第 5 章

CHAPTER 5

VHDL 设计进阶

第 4 章中，通过几个典型的实例，对 VHDL 的结构、某些语言规则和语句类型等做过部分针对性的介绍，本章将对 VHDL 的语言规则和语句类型等做出更系统的叙述，对于在第 4 章中已出现过的内容，本章根据实际需要仅做简要介绍或归纳。

5.1 VHDL 要素

VHDL 具有计算机编程语言的一般特性，其语言要素是编程语句的基本单元，是 VHDL 作为硬件描述语言的基本结构元素，反映了 VHDL 重要的语言特征。准确无误地理解和掌握 VHDL 的语言要素的基本含义和用法，对于正确地完成 VHDL 程序设计十分重要。

VHDL 的语言要素主要有 VHDL 文字规则、数据对象(Data Objects)、数据类型(Data Type)、各类操作数(Operands)和运算操作符(Operator)等。

5.1.1 VHDL 文字规则

VHDL 除了具有类似于计算机高级语言所具有的一般文字规则外，还包含许多特有的文字规则和表达方式，在编程中需认真遵循。

1. 数字

数字型文字的值有多种表达方式，现列举如下。

(1) 整数文字：整数文字都是十进制的数。例如：

```
5,678,0,156E2( = 15600),12_345_678( = 12345678)
```

注意：数字间的下画线仅仅是为了提高文字的可读性，相当于一个空的间隔符，而没有其他的意义，因而不影响文字本身的数值。

(2) 实数文字：实数文字也都是十进制的数，但必须带有小数点。例如：

```
188.993,88_670_551.453_909( = 88670551.453909),1.0,44.99E - 2( = .4499)
```

(3) 以数制基数表示的文字：用这种方式表示的数由五个部分组成，可以表示为：

```
基数 # 基于该基的整数[.基于该基的整数]#E 指数
```

其中，第一部分是用十进制数标明数制进位的基数；第二部分是数制隔离符号“#”；

第三部分是表达的文字，可以为整数，也可以为实数；第四部分是指数隔离符号“#”；第五部分用字符“E”加十进制表示的指数部分，这一部分的数如果是 0 则可以省去不写。现举例如下：

```
10#254#              --(十进制数表示,等于 254)
2#1111_1110#         --(二进制数表示,等于 254)
16#FE#               --(十六进制数表示,等于 254)
8#376#               --(八进制数表示,等于 254)
```

（4）物理量文字（VHDL 综合器不接受此类文字）。例如：

```
60s(60 秒),100m(100 米),1kΩ(1000 欧姆),10A(10 安培)
```

2. 字符与字符串

字符是用单引号引起来的 ASCII 字符，可以是数值，也可以是符号或字母，如‘R’，‘A’，‘*’，‘0’。而字符串则是一维的字符数组，须放在双引号中。VHDL 中有两种类型的字符串：文字字符串和数位字符串。

（1）文字字符串：文字字符串是用双引号引起来的一串文字。例如：

```
"ERROR","BOTH S AND Q EQUAL TO L","X","BB$CC"
```

（2）数位字符串：数位字符串也称位矢量，是用字符形式表示的多位数码，它们所代表的是二进制、八进制或十六进制的数组，其位矢量的长度即为等值的二进制数的位数。数位字符串的表示首先要有计数基数，然后将该基数表示的值放在双引号中，基数符以 B、O 和 X 表示，并放在字符串的前面。它们的含义分别如下。

B：二进制基数符号，表示二进制数位 0 或 1，在字符串中每一个位表示一个 BIT。

O：八进制基数符号（0～7），在字符串中的每一个数代表一个八进制数，即代表一个 3 位（BIT）的二进制数。

X：十六进制基数符号（0～F），在字符串中的每一个数代表一个十六进制数，即代表一个 4 位的二进制数。

例如：

```
B"1_1101_1110"       -- 二进制数数组,位矢数组长度是 9
O"15"                -- 八进制数数组,位矢数组长度是 6
X"AD0"               -- 十六进制数数组,位矢数组长度是 12
```

