируется временная диаграмма одного периода сигнала. Предложение wait for ожидает до конца периода, в течение которого запланированные значения успевают появиться на clk.

```
architecture sequential2 of hilow_clock;
begin
 period_wait: process
 begin
   clk <= '1', '0' after high_time;
   wait for high_time+low_time;
 end process;
end sequential2;
```

Рис. 21. Альтернативная архитектура для hilow_clock

Когда процесс period_wait выполняется впервые (в начале моделирования), он планирует переключение в '1' на clk с дельта задержкой и в '0' через время, равное high_time. Затем процесс ждет в течение времени high_time+low_time (общий период) и повторяется снова.

Можно сформировать более сложную временную диаграмму для генерации сигналов специальной формы. Например, пусть необходимо описать синхросигнал с временной диаграммой, показанной на рис. 22. Назначение, выполняющееся в процессе complex_single (рис. 23), создает такую временную диаграмму, после чего процесс ожидает 150 нс, для того чтобы все элементы этой диаграммы последовательно появились в качестве значения сигнала clk.

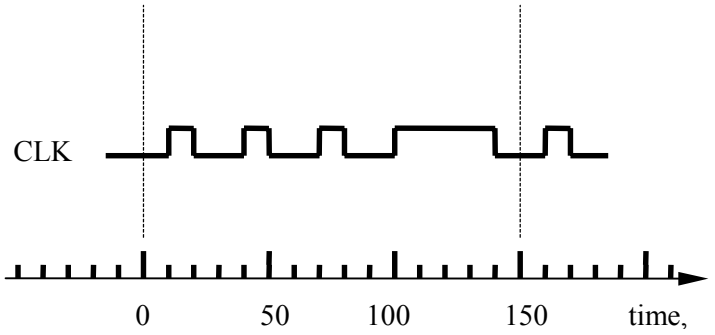

Рис. 22. Временная диаграмма сложного сигнала

```
complex_single: process
begin
   clk<= '1' after 10 ns, -- планирование отдельных
         '0' after 20 ns, -- переключений
         '1' after 40 ns,
         '0' after 50 ns,
         '1' after 70 ns,
         '0' after 80 ns,
         '1' after 100 ns,
```

```
         '0' after 140 ns,
   wait for 150 ns;       -- 150 нс - общий период
end process;
```

Рис. 23. Генератор синхросигнала со сложной временной диаграммой

Приведенные выше примеры моделировались с применением поведенческого стиля описания. Однако генератор можно описать и в потоковом стиле, как показано на рис. 24. Данный генератор вырабатывает четырехфазный синхросигнал, реализованный с помощью сигнала типа bit_vector длиной в 4 бита.

```
signal phases : bit_vector (1 то 4):="0000";
    ...
with phases select
   phases<="1000" after 100 ns when "0000",
           "1000" after 25 ns when "0001",
           "0100" after 20 ns when "1000",
           "0010" after 25 ns when "0100",
           "0001" after 30 ns when "0010",
           "0000" when others;
```

Рис. 24. Четырехфазный генератор

Мы используем в блоке предложение назначения с выбором сигнала для инициализации phases и корректировки его значения. Сначала phases будет равен "0000". После выполнения назначения phases установится в "1000" с задержкой относительно начала, равной 100 нс. Затем при каждом изменении phases назначение будет выполняться вновь и вычислять новое значение. Все возможные значения должны быть учтены, и поэтому было использовано ключевое слово others для реакции на другие (недопустимые) состояния (в случае, когда имеются другие формирователи этого сигнала, которые могут привести к недопустимому состоянию). В этом случае все биты phases устанавливаются в "0" с дельта задержкой и генератор возвращается в начальное состояние.

