V.II Основы языка VHDL

(Very high speed integration circuits Hardware Description Language)
Стандарт VHDL-87, Стандарт VHDL-93, Стандарт VHDL-AMS

Язык VHDL используется для:

- описания поведения цифровых устройств во времени и при изменении входных воздействий;
- описания структуры цифровых устройств с различной степенью детализации (на системном и блочном уровнях, на уровне регистровых передач, на уровне вентилей);
- моделирования цифровых устройств;
- описания тестовых воздействий при моделировании устройств;
- автоматизации преобразования исходного описания схемы в описание на более низком уровне (вплоть до вентильного уровня).

Стили описания:

- **поведенческий стиль**, при котором для описания проекта используются причинно-следственные связи между событиями на входах устройства и событиями на его выходах (без уточнения структуры);
- **структурный стиль описания**, при котором устройство представляется в виде иерархии взаимосвязанных простых устройств (подобно стилю, принятому в схемотехнике);
- **потоковый стиль описания** устройства основан на использовании логических уравнений, каждое из которых преобразует один или несколько входных информационных потоков в выходные потоки.

Объекты VHDL и их типы

Объекты могут относится к следующим категориям:

- константы;
- сигналы;
- переменные;
- файлы.

В VHDL применяется строгая типизация, при которой для выполнение действий над разнотипными объектами требуется явно преобразовать их к одному и тому же типу.

Упрощенный синтаксис описания типа и подтипа:

<Тип>::= **TYPE** <Идентификатор> **IS** <Описание типа> ;
<Подтип>::= **SUBTYPE** <Идентификатор> **IS** <Описание подтипа> ;

```
TYPE Multi_Level_Logic IS (LOW, HIGH, RISING, FALLING, AMBIGUOUS);
TYPE BIT IS ('0','1') ;
TYPE BYTE_LENGTH_INTEGER IS RANGE 0 TO 255;
TYPE WORD_INDEX IS RANGE 31 DOWNTO 0;
SUBTYPE HIGH_BIT_LOW IS BYTE_LENGTH_INTEGER RANGE 0 TO 127;
TYPE MY_WORD IS ARRAY (0 TO 31) OF BIT ;
TYPE STATE_TYPE IS (s0,s1,s2,s3);
```

Базовые типы данных, описанные в библиотеке STANDARD
(автоматически подключается для всех проектов во всех САПР)

Тип	Значения/Диапазон
Перечислимые типы	
BOOLEAN	TRUE,FALSE
BIT	‘0’,’1’
BIT_VECTOR	Массив элементов BIT с индексами от 0 до +2147483647
SEVERITY_LEVEL	NOTE, WARNING, ERROR, FAILURE
FILE_OPEN_KIND	READ_MODE, WRITE_MODE, APPEND_MODE
FILE_OPEN_STATUS	OPEN_OK, STATUS_ERROR, NAME_ERROR, MODE_ERROR
Целочисленные типы	
INTEGER	−2147483647 … +2147483647
POSITIV	1…… +2147483647
NATURAL	0…… +2147483647
Типы чисел с плавающей запятой	
REAL	−1.0E38 … +1.0E38
Символьные типы	
CHARACTER	Nul,..,’0’,…,’9’,’@’,’:’,’;’,…,’A’,…’Z’,’a’,…,’z’,DEL
STRING	Массив элементов CHARACTER с индексами от 1 до +2147483647
Физические типы	
TIME	−2147483647 … +2147483647
DELAY_LENGTH	0…+2147483647

Атрибуты

Объявление атрибута:

<Объявление Атрибута> ::=
ATTRIBUTE <Идентификатор>: <Тип>;

Спецификация атрибута:

<Спецификация атрибута> ::=
ATTRIBUTE <Идентификатор> of <Описание> : <Класс> **IS** <Выражение>;
<Описание> ::= <Имя> [{ , <Имя> }] | **OTHERS | ALL**
<Класс> ::= **ENTITY | ARCHITECTURE | CONFIGURATION**
| PROCEDURE | FUNCTION | PACKAGE
| TYPE | SUBTYPE | CONSTANT
| SIGNAL | VARIABLE | COMPONENT
| LABEL | LITERAL | UNITS
| GROUP | FILE
<Имя> ::= <Таг> [<Сигнатура>]
<Таг> ::= <Идентификатор> | <Символьный литера> | <Символ оператора>

```
ATTRIBUTE state_vector : string;
  ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";
```

Константы и сигналы

<Константа> ::=
CONSTANT <Список идентификаторов> : <Тип> [:= <Выражение>]

CONSTANT A: TIME := 30ns;

Сигналы – это объекты, обладающие историей изменения (прошлым состоянием и текущим состоянием)

<Сигнал> ::=
SIGNAL <Список идентификаторов> : <Тип> [<Вид>] [:= <Выражение>] ;
<Вид сигнала> ::= **REGISTER | BUS**

Для каждого сигнала предопределены следующие атрибуты, поддерживаемые программами синтеза (на примере сигнала s):

- s'**stable**[(T)] – атрибут равен значению true, если за время T сигнал не изменял своего состояния.
- s'**transaction** – атрибут типа bit, изменяющий состояние на противоположное при каждом присваивании нового значения сигналу s.
- s'**event** – атрибут, равный true, если в данном цикле моделирования произошло изменение сигнала s.
- s'**active** – атрибут равен true, если в данном цикле моделирования произошло присваивание нового значения сигналу s.
- s'**last_value** - атрибут, равный сигналу s до момента его последнего изменения.

