ФЕДЕАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВНИЮ

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ С ПОМОЩЬЮ ЯЗЫКА ОПИСАНИЯ АППАРАТУРЫ VHDL

Учебное пособие для вузов

Составители:
А.М. Бобрешов, А.В. Дыбой

ВОРОНЕЖ
2007

Утверждено Научно-методическим советом физического факультета 6 марта 2007г., протокол №3

Рецензент ассистент кафедры радиофизики к.ф.-м.н. Е.В. Литвинов

Учебное пособие подготовлено на кафедре электроники физического факультета Воронежского государственного университета.

Рекомендовано для студентов физического факультета Воронежского государственного университета всех форм обучения.

Для специальности: (010801) – Радиофизика и электроника

Настоящее учебное пособие предназначено для студентов, обучающихся на физическом факультете по специальности 013800 “Радиофизика и электроника”.

Языки описания цифровой аппаратуры, в том числе VHDL, являются мощным средством для ускоренной разработки и верификации сложных цифровых схем и проектов. Постепенно вытесняя графические средства описания схем (так называемые схемотехнические редакторы), языки описания аппаратуры на порядок сокращают время разработки, делают процесс проектирования более “человеко-ориентированным” и похожим на процесс написания прикладных программ на языках высокого уровня, таких, как Паскаль. Одним из наиболее полезных свойств VHDL является возможность применения так называемого “функционального” описания, когда разработчик не конкретизирует структуру схемы, а лишь описывает ее поведение, то есть то, как она должна работать. Составление же самой электрической схемы возлагается на программу-компилятор, которая сегодня легко справляется с этой задачей. На сегодняшний день созрела и аппаратная база для применения VHDL. Современные доступные программируемые логические интегральные схемы (ПЛИС) имеют емкость в сотни тысяч эквивалентных логических вентилей, что может быть достаточно даже для реализации не очень сложного микропроцессора. Проектирование схем такой сложности с помощью традиционных схемотехнических редакторов требует больших сроков и чаще всего выполняется коллективами разработчиков. В то же время, применение современных средств проектирования, в первую очередь ориентированных на работу с языками описания аппаратуры, позволяет даже отдельным специалистам решать такого рода задачи в сжатые сроки.

Поскольку в настоящее время доступны книги другие исчерпывающие источники информации по VHDL, данное пособие призвано дать лишь необходимый для начала самостоятельной работы объем знаний.

Содержание

Введение 5
Язык VHDL и примеры его использования 7
Алфавит моделирования 13
Типы данных и декларации объектов 14
  Предопределенные типы данных 16
  Типы, вводимые пользователем 18
Сигналы и переменные. Оператор PROCESS 22
Атрибуты в языке VHDL 26
Последовательные операторы 29
  Операторы присваивания 29
  Оператор условия и оператор выбора (if и case) 30
  Оператор ожидания 31
  Операторы повторения 32
  Оператор проверки 34
Параллельные операторы 35
  Оператор процесса 35
  Параллельное присваивание 35
  Оператор блока 37
Подпрограммы 39
Разрешаемые сигналы и шины 42
Компоненты 45
Примеры описания устройств 47
  Передатчик RS-232 48
Литература: 51

Введение

Развитие современной элементной базы тесно связано с развитием методик проектирования сложных цифровых устройств. Разработка современных устройств невозможна без применения систем автоматизированного проектирования (САПР), которые выполняют значительную часть рутинной работы. Современные САПР обладают достаточным “интеллектом” для того, чтобы вместе с человеком участвовать в разработке схемы. Однако реализация открывающихся возможностей требует использования новых, современных методов проектирования.

На сегодняшний день наиболее эффективными считаются методы, использующие текстовые описания проектов вместо традиционных графических редакторов схем. В этом случае проект представляет собой текст, сохраняемый в файле или нескольких файлах, которые в совокупности определяют представления разработчика об устройстве и используются на всех этапах проектирования, в том числе при моделировании устройства и синтезе его принципиальной схемы. Проектирование цифровых схем с использованием языков описания аппаратуры позволяет полностью отказаться от графических редакторов принципиальных схем, при этом повысить скорость работы, надежность и удобство отладки. Тексты описания в большинстве языков проектирования дискретных устройств по составу синтаксических конструкций очень схожи с традиционными языками программирования. Поэтому часто такое текстовое описание называют программой на языке проектирования, или, коротко, HDL-программой, а конструкции, описывающие формирование принципиальной электрической схемы, называют операторами.

Другими словами, HDL-программа – это знаковая модель дискретного устройства. Такая модель может быть описана с различной степенью конкретизации. Давая возможность достаточно детально описывать структуру проекта, часто, вплоть до отдельных вентилей, языковое описание также позволяет описывать устройство на уровне алгоритмов его работы, возлагая при этом синтез самой структуры на систему автоматизированного проектирования (САПР). В этом заключается мощное превосходство языков описания аппаратуры перед графическими редакторами схем, где проектировщик вынужден полностью рисовать структуру устройства с максимальной степенью детализации.

Следует заметить, что синтез цифровых электрических схем – лишь одно из направлений использования языков описания аппаратуры. В процессе разработки сложных цифровых устройств огромное значение придается компьютерному моделированию. Этап временного моделирования схем так же важен, как этап отладки прикладных

программ. При этом часто возникает потребность спрогнозировать работу проектируемого устройства совместно с другими устройствами, модель которых может быть задана в виде HDL-программ. Кроме того, даже если современная система проектирования рассчитана на традиционный, схемотехнический подход (например, ORCAD), модели библиотечных цифровых компонентов чаще всего описываются с помощью HDL-языков. В силу сказанного многие языки описания аппаратуры включают конструкции, которые не могут быть синтезированы в виде схемы, а используются только в процессе моделирования. Например, это касается работы с файлами, вывода различных сообщений на экран. Большинство операций, имеющих сложную аппаратную реализацию, тоже не могут быть синтезированы, например, операции с плавающей точкой и некоторые другие. То есть нужно представлять, что HDL-языки имеют так называемое синтезируемое подмножество языка, которое включает те конструкции, которые могут быть автоматически преобразованы в электрическую схему.

Существует два наиболее распространенных универсальных языка описания аппаратуры – VHDL, который по своим синтаксическим конструкциям напоминает Паскаль, и Verilog, имеющий некоторое сходство с языком Си. Это сходство весьма условно, поскольку и VHDL, и Verilog являются языками структурного описания электрических схем, а не языками программирования. Остановимся на изучении языка VHDL, поскольку он имеет большее распространение на территории России.

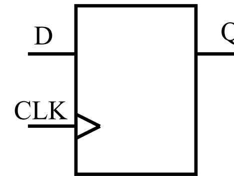
Язык VHDL и примеры его использования

Язык описания цифровой аппаратуры VHDL (Very high speed integrated circuits Hardware Description Language) был разработан по заказу Министерства обороны США в начале 80-х годов. Язык был стандартизован, после чего упомянутое Министерство обязало своих поставщиков цифровых микросхем представлять в составе документации их описание на VHDL. С самого начала VHDL выполнял роль стандарта и поэтому получил огромное распространение в США и Европе. Это позволило во многом решить актуальный на то время вопрос о моделировании проектов цифровых устройств, содержащих схемы от разных производителей. Таким образом, первоначально языки описания аппаратуры использовались только для моделирования. В начале девяностых годов начали разрабатываться прямые компиляторы VHDL-программ в аппаратные реализации различных классов. Создание таких компиляторов представляло собой весьма сложную задачу, поскольку не существует прямой и однозначной взаимосвязи между текстовым, преимущественно поведенческим описанием устройства, и его схемотехническим, структурным представлением. На создание эффективно работающих компиляторов потребовалось почти 10 лет. Язык за время своего существования претерпел несколько изменений, наиболее актуальным на сегодняшний день является стандарт VHDL’93, а также новый стандарт, принятый в 1999г., называемый VHDL-AMS. Наиболее существенным нововведением VHDL-AMS является появление конструкций для описания аналоговых и смешанных аналого-цифровых устройств. За основу изучения возьмем VHDL’93, поскольку именно этот вариант поддерживается сегодня большинством САПР.

Поскольку перечисление синтаксических конструкций не способно дать правильное представление об отличиях VHDL от традиционных языков программирования высокого уровня, целесообразно начать его изучение с простых примеров. Данные примеры содержат описания некоторых простейших устройств цифровой схемотехники. Поскольку в VHDL не делается различий между прописными и строчными буквами, для удобства чтения примеров ключевые слова будем записывать строчными символами, а определяемые имена – прописными.