3. 标识符

标识符是 VHDL 中各种成分的名称，这些成分包括常量、变量、信号、端口、子程序或参数等。定义标识符需要遵循以下规则。

（1）有效的字符：包括 26 个大小写英文字母，数字 0～9 以及下画线“_”。

（2）任何标识符必须以英文字母开头。

（3）必须是单一下画线“_”，且其前后都必须有英文字母或数字。

（4）标识符中的英文字母不分大小写。

（5）允许包含图形符号（如回车符、换行符等），也允许包含空格符。

（6）VHDL 的保留字不能用作标识符。

以下是几种合法和非法标识符的示例。

合法的标识符：Decoder_1，FFT，abc123。

非法的标识符：

```
_Decoder_1                    -- 起始为非英文字母
2 FET                         -- 起始为数字
Not - RST                     -- 符号" - "不能作为标识符的构成
RyY_RST_                      -- 标识符的最后不能是下画线
Data_ _BUS                    -- 标识符中不能有双下画线
Begin                         -- 关键字不能作为标识符
resΩ                          -- 使用了无效字符"Ω"
```

4. 下标名及下标段名

下标名用于指示数组型变量或信号的某一元素，而下标段名则用于指示数组型变量或信号的某一段元素，其语句格式如下。

```
数组类型信号名或变量名(表达式 1[TO/DOWNTO 表达式 2]);
```

表达式的数值必须在数组元素下标范围以内，并且必须是可计算的。TO 表示数组下标序列由低到高，如“ 2 TO 8”；DOWNTO 表示数组下标序列由高到低，如“ 8 DOWNTO 2”。

如果表达式是一个可计算的值，则此操作数可很容易地进行综合。如果是不可计算的，则只能在特定的情况下综合，且耗费资源较大。

下面是下标名及下标段名使用示例。

```
SIGNAL a,b,c: BIT_VECTOR(0 TO 7);
SIGNAL m:    INTEGER RANGE 0 TO 3;
SIGNAL y,z: BIT;
y <= a(m);                    -- m 是不可计算型下标表示
z <= b(3);                    -- 3 是可计算型下标表示
c(0 TO 3)<= a(4 TO 7);        -- 以段的方式进行赋值
c(4 TO 7)<= a(0 TO 3);        -- 以段的方式进行赋值
```

5.1.2 VHDL 数据对象

尽管信号和变量在第 4 章的示例中已出现多次，但没有做更详细的解释，为了更好地理解 VHDL 程序，以下对它们做进一步的说明。

在 VHDL 中，凡是可以赋予一个值的对象就称为数据对象(Data Objects)，它类似于一种容器，可接受不同数据类型的赋值。在 VHDL 中，数据对象有三种，即常量(CONSTANT)、变量(VARIABLE)和信号(SIGNAL)。前两种数据对象可以从传统的计算机高级语言中找到对应的数据类型，其语言行为与高级语言中的常量和变量十分相似。但信号的表现较为特殊，它是具有更多的硬件特征的特殊数据对象，是 VHDL 中最有特色的语言要素之一。

1. 常量

常量(CONSTANT)就是指在设计实体中不会发生变化的值，它可以在很多部分进行说明，并且可以是任何的数据类型。常量的定义和设置主要是为了使设计实体中的常数更容易阅读和修改。例如，将逻辑位的宽度定义为一个常量，只要修改这个常量就能很容易改

变宽度，从而改变硬件结构。在程序中，常量是一个恒定不变的值，一旦做了数据类型的赋值定义后，在程序中不能再改变，因而具有全局意义。常量的定义形式如下。

```
CONSTANT 常量名: 数据类型[: =表达式];
```

例如：

```
CONSTANT  fbt: STD_LOGIC_VECTOR : ="010110";    --标准位矢类型
CONSTANT  vcc: REAL: =5.0;                      --实数类型
CONSTANT dely: TIME: =25ns;                     --时间类型
```