Пример декларации и инициализации сигнала:

```
SIGNAL B: BIT_VECTOR(3 DOWNTO 0):=”0000”;
```

Пример присвоения значения сигналу:

```
B<=”0001”;
```

Пример использования атрибутов сигнала:

```
IF CLK’event AND CLK=’1’ THEN
B<=D;  --Изменить значение по фронту синхросигнала
END IF;
```

Переменные

<Переменная> ::=
[**SHARED**] **VARIABLE** <Список идентификаторов> : <Тип> [:= <Выражение>] ;

Пример декларации переменной:

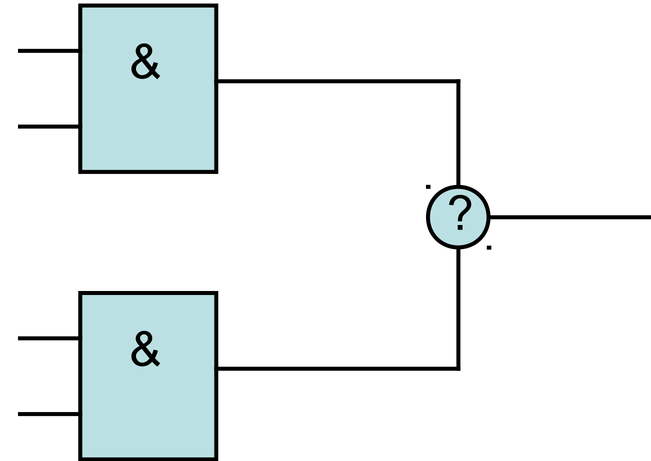
```
VARIABLE C: INTEGER := 100;
```

Пример присвоения значения переменной:

```
B:=200;
```

Сигнал типа STD_LOGIC

'U' – не инициализировано;
'X' – неизвестное значение (сильное);
'0' – логический 0 (сильное);
'1' - логическая 1 (сильное);
'Z' – третье состояние;
'W' - неизвестное значение (слабое);
'L' - логический 0 (слабое);
'H' - логическая 1 (слабое);
'-' – Неопределенное значение.



Функцией разрешения называется функция, по которой определяется результирующее значение сигнала от многих источников.

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

Сигнал типа STD_LOGIC

•‘U’ – цепь не инициализирована, используется при моделировании. Например если мы попытаемся прочитать ячейку память в котороюо до этого не было записи то получим такое значение.

•‘X’ – неизвестное значение. Отражает ситуацию подключения к одной и той же цепи двух драйверов, причем один выдает ‘1’, а другой ‘0’.

•‘0’, ‘1’ – логический ‘0’ или ‘1’ соответственно.

•‘Z’ – высокий импеданс.

•‘L’ – цепь подключена к терминальному резистору уровня ‘0’. Например мы имеем тристабильный буфер, выход которого подключен через резистор к земле, притом сам буфер выключен.

•‘H’ – цепь подключена к терминальному резистору уровня ‘1’. Например мы имеем тристабильный буфер, выход которого подключен через резистор к питанию, притом сам буфер выключен.

•‘W’ – цепь одновременно подключена к терминальному резистору уровня ‘0’ и терминальному резистору уровня ‘1’.

•‘-‘ – не имеет значения (безразлично). Используется при описании устройств, в которых значение на выходе схемы в некоторых случаях не зависит от значения на входе, например, шифратора. (В картах Карно такие значения обозначались как α)

Операции VHDL

Тип операций	Обозначения	Комментарий
Логические	AND OR NAND NOR XOR XNOR	Логическое И Логическое ИЛИ Логическое И-НЕ Логическое ИЛИ-НЕ Исключающее ИЛИ Эквивалентность
Сравнения	= /= < > <= >=	Равенство Неравенство Меньше Больше Меньше или равно Больше или равно
Сдвига	SLL SRL SLA SRA ROL ROR	Сдвиг влево логический Сдвиг вправо логический Сдвиг влево арифметический Сдвиг вправо арифметический Сдвиг влево циклический Сдвиг вправо циклический
Знака и сложения	+ - &	Сложение (знак +) Вычитание (знак -) Конкатенация
Умножения	* / MOD REM	Умножение Деление (синтезируются ограничено) Модуль (синтезируются ограничено) Остаток (синтезируются ограничено)
Смешанные	ABS NOT **	Абсолютное значение Отрицание Степень