Пример 1. Синхронный D-триггер, тактируемый передним фронтом

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DFF is
  port (DATA, CLK: in std_logic;
  Q: out std_logic);
end DFF;

architecture DFF_ARCH of DFF is
begin
process (CLK)
 begin
  if CLK'event and CLK='1' then
    Q<=DATA;
  end if;
 end process;
end DFF_ARCH;
```



Пример начинается с подключения пакета, называемого ***std_logic_1164*** и содержащего стандартные расширения языка VHDL. Пакет представляет собой набор объявлений, схемотехнических компонентов и подпрограмм, которые могут быть использованы в разных проектах. В частности, в пакете ***std_logic_1164*** размещено объявление типов ***std_logic*** и ***std_logic_vector***, которые, как будет показано ниже, являются основными типами определения сигналов и шин. Определения пакетов содержатся в библиотеках. Соответственно, сначала требуется подключить библиотеку с помощью конструкции ***library <имя библиотеки>***. Затем с помощью ключевого слова ***use*** можно подключить необходимый пакет из библиотеки. Поскольку на практике пакеты могут иметь много объявлений, в том числе вложенных пакетов, можно явно указать, какие из объявлений необходимо включать в проект. Для того чтобы подключить все доступные объявления на указанном иерархическом уровне, используется ключевое слово ***all***. Например, запись ***use IEEE.std_logic_1164.all*** означает, что необходимо подключить все объявления пакета ***std_logic_1164***, входящего в библиотеку ***IEEE***. По возможности следует избегать подключения лишних объявлений. Например, конструкция ***use IEEE.all*** приведет к подключению всего содержимого библиотеки IEEE. С одной стороны, это приведет к увеличению времени обработки проекта. С другой, поскольку разные пакеты могут содержать объявления с одинаковыми именами, это может привести к конфликтным ситуациям, что потребует в явном виде указывать имена пакетов при использовании таких деклараций.

После объявления используемых пакетов идет раздел, называемый *интерфейсом объекта*. Объект проекта может представлять всю проектируемую систему, некоторую подсистему, устройство, узел, стойку, плату, кристалл, макроячейку, логический элемент и т.п. Раздел начинается с ключевого слова ***entity***, после которого следует произвольное название, или имя объекта. В этом разделе объект описывается в виде “черного ящика”, то есть определяются сигналы, которыми он обменивается с окружающими объектами, их тип и направление. В нашем примере схемотехнический объект имеет 2 входных сигнала ***D*** и ***CLK*** и один выходной сигнал ***Q***. Сигналы, определяемые в разделе интерфейса, называются *портами*. Входные порты помечаются ключевым словом ***in***, выходные – ***out***. Существуют двунаправленные порты, которые доступны как для управления, так и для опроса. Такие порты помечаются ключевым словом ***inout***. Заканчивается раздел ключевым словом ***end***, за которым следует имя объекта.

Следующий раздел начинается с ключевого слова ***architecture*** и называется архитектурным телом, которое содержит функциональное описание объекта. Каждый объект должен иметь как минимум одну архитектуру. Архитектура должна иметь уникальное имя (в рассматриваемом примере это DFF_ARCH). Собственно функционирование объекта описывается с помощью операторов, расположенных между ключевыми словами ***begin*** и ***end <имя архитектуры>***. Между ключевыми словами ***architecture*** и ***begin*** располагается раздел деклараций архитектуры, в котором определяются константы, пользовательские типы, сигналы, описываются компоненты используемых объектов. В нашем простейшем примере объявления новых типов, сущностей и объектов не используется (для более полного знакомства со структурой проектов VHDL рекомендуется рассмотреть пример в конце пособия). Собственно архитектурное тело содержит набор операторов, которые могут быть простыми или составными. Типичным составным оператором является оператор *процесса*. В нашем случае архитектура содержит всего один процесс, который начинается с ключевого слова ***process*** и заканчивается ***end process***. Тело любого процесса выполняется каждый раз, когда происходит любое изменение сигналов, перечисленных в его списке чувствительности (в скобках после ключевого слова ***process***). В теле процесса рассматриваемого примера происходит проверка условия и присваивание нового значения сигналу Q в зависимости от его выполнения:

```
if CLK’event and CLK=’1’ then
   Q<=DATA;
end if;
```

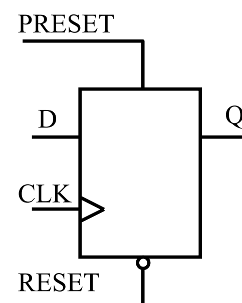
Каждый сигнал имеет набор так называемых атрибутов, которые позволяют получить доступ к их дополнительным свойствам. Другими словами, сигналы, в отличие от переменных в языках программирования, имеют не только текущее значение, но и свойство изменяться во времени, что дает возможность обращаться к их предшествующим значениям. Атрибут ***‘event*** является самым “используемым”. Он возвращает значение логической истины только в момент изменения сигнала. Таким образом, если в списке чувствительности имеется несколько разных сигналов, применение ***‘event*** позволяет определить, с изменением какого из них связано выполнение текущей итерации процесса. Его применение также необходимо при описании синхронных схем. В этих случаях атрибут указывает на тот сигнал, который является задающим (тактовым) сигналом.

Из сказанного выше легко понять, как работает описанный в примере объект. По переднему фронту сигнала ***CLK*** выходному сигналу ***Q*** присваивается значение входного сигнала ***D***. Такое поведение соответствует поведению стандартного элемента – синхронного D-триггера.

Рассмотрим еще несколько примеров, дающих начальное представление об использовании VHDL для описания простых устройств.

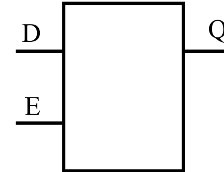
Пример 2. Триггер, тактируемый передним фронтом, с асинхронным сбросом и предустановкой.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity DFF_ASYNC is
  port (PRESET,RESET,DATA, CLK: in std_logic;
  Q: out std_logic);
end DFF_ASYNC;
architecture DFF_ASYNC_ARCH of DFF_ASYBC is
  begin
    process (CLK)
      begin
        if RESET='0' then
          Q<='0';
        elsif PRESET='1' then
          Q<='1';
        elsif CLK'event and CLK='1' then
          Q<=DATA;
        end if;
    end process;
  end DFF_ASYNC_ARCH;
```



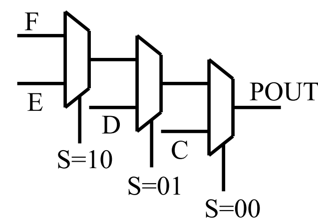
Пример 3. Защелка на основе D-триггеров

```
library IEEE;
use IEEE.std_logic_1164.all;

entity D_LATCH is
  port (D, E: in std_logic;
 Q: out std_logic);
end D_LATCH;

architecture D_LATCH_ARCH of D_LATCH is
  begin
    process (D, E)
      begin
        if  E='1' then
          Q<=D;
        end if;
      end process;
end D_LATCH_ARCH;
```



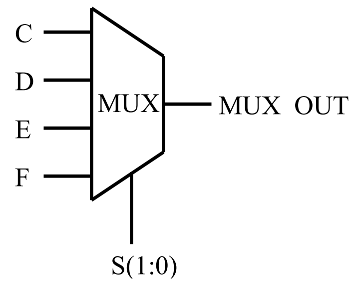
Пример 4. Приоритетный шифратор

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MY_IF is
   port (C,D,E,F: in std_logic;
  S: in std_logic_vector(1 downto 0);
  POUT: out std_logic);
end  MY_IF;

architecture MY_IF_ARCH of MY_IF is
  begin
    process(S,C,D,E,F)
      begin
        if S="00" then          POUT<=C;
        elsif S="01" then       POUT <=D;
        elsif S="10" then       POUT <=E;
        else                    POUT <=F;
        end if;
     end process;
end MY_IF_ARCH;
```



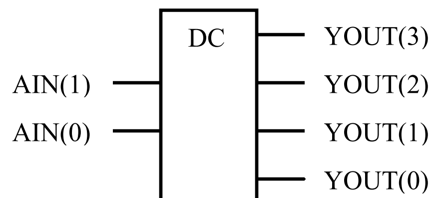
Пример 5. Мультиплексор 4:1

```
library IEEE;
use IEEE.std_logic_1164.all;

entity MUX is
  port(C,D,E,F: in std_logic;
  S: in std_logic_vector(1 downto 0);
  MUX_OUT: out std_logic);
end MUX;

architecture MUX_ARCH of MUX is
  begin
    process(S,C,D,E,F)
      begin
        case S is
          when “00” =>        MUX_OUT <=C;
          when “01” =>        MUX_OUT <=D;
          when “10” =>        MUX_OUT <=E;
          when others =>      MUX_OUT <=F;
        end case;
    end process;
end MUX_ARCH;
```



Пример 6. Дешифратор

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DECODE is
   port (AIN: in std_logic_vector(1 downto 0);
  EN: in std_logic;
  YOUT: out std_logic_vector(3 downto 0));
end DECODE;

architecture DECODE _ARCH of DECODE is
  begin
    process (AIN, EN)
      begin
        if EN=’0’ then YOUT<=(others=>’0’);
        else
          case AIN is
            when “00” =>     YOUT <= “0001”;
            when “01” =>     YOUT <= “0010”;
            when “10” =>     YOUT <= “0100”;
            when “11” =>     YOUT <= “1000”;
```



```
        when others =>   YOUT <= “0000”;
      end case;
    end if;
  end process;
end DECODE_ARCH;
```

Алфавит моделирования

Важной характеристикой при проектировании устройств и моделировании является количество различимых состояний сигналов. Набор всех состояний сигналов составляет алфавит. Простейшим алфавитом является набор из двух символов: ***0*** и ***1***. Взаимодействие сигналов описывается правилами математической логики. Однако такой алфавит не является достаточным и ограничивает возможности моделирования и проектирования. Например, невозможно описание шинной логики, в том числе схем, имеющих высокоимпедансное состояние на выходе (Z-состояние), схем с открытым коллектором и т.д. Затруднено воспроизведение сбойных ситуаций, например, вызванных подачей управляющих сигналов на триггеры во время, когда информационные сигналы еще не установлены. Поэтому широко распространен алфавит из 4-х символов: ***0***, ***X***, ***1*** и ***Z***. ***X*** обозначает неопределенное состояние. В таком состоянии находится, например, выход логического элемента во время переходного процесса. Символом ***Z*** обозначается высокоимпедансное состояние порта или отключенная линия. Некоторые упрощенные диалекты языка VHDL используют четырехсимвольный алфавит.