VHDL要求所定义的常量数据类型必须与表达式的数据类型一致。如果常量的定义形式写成：

```
CONSTANT  vcc: REAL: =25ns;
```

这样的定义显然是错误的。

常量定义语句所允许的设计单元有实体、结构体、程序包、块、进程和子程序。在程序包中定义的常量可以暂不设具体数值，它可以在程序包体中设定。

使用时，注意常量的可视性，即常量的使用范围取决于它被定义的位置。在程序包中定义的常量具有最大全局化特征，可以用在调用此程序包的所有设计实体中；定义在设计实体中的常量，其有效范围为这个实体定义的所有结构体；定义在设计实体的某一结构体中的常量，则只能用于此结构体；定义在结构体的某一单元的常量，如一个进程中，则这个常量只能用在这一进程中。这就是常量的可视性规则。

2. 变量

变量(VARIABLE)是指在设计实体中会发生变化的值。在VHDL语法规则中，变量是一个局部量，只能在进程和子程序中使用。变量不能将信息带出对它做出定义的当前结构体。变量的赋值是理想化的数据传输，是立即发生的，不存在任何延时的行为。变量的主要作用是在进程中作为临时的数据存储单元。定义变量的语法格式如下。

```
VARIABLE 变量名: 数据类型[: =初始值];
```

例如：

```
VARIABLE    a   : INTEGER    RANGE    0  TO  15;
VARIABLE    b,c : INTEGER: =2;
VARIABLE    d   : STD_LOGIC;
```

分别定义a的取值范围为0～15的整数型变量；b和c为初始值为2的整型变量；d为标准位类型的变量。

变量作为局部量，其适用范围仅限于定义了变量的进程或子程序的顺序语句中，在这些语句结构中，同一变量的值将随着变量赋值语句的运算而改变。

变量定义语句中的初始值可以是一个与变量具有相同数据类型的常数值，也可以是一个全局静态表达式，这个表达式的数据类型必须与所赋值的变量一致。此初始值不是必需的，由于硬件电路上电后的随机性，因此综合器并不支持设置初始值。变量赋值的一般表达式如下。

```
目标变量名: = 表达式;
```

注意：变量赋值符号是“：＝”，变量数值的改变是通过变量赋值来实现的。赋值语句右方的“表达式”必须是一个与“目标变量名”具有相同数据类型的数值，这个表达式可以是一个运算表达式，也可以是一个数值。通过赋值操作，新的变量值的获得是立刻发生的。变量赋值语句左边的目标变量可以是单值变量，也可以是一个变量的集合，如位矢量类型的变量。例如：

```
VARIABLE x,y: REAL;
VARIABLE a,b: STD_LOGIC_VECTOR(7 DOWNTO 0)
  x: = 100.0;                    -- 实数赋值,x 是实数变量
  y: = 1.5 + x;                  -- 运算表达式赋值,y 也是实数变量
  a: = "10111011";               -- 位矢量赋值
  a(0 TO 5): = b(2 TO 7);        -- 段赋值
```

3. 信号

信号(SIGNAL)是描述硬件系统的基本数据对象，它类似于电子电路内部的连接线。信号可以作为设计实体中并行语句模块间的信息交流通道。在 VHDL 中，信号及其相关的信号赋值语句、决断函数、延时语句等很好地描述了硬件系统的许多基本特征。如硬件系统运行的并行性、信号传输过程中的惯性延时特性、多驱动源的总线行为等。

信号作为一种数值容器，不但可以容纳当前值，也可以保持历史值。这一属性与触发器的记忆功能有很好的对应关系。信号的定义格式如下。

```
SIGNAL 信号名: 数据类型[: = 初始值];
```

信号初始值的设置不是必需的，而且初始值仅在 VHDL 的行为仿真中有效。与变量相比，信号的硬件特征更为明显，它具有全局性特征。例如，在程序包中定义的信号，对于所有调用此程序包的设计实体都是可见的；在实体中定义的信号，在其对应的结构体中都是可见的。

事实上，除了没有方向说明以外，信号与实体的端口概念是一致的。对于端口来说，其区别只是输出端口不能读入数据，输入端口不能被赋值。信号可以看成是实体内部的端口。反之，实体的端口只是一种隐形的信号，端口的定义实际上是做了隐式的信号定义，并附加了数据流动的方向。信号本身的定义是一种显式的定义，因此，在实体中定义的端口，在其结构体中都可以看成一个信号，并加以使用而不必另做定义。以下是信号的定义示例。

```
SIGNAL s1: STD_LOGIC: = '0';       -- 定义了一个标准位的单值信号 s1,初始值为低电平
SIGNAL s2,s3: BIT;                 -- 定义了两个位(BIT)的信号 s2 和 s3
SIGNAL s4: STD_LOGIC_VECTOR(15 DOWNTO 0);
  -- 定义了一个标准位矢的位矢量(数组、总线)信号,共有 16 个信号元素
```