Примеры выполнения операций над объектами:

```
result <= a + b;--сумматор (поведенческий стиль)
sum <= (a XOR b) XOR cin;--сумматор (потоковый или структурный стиль)
cout  <= (a AND b) OR (a AND cin) OR (b AND cin);
equal <= '1' WHEN a = b ELSE '0'; --компаратор
outp <=  "00000001" SLL conv_integer(inp); --дешифратор (поведенческий стиль)
```

Интерфейс и архитектура устройств

```
<Интерфейс> ::=
ENTITY <Идентификатор> IS
[GENERICS  <Список настроечных констант>]
[PORT <Список портов>]
      <Декларативная часть>
[ BEGIN
      <Пассивные конструкции языка> ]
END [ ENTITY ] [ <Идентификатор>] ;
```

Упрощенный синтаксис декларации порта

```
<Описание портов>::=
PORT(<Описание порта> {;<Описание порта>});
<Описание порта>::= <Список идентификаторов> : <Режим>  <Тип>
<Режим> ::= BUFFER|IN|OUT|INOUT|LINKAGE
```

Пример конструкции ENTITY:

```
ENTITY Decoder IS
      PORT (        inp:  IN std_logic_vector(2 DOWNTO 0);
                    outp: OUT std_logic_vector(7 DOWNTO 0));
END Decoder;
```

Упрощенный синтаксис конструкции ARCHITECTURE:

```
<Архитектура> ::=
ARCHITECTURE <Идентификатор> OF <Интерфейс> IS
      <Декларативная часть>
BEGIN
      <Параллельные операторы>
END [ ARCHITECTURE ] [<Идентификатор>] ;
```

Пример описания устройства:

```
-- Комментарий: Интерфейс полного сумматора
ENTITY Full_Adder IS
PORT (  X, Y, Cin:IN Bit;
                    Cout, Sum:OUT Bit) ;
END Full_Adder ;
-- Комментарий: Архитектура 1 полного сумматора  (потоковый
стиль)
ARCHITECTURE DataFlow1 OF Full_Adder IS
SIGNAL A,B: Bit;
```

```
BEGIN
      A <= X XOR Y;
      B <= A AND Cin;
      Sum <= A XOR Cin;
      Cout <= B OR (X AND Y);
END ARCHITECTURE DataFlow1 ;
-- Комментарий: Архитектура 2 полного сумматора  (потоковый стиль)
ARCHITECTURE DataFlow2 OF Full_Adder  IS
BEGIN
      Sum <= (X XOR Y) XOR Cin;
      Cout  <= (X AND Y) or (X AND Cin) OR (Y AND Cin);
END DataFlow2;
```

Пакеты и библиотеки

Декларация пакета состоит из описания интерфейса и описания тела.

<Интерфейс пакета> ::=
PACKAGE <Имя пакета> **is**
    <Декларативная часть>
END [**PACKAGE**] [<Имя>] ;
<Тело пакета> ::=
PACKAGE BODY <Имя пакета> **is**
    <Декларативная часть>
END [**PACKAGE**] [<Имя пакета>] ;

Пакеты объединяются в библиотеки (LIBRARY)

<Подключение библиотеки> ::=
LIBRARY <Имя библиотеки> { , <Имя библиотеки> };
<Использование Пакета>::=
<Имя библиотеки>.<Имя пакета>.<Символьное имя> | <Идентификатор> | **ALL**;

```
PACKAGE TriState IS
    TYPE Tri IS ('0', '1', 'Z', 'E');
    FUNCTION BitVal (Value: Tri) RETURN Bit ;
    FUNCTION TriVal (Value: Bit) RETURN Tri;
    TYPE TriVector IS ARRAY (Natural RANGE <>) OF Tri ;
    FUNCTION Resolve (Sources: TriVector) RETURN Tri ;
END PACKAGE TriState ;
PACKAGE BODY TriState IS
    FUNCTION BitVal (Value: Tri) RETURN Bit IS
        CONSTANT Bits : Bit_Vector := "0100";
    BEGIN
        RETURN Bits(Tri'Pos(Value));
    END;
    FUNCTION TriVal (Value: Bit) RETURN Tri IS
    BEGIN
        RETURN Tri'Val(Bit'Pos(Value));
    END;
```

```
        FUNCTION Resolve (Sources: TriVector) RETURN Tri IS
                VARIABLE V: Tri := 'Z';
        BEGIN
                FOR i IN Sources'Range LOOP
                        IF Sources(i) /= 'Z' THEN
                                IF V = 'Z' THEN
                                        V := Sources(i);
                                ELSE
                                        RETURN 'E';
                                END IF;
                        END IF;
                END LOOP;
                RETURN V;
        END;
END PACKAGE BODY TriState ;
```

Параллельные операторы

Операторы данной группы описывают параллельно функционирующие части устройства. К таким операторам относятся:

- •Блоки (BLOCK);
- •Процессы (PROCESS);
- •Параллельные вызовы процедур;
- •Параллельные присваивания сигналов (<=);
- •Параллельные операторы – ловушки (ASSERT);
- •Экземпляры компонентов (COMPONENT);

Оператор BLOCK

Оператор BLOCK позволяет описать группу параллельных операторов:

```
<Блок> ::=
[<Метка>] :  BLOCK [( <Охранное выражение> )] [IS]
    [GENERIC (<Объявление настроечных констант>);]
    [GENERIC MAP ( Список связывания настроечных констант );]
    [PORT (<Объявление портов>);]
    [PORT MAP ( <Список Связывания портов> )]
    <Декларативная часть>
 BEGIN
    <Параллельные операторы>
 END BLOCK [<Метка>];
```

Пример описания устройства с использованием иерархии блоков:

```
C: BLOCK
BEGIN
    X: BLOCK
    PORT (P1, P2: INOUT BIT); --объявление портов
    PORT MAP (P1 => S1, P2 => S2); --связывание портов
    CONSTANT Delay: DELAY_LENGTH := 1 ms; --декларации
    SIGNAL P3: BIT;
    BEGIN
        P3 <= P1 AFTER Delay; --Присвоение
        •
        B: BLOCK
        •
        BEGIN
                •
        END BLOCK B;
    END BLOCK X;
END BLOCK C;
```

Оператор PROCESS

Оператор PROCESS описывает независимые группы последовательных операторов в виде параллельных процессов. Упрощенный синтаксис оператора PROCESS:

```
<Процесс>::=
[<Метка>:][POSTPONED] PROCESS (<Список чувствительности>) [IS]
      <Декларативная часть>
BEGIN
      <Последовательные операторы>
END [POSTPONED] PROCESS [<Метка>] ;
```

Пример описания динамического триггера с использованием процесса:

```
FF: PROCESS (CLK,RST,WE)
BEGIN
    IF RST='1' THEN
            D<='0';
    ELSIF ((CLK='1') AND (CLK'event)) THEN
            IF (WE='1') THEN
                D<=D_IN;
            END IF;
    END IF;
END PROCESS;
```

Оператор параллельного присваивания сигнала

Оператор параллельного присваивания сигнала эквивалентен процессу, в котором, при определенных условиях, происходит присваивание сигналу нового значения

<Параллельное присваивание сигнала>::=
[<Метка> :] [**POSTPONED**] <Условное присваивание сигнала>
| [<Метка> :] [**POSTPONED**] <Присваивание сигнала>

<Условное присваивание сигнала> ::=
<Сигнал>**<=** <Опции> { <Образец> **WHEN** <Условие> **ELSE** }
<Образец> [**WHEN** <Условие>];

<Присваивание сигнала> ::=
WITH <Выражение> **SELECT**
<Сигнал> **<=** <Опции> { <Образец> **WHEN** <Значение>, }
<Образец> **WHEN** <Значение>;

<Опции> ::= [**GUARDED**] [<Способ задержки>]

<Образец> ::=
<Выражение> [**AFTER** <Время>]| **NULL** [**AFTER** <Время>]

Динамический триггер:

```
ENTITY ff IS
      PORT (RST,CLK,WE: IN std_logic;
            D_IN: IN std_logic;
            D: OUT std_logic);
END ff;
ARCHITECTURE DataFlow OF ff IS
BEGIN
D<='0'    WHEN RST='1' ELSE
     D_IN         WHEN (CLK='1') AND (CLK'event) AND (WE='1');
END DataFlow;
```

Дешифратор:

```
LIBRARY ieee; --Описание подключаемых библиотек
USE ieee.std_logic_1164.ALL;
ENTITY decoder IS
     PORT ( inp:  IN std_logic_vector(2 DOWNTO 0);
           outp: OUT std_logic_vector(7 DOWNTO 0));
END decoder;
```

```
ARCHITECTURE DataFlow OF decoder IS
BEGIN
        outp(0) <= '1' WHEN inp = "000" ELSE '0';
        outp(1) <= '1' WHEN inp = "001" ELSE '0';
        outp(2) <= '1' WHEN inp = "010" ELSE '0';
        outp(3) <= '1' WHEN inp = "011" ELSE '0';
        outp(4) <= '1' WHEN inp = "100" ELSE '0';
        outp(5) <= '1' WHEN inp = "101" ELSE '0';
        outp(6) <= '1' WHEN inp = "110" ELSE '0';
        outp(7) <= '1' WHEN inp = "111" ELSE '0';
END DataFlow;
```

Процедуры и функции

Другим способом группировки последовательных действий являются процедуры и функции:

<Процедура> ::=
PROCEDURE <Имя> [(<Список формальных параметров >)]
|[**PURE**|**IMPURE**] **FUNCTION** <Имя> [(< Список формальных параметров>)]
RETURN <Тип>