Более сложный алфавит состоит из 9 символов:

- ***U*** – не инициализировано. Если сигналу ранее не присваивалось какое-либо значение, то он будет находиться в этом состоянии. Символ используется только при моделировании для выявления ошибок инициализации, поскольку реальный сигнал будет находиться в каком-либо определенном состоянии или в состоянии X (неопределенность);
- ***X*** – активное неопределенное состояние;
- ***0*** – активный ноль;
- ***1*** – активная единица;
- ***Z*** – высокоимпедансное состояние;
- ***L*** – слабый ноль;
- ***H*** – слабая единица;
- ***W*** – слабое неопределенное состояние;

– **-** – не важно (любое состояние). Этот символ введен для оптимизации синтезируемой схемы. Если реализация алгоритма на некотором интервале времени не зависит от значения сигнала, то имеет смысл присвоить этому сигналу значение '**-**'. При этом компилятор сам подставляет то конкретное значение, которое приводит к более оптимальной реализации устройства или проще реализуется.

Разница между слабыми и активными состояниями заключается в том, что слабый сигнал формируется от источников, имеющих повышенное выходное сопротивление по сравнению с активными источниками. Поэтому источник, выдающий активный сигнал, подавляет слабый. Например, буфер с открытым коллектором генерирует слабую единицу, но активный ноль. Наличие такого сложного алфавита позволяет описывать такие схемотехнические приемы, как использование монтажной логики и др.

При записи программ на VHDL пользователь может сам задавать алфавит моделирования тех или иных конструкций, задавая тип сигнала. Например,

signal S1: bit;
signal S2: std_logic;

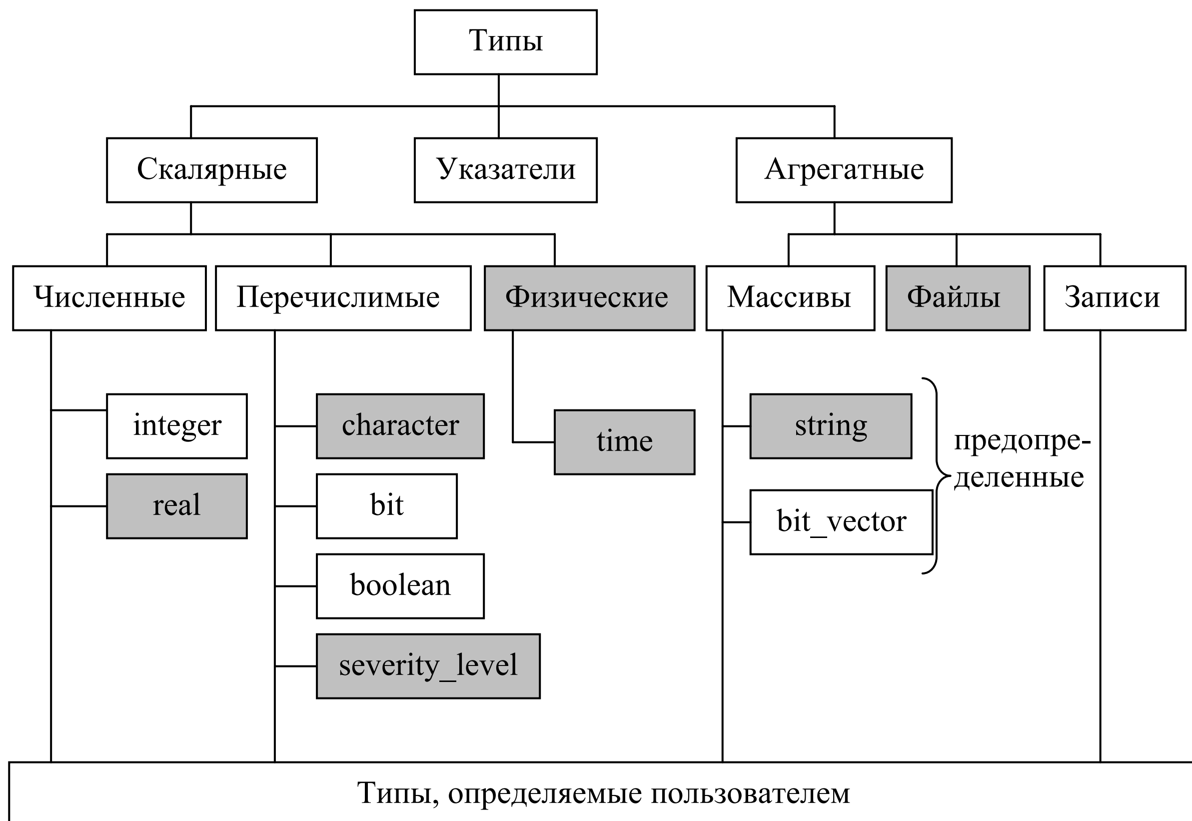
Сигнал ***S1*** имеет тип ***bit***, который включает только два значения: ***0*** и ***1***. Тип ***std_logic*** (сигнал ***S2***) включает алфавит из 9 элементов, рассмотренный выше.

Типы данных и декларации объектов

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

Данные, используемые в программах, относятся к одной из следующих категорий: кон*станты, переменные, сигналы и файлы.* Различие между сигналами и переменными будет рассмотрено позже.

На следующей диаграмме изображена структура типов данных в языке VHDL. Серым цветом отмечены типы, не поддерживаемые в синтезируемом подмножестве языка и используемые только для компьютерного моделирования.



Декларация объектов имеет следующий синтаксис:

<категория> <имя>«,<имя>» :<тип> [:=<выражение>] ;
<категория> ::= constant | variable | signal

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

constant A: integer:=15;
variable B,C : bit;
signal D,E : DOUBLE_WORD;

В последнем примере предполагается, что тип ***DOUBLE_WORD*** был ранее определен пользователем.

Если декларируемый объект является агрегатом (см. агрегатные типы на диаграмме), то в одной декларации можно определить все компоненты агрегата. Например, пусть определен тип ***STACK*** как массив из восьми целых. Для того чтобы объявить переменную этого типа и задать ее начальное значение, можно записать:

variable STACK_INSTANTION: STACK:= (4, 5, 7,129, 64, 7, 87, 67) ;

Язык VHDL предопределяет некоторый базовый набор типов данных, которые не требуют объявления в программе пользователя. Кроме того, пользователь может определять свои типы данных. Различают скалярные типы данных и агрегатные типы. Объект, отнесенный к скалярному типу, рассматривается как законченная единица информации. Агрегат представляет упорядоченную совокупность скалярных единиц, объединенных одинаковым именем.

Предопределенные типы данных

Сначала остановимся на предопределенных типах.

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

Типы ***integer*** и ***real*** определяют численные данные — целые и действительные, соответственно. Диапазон представления чисел может зависеть от реализации, но стандартными считаются диапазоны $\{-2^{31}+1, +2^{31}-1\}$ для типа ***integer*** и $\{10^{38}, -10^{38}\}$ для ***real***. Надо иметь в виду, что число фактически является укороченной записью обрабатываемых кодов, но над этими кодами определены арифметические операции:

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

Определены для этих кодов также и операции арифметического отношения: =, /=, <, <=, >=, >, которые дают результат типа ***boolean***. Приоритет операторов такой же, как и в большинстве языков программирования.

Данные типа ***bit*** могут принимать значения из множества ***0*** и ***1***. Для данных типа ***bit*** определены следующие логические операции:

- ***not*** - инверсия;
- ***or*** - операция ИЛИ;
- ***nor*** — операция ИЛИ-НЕ;
- ***and*** — операция И;
- ***nand*** - операция И-НЕ;

– ***xor*** - неравнозначность;
– ***xnor*** - равнозначность.

Данные типа ***boolean*** также могут принимать два значения: {***true, faslse***}, и на них определены те же операции, что и над данными типа ***bit***. Различие между типами ***bit*** и ***boolean*** состоит в том, что первые используются для представления уровней логических сигналов в аппаратуре, а вторые для представления обобщенных условий, например, результатов сравнения. Так, если переменная ***SELECT*** определена как бит, то нельзя записать условный оператор в виде

if SELECT then . . .

следует записывать

if SELECT='l' then . . .

Если бы переменная ***SELECT*** была определена как ***boolean***, то, наоборот, первый вариант был бы допустим, а второй – нет.

Данные разных типов несовместимы, поэтому недопустимо выражение типа

'0' and true

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

Тип ***time*** — время — используется для задания задержек элементов и времени приостанова процессов при моделировании. Определены такие единицы измерения времени: ***fs*** — фемтосекунда, ***ps*** — пикосекунда, ***ns*** — наносекунда, ***us*** — микросекунда, ***ms*** — миллисекунда, ***s*** — секунда.

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

Типы ***file_open_status*** и ***file_open_kind*** обеспечивают возможность работы с файловой системой инструментального компьютера.

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

Помимо этого, пользователь может самостоятельно определять и использовать новые типы.

Типы, вводимые пользователем

VHDL предусматривает возможность вводить пользовательские типы и затем использовать их наряду с предопределенными.

Перечислимые типы, вводимые пользователем:

type STATE is (S0, S1, S2, S3);

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

type COLOUR is (WHITE, BLACK, RED, GREEN, BLUE, YELLOW, ARGENDA);

type std_ulogic is ('U', 'X' ,' 0 ',' 1' , 'Z', 'W , 'L' , 'H' ,'-');

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

Численные типы, вводимые пользователем

Определение численных типов пользователя целесообразно, во-первых, для контроля совместимости данных в программах, а во-вторых, для точного задания разрядности слов, представляющих данные в проектируемом объекте.

type UNSIGNED_SHORT is integer range 0 to 255;
*type MY_DATA is integer range -2**(n-l)+l to 2**(n-l)-l;*
type INPUT_LEVEL is +10.0 downto -10.0;

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

|При объявлении типа ***INPUT_LEVEL*** базовый тип явно не задан, используется тип ограничений в соответствии с типом их фактических значений.