3.5. Контроль ошибок в VHDL

В VHDL предусмотрен оператор для отслеживания ошибок, возникающих при функционировании моделируемого устройства. Это о*ператор утверждения* (assertion_statement), который проверяет истинность заданного условия и формирует сообщение об ошибке, если это условие ложно. Например:

```
assert not(s='1' and r='1')
report "input error: s,r=1"
severity warning;
```

Сигналы s и r не должны одновременно принимать значение '1'. Ключевые слова report и severity входят в синтаксис оператора assert. Описание сообщения ("input error: s,r=1") содержит выражение предопределенного типа STRING, задающее текст сообщения. Описание серьезности содержит выражение предопределенного перечислимого типа SEVERITY_LEVEL, задающее уровень серьезности утверждения. Возможные значения типа SEVERITY_LEVEL следующие: NOTE, WARNING, ERROR, FAILURE.

При отсутствии описания сообщения в конкретном утверждении неявным значением строки сообщения является строка "Assertion violation". При отсутствии описания серьезности в конкретном утверждении неявным значением уровня серьезности является ERROR.

Вычисление оператора утверждения состоит из вычисления логического булева выражения, задающего условие. Если это выражение вырабатывает значение FALSE, то имеет место *нарушение утверждения*. Если это происходит, то заданная строка сообщения и уровень серьезности (или соответствующие неявные значения) используются для формирования сообщения об ошибке.

Сообщение об ошибке состоит по крайней мере из:

1) указания, что данное сообщение сгенерировано утверждением;

2) значения уровня серьезности;

3) значения строки сообщения;

4) имени модуля проекта, содержащего это утверждение.

Операторы контроля могут включаться в описание интерфейса или в архитектурное тело объекта проектирования. Если операторы контроля помещены в архитектурное тело, они будут распространяться только на конкретную реализацию объекта проектирования. Если же их поместить в описание интерфейса, их можно будет использовать для проверки временных параметров входных и выходных сигналов любого архитектурного тела объекта проектирования.

Приведем пример контроля временных параметров:

```
assert STRB'STABLE(W)
report "Minimum pulse width failure";
-- "ошибка минимальной длительности импульса"
```

4. ПОДМНОЖЕСТВА ЯЗЫКА VHDL

4.1. Стандартные подмножества языка VHDL

В настоящий момент существуют различные системы моделирования аппаратного обеспечения, реализующие такие подмножества VHDL, которые соответствуют существующим у них средам проектирования. Если в каждой подобной системе внедрить свое собственное подмножество языка, то это разрушит главную цель VHDL : достижение переносимости и взаимодействия проектов.

С этой целью специальной группой VDEG (VHDL Design Exchange Group) для задач моделирования были выделены три стандартных уровня языка и для них были выбраны соответствующие подмножества VHDL-конструкций /2/. Эти подмножества приведены ниже.

S0 - подмножество конструкций VHDL, необходимых при использовании VHDL в среде отдельной системы моделирования; это подмножество определяет минимальный набор конструкций VHDL: объявление объекта (entity_declaration), тело архитектуры (architecture_body), объявление подпрограммы (subprogram_declaration), список портов (port_list) и др., требующихся для выполнения модели проекта; это подмножество не гарантирует взаимодействия (совместного функционирования) моделей, т.к. не допустимо использование пакетов, объявления новых типов, параметров настройки и пр.

S1 - подмножество конструкций VHDL, дополненное возможностями создания взаимодействующих моделей; в это подмножество добавлены такие конструкции, как объявление пакета (package_declaration), тело пакета (package_body), объявление типа (type_declaration), описание настройки (generic_clause) и др.

S2 - полная реализация VHDL, куда добавлены такие конструкции, как объявление дополнительного имени (alias_declaration), объявление атрибутов (attribute_declaration), спецификация атрибута (attribute_specification), объявление константы (constant_declaration), объявление подтипа (subtype_declaration), параллельный вызов процедуры (concurrent_procedure_call), параллельный оператор утверждения (concurrent_assertion_statement) и др.

Кроме того, VHDL поддерживает три различных стиля для описания аппаратных архитектур: - структурное, потоковое и поведенческое. Поэтому, на основании представленной классификации, можно представить 9 (3x3) различных подмножеств языка VHDL, необходимых для решения соответствующих задач.