Пример объявления и параллельного вызова процедуры:

```
ENTITY sort4 IS
	GENERIC (top : integer :=3);
	PORT (
		a, b, c, d : IN bit_vector(0 TO top);
		ra, rb, rc, rd : OUT bit_vector(0 TO top)
	);
END sort4;
ARCHITECTURE muxes OF sort4 IS
	PROCEDURE sort2(SIGNAL x, y : IN bit_vector(0 TO top);
			SIGNAL g, l : OUT bit_vector(0 TO top)
	) IS
	BEGIN
```

```
IF x > y THEN
                    g <= x;
                    l <= y;
              ELSE
                    l <= x;
                    g <= y;
END IF;
       END sort2;
SIGNAL v1,v2,v3,v4,v5,v6,v7,v8,v9,v10 : bit_vector(0 TO top);

BEGIN
--Параллельные вызовы процедуры
      sort2(a, c, v1, v2);
      sort2(b, d, v3, v4);
      sort2(v1, v3, v5, v6);
      sort2(v2, v4, v7, v8);
      sort2(v6, v7, v9, v10);
-- Параллельные присваивания сигналов
      ra <= v8;
      rb <= v10;
      rc <= v9;
      rd <= v5;
END muxes;
```

Оператор - ловушка ASSERT

Для упрощения отладки в языке VHDL используется оператор - ловушка ASSERT, позволяющий выполнить в ходе моделирования проверочное сравнение, и, при обнаружении ошибки (ложном значении выражения), выдать заданное сообщение.

<Сообщение> ::=
[<Метка> :] [**POSTPONED**] **ASSERT** <Выражение> [**REPORT** <Сообщение>]
[**SEVERITY** <Тип сообщения>];

Пример сообщения:

```
ASSERT D ='1'
REPORT "Ошибка, D не равно 1";
```

Для использования устройства в виде компонента другого устройства необходимо объявить его в декларативной части конструкции ARCHITECTURE:

Использования компонентов

Для использования устройства в виде компонента другого устройства необходимо объявить его в декларативной части конструкции ARCHITECTURE:

```
<Объявление компонента> ::=
COMPONENT <Имя компонента>
GENERIC          <Список настроечных параметров>;
PORT             <Список портов>;
END COMPONENT;
```

После этого можно создать экземпляр компонента в описательной части ARCHITECTURE при помощи конструкции использования:

```
<ЭКЗЕМПЛЯР компонента> ::=
<Метка экземпляра>: <Имя компонента>
GENERIC MAP ( <Значения настроечных констант> )
PORT MAP ( <Список соответствий сигналов> );
```

```
-- Пример использования компонент
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;
ENTITY adder IS
      GENERIC ( n:                          INTEGER:=16);
      PORT (      a,b: IN std_logic_vector (n-1 DOWNTO 0);
                  result: OUT std_logic_vector(n-1 DOWNTO 0));
END adder;
ARCHITECTURE behave OF adder IS
BEGIN
      result <= a + b;
END;
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.ALL;
ENTITY adders IS
      GENERIC(   op_n:       INTEGER := 16;
              sum_n:          INTEGER :=16);
```

```
PORT(   a,b,c: IN std_logic_vector(op_n-1 DOWNTO 0);
                      result: OUT std_logic_vector(sum_n-1 DOWNTO 0));
END adders;
ARCHITECTURE behave OF adders IS
      COMPONENT adder
                         GENERIC ( n:INTEGER:=op_n);
            PORT ( a,b: IN std_logic_vector(n-1 DOWNTO 0);
                 result: OUT std_logic_vector(n-1 DOWNTO 0));
      END COMPONENT;
      SIGNAL rez: std_logic_vector(sum_n-1 DOWNTO 0);
BEGIN
      a1:adder
            PORT MAP (      a => a, b=>b, result=>rez);
      a2:adder
            PORT MAP (      a=>rez,
                                               b=>c,
                      result => result);
END behave;
```

Оператор GENERATE

Оператор GENERATE обеспечивает возможность итеративного или условного повторения деклараций и операторов.

<Генерация> ::=
[<Метка>:]<Способ генерации> **GENERATE**
[<Декларации> **BEGIN**]
[<Параллельные операторы>]
END GENERATE [<Метка>];
<Способ генерации> ::= **FOR** <идентификатор> **IN** <диапазон> | **IF** <условие>

Пример конструкции GENERATE:

```
Gen: BLOCK --двоичное дерево
BEGIN
    L1: CELL PORT MAP (Top, Bus(1), Bus(2)) ; --Top,Left,Right
    L2: FOR I IN 1 to 7 GENERATE
        L3: FOR J IN 0 TO 2**I-1 GENERATE
        L5: CELL PORT MAP (Bus(2**I+J-1), Bus((2**I+J)*2-1), Bus(2**I+J)*2);
        END GENERATE ;
    END GENERATE ;
END BLOCK Gen;
```

Последовательные операторы

Последовательные операторы используются для описания алгоритма функционирования параллельных процессов и подпрограмм. Их исполнение происходит в той последовательности, в которой они описаны в программе. Последовательными операторами являются:

- оператор WAIT;
- оператор – ловушка ASSERT;
- оператор сообщения REPORT;
- оператор присваивания сигнала и оператор условного присваивания сигнала;
- оператор присваивания переменной;
- вызов процедуры;
- оператор IF;
- оператор CASE;
- оператор цикла LOOP;
- оператор NEXT;
- оператор EXIT;
- оператор RETURN;
- пустой оператор NULL.