Пусть пользователь в одном проекте вводит два типа:
type DATA is integer range 0 to 15;
type CONTROLE is integer range 0 to 15;
Хотя с точки зрения представления эти типы равноценны, однако они несовместимы, что облегчает контроль корректности описания.

Физические типы, вводимые пользователем

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

Пример:

```
type VOLTAGE is range -5e6 to +5e6;
  units uV;     - базовая единица - микровольт
  mV= 1000 uV;  - милливольт
   V=1000 mV;   - вольт
end units VOLTAGE;
```

Введение такого типа позволяет описывать и моделировать сопряжение цифровой логической схемы с аналоговыми источниками.

Пусть входной порт ***ANALOG_INPUT*** и константа ***THRESHOLD*** (порог) объявлены как ***VOLTAGE***, а сигнал ***COMPARE*** (сравнение) как ***bit***. Тогда можно записать:

COMPARE <= '1' when ANALOG_INPUT > THRESHOLD else '0';

Массивы и записи

Как в большинстве языков программирования, массивы и записи относятся к пользовательским типам.

Массив - это набор данных, объединенных общим именем и различаемых по порядковым номерам (индексам). Для того чтобы ввести объект типа массив, необходимо задать базовый тип и диапазон значений индексов.

array (<диапазон> «, <диапазон> ») of <тип элемента массива>

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

Такое определение предполагает задание диапазона при определении конкретного экземпляра объекта.

Примеры:

*type RAM is array (LENGTH-1 downto 0) of integer range 2**WIDTH-l downto 0;*
type RAM1 is array (LENGTH-1 downto 0,WIDTH-1 DOWNTO 0) of std_logic;
type RAM2 is array (integer range<>, integer range<>) of std_logic;

Во всех приведенных декларациях объявляется в сущности одно и то же, а именно матрица ячеек памяти емкостью ***LENGTH*** слов по ***WIDTH*** разрядов в каждом, причем предполагается, что эти параметры были ранее определены. Однако выполнено это разными способами, а значит, и ссылаться на эти типы следует по-разному. ***RAM*** и ***RAM1*** определены как ограниченные типы массивов, ***RAM*** — как одномерный массив целых, а ***RAM1*** — как двумерный массив битов. ***RAM2*** определен как неограниченный тип и требует задания границ индексов при декларации объектов выбранного типа.

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

variable RAM_INSTANCE: RAM;
variable RAM1_INSTANCE: RAM1;
variable RAM2_INS: RAM2(LENGTH-1 downto 0, WIDTH-1 downto 0);

При обращении к элементам массива в программе индексы помещаются в скобках следом за именем массива:

RAM2_INS(y, 5) :=’1’;

Для одномерных массивов определено несколько групповых операций, в которых массив рассматривается как единое целое. Это, прежде всего, операция конкатенации ***&*** (объединение строк). Например, приведенная ниже последовательность операторов присваивает сигналу значение "10011101".

A:="1001";
B<= "1101" & a;

Здесь: ***A*** и ***B*** — строки или битовые векторы, причем ***A*** -

переменная, а ***B*** - сигнал (о сигналах и переменных будет сказано ниже).

Операции сдвига определены для одномерных массивов типа ***bit*** или ***boolean*** и записываются следующим образом:

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

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

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

Запись - эта структура данных, каждая информационная единица которой, называемая полем записи, имеет индивидуальное имя и тип. Обычно записи используются для объединения разнородных данных, характеризующих один объект. Для использования записей как переменных сначала надо объявить соответствующий тип:

```
type PIXEL is
  record RED, GREEN, BLUE: integer range 0 to 255;
end record;
```

Тогда тип "видеопамять" может быть определен как

```
type VIDEO_RAM is array(integer range<>, integer range<>) of PIXEL;
```

Экземпляр видеопамяти будет определяться, например, следующим образом:

```
signal VRAM : VIDEO_RAM (479 downto 0, 639 downto 0);
```

Этот экземпляр может сохранять информацию об изображении размером 680 строк по 840 элементов в строке. Выборка значения красной составляющей верхнего левого элемента изображения из такой памяти описывается оператором

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

Подтипы

Специфическим понятием языка VHDL является подтип. Объекты, отнесенные к подтипу, сохраняют совместимость с базовым типом. Однако введение подтипа определяет множество допустимых значений данных подтипа, а также позволяет вводить дополнительные функции преобразования, определяемые только для данных подтипа.

subtype BIT_IN_WORD_NUMBER is integer range 31 downto 0;

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

Сигналы и переменные. Оператор PROCESS

Любой проект является описанием явлений в дискретных системах. Эти явления могут представляться тремя различными категориями данных: константы, переменные и сигналы. ***signal*** - это информация, передаваемая между модулями проекта или представляющая входные и выходные данные проектируемого устройства. Сигналу присваиваются свойства изменения во времени. ***variable*** - это вспомогательная информационная единица, используемая для описания внутренних операций в программных блока. В любом случае, и сигналы, и переменные VHDL в конечном счете интерпретируются как аналоги электрических сигналов в схеме. Однако они обладают разным поведением, и чаще всего применение сигналов и переменных приводит к получению разных электрических схем в ходе синтеза. Для того чтобы подчеркнуть различие между переменными и сигналами, VHDL использует для них разные формы операторов присваивания. Присвоение значения сигналу отображается знаком **<=**, а переменной - знаком **:=**.

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

Исполнение параллельных операторов инициируется не по последовательному, а по событийному принципу, т. е. они исполняются тогда, когда реализация других операторов программы создала условия для их исполнения. Параллельные операторы представляют части алгоритма, которые в реальной системе могут исполняться одновременно. Эти части взаимодействуют между собой и с окружением проектируемой

системы. Параллельные операторы могут быть простыми и составными. Составной оператор включает несколько простых операторов, для которых определены общие условия инициализации. Такая совокупность операторов называется телом составного оператора.

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

```
process(CLK)
begin
if CLK'event and CLK='1' then
        Q<=D;
end if;
end process;
```

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

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

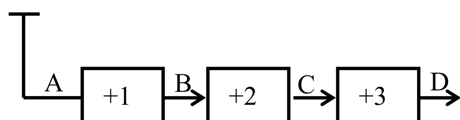
Наиболее явно разница между сигналами и переменными проявляется при интерпретации операторов последовательных присвоений. Для обоих видов сохраняется общее для последовательных операторов правило начала исполнения: первый оператор в процессе исполняется после выполнения условий инициализации процесса,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

исполнения до конца процессов и других параллельных операторов, инициированных общим событием, или после исполнения оператора останова ***wait***. Рассмотрим пример. Предположим, что в некотором процессе мы ввели несколько переменных

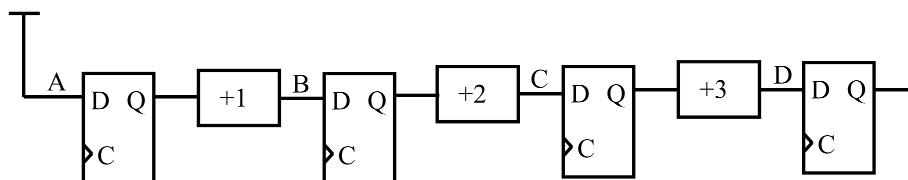
variable A: integer range 0 to 127:=0;
variable B: integer range 0 to 127:=0;
variable C: integer range 0 to 127:=0;
variable D: integer range 0 to 127:=0;

Далее встречается следующая последовательность операторов:
A:=1; B:=A+1; C:=B+2; D:=C+3;

После выполнения первой итерации процесса сигнал ***A*** будет равен 1, ***B*** будет равен 2, ***C*** – 4, а ***D*** – 7. Схемотехническая реализация данных операторов будет выглядеть примерно следующим образом:



Теперь предположим, что речь идет о последовательном присваивании для сигналов. В этом случае определения сигналов будут присутствовать не в процессе, а в разделе деклараций архитектурного тела:



signal A: integer range 0 to 127:=0;
signal B: integer range 0 to 127:=0;
signal C: integer range 0 to 127:=0;
signal D: integer range 0 to 127:=0;

В теле процесса присутствуют следующие операторы:
A<=1; B<=A+1; C<=B+2; D<=C+3;

Тогда после первого выполнения тела процесса ***A*** будет равен 1, ***B*** также будет равен 1, ***C*** будет равен 2, а ***D*** – 3. Будет синтезирована следующая аппаратура:

Общие правила интерпретации оператора ***process*** можно свести к следующим:

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

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

Все операторы раздела операторов выполняются подряд друг за другом от начала до конца, за исключением случаев приостановки исполнения действий оператором ***wait***. Тогда после приостановки может быть инициировано исполнение других процессов и параллельных операторов, а реализация операторов, следующих за оператором ***wait***, продолжится после наступления события, объявленного в этом операторе.

```
process(CLK)
begin
if CLK'event and CLK='1' then
   Q<=D;
end if;
end process;

process
begin
WAIT for CLK'event;
Q<=D;
end process;
```

Приведенный выше пример содержит два возможных варианта процессов, функционирующих одинаково и описывающих одну и ту же схему. Однако первый вариант является все же предпочтительным, потому что оператор ожидания ***wait*** поддерживается не всеми средствами синтеза и может относиться к несинтезируемому подмножеству языка.