4.2. Реализации подмножеств VHDL в учебно-промышленных САПР БИС.

Учебное программное обеспечение характеризуется, как правило, ограниченным набором свойств по сравнению с коммерческими промышленными программами. Рассматриваемые ниже программные средства, разработанные для учебных целей, обладают некоторыми ограничениями, тем не менее, они вполне могут применяться как в учебном процессе, так и при разработке реальных проектов.

4.2.1. Подмножество VHDL системы синтаксического и семантического контроля VHDL-описаний “VHDL-анализатор”

Система синтаксического и семантического контроля VHDL-описаний “VHDL-анализатор” (VA) поддерживает стандартное подмножество VHDL S2 /1/. Это позволяет проверять практически любые конструкции языка. Однако данный пакет имеет серьезное ограничение, сужающее его практическое применение: объем анализируемого модуля проекта не должен превышать 64 Кбайт. Синтаксис описаний проверяется полностью. При семантическом анализе выявляются только возможные статические ошибки описания модуля проекта.

Согласно требованиям стандарта (IEEE Standard 1076 VHDL) система VA позволяет создать и использовать любое число библиотек ресурсов. Для ассоциирования логического имени библиотеки с соответствующим ей фактическим именем в VA предусмотрен специальный механизм установки внешних ссылок. Наличие развитых средств работы с библиотеками, а также поддержка стандартного подмножества VHDL позволяют эффективно использовать данный программный продукт.

4.2.2. Подмножество VHDL САПР СБИС Alliance

Учебно-промышленная САПР Alliance предназначена для разработки цифровых СБИС. Версия САПР 3.2b (1997-1998 гг.) поддерживает ограниченное подмножество VHDL, которое можно отнести к стандартному подмножеству S0. Несмотря на это, Alliance широко используется как для обучения основам проектирования СБИС, так и для разработки коммерческих проектов /3/.

Рассмотрим особенности подмножества VHDL этой САПР.

Из трех существующих стилей описания Alliance поддерживает структурное и потоковое представления. Для обеспечения автоматической трансляции из структурного стиля VHDL в другие структурные форматы (EDIF, COMPASS, VERILOG, ALLIANCE, и др.) исключена возможность объединять в описании одного объекта проекта оба стиля.

Типичная VHDL модель проекта, предлагаемая в Alliance, состоит из иерархии структурных описаний компонент и потоковых описаний простейших логических элементов нулевого и первого уровня.

САПР поддерживает несколько предопределенных типов данных, достаточных для описания проектов различного уровня сложности. Однако пользователь лишен возможности создавать свои альтернативные типы данных. Рассмотрим предлагаемый набор предопределенных типов.

Тип *bit* - предопределенный перечислимый тип, который может принимать значения '0', '1' и 'D'. Дополнительное значение 'D' (don't care) введено для обозначения неопределенного состояния. Когда какой-либо сигнал имеет значение 'D', утилита логического синтеза может установить его в '0' или '1'. Это единственное расширение языка, допущенное в САПР Alliance, которое не соответствует стандарту IEEE Standard 1076 VHDL., то есть не является частью любого стандартного подмножества VHDL (S0, S1, S2).

Тип *bit_vector* - предопределенный массивный тип, элементами которого являются объекты типа bit.

Тип *mux_bit* - предопределенный подтип типа bit, связанный с функцией разрешения *mux*. Эта функция следит, чтобы только один драйвер был подключен к сигналу. Эффективным значением сигнала является значение активного драйвера. Если все драйверы отключены, значение сигнала устанавливается в '1'. Сигнал типа mux_bit должен быть объявлен как шина (bus), например:

```
signal data: mux_bit bus;
```

Тип *mux_vector* - предопределенный массивный тип, элементами которого являются объекты типа mux_bit.

Тип *wor_bit* - предопределенный подтип типа bit, связанный с функцией разрешения *wor*. Эта функция допускает, чтобы к одному сигналу были подключены несколько драйверов. Активные драйверы должны иметь одинаковое значение. Эффективным значением сигнала является значение активных драйверов. Если все драйверы отключены, значение сигнала устанавливается в '1'. Сигнал типа wor_bit должен быть объявлен как шина (bus).