Оператор ожидания WAIT

Оператор ожидания WAIT позволяет приостановить исполнение процесса или подпрограммы.

<Ожидание> ::=
[<Метка :] **WAIT** [**ON** <Список чувствительности>] [**UNTIL** <Условие>] [**FOR** <Задержка>];

Пример использования оператора WAIT:

```
WAIT ON Clk;
```

Оператор-ловушка ASSERT

<Ловушка> ::=
[<Метка> :] **ASSERT** <Выражение> [**REPORT** <Сообщение>] [**SEVERITY** <Тип сообщения>];

Пример использования ASSERT и REPORT:

```
ASSERT (IRDY_N = '0')
REPORT "Target device: FRAME signal deassertion error. IRDY is not asserted."
SEVERITY ERROR;
```

Последовательный оператор присваивание сигнала

Последовательный оператор присваивание сигнала служит для изменения значения сигнала и используется внутри процесса.

<Присваивание сигнала> ::=
[<метка> :] <сигнал> <= [<способ задержки>] <образец> ;
<способ задержки> ::= **TRANSPORT**
|[**REJECT** <время>] **INERTIAL**
<образец> ::=
<выражение> [**AFTER** <время>]|| **NULL** [**AFTER** <время>]

Пример присваивания сигнала:

```
P3 <= P1 AFTER Delay;
Output_pin <= TRANSPORT Input_pin AFTER 10 ns;
```

Присваивание значения переменной

Присваивание значения переменной подобно операторам присваивания в языках программирования. Синтаксис оператора присваивания переменной:

<Присваивание переменной> ::=
[<метка> :] <Переменная> := <Выражение>;

Пример присваивания переменной:

```
i  := i - 1;
```

Оператор вызова подпрограммы

Оператор вызова подпрограммы служит для исполнения алгоритма, описанного в процедуре или функции:

<Вызов подпрограммы> ::=
[<Метка> :] <Имя процедуры или функции> [(<Список формальных параметров>)];

Условный оператор IF

Условный оператор IF позволяет описать устройство, в алгоритме работы которого по результатам проверки заданных условий разрешается исполнение одной (или нуля) групп последовательных операторов. Выполнение последовательных операторов разрешается, если заданное условие истинно.

```
<Условный оператор> ::=
[ <Метка> :] IF <Условие> THEN
      <Последовательные операторы>
{ ELSIF <Условие> THEN
      <Последовательные операторы>}
[ ELSE
      <Последовательные операторы>]
END IF [ <Метка> ];
```

Пример описания шифратора с помощью условного оператора:

```
ENTITY encoder IS
     PORT ( in1     :IN   std_logic_vector(7 DOWNTO 0);
            out1    :OUT  std_logic_vector(2 DOWNTO 0));
END encoder;
ARCHITECTURE behave OF encoder IS
BEGIN
    PROCESS (in1)
    BEGIN
        IF            in1(7) = '1' THEN out1 <= "111";
        ELSIF in1(6) = '1' THEN out1 <= "110";
        ELSIF in1(5) = '1' THEN out1 <= "101";
        ELSIF in1(4) = '1' THEN out1 <= "100";
        ELSIF in1(3) = '1' THEN out1 <= "011";
        ELSIF in1(2) = '1' THEN out1 <= "010";
        ELSIF in1(1) = '1' THEN out1 <= "001";
        ELSIF in1(0) = '1' THEN out1 <= "000";
        ELSE  out1 <= "XXX";
        END IF;
    END PROCESS;
END behave;
```

Оператор выбора CASE

Оператор выбора CASE разрешает исполнение одного из многих альтернативных блоков последовательных операторов. Выбор блока среди альтернатив выполняется при совпадении значения выражения с образцом.

<Выбор> ::=
[<метка> :]
case <выражение> **is**
when <образец> => <последовательные операторы>
{ **when** <образец> => <последовательные операторы>}
end case [<метка>] ;

Пример использования оператора выбора при описании дешифратора:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY decoder IS
      PORT (   inp:  IN std_logic_vector(2 DOWNTO 0);
               outp: OUT std_logic_vector(7 DOWNTO 0));
END decoder;
```

```
ARCHITECTURE behave OF decoder IS
  BEGIN
  PROCESS (inp) BEGIN
    CASE inp IS
        WHEN "000"  => outp <= "00000001";
        WHEN "001"  => outp <= "00000010";
        WHEN "010"  => outp <= "00000100";
        WHEN "011"  => outp <= "00001000";
        WHEN "100"  => outp <= "00010000";
        WHEN "101"  => outp <= "00100000";
        WHEN "110"  => outp <= "01000000";
        WHEN "111"  => outp <= "10000000";
        WHEN OTHERS => outp <= "XXXXXXXX";
    END CASE;
  END PROCESS;
END behave;
```

Оператор цикла

Оператор цикла позволяет описать часть устройства, алгоритм работы которой представляет из себя повторяющиеся действия.