Укажем на некоторые наиболее существенные различия сигналов и переменных:

- переменные меняют значения сразу после присвоения, и новые значения непосредственно учитываются во всех преобразованиях, записанных в теле процесса после такого присвоения;
- значение сигнала меняется не сразу после выполнения присвоения. Оператору присваивания сопоставляется некий буфер, называемый контейнером или, чаще, драйвером

сигнала. Оператор присваивания <= передает новое значение драйверу сигнала, и лишь после того, как выполнены преобразования во всех процессах, инициированных общим событием, содержание драйвера передается сигналу;

- переменная определена только внутри тела процесса, сигнал - во всем архитектурном теле;
- переменной можно переприсваивать значение в теле процесса. Сигнал внутри одного процесса может иметь только один драйвер. То есть присвоение значения сигналу может быть выполнено только один раз в теле процесса.

Атрибуты в языке VHDL

Атрибуты – это скаляры, отражающие некоторые свойства объектов, используемых в программных модулях (типов, переменных, агрегатов). Например, атрибуты типа используются для сжатого представления информации о множестве значений, объединенных типом, а атрибуты сигнала — для представления временных свойств сигнала.

Бывают предопределенные атрибуты и атрибуты, вводимые программистом. Ограничимся только представлением наиболее употребительных предопределенных атрибутов.

Предопределенные атрибуты типов

Атрибут	***Тип***	***Значение***
T'left	Любой скалярный	Левая граница значений T
T'right	Любой скалярный	Правая граница значений T
T'low	Численный, физический	Нижняя граница значений T
T'high	Численный, физический	Верхняя граница значений T
T'image(x)	Любой	Строка символов, представляющая значение x
T'pos(x)	Перечислимый	Позиция значения x в наборе значений T
T'val(N)	Перечислимый, физический, целый	Значение элемента в позиции N в наборе значений T
T' leftof(X)	Перечислимый, физический, целый	Значение в наборе значений T, записанное в позиции слева от X
T' rightof(X)	Перечислимый, физический, целый	Значение в наборе значений T, записанное в позиции справа от X

T'pred(X)	Перечислимый, физический, целый	Значение в наборе значений Т на одну позицию меньшее Х
T'succ(X)	Перечислимый, физический, целый	Значение в наборе значений Т на одну позицию большее Х

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

T'leftof = T'pred　　　*T'left = T'low*　　　*T'right = T'high*

T'rightof = T'succ.

Смысл конструкторов ***leftof***, ***pred***, ***left***, ***low***, ***right***, ***high***, ***rightof***, ***succ*** для целых и физических типов зависит от направления (***to*** или ***downto***), объявленного в декларации типа. Если объявлено обратное направление, то ***T'left= m'high T'leftof= T'succ*** и т. д.

Пусть совокупность возможных состояний некоторого устройства объявлена типом

type STATE is(s0,   s1,   s2,   s3,   s4,   s5);

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

```
process
  if CURRENT_STATE = STATE'right then
      CURRENT_STATE<=STATE'left;
  else current_state <=state'succ(current_state);
      wait 100 nS;
  end if;
end process;
```

В соответствии с приведенным фрагментом через каждые 100 нс происходит инициализация процесса и изменение состояния. Устройство поочередно принимает состояния в порядке записи в списке значений от ***S0*** до ***S5***, а из ***S5*** выполняется переход в начальное состояние ***S0***.

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

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

Атрибуты сигналов являются эффективным средством анализа поведения сигнала во времени.

Имя атрибута	*Тип атрибута*	*Значение*
S'delayed (T)	То же, что у ***S***	Значение ***S***, существовавшее на время ***T*** перед вычислением атрибута
S'event	***boolean***	Сигнализирует об изменении сигнала
S'stable	***boolean***	***S ' stable = not S'event***
S'active	***boolean***	true, если присвоение сигналу выполнено, но значение еще не изменено (не кончен временной интервал, заданный выражением after)
S'quiet	***boolean***	***S'active = not S'quiet***
S'last_event	***time***	Время от момента вычисления атрибута до последнего перед этим изменения сигнала
S'last_active	***time***	Время от момента вычисления атрибута до последнего присвоения значения сигналу (не совпадает с ***last_event*** при наличии слова after в определяющем выражении)

Например, в конструкции

if CLK' event and CLK='1' then <оператор>;

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

Последовательные операторы

Последовательные операторы по характеру исполнения подобны операторам традиционных языков программирования. Операторы этого типа обязательно "вложены" в оператор ***process*** или подпрограмму и выполняются последовательно друг за другом в порядке записи. Результаты исполнения последовательных операторов недоступны прочим программным модулям по крайней мере до того, как будет выполнен оператор ожидания ***wait***, или не будут выполнены до конца все процессы, инициированные общим событием. Это можно трактовать так, что с точки зрения "окружения" все операторы, в теле процесса от его начала до оператора ***wait***, а при отсутствии ***wait*** — до конца тела, исполняются одномоментно.

Операторы присваивания

Оператор присваивания значения переменной **:=** всегда является последовательным оператором. Результат присваивания значения переменной становится доступен сразу после выполнения оператора. Сигнальное присваивание (**<=**) бывает последовательным и параллельным в зависимости от расположения оператора. Синтаксис оператора присваивания значения сигналу имеет вид:

<приемник> <= [<модель задержки>] <временная диаграмма>;
<модель задержки> : :=transport | inertial <выражение времени>
<временная диаграмма> ::= элемент поведения> «, элемент поведения> »
<элемент поведения> : := <выражение> [after<выражение времени)

Временные интервалы для определения переходов временной диаграммы задаются относительно времени возникновения инициирующего события. Например, в следующем операторе присваивания все временные интервалы отсчитываются относительно момента его срабатывания:

X<=’1’ after 5ns, ‘0’ after 20ns, ‘Z’ after 50 ns,’0’ after 70 ns;

Различают транспортную и инерционную модели задержки. Если в операторе присваивания присутствует ключевое слово ***transport***, предполагается транспортная задержка. Транспортная модель предполагает идеализацию поведения устройства так, что любой импульс, сколь коротким он бы ни был, воспроизводится на выходе.

Y<= transport 'X' after 35 ns;

По умолчанию, а также если использовано объявление ***inertial***, предполагается инерционная задержка. В VHDL'87 слово ***inertial*** не определено, инерционная задержка выбирается по умолчанию при

отсутствии опции "модель задержки".

Инерционная модель используется для описания устройств, не реагирующих на импульсы, длительность которых меньше некоторого наперед заданного значения. По умолчанию это значение равно значению времени, указанному после слова ***after***. В следующем примере сигнал ***Y*** не переходит в состояние '***Z***', поскольку длительность этого состояния сигнала ***X*** меньше времени задержки.

X<=’1’, ‘Z’ after 30 ns, ‘0’ after 40 ns;
Y<= inertial ‘X’ after 20 ns;

Оператор условия и оператор выбора (if и case)

if B_1 *then* S_1
elsif B_2 *then* S_2
elsif B_3 *then* S_3
...
elsif B_n *then* S_n
else S_{n+1}
end if;

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

Если при каком-либо условии никаких действий не предусмотрено, то все равно после этого выражения проверки данного условия должен размещаться оператор, в данном случае пустой оператор ***null***.

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

Оператор ***case*** имеет вид:
Пусть ***X*** – переменная или сигнал типа ***integer***

```
case X  is
when 1  =>
        <оператор> « <оператор> »
when 2 | 3 =>
        <оператор> «<оператор> » »
when 4 TO 6 =>
        <оператор> «<оператор> » »
when others =>
        <оператор> «<оператор> » »
end case;
```

В качестве разделителя в списках выбираемых вариантов используется вертикальная черта.

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

Оператор ожидания

Исполнение операторов, записанных в теле процесса, приостанавливается, если очередной оператор является оператором ожидания WAIT. При этом результаты исполнения предшествующих операторов заносятся в календарь событий и могут быть инициализированы другие процессы. Определено несколько модификаций оператора ***wait***:

wait;
wait on <имя сигнала> «,<имя сигнала> »;
wait until <условие>;
wait for <выражение времени>;

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

Список сигналов в варианте ***wait on*** эквивалентен списку инициализаторов процесса: продолжение исполнения будет после того, как один из сигналов списка изменит свое значение.

Пауза, заданная конструкцией ***wait until***, заканчивается, когда выполнено заданное оператором условие, т. е. соответствующее выражение принимает значение ***true***.

Процесс, содержащий оператор ***wait***, не может иметь списка инициализаторов. Это связано с тем, что трудно описать систему, в которой может произойти повторная инициализация действий, в то время как реакция на предыдущее событие еще не реализована, например, произошло изменение одного из инициирующих сигналов, когда время ожидания еще не вышло. Оператор ***wait***, как и другие последовательные

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

Операторы повторения

Операторы повторения ***loop*** позволяют записывать циклы.

Формат оператора:

[<метка>:] [-«итерационная схема>] loop

<оператор> « <оператор> »

end loop [<метка>];

«Итерационная схема> : :=while <условие> | for <имя переменной> in <диапазон>

Последовательность операторов (здесь могут быть только последовательные операторы), заключенная между словами ***loop*** и ***end loop***, называется телом цикла. Операторы в теле цикла выполняются друг за другом в порядке записи. Число повторений определяется итерационной схемой.