Тип *wor_vector* - предопределенный массивный тип, элементами которого являются объекты типа wor_bit.

Тип *reg_bit* - предопределенный подтип типа bit, связанный с функцией разрешения *reg*. Эта функция следит, чтобы только один драйвер был подключен к сигналу. Эффективным значением сигнала является значение активного драйвера. Если все драйверы отключены, значение сигнала устанавливается в '1'. Сигнал типа reg_bit должен быть объявлен как регистр (register), например:

```
signal data: reg_bit register;
```

Тип *reg_vector* - предопределенный массивный тип, элементами которого являются объекты типа reg_bit.

Поскольку возможность создавать пользовательские типы данных отсутствует, среди массивных типов доступны предопределенные bit_vector, reg_vector, mux_vector и wor_vector.

Для всех предопределенных типов поддерживаются следующие операции:

```
not, and, or, xor, nor, nand, &, =, /=
```

Другие стандартные операторы VHDL (+, -, >, <, и др.) в текущей версии САПР пока не поддерживаются.

Функции разрешения для контроля над сигналом и его драйверами используют оператор утверждения (assertion_statement), о котором шла речь в разделе "Контроль ошибок в VHDL". В Alliance определены только два значения уровня серьезности (severity levels) в данном операторе:

1) warning - при этом выводится предупреждающее сообщение, если утверждаемое условие не выполнено;
2) error - при этом значении выводится сообщение об ошибке, если утверждаемое условие не выполнено, и моделирование останавливается.

Рассмотрим теперь более детально потоковое и структурное подмножества VHDL, поддерживаемые Alliance.

4.2.2.1. Потоковое подмножество

Потоковое подмножество VHDL САПР Alliance носит название *vbe*. Такое же расширение по умолчанию имеют файлы с описанием в потоковом стиле.

В описании архитектуры допускается использовать следующие параллельные предложения (оператор process не поддерживается):

1) простое назначение сигнала (simple_signal_assignment), например:

```
command <= '1';
```

2) условное назначение сигнала (conditional_signal_assignment), например:

```
Name_34 <= guarded
                    null      when     des=hn
             else   a         when     j=mn
             else   l         when     s=mn
             else   rith ;
```

3) выборочное назначение сигнала (selected_signal_assignment), например:

```
with  command select
      name_Numb_1 <= guarded  null when '1',
                              m    when '0';
```

4) параллельный оператор утверждения (concurrent_assertion_statement);

5) оператор блока (block_statement).

Необходимость использовать только параллельные назначения сигналов затрудняет создание моделей *шин*, управляемых несколькими драйверами. Это вызвано тем, что каждый драйвер может быть назначен сигналу лишь один раз. Поэтому для определения эффективного значения сигнала используют уже обсуждавшиеся выше функции разрешения.

Управляемые сигналы (guarded_signals) являются разрешенными сигналами, они связаны с некоторой функцией разрешения, которая позволяет драйверам быть отключенными. Управляемый сигнал должен быть назначен внутри оператора блока (block_statement) с помощью управляемого назначения (guarded_signal_assignment). В качестве примера приведем описание распределенного мультиплексора:

```
signal Distributed_Mux : mux_bit bus;
begin
first_driver_of_mux : block (Sel1 = '1')
begin
      Distributed_Mux <= guarded Data1;
end block;
second_driver_of_mux : block (Sel2 = '1')
begin
      Distributed_Mux <= guarded Data2;
end block;
```

Последовательностные элементы, такие как *триггеры* и *регистры*, должны быть явно объявлены с предопределенными типами reg_bit и reg_vector в качестве регистров (register). Назначения этим элементам должны выполняться внутри оператора блока (block_statement) с помощью управляемого назначения (guarded_signal_assignment):

```
-- Falling edge triggered D flip flop :
signal Reg : reg_bit register;
begin
flip_flop : block (ck = '0' and not ck'STABLE)
begin
      Reg <= guarded Din;
end block;

-- Level sensitive latch:
signal Lat : reg_bit register;
begin
latch : block (ck = '1')
begin
      Lat <= guarded Din;
end block;
```

В обоих случаях, рассмотренных в последнем примере, управляющее выражение должно зависеть только от одного сигнала, если описание будет использоваться в качестве исходного задания для утилит логического синтеза Alliance.