信号的使用和定义范围是实体、结构体和程序包。在进程和子程序中不允许定义信号。在进程中，只能将信号列入敏感表，而不能将变量列入敏感表。可见进程只对信号敏感，而对变量不敏感。

当信号定义了数据类型和表达方式后，在 VHDL 中就能对信号进行赋值了。信号的赋

值语句表达式如下。

```
目标信号名<=表达式;
```

这里的“表达式”可以是一个运算表达式，也可以是数据对象（变量、信号或常量）。数据信息的传入可以设置延时量，因此目标信号获得传入的数据并不是即时的。即使是零延时（不做任何显式的延时设置），也要经历一个特定的延时，即 δ 延时。因此，符号“<=”两边的数值并不总是一致的，这与实际器件的传播延迟特性是吻合的，因此，信号赋值与变量赋值的过程有很大差别。

下面列举了几个信号赋值的语句。

```
a<=y;
a<='1';
s1<=s2 AFTER 10ns;
```

这里 a、s1、s2 均为信号。AFTER 后面是延迟时间，即 s2 经过 10ns 的延迟后，其值才赋值到 s1 中，这一点是与变量完全不同的。

信号的赋值可以出现在一个进程中，也可以出现在结构体的并行语句结构中，但它们运行的含义是不一样的。前者属于顺序信号赋值，这时的信号赋值操作要视进程是否已被启动。后者属于并行信号赋值，其赋值操作是各自独立并行地发生的。

在进程中，可以允许同一信号有多个驱动源，即在同一进程中存在多个同名的信号被赋值，其结果只有最后的赋值语句被启动，并进行赋值操作，例如：

```
SIGNAL a,b,c,x,y: INTEGER;
…
PROCESS(a,b,c)
BEGIN
  x<=a*b;
  y<=c-a;
  x<=b;
```

上例中，信号 a、b、c 被列入进程表，当进程被启动后，信号赋值将自上而下顺序执行，但第一项赋值操作并不会发生，这是因为 x 的最后一项驱动源是 b，因此 x 被赋值为 b。但在并行赋值语句中，不允许同一信号有多个驱动源的情况。

4. 信号与变量的区别

信号与变量都是 VHDL 中的重要对象，由于它们存在某些相似之处，因此，人们在使用时常常将两者混淆。下面讨论两者之间存在的区别。

（1）信号赋值至少有 δ 延时，而变量赋值没有延时。

（2）信号除当前值外有许多相关的信息，而变量只有当前值。

（3）进程对信号敏感而对变量不敏感。

（4）信号可以是多个进程的全局信号，而变量只在定义它们的顺序域可见（共享变量除外）。

（5）信号是硬件中连线的抽象描述，它们的功能是保存变化的数据和连接子元件，信号在元件的端口连接元件。变量在硬件中没有类似的对应关系，它们用于硬件特性的高层次建模所需要的计算中。

(6) 信号赋值和变量赋值分别使用不同的赋值符号“<=”和“:=”,信号类型和变量类型可以完全一致,也允许两者之间相互赋值,但要保证两者的类型相同。

关于信号和变量赋值的区别的具体示例见 5.2 节。

5.1.3 VHDL 数据类型

VHDL 有很强的数据类型,它对运算关系与赋值关系中各操作数的数据类型有严格要求,它要求设计实体中的每一个常量、信号、变量、函数以及设定的各种参量都必须具有确定的数据类型,只有相同数据类型的量才能相互传递和作用。VHDL 作为强类型语言的好处是使 VHDL 编译或综合工具很容易找出设计中的各种常见错误。VHDL 中的数据类型可以分成以下四大类。