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

```
entity SEQUENCE_OF_NUMBERS is
  port ( OUT_DATA: OUT INTEGER range 0 TO MAX_VALUE;
  CLK : inout bit);
end SEQUENCE_OF_NUMBERS;

arhitecture BEHAVIOR of SEQUENCE_OF_NUMBERS is begin

  CLOCK_GENERATOR: process
    begin
```

```
        CLK<='0';
        loop wait for 50 NS;
                CLK<= NOT CLK;
        end loop;
  end process CLOCK_GENERATOR;


EXECUTE : process
variable INT_STATE: integer range 0 to 63:=0;
begin
  OUT DATA<=0;
  loop
        wait until(CLK='L'   AND CLK'EVENT);
        if  (INT_STATE=MAX VALUE)   THEN INT_STATE:=0;
        else  INT_STATE:=INT_STATE+L;
        end if;
        OUT_DATA<= INT_STATE;
  end loop;
end process  EXECUTE;
end BEHAVIOR;
```

Оператор с ключевым словом ***while*** обязательно содержит в теле цикла операторы, изменяющие описанное в итерационной схеме условие. Операторы цикла повторяются, пока при вычислении условия не получается значения ***false***. Условие проверяется каждый раз перед исполнением тела цикла. Например, оператор повторения с ключевым словом ***for*** повторяется для всех значений переменной из заданного итерационной схемой диапазона.

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

Число устройств определяется итерационной схемой. Для оператора с ключевым словом ***for*** – это просто число значений переменной в объявленном диапазоне. Для варианта с ключевым словом ***while*** условие не может быть связано с сигнальными данными, способными изменяться в реальном устройстве. Как правило, для аппаратной реализации оператор повторения должен иметь логически постоянные границы. В противном

случае не ясно, сколько повторяющихся блоков в устройстве реально потребуется.

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

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

Оператор записывается следующим образом:

[<метка>:] next [<метка оператора повторения>] [when <условие>]

Оператор ***exit*** прекращает исполнение не только текущего цикла, но всех последующих циклов, заданных итерационной схемой исполняемого оператора.

[<метка>:] exit [<метка оператора повторения>] [when <условие>]

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

Оператор проверки

Оператор проверки ***assert*** относится к категории конструкций, не подлежащих реализации в аппаратуре. Оператор служит для выявления специфических ситуаций, которые могут возникать в процессе компиляции и моделирования, и выдачи сообщений разработчику.

assert <булевское выражение>[report<сообщение>][severity<важность>];

При выполнении этого оператора в процессе моделирования проверяется условие, и если получено значение ***true***, выполняется переход к следующему оператору программы. В противном случае на терминал выводится сообщения. Если опция ***report <строка сообщения>*** отсутствует, выводится стандартное сообщение "*Assertion violation*" (нарушение условия проверки). После этого поведение моделирующей программы определяется значением уровня важности. Уровень важности – это выражение (обычно константа) типа ***SEVERITY_LEVEL***. Данные этого типа могут принимать четыре значения, причем значения ***NOTE*** и ***WARNING*** имеют чисто информационный характер, и в этом случае моделирование после выдачи сообщения продолжается, а значения ***ERROR*** и ***FAILURE*** используются для обнаружения полностью некорректных ситуаций, требующих прекращения моделирования. По

умолчанию подразумевается уровень ***ERROR***.

Параллельные операторы

Параллельные операторы выполняются при любом изменении сигналов, используемых в качестве его исходных данных. Результаты исполнения оператора доступны для других параллельных операторов не ранее, чем будут выполнены все операторы, инициализированные общим событием (а может быть и позже, если присутствуют выражения задержки).

Оператор процесса

Оператор процесса уже рассматривался ранее. Этот оператор определен как составной оператор параллельного типа. Оператор процесса начинает исполняться при изменении сигналов, входящих в список инициализаторов (при отсутствии такого списка — безусловно после выполнения всех вложенных операторов), а результаты его исполнения доступны другим параллельным операторам только после исполнения всех операторов, инициируемых теми же событиями, в том числе процессов.

Параллельное присваивание

Параллельное присваивание определено в трех различных формах: безусловное параллельное присваивание, условное присваивание, присваивание по выбору.

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

Различия между параллельным и последовательным присваиванием заключаются в следующем:

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

В обоих случаях результаты присвоения сначала фиксируются в драйвере сигнала и передаются сигналу, т. е. могут влиять на другие операторы, только после исполнения всех операторов и процессов,

инициированных одним событием, или через интервал модельного времени, заданный опцией ***after***.

Условное присваивание и присваивание по выбору (параллельное) во многом сходны с условным оператором и оператором выбора (последовательными), так как описанные действия выполняются при определенных условиях. Однако условный оператор и оператор выбора являются составными, т. е. условие может задавать в них исполнение последовательности действий, а в операторах присваивания — только присвоение одного значения.

<условное присваивание>: : =
<приемник> <= [guarded] [<модель задержки>] «<прогноз поведения> when <условие> else » <прогноз поведения>;

Например, двухвходовый буфер с тремя состояниями на выходе может быть представлен следующим оператором:
OUT<= transport X0 after 2 ns when (ADR = '0' and EN='L') else X1 after 2 ns when (ADR='L' and EN='L') else 'Z' after 5 ns;

Здесь битовый сигнал ***ADR*** задает номер включаемого канала, а ***EN*** – разрешение передачи.

Параллельный оператор присваивания по выбору имеет вид:
<присваивание по выбору> : : =
with <ключевое выражение> select
<приемник> <= [guarded] [<модель задержки>] «<прогноз поведения> when <вариант>,» <прогноз поведения> when <вариант>;

Следующий оператор описывает буфер с тремя состояниями:

```
with A & B select
Z_OUT<= transport X0 after 2 ns  when   "01",
X1 after 2 ns  when   "11",
'Z' after 5 ns  when  others;
```

Условный оператор ***if*** и оператор выбора ***case*** не могут выполняться над данными, вырабатываемыми различными операторами процесса, а условное присваивание и присваивание по выбору позволяют описывать такие ситуации.

```
entity THREE_CHANNEL is
    port ( DATA_IN: in integer;  -- условные входные данные
```

```
    CHANNEL_SELECT: in integer range 1 to 3;
    Z: out integer);
end THREE_CHANNEL;

architecture SKELETON of THREE_CHANNEL is
signal DATAL,DATA2, DATA3 : integer;
  begin
     with cHANNEL_SELECT select
        Z<= DATA1        when 1,
        DATA2            when 2,
        DATA3            when 3,
        'z'          when OTHERS;
     process.....
       begin      DATA1<=    .....
    end  process;
    process.....
      begin      DATA2<=.....
    end   process;
    process.....
       begin      DATA3<=    .....
    end  process;
end SKELETON;
```

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

Оператор блока

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

В этом смысле операторы, включенные в блок, не отличаются от

"индивидуальных" параллельных операторов.

Объединение операторов в блоки обеспечивает следующие возможности:

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

Упрощенные правила записи оператора блока определены таким образом:

(оператор блока> : : =
<метка блока>: block [(охранное выражение)] [is] [<раздел деклараций блока>]
begin
<раздел операторов блока>
End block [<метка блока>] ;

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

Охранное выражение – это любое выражение логического типа, аргументом которого являются сигналы. Любое изменение сигналов, входящих в охранное выражение, вызывает вычисление значения этого выражения и присвоение полученного значения предопределенной переменной ***GUARD***. Область действия переменной ***GUARD*** — все тело блока, и она может использоваться как обычная логическая переменная во вложенных операторах блока. Например, узел выборки данных из тридцатидвухразрядного регистра на восьмиразрядную линию:

```
SELECT_BYTE:  block   (SELECT='L'   AND READ='L')   is
begin
        DBUS<=REG(7  DOWNTO  0)   when GUARD and BYTE_SEL="00"
        else
        REG(15 DOWNTO 0)     when GUARD and BYTE_SEL="01"    else
        REG(23 DOWNTO  16)   when GUARD and BYTE__SEL ="10"  else
        REG(31 DOWNTO 24)   when GUARD and BYTE_SEL ="11" else
        "ZZZZZZZZ";
end block SELECT_BYTE;
```

Охраняемый оператор присваивания использует значение переменной ***GUARD***. Если ***GUARD*** ='***0***', то исполнение операторов присваивания, содержащих ключевое слово ***quarded***, в таком блоке запрещено.

Например:

```
architecture GUARD_EXAMPLE of TWO_BLOCK is
signal  DATA_BUS: std_logic_vector (N-1 downto 0);
begin
<описание других блоков системы>

UNIT1: block ( ADR='0' and READ_DATA='1')
signal DATA0 : std_logic_vector (N-1 downto 0) ;
  begin
        DATA_BUS<=quarded DATA0;
        process
          begin
        <вычисление DATA0>
end process;
end block UNIT1;

UNIT2: block ( ADR='L' and READ_DATA='1')
signal DATAL : std_logic_vector (N-1 downto 0);
  begin
dATA_BUS<=quarded DATA1;
process
  begin
        <вычисление DATA1;
        end process;
end block UNIT2;
```

Подпрограммы

Подпрограммы в VHDL, как и в других алгоритмических языках, обеспечивают, во-первых, структуризацию описания проекта, а во-вторых, являются средством экономии времени проектировщика, позволяя заменить несколько описаний сходных фрагментов алгоритма одним объявлением подпрограммы и соответствующими вызовами в основном тексте.

Каждая подпрограмма, используемая в проектном модуле, должна быть представлена телом подпрограммы в разделе деклараций этого

модуля или проектного модуля, иерархически старшего по отношению к данному. Различают два вида подпрограмм - процедуры (***procedure***) и функция (***function***).

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

Спецификация подпрограммы определяет ее интерфейс - имя, входные и выходные данные. Входные данные подпрограммы специфицируются направлением ***in***, выходные — направлением ***out***, а данные, которые воспринимаются подпрограммой и возвращаются в вызывающую программу измененными, — как ***inout***. Указание категории элемента списка (***constant***, ***variable*** или ***signal***) обеспечивает контроль корректности использования подпрограммы. По умолчанию определено, что сопоставляемый объект относится к категории ***variable***. Несоответствие типа или категории фактического или формального параметра является ошибкой.