```
<Цикл> ::=
[ <Метка> : ]
[ WHILE <Условие> | FOR <Идентификатор> IN <Диапазон>]
LOOP
      <Последовательные операторы>
END LOOP [ <Метка> ];
```

Пример использования оператора цикла при описании шифратора:

```
ENTITY encoder IS
   PORT ( a, b, c, d, e, f, g, h : IN std_logic;
            out2, out1, out0 : OUT std_logic);
END encoder;
ARCHITECTURE behave OF encoder IS
BEGIN
 PROCESS (a, b, c, d, e, f, g, h)
      VARIABLE inputs : std_logic_vector (7 DOWNTO 0);
      VARIABLE i : INTEGER ;
```

```
  BEGIN
    INPUTS := (h, g, f, e, d, c, b, a);
    i := 7;
    WHILE i >= 0 AND inputs(i) /= '1' LOOP
      i  := i - 1;
    END LOOP;
    IF ( i < 0) THEN
       i := 0;
    END IF;
     -- conv_std_logic_vector (i, 3) - функция преобразования
     -- переменной типа integer в сигнал типа std_logic_vector
     -- Второй аргумент определяет размер вектора.
   (out2, out1, out0) <= conv_std_logic_vector (i, 3);
 END process;
END behave;
```

Пример описание устройств с тремя состояниями выходов

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- Синтезируемое описание простого выходного буфера (драйвера)
ENTITY tristate1 IS
      PORT ( input, enable : IN std_logic;
            output : OUT std_logic) ;
END tristate1 ;
ARCHITECTURE single_driver OF tristate1 IS
BEGIN
      output <= input WHEN enable = '1' ELSE 'Z' ;
END single_driver ;
-- Синтезируемое описание драйвера шины
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY tristate2 IS
      PORT ( input3, input2, input1, input0: IN std_logic_vector (7 DOWNTO 0);
           enable : IN std_logic_vector (3 DOWNTO 0);
           output : OUT std_logic_vector (7 DOWNTO 0) );
END tristate2 ;
ARCHITECTURE multiple_drivers of tristate2 IS
BEGIN
```

```
output <= input3 WHEN enable(3) = '1' ELSE "ZZZZZZZZ" ;
      output <= input2 WHEN enable(2) = '1' ELSE "ZZZZZZZZ" ;
      output <= input1 WHEN enable(1) = '1' ELSE "ZZZZZZZZ" ;
      output <= input0 WHEN enable(0) = '1' ELSE "ZZZZZZZZ" ;
END multiple_drivers;
-- Синтезируемое описание шинного формирователя
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY bidir IS
   PORT ( input_val, enable: IN std_logic;
            output_val : OUT std_logic;
            bidir_port : INOUT std_logic) ;
END bidir ;
ARCHITECTURE tri_state OF bidir IS
BEGIN
      bidir_port <= input_val WHEN enable = '1' ELSE 'Z' ;
      output_val <= bidir_port;
END tri_state;
```

Пример описания ОЗУ

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE ieee.std_logic_unsigned.ALL;
ENTITY MEM_8_16_BE IS
  PORT (
    A : IN std_logic_vector(2 DOWNTO 0);
    BE: IN std_logic_vector(1 DOWNTO 0);
    WE: IN std_logic;
    CLK: IN std_logic;
    DIN: IN std_logic_vector(15 DOWNTO 0);
    DOUT: OUT std_logic_vector(15 DOWNTO 0)
    );
END MEM_8_16_BE;
ARCHITECTURE BEHAV OF MEM_8_16_BE IS
TYPE MEM_8_16 IS ARRAY (0 TO 7) OF std_logic_vector(7 DOWNTO 0);
SIGNAL RAM0: MEM_8_16;
SIGNAL RAM1: MEM_8_16;
SIGNAL D:std_logic_vector(15 DOWNTO 0);
```