В САПР Alliance потоковое описание задействуется при достижении следующих целей:

1) моделирование и проверка правильности ввода описания (спецификации) устройства;

2) синтез аппаратных структур на основе исходного описания.

На этих этапах в предлагаемом маршруте проектирования не используется информация о реальных временных интервалах, поэтому потоковое подмножество VHDL САПР не поддерживает временные спецификации, в частности, выражение *after* не обрабатывается, а моделирование выполняется с использованием дельта-задержки.

Ниже приведен пример сумматора с аккумулятором, выполненном на основе регистра. Описание выполнено в потоковом стиле.

```
entity add_accu is
       port (
            clk      : in  bit;
            command  : in  bit;
            data_in  : in  bit_vector (31 downto 0);
            data_out : out bit_vector (31 downto 0); cry_out  : out bit;
            vdd      : in  bit;
            vss      : in  bit );
end add_accu;

architecture data_flow of add_accu is

signal eff_data  : bit_vector (31 downto 0);
-- effective operand
signal adder_out : bit_vector (31 downto 0);
-- adder's result
```

```
signal adder_cry : bit_vector (32 downto 0);
-- adder's carry
signal accum_reg : reg_vector (31 downto 0) register;
-- accumulator
constant initialize : bit := '0';
constant accumulate : bit := '1';
begin
-- select the effective operand
      with command select
      eff_data <= X"0000_0000" when initialize,
                  accum_reg    when accumulate;
-- compute the result out of the adder
      adder_out <= eff_data xor data_in xor adder_cry;
      adder_cry (0)  <= '0';
      adder_cry (32 downto 1) <= (eff_data and adder_cry (31 downto 0)) or
                                 (data_in  and adder_cry (31 downto 0)) or
                                 (eff_data and data_in);
-- write the result into the register on the falling
-- edge of clk
      write : block (clk = '0' and not clk'STABLE)
      begin
            accum_reg <= guarded adder_out;
      end block;
-- assign outputs
      cry_out  <= adder_cry (32);
      data_out <= accum_reg     ;
-- check power supply
assert (vdd = '1' and vss = '0')
report "power sypply is missing"
severity ERROR;
end;
```

4.2.2.2. Структурное подмножество

Структурное описание - это набор предложений конкретизации компонентов. Порты отдельных экземпляров соединяются между собой через сигналы с помощью карты портов (port map). В фактической части спецификации карты портов из операторов допускается использовать только оператор конкатенации (&).

В описываемой версии компилятора VHDL не допускается оставлять неподсоединенные порты.

В соответствии со стандартом VDHL, декларативная часть (область объявлений) структурного описания в Alliance включает объявления сигналов и компонент.

Допускается объявлять сигналы всех предопределенных типов, кроме reg_bit и reg_vector, которые не имеют смысла в контексте структурного стиля.

Компоненты должны объявляться с такими же описаниями портов, какие были в их описаниях объектов (entity), то есть должны совпадать имена портов, порядок их перечисления, их виды (in, out) и типы.