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

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

Оператор вызова процедуры является последовательным, если локализован в теле процесса или теле другой подпрограммы. В иных случаях оператор вызова подпрограммы интерпретируется как параллельный оператор. Одна и та же подпрограмма может вызываться как параллельным, так и последовательным оператором. Параллельный вызов процедуры фактически эквивалентен процессу, тело которого совпадает с телом процедуры с точностью до обозначений, а список инициализаторов содержит входные фактические параметры оператора вызова.

Ниже представлено несколько узлов суммирования и вычитания в различной форме.

```
architecture ADD_EXAMPLES of SOMETHIN is
procedure ADD_SUBB(
        signal A,B:in integer;
        signal OPERATION : in std_logic;
        signal RESULT: out integer) is
begin
        if OPERATION='0' then RESULT<= A+B;
        else RESULT<=A-B;
        end if;
end ADD_SUBB;

begin
FIRST:  ADD_SUBB(X1,Y1,CONTROLE, Z1);
SECOND: ADD_SUBB( A=>X1,B=>Y1, RESULT=>Z2,
OPERATION=>CONTROLE);
```

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

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

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

```
architecture NOTHING of SOMETHING is
type V is array (integer range<>) of integer;
function SUM_UNTIL_ZERO (signal DATA:in V) return integer is
variable I,SUM:integer;
begin
	for I in DATA'range loop
		if DATA(I) =0 then return 0;—обнаружен нулевой элемент
		else SUM:= SUM+DATA(I);
		end if;
	end loop;
	return SUM;
end SUM_UNTIL_ZERO;

signal DATA_ARRAY : V ( 0 to 15);
begin
	process(DATA_ARRAY)
	begin
		RESULT <= INPUT+ SUM_UNTIL_ZERO(DATA_ARRAY);
	end process;
end NOTHING;
```

В связи с рассмотренным примером отметим еще одно обстоятельство. Здесь в описании функции тип формального параметра определен как неограниченный массив целых. Фактический параметр отнесен к тому же типу, но при декларации сигнала определен фактический его размер. Размер массива в подпрограмме автоматически устанавливается равным размеру фактического массива.

Разрешаемые сигналы и шины

Если сигнал имеет один драйвер, то его значение определяется достаточно просто. Однако во многих системах к одной линии подключено несколько источников. Например, шина данных компьютера может принимать сигналы от процессора, памяти, периферийных устройств, а к линии данных матрицы запоминающего устройства подключается достаточно много ячеек памяти, а также буферы ввода/вывода. Этому соответствуют программные модели, в которых один сигнал назначается в

нескольких параллельных операторах и процессах. Сигналы, значения которых автоматически определяются исходя из состояний нескольких источников (драйверов), называют *разрешаемыми* (resolved). Разрешаемым может быть объявлен конкретный сигнал или подтип данных, к которому такой сигнал отнесен при его декларации. Декларация подтипа может содержать имя *функции разрешения* (resolution function). Функция разрешения определяет правило вычисления сигнала, формируемого несколькими независимыми источниками.

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

Рассмотрим в качестве примера представление линии, к которой подключаются элементы с выходами типа "открытый коллектор". Если хоть один источник выдает на линию уровень логического нуля, то на линии устанавливается логический нуль. Тип ***bit*** не может быть использован в подобной ситуации, ибо является неразрешаемым. Введем тип ***odw***, как разрешаемый подтип типа ***bit***, и соответствующую функцию разрешения:

```
function RESOLVED_ODW( S:bit_vector) return bit is
variable C: bit:='1';
  begin
       for I in S'range loop
              if S(I)='0' then
                     C:='0';
                     exit;
              end if;
       end loop;
       return C;
end RESOLVED_ODW;
subtype ODW is RESOLVED_ODW bit;
```

При проектировании чаще всего не требуется создание собственных разрешаемых подтипов данных. В современных системах интерпретации всегда присутствует пакет ***std_logic_ll64***, в котором, в частности, определен подтип ***std_logic*** как разрешаемый подтип девятизначного перечислимого типа ***std_ulogic***

```
type stdlogic_table is array(std_ulogic,   std_ulogic)   of std_ulogic;
constant  resolution_table   :   stdlogic_table   :=   (
--      U     X     0     1     Z     W     L     H     -

(       'U',  'U',  'U',  'U',  'U',  'U',  'U',  'U',  'U'),  -- |U|
(       'U',  'X',  'X',  'X',  'X',  'X',  'X',  'X',  'X'),  -- |X|
(       'U',  'X',  '0',  'X',  '0',  '0',  '0',  '0',  '0'),  -- |0|
(       'U',  'X',  'X',  '1',  '1',  '1',  '1',  '1',  '1'),  -- |1|
(       'U',  'X',  '0',  '1',  'Z',  'W',  'L',  'H',  'Z'),  -- |Z|
(       'U',  'X',  '0',  '1',  'W',  'W',  'W',  'W',  'W'),  -- |W|
(       'U',  'X',  '0',  '1',  'L',  'W',  'L',  'W',  'L'),  -- |L|
(       'U',  'X',  '0',  '1',  'H',  'W',  'W',  'H',  'H'),  -- |H|
(       'U',  'X',  '0',  '1',  'Z',  'W',  'L',  'H',  '-')   -- |-|

function RESOLVED ( S : std_ulogic_vector) return std_ulogic is
variable RESULT : std_ulogic :='-'; -- значение по умолчанию - самый
слабый сигнал
begin
   if    (S'length = 1) then   return S(S'low);
   else                              -- ПРОСМОТР ВСЕХ ДРАЙВЕРОВ
        for I in S'range loop
                RESULT := RESOLUTION_TABLE(RESULT, S(I));
        end loop
        return RESULT;
end if;
end RESOLVED;
subtype STD_LOGIC is RESOLVED std_ulogic;
```

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

Использование подтипа ***std_logic*** позволяет облегчить решение многих проблем, возникающих при моделировании шинных соединений. Этот подтип и порождаемый на его основе тип ***std_logic_vector*** стали фактическим стандартом. Данные этого типа нередко используют и в случаях, когда сигналы имеют единственный драйвер и даже если при моделировании достаточно воспроизводить меньшее число состояний.

Однако платой за такую универсализацию может стать увеличение затрат машинного времени при моделировании.

Компоненты

Компоненты являются средством, позволяющим использовать уже существующие объекты. Кроме того, они являются важным инструментом для структурного описания проектов. Используемые компоненты должны быть предварительно описаны, а также объявлены в теле архитектуры или пакета. Компоненты имеют локальную область видимости, то есть могут использоваться только в тех архитектурах, где они объявлены. Исходное описание компонента должно присутствовать либо в том же исходном файле, либо в другом исходном файле, который подключен к проекту. VHDL не устанавливает каких-либо стандартов, описывающих структуру проекта, например, формат записей, указывающих, какие модули входят в проект. Каждая система проектирования решает этот вопрос по-своему. С точки зрения пользователя важно, что он имеет возможность объединить несколько модулей, хранящихся в отдельных файлах, в один проект, который содержит общий набор объектов. Из этих объектов можно строить иерархическую структуру. Кроме того, можно использовать объекты, которые хранятся в пакетах и подключены с помощью директивы ***use***.

Объявление компонент имеет следующий формат:

component <имя>
[generic (описание локальных констант);]
port (описание портов);
end component;

Раздел описания локальных констант может отсутствовать. Как правило, он используется для того, чтобы позволить с помощью одних и тех же компонентов работать с данными, имеющими разную разрядность.

Установка компонента в проект напоминает процесс установки нового компонента в электрическую схему (при работе со схемотехническим редактором) и имеет следующий вид:

Имя элемента : имя компонента
[generic map (имя локальной константы => значение
{ ,имя локальной константы => значение })]
port map (
[имя порта =>] имя сигнала
{ , [имя порта =>] имя сигнала }
);

Встраивание компонента заключается в том, что создается его экземпляр, который имеет уникальное имя (в локальной области видимости). Раздел generic map позволяет указать конкретные значения локальным константам. В разделе port map порты созданного экземпляра подключаются к сигналам, которые определены в архитектуре. При соединении компонентов и сигналов необходимо придерживаться элементарных схемотехнических правил. Например, если один и тот же сигнал будет подключен к выходам нескольких компонентов, то при этом могут возникнуть конфликты, связанные с попытками разных компонентов присвоить сигналу разные значения. Как при задании значении локальных констант, так и при подключении сигналов к портам можно использовать две синтаксические формы записи – полную и сокращенную. Полная форма записи выглядит следующим образом: *имя порта => имя сигнала* или *имя локальной константы => значение.* При этом порты или имена констант могут перечисляться в произвольном порядке. В сокращенной форме записи имена портов (локальных констант) вообще не используются. Вместо этого подключаемые сигналы (или значения) перечисляются через запятую в том же порядке, в котором перечислены порты (константы) при объявлении компонента.