(1) 标量类型(Scalar Type):是最基本的数据类型,通常用于描述一个单值数据对象,包括实数类型、整数类型、枚举类型和物理类型。

(2) 复合类型(Composite Type):可以由小的数据类型复合而成,如可由标量类型复合而成。复合类型主要有数组型和记录型。

(3) 存取类型(Access Type):为给定的数据类型的数据对象提供存取方式。

(4) 文件类型(File Type):用于提供多值存取类型。

这些数据类型又可分成在现成程序包中可以随时获得的预定义数据类型和用户自定义数据类型两大类别。预定义的 VHDL 的数据类型是 VHDL 最常用、最基本的数据类型。这些数据类型都已在 VHDL 的标准程序包 STANDARD 和 STD_LOGIC_1164 及其他的标准程序包中做了定义,并可在设计中随时调用。

VHDL 还允许用户自己定义其他的数据类型及子类型。通常,新定义的数据类型和子类型的基本元素一般仍属于 VHDL 的预定义类型。

1. VHDL 的预定义数据类型

VHDL 的预定义数据类型都是在 VHDL 标准程序包 STANDARD 中定义的,在实际使用中,它会自动包含进 VHDL 的源文件中,因而不必通过 USE 语句以显式调用。

1) 布尔(BOOLEAN)数据类型

布尔数据类型常用来表示信号的状态或者总线上的情况,它实际上是一个二值枚举型数据类型,它的取值有 FALSE 和 TRUE 两种。综合器将用一个二进制位表示 BOOLEAN 型变量或信号。布尔量没有数值含义,不能进行算术运算,但可以进行关系运算。

例如,当 a 大于 b 时,在 IF 语句中的关系运算表达式(a > b)的结果是布尔量 TRUE,反之为 FALSE。综合器将其变为 1 或 0 信号值。

程序包 STANDARD 中定义布尔数据类型的源代码如下。

```
TYPE BOOLEAN IS(FALSE,TRUE);
```

2) 位(BIT)数据类型

位数据类型也属于枚举型,取值只能是‘1’或‘0’。这与整数中的 1 或 0 不同,‘1’和‘0’只表示一个位的两种取值。位数据类型的数据对象,如变量、信号等,可以参与逻辑运算,运算结果仍是位的数据类型。VHDL 综合器用一个二进制位表示 BIT。在程序包 STANDARD 中定义的源代码是:

```
TYPE BIT  IS('0','1');
```

下面是几个关于位类型的例子。

```
CONSTANT c: BIT : = '1';          -- 值为 1 的位类型常量 c
VARIABLE q: BIT : = '0';          -- 值为 0 的位类型变量 q
SIGNAL a,b: BIT;                  -- 两个位类型的信号
```

3）位矢量(BIT_VECTOR)数据类型

位矢量只是基于 BIT 数据类型的数组，它是使用双引号括起来的一组位数据，如“10110101”。在程序包 STANDARD 中定义的源代码是：

```
TYPE BIT_VECTOR IS ARRAY(Natural Range <>)OF BIT;
```

使用位矢量必须注明位宽，即数组中的元素个数和排列，例如：

```
SIGNAL a: BIT_VECTOR(7 TO 0);
```

信号 a 被定义为一个具有 8 位位宽的矢量，它的最左位是 a(7)，最右位是 a(0)。

使用位矢量数据可以形象地表示总线的状态。

4）字符(CHARACTER)数据类型

字符类型通常用单引号引起来，如‘A’。字符类型区分大小写，如‘B’不同于‘b’。字符类型也已在 STANDARD 程序包中做了定义，在 VHDL 程序设计中，标识符的大小写一般是不分的，但用了单引号的字符的大小是有区别的。

5）整数(INTEGER)数据类型

整数类型的整数与数学中的定义相同，但是它的描述是有范围的。在 VHDL 中，整数的取值范围是－2 147 483 647～＋2 147 483 647，即可用 32 位有符号的二进制数表示，范围为$-(2^{31}-1)\sim(2^{31}-1)$。在实际应用中，VHDL 仿真器通常将 INTEGER 类型作为有符号数处理，而 VHDL 综合器则将 INTEGER 作为无符号数处理。在使用整数时，VHDL 综合器要求用 RANGE 子句为所定义的数限定范围，然后根据所限定的范围来决定表示此信号或变量的二进制数的位数，因为 VHDL 综合器无法综合未限定的整数类型的信号或变量。