Описание в структурном стиле сумматора с аккумулятором, выполненном на основе регистра, приведено в следующем примере.

```
entity add_accu is
port (
      clk         : in  bit;
      command     : in  bit;
      data_in     : in  bit_vector (31 downto 0);
      data_out    : out bit_vector (31 downto 0);
      cry_out     : out bit;
      vdd         : in  bit;
      vss         : in  bit
      );
end add_accu;

architecture structural of add_accu is
signal eff_data  : bit_vector (31 downto 0);
-- effective operande
signal adder_out : bit_vector (31 downto 0);
-- adder's result
signal accu_out  : bit_vector (31 downto 0);
-- accumulator
```

```
component adder
port (a   : in  bit_vector (31 downto 0);
      b   : in  bit_vector (31 downto 0);
      res : out bit_vector (31 downto 0));
end component;

component and_32
port (a   : in  bit_vector (31 downto 0);
      cmd : in  bit;
      res : out bit_vector (31 downto 0));
end component;

component falling_edge_reg
port (din  : in  bit_vector (31 downto 0);
      clk  : in  bit;
      dout : out bit_vector (31 downto 0));
end component;

begin

my_adder : adder
      port map (a => eff_data, b => accu_out, res => adder_out);

my_mux : and_32
      port map (cmd => command, a => accu_out, res => eff_data);

my_reg : falling_edge_reg
      port map (din => adder_out, clk => clk, dout => accu_out);

end;
```

5. РЕШЕНИЕ ЗАДАЧ ЭТАПОВ РАЗРАБОТКИ БИС С ПРИМЕНЕНИЕМ VHDL

5.1. Проектирование

Разработку интегральных схем можно выполнять, опираясь на различные уровни абстрактного представления проекта. Наиболее высокий уровень абстракции можно представить алгоритмическими выражениями, описывающими логику работы устройства.

Язык VHDL представляет для этих целей развитые средства поведенческого стиля описания. Кроме того, стандарт IEEE Standard 1076 VHDL позволяет компиляторам, реализующим VHDL, допускать встраивание описаний различных подпрограмм на языках программирования высокого уровня (Си, Паскаль и др.) в качестве реализации функций и процедур. В этом стиле удобно разрабатывать описания устройств высокого уровня сложности (ЭВМ, процессоры, контроллеры различных устройств, арбитры шин, сложные последовательностные схемы и т.д.).

Потоковый стиль позволяет описывать более простые устройства, такие как сумматоры, мультиплексоры и т.д.

Описания этих двух стилей, как правило, служат исходным заданием для программ логического синтеза. Программы генерируют на основе входных данных структурное представление проекта. Таким образом, подход, основанный на разработке поведенческого и потокового описания, позволяет уделять больше внимания непосредственно фунции устройства, не останавливаясь на его структуре.

Другой подход основан на непосредственной разработке архитектуры ВС. Подмножество структурного стиля VHDL обеспечивает разработчика средствами описания архитектурной иерархии проекта. Потоковые или поведенческие описания компонент могут вводиться на любых иерархических уровнях. Это позволяет пользоваться библиотечными наборами готовых логических и архитектурных решений. Преимущества данного подхода заключаются в возможности тщательной "ручной" оптимизации структуры ВС.

Компромиссным и наиболее оптимальным представляется подход, заключающийся в ручной "доводке" синтезированных проектов. Следует отметить, что непрерывное совершенствование качества выходного кода компиляторов VHDL, происходящее в настоящее время, позволит получать достаточно качественные крупные проекты при приемлемых затратах времени.

Немаловажное значение на качество проектирования влияет оптимизация исходного поведенческого или потокового описания. Так, например, введение неоправданно большого числа дополнительных (промежуточных) сигналов и переменных способно привести к совершенно неработоспособному проекту.

5.2. Верификация

При проектировании сложных специализированных СБИС верификация играет настолько важную роль, что большинство маршрутов проектирования содержат операции проверки практически после каждой проектной процедуры.

Любой стиль описания в VHDL позволяет выполнять моделирование ВС. Оценка значений сигналов, а также выполнение операторов утверждения (assertion_statement) позволяют судить о наличии ошибок в проекте и работоспособности устройства. Приведем перечень проверок, которые можно выполнить, используя VHDL описания ВС:

1) при моделировании исходного описания (любого стиля) в первую очередь выявляются ошибки ввода;

2) при непосредственной "ручной" разработке структурного вида моделирование позволяет выявить ошибочные и неразведенные связи, ошибки структурной схемы, допущенные разработчиком и т.п.;

3) при моделировании исходного высокоуровневого поведенческого или потокового описания ВС проверяется на соответствие заданной функции;

4) системы, генерирующие структурное представление схемы на основе ее поведенческого (или потокового) описания, проверяют соответствие выполняемой функции синтезированного устройства функции исходного описания;

5) в некоторых САПР проводится сравнение между структурной схемой, полученной экстракцией из реальной топологии, со структурным описанием, на основе которого была получена топология.