```
BEGIN
OUT0: PROCESS (CLK,WE,A,BE)
BEGIN
  IF (CLK'EVENT AND CLK='1') THEN
    IF ((WE = '1' OR WE = 'H') AND BE(0)='0') THEN
    RAM0(conv_integer(unsigned(A)))(7 DOWNTO 0) <= DIN(7 DOWNTO 0);
        END IF;
  END IF;
END PROCESS OUT0;
DOUT(7 DOWNTO 0) <= RAM0(conv_integer(unsigned(A)))(7 DOWNTO 0);
OUT1: PROCESS (CLK,WE,A,BE)
BEGIN
  IF (CLK'EVENT AND CLK='1') THEN
    IF ((WE = '1' OR WE = 'H') AND BE(1)='0') THEN
    RAM1(conv_integer(unsigned(A)))(7 DOWNTO 0) <= DIN(15 DOWNTO 8);
        END IF;
  END IF;
END PROCESS OUT1;
DOUT(15 DOWNTO 8) <= RAM1(conv_integer(unsigned(A)))(7 DOWNTO 0);
END behav;
```

Правила описания синтезируемых конструкций на VHDL

- Необходимо максимально использовать поведенческий стиль описания. При структурном описании нижние модули иерархии должны выполнять законченные функции. Обычно поведенческое описание модуля содержит 30 – 500 строк кода или от 1 до 10 процессов.

- Процесс, по возможности должен содержать один – два оператора на верхнем уровне.

- Для создания триггера, защелкиваемого по перепаду сигнала можно использовать условие CLK'event and CLK='1' или CLK'event and CLK='0' (в первом случае отслеживается фронт, во втором спад.) Это условие используется совместно с оператором IF, причем если оно следует после ELSIF то после IF этого же оператора обычно описывается условие асинхронного сброса. Для всех сигналов, имеющих присвоения внутри ветки с условием, создаются триггеры

Правила описания синтезируемых конструкций на VHDL

- Использование условий (CLK’event and CLK=’1’) или (CLK’event and CLK=’0’) в двух и более ветках оператора IF означает что описывается триггер с двумя тактовыми сигналами. Таких триггеров в ПЛИС нет, а применение таких триггеров в ASIC нежелательно.
- Все последовательные описания синтезируется в комбинационную схему, если описание заключено в ветку IF с условием (CLK’event and CLK=’1’) или (CLK’event and CLK=’0’) , то к выходу такой комбинационной схемы подключается один или несколько триггеров (в зависимости от количества сигналов, используемых в описании и их разрядности).
- Если последовательное описание содержит исключительно асинхронную логику, то необходимо, чтобы используемые в нем сигналы имели назначение в каждой из альтернативных веток. Отсутствие такого назначения приводит к тому, что при выполнении этой ветки сигнал должен принимать свое старое значение). На самом деле для таких сигналов синтезируются триггеры, управляемые уровнем (LATCH).

Правила описания синтезируемых конструкций на VHDL

Latch- это триггер, управляемый уровнем разрешающего сигнала (см. рис 2.1). Такой триггер формируется при использовании описаний, подобных представленному на следующем листинге:

```
Latch_Data: process(Latch_Open, D_Input)
begin
if (Latch_Enable = '1') then
        Q <= D;
-- If Latch_Enable = 0, then Q  сохраняет свое старое значение,
-- то есть Latch закрыта.
end if;
end process Latch_Data;
```

Использование LATCH - триггеров имеет следующие недостатки:

•В случае появления помех, вызванных гонками сигналов, на линии данных, в момент, когда защелка открыта, эти помехи проходят на выход.

•Помехи сигнала разрешения, также проходят на выход.

•Обычно LATCH выполняются на последовательной логике путем добавления обратных связей, а обычный триггер, тактируемый перепадом является готовым законченным узлом внутри ПЛИС и занимает меньше места.

•Путь сигнала проходящего через LACH плохо поддается временному анализу.

Правила описания синтезируемых конструкций на VHDL

•Операторы IF, WHEN/ELSE приводят к созданию приоритетной логики.
•Операторы CASE и WITH/SELECT/WHEN, приводят к созданию безприоритетной логики.
•Количество используемых LUT и уровень их каскадирования для реализации некоторой функции можно оценить следующим образом: 1) определяется количество входов функции n, 2) определяется минимальное количество каскадов LUT]log4n[. (для n=1..4 – 1 уровень, для n=5..16 – 2 уровня LUT).

Правила описания синтезируемых конструкций на VHDL

•Тактовые сигналы имеют обычно иные электрические характеристики, чем сигналы дынных. Это связано с большим коэффициентом разветвления тактового сигнала, а так же с необходимостью одновременного тактирования нескольких триггеров, что обуславливается минимальным временем задержки на линии тактового сигнала. Поэтому для тактирования используются только тактовые сигналы, а не линии данных. То есть с выхода триггера нельзя брать тактовый сигнал. Если все-таки необходимо тактировать схему с выхода триггера, то между ним и тактовым входом другого триггера вставляется специальный буфер. Тип и название буфера определяется технологией ПЛИС (или ASIC), для XILINX, в библиотеке UNISIM имеется буфер GCLK.
•Тактирование схемы должно быть однокаскадным. Однако это не мешает строить несколько схем (модулей), с тактированием разными тактовыми сигналами (тактовыми доменами).
•Если необходимо управлять триггером более медленным тактовым сигналом, кратным основному тактовому, можно использовать сигнал разрешения тактирования.