Рассмотрим тривиальный пример, который иллюстрирует правила работы с компонентами. Пусть мы описываем схему, которая содержит несколько операций сложения с разной разрядностью.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity add is
  generic (N: integer:=8);  -- задано значение по умолчанию – 8
  port(
    S1: in integer range 0 to 2**N-1;
    S1: in integer range 0 to 2**N-1;
    SUM: out integer range 0 to 2**(N+1)-1);
  end add;
architecture add_arch of add is
  begin
    SUM<=S1+S2;
  end add_arch;

library IEEE;
use IEEE.std_logic_1164.all;
entity add_test is
  port (
```

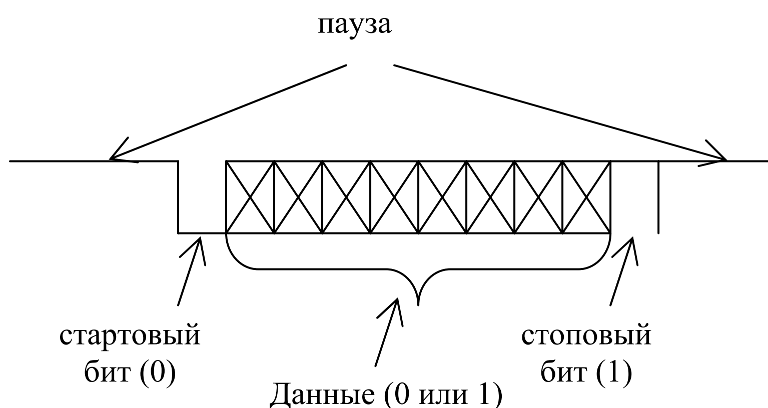
```
      A,B: in integer range 0 to 255;
      C,B: in integer range 0 to 15;
      S1: out integer range 0 to 511;
      S2: out integer range 0 to 31);
end add_test;
architecture behavioral of add_test is

component add is  -- объявление компонента
   generic (N: integer);
   port(
      S1: in integer range 0 to 2**N-1;
      S1: in integer range 0 to 2**N-1;
      SUM: out integer range 0 to 2**(N+1)-1);
   end component;

begin
   U1: add generic map(N=>8) port map(S1=>A, S2=>B, SUM=>S1);
   U1: add generic map(N=>4) port map(S1=>C, S2=>D, SUM=>S2);
end add_test;
```

Примеры описания устройств

Рассмотрим более сложный пример, который обобщает рассмотренный материал. В основном, реальные проекты содержат не один, а как минимум несколько процессов. Легко представить себе цикл проектирования как разработку функциональной схемы, которая состоит из набора функциональных блоков, которые, в свою очередь, также представляются в виде набора блоков и связей между ними. Так может быть образована многоуровневая иерархическая структура, которая задается в VHDL-проекте с помощью механизмов структурного описания, в том числе путем включения компонентов. И лишь на нижних уровнях иерархии используется поведенческое описание. Основную роль при этом играют процессы, которые содержат блоки последовательных операторов. Проектирование традиционным способом, то есть с помощью графического схемного редактора, опирается только на структурное описание. Поэтому, при переходе к использованию языков описания аппаратуры нужно научиться правильно комбинировать элементы поведенческого и структурного описания. С одной стороны, требуется обеспечить высокую скорость проектирования, легкую читаемость описания и возможность вносить в него оперативные изменения, что создается элементами поведенческого описания. В то же время, применение структурных методов описания, а также уменьшение

сложности компонентов и процессов на нижнем уровне иерархии позволяет сохранить контроль над синтезируемой схемой.

Рассмотрим реализацию протокола последовательного асинхронного передатчика. Логическая часть указанного протокола совместима с RS-232. Таким образом, можно использовать передатчик для организации обмена данными между стандартным компьютером и внешним устройством.



Данные передаются пакетами, структура которых изображена ниже.

Пакет содержат стартовый бит, или признак начала пакета, имеющий фиксированное состояние логического ***0***, 8 бит данных, и стоповый бит, логическое состояние которого всегда ***1***. Пауза между пакетами имеет произвольную длительность, определяемую передатчиком, и имеет состояние логической ***1***. Таким образом, длительность стопового бита представляет собой, по сути, минимальную длительность паузы между пакетами. Данные в пакете передаются, начиная с младшего значащего разряда. Все передаваемые биты, включая стартовый, имеют одинаковую длительность, которая зависит от настроек протокола обмена данными. Например, если битовая скорость (bitrate) составляет 9600 (указанное значение входит в набор стандартных), то длительность каждого бита составляет 1/9600 сек. Исключение составляет последний, стоповый бит, длительность которого может составлять ***1***, ***1.5*** и ***2***. Соответственно, временной интервал удержания стопового бита может соответствовать длительности всех остальных битов или быть в 1.5 или 2 раза больше.

Передатчик RS-232

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RS232_TX is
 port (
   RST                          : in std_logic;
   DATA                         : in std_logic_vector(7 downto 0);
```

```
  DATA_READY            : in std_logic;
  CLK                   : in std_logic;       --bitrate x 16 clock;
  BUSY                  : out std_logic;
  TxD                   : out std_logic);
end RS232_TX;

architecture RS232_TX_ARCH of RS232_TX is
type STATES is (WAIT_FOR_BYTE, BYTE_TRANSFER);
signal STATE: STATES:=WAIT_FOR_BYTE;
signal REG: std_logic_vector(9 downto 0);
signal TICK: integer range 0 to 255:=0;
  begin
    TxD<=REG(0);
    BUSY<='0' when state=WAIT_FOR_BYTE else '1';

   process(RST,CLK)
      begin
         if RST='1' then STATE<=WAIT_FOR_BYTE;
          elsif CLK'event and CLK='1' then
                case (STATE) is
                  when WAIT_FOR_BYTE =>
                    if DATA_READY='1' then STATE<=BYTE_TRANSFER;
                    else null;
                    end if;
                 when BYTE_TRANSFER =>
                   if TICK=160 then
                     STATE<=WAIT_FOR_BYTE;
                   else null;
                   end if;
                end case;
          end if;
        end process;

  process(RST, STATE, CLK)
    begin
         if RST='1' or STATE= WAIT_FOR_BYTE  then TICK<=0;
         elsif CLK'event and CLK='1' then TICK<=TICK+1;
         end if;
   end process;

  process (RST, CLK)
    begin
        if RST='1' then REG<=(others=>'1');
        elsif CLK'event and CLK='1' then
          if state=WAIT_FOR_BYTE then REG<=DATA&"01";
          elsif (TICK=1 or TICK=17 or TICK=33 or TICK=49 or TICK=65 //
           or TICK=81 or TICK=97 or TICK=113 or TICK=129 or TICK=145)
          then REG(9 downto 0) <= '1' & REG(9 downto 1);
          else null;
          end if;
```

```
        end if;
    end process;
end rs232_tx_arch;
```

В интерфейсной части компонента представлены следующие сигналы (порты).

RST - Асинхронный сброс. Активное состояние - логическая 1. Если не используется (в большинстве случаев), то должен быть привязан к состоянию логического 0.

DATA - передаваемые данные.

DATA_READY - готовность данных. После того, как на этом входе установлено состояние логической 1, начинается передача. Состояние *DATA* и *DATA_READY* должно удерживаться как минимум до ближайшего переднего фронта сигнала *CLK*.

CLK - сигнал тактирования компонента. Частота должна составлять 16 x baudrate, то есть в 16 раз превышать значение скорости в настройках интерфейса.

BUSY - сигнализирует о занятости компонента. Логическая 1 выставляется в том случае, когда идет передача данных.

TXD - линия передатчика. С помощью этого сигнала производится передача данных в виде описанных выше пакетов.

Архитектурное тело содержит 3 процесса. Хотя функционирование протокола можно описать в виде единственного процесса, приведенное описание позволяет более четко представлять схемотехническую реализацию компонента.

Первый процесс описывает функционирование конечного автомата с двумя состояниями: *WAIT_FOR_BYTE* и *BYTE_TRANSFER*. В первом состоянии компонент ожидает активного состояния сигнала *DATA_READY*, после чего переходит в состояние, когда происходит передача пакета. Передача пакета занимает 160 периодов тактового сигнала *CLK*, после чего компонент автоматически возвращается в состояние *WAIT_FOR_BYTE.*

Второй процесс реализует счетчик, который считает периоды тактового сигнала в ходе передачи пакета. В состоянии *WAIT_FOR_BYTE* счетчик сбрасывается. Значения счетчика используется для контроля временных интервалов в ходе передачи пакета.

Третий процесс реализует сдвиговый регистр, выполняющий параллельно-последовательное преобразование, а также управляющую логику для этого регистра.

Передаваемый пакет содержит стартовый бит, 8 бит данных, а также стоповый бит, длительность которого равна 1. Компонент содержит только синтезируемые конструкции и операторы.

Литература:

1. Бибило П.Н. Основы языка VHDL / П.Н. Бибило. – М. : Солон-Р, 1999. – 200 с.
2. Грушвицкий Р.И. Проектирование систем на микросхемах программируемой логики / Р.И. Грушвицкий, А.Х. Мурсаев, Е.П. Угрюмов. – СПб. : БХВ-Петербург, 2002. – 608 с.
3. Шалыто А.А. Методы аппаратной и программной реализации алгоритмов / А.А. Шалыто. – СПб. : Наука, 2000. – 780 с.
4. VHDL для моделирования, синтеза и формальной верификации аппаратуры: сб. ст. / под. ред. Ж. Мермье. – М. : Радио и связь, 1995. – 360 с.
5. VHDL'92. Новые свойства языка описания аппаратуры VHDL / Ж.-М. Берже. [и др.]. – М. : Радио и связь, 1995. – 256 с.
6. Угрюмов Е.П. Цифровая схемотехника / Е.П. Угрюмов. – СПб. : БХВ-Петербург, 2000. – 528 с.
7. Уэйкерли Дж. Ф. Проектирование цифровых устройств : в 2-х т. / Дж. Ф. Уэйкерли. – М. : Постмаркет, 2002.

Учебное издание

ПРОЕКТИРОВАНИЕ ЦИФРОВЫХ УСТРОЙСТВ С ПОМОЩЬЮ ЯЗЫКА ОПИСАНИЯ АППАРАТУРЫ VHDL

Учебное пособие для вузов

Составители:
Бобрешов Анатолий Михайлович,
Дыбой Александр Вячеславович

Редактор А.П. Воронина