Данные проверки выполняются в САПР специализированных БИС Alliance, при необходимости перечень может быть расширен.

5.3. Сопровождение и модификация проектов

В настоящее время VHDL используется для работы с ВС всех уровней сложности. Различные этапы проекта могут выполняться отдельными группами разработчиков. Поведенческое и потоковое описания устройства могут служить коммерческим результатом проекта. Эти данные после передачи заказчику могут быть использованы для дальнейшей проработки и структурной конкретизации проекта.

Стандартизация подмножеств VHDL обеспечивает беспрепятственный обмен проектными данными и моделями на всех этапах разработки. Общий набор предопределенных элементов поддерживается благодаря соглашению об общих стандартных пакетах: STANDARD и TEXTIO.

Пакет STANDARD предопределяет ряд типов, подтипов и функций. Предполагается, что в начале каждого модуля проекта присутствует неявное описание контекста, ссылающееся на этот пакет. Пакет TEXTIO содержит объявления типов и подпрограмм, обеспечивающих форматированные операции ввода-вывода с символами ASCII. Оба пакета не могут быть изменены пользователем.

Необходимость модификации проектов может возникнуть как на этапе проектирования (в результате выявления ошибок разработчика и ошибок САПР), так и в случае изменения требований технического задания.

Наиболее просто вносить модификации в разработку при использовании нисходящего проектирования. В этом случае достаточно изменить соответствующие исходные параметры в поведенческом или потоковом описании. Окончательный проект создается, как правило, в автоматическом режиме.

Более сложно модифицировать проект при использовании восходящей методологии. Однако строгая модульная организация иерархии проекта, предлагаемая структурным стилем, облегчит выполнение этой задачи.

ЗАКЛЮЧЕНИЕ

На сегодняшний день специализированные СБИС составляют основу элементной базы большей части радиоэлектронных и электронно-вычислительных устройств. В связи с этим особую важность приобретает развитие средств разработки новых устройств и, в частности, совершенных языков описания аппаратуры. В пособии изложены основные понятия универсального ЯОА VHDL, получившего наиболее широкое распространение при проектировании СБИС и, особенно, специализированных схем.

Заложенные в язык развитые описательные возможности обеспечивают эффективное применение VHDL для проектирования, верификации, модификации и сопровождения проектов. Приведенный материал предназначен для начального знакомства с новым языком и содержит разделы, необходимые при освоении заложенных в VHDL средств, используемых в практике проектирования ВС.

Рассмотрены основные лексические элементы языка, принципы организации объектов данных и работа с ними, формирование иерархии описания проектов и подходы к многоуровневому проектированию ВС. Приведенные примеры иллюстрируют теоретический материал и дополняют его благодаря комментариям и наглядности.

Описаны подмножества VHDL учебно-промышленных доступных и популярных программных пакетов. Показаны пути преодоления ограничений, присущих подобным продуктам.

Применение языка VHDL на различных этапах разработки СБИС значительно повышает эффективность проектирования. В пособии описываются подходы в использовании VHDL в новых маршрутах проектирования интегральных схем.

ЛИТЕРАТУРА

1. Система синтаксического и семантического контроля VHDL-описаний “VHDL-анализатор”. Руководство пользователя. М.: РосНИИИС, 1991. 112 с.

2. Резидентный справочник по языку VHDL. Руководство пользователя. М.: РосНИИИС, 1993. 21 с.

3. Alliance: A Complete CAD System for VLSI Design. Users manual. Paris: Universite Pierre et Marie Curie, 1997.

СОДЕРЖАНИЕ