如语句“SIGNAL type1：INTEGER RANGE 0 TO 15；”规定整数 type1 的取值范围是 0～15 共 16 个值，可用 4 位二进制数来表示，因此，type1 将被综合成由 4 条信号线构成的信号。

不同进制整数常量的书写方式示例如下。

```
2                                 -- 十进制整数
10E4                              -- 十进制整数
16#D2#.                           -- 十六进制整数
2#11011010#                       -- 二进制整数
```

6）自然数(NATURAL)和正整数(POSITIVE)数据类型

自然数是整数的一个子类型，非负的整数，即零和正整数；正整数也是整数的一个子类型，包括整数中非零和非负的数值。它们在 STANDARD 程序包中定义的源代码如下。

```
SUBTYPE NATURAL IS INTEGER RANGE 0 TO INTEGER'HIGH;
```

```
SUBTYPE POSITIVE IS INTEGER RANGE 1 TO INTEGER'HIGH;
```

7）实数(REAL)数据类型

VHDL的实数类型类似于数学上的实数，或称浮点数。实数的取值范围为－1.0E38～＋1.0E38。通常情况下，实数类型仅能在VHDL仿真器中使用，VHDL综合器不支持实数，因为实数类型的实现相当复杂，目前在电路规模上难以承受。

不同进制实数常量的书写方式举例如下。

```
－1.0                          -- 十进制实数
65971.333333                   -- 十进制实数
8＃43.6＃E＋4                  -- 八进制实数
43.6E－4                       -- 十进制实数
```

有些数可以用整数表示也可以用实数表示。例如，数字1的整数表示为1，而用实数表示则为1.0。两个数的值是一样的，但数据类型却不一样。在实际应用中不能把一个实数赋予一个整数变量，这样会造成数据类型不匹配。

8）字符串(STRING)数据类型

字符串数据类型是字符数据类型的一个非约束型数组，或称为字符串数组。字符串必须用双引号标明，VHDL综合器支持字符串数据类型。字符串数据类型示例如下。

```
VARIABLE string_var : STRING(1 TO 7);
…
string_var: "a b c d";
```

9）时间(TIME)数据类型

VHDL中唯一的预定义物理类型是时间。完整的时间类型包括整数和物理量单位两部分。整数和单位之间至少留一个空格，如55ms、20ns。STANDARD程序包中也定义了时间，定义如下。

```
TYPE time IS RANGE －2147483647 TO 2147483647
   units
   fs:                         -- 飞秒，VHDL中的最小时间单位
   ps＝1000fs;                 -- 皮秒
   ns＝1000ps;                 -- 纳秒
   us＝1000ns;                 -- 微秒
   ms＝1000us;                 -- 毫秒
   sec＝1000ms;                -- 秒
   min＝60sec;                 -- 分
   hr＝60min;                  -- 时
   end units;
```

在系统仿真时，利用时间类型数据表示信号延时，可以使模型更接近系统的运行环境。

10）错误等级(Severity Level)

错误等级类型数据用来表征系统的状态，它共有4种：NOTE(注意)、WARNING(警告)、ERROR(出错)和FAILURE(失败)。在系统仿真过程中可以用这4种状态来提示系统当前的工作情况。这样可以使操作人员随时了解当前系统工作的情况，并根据系统的不

同状态采取相应的对策。

11）综合器不支持的数据类型

下面列举的这些数据类型虽然仿真器支持，但是综合器是不支持的。

（1）物理类型：综合器不支持物理类型的数据，如具有量纲型的数据，包括时间类型。这些类型只能用于仿真过程。

（2）浮点型：如 REAL 型。

（3）Access 型：综合器不支持存取型结构，因为不存在这样对应的硬件结构。

（4）File 型：综合器不支持磁盘文件型，硬件对应的文件仅为 RAM 和 ROM。

2. IEEE 预定义标准逻辑位与矢量

在 IEEE 库的程序包 STD_LOGIC_1164 中，定义了两个非常重要的数据类型，即标准逻辑位（STD_LOGIC）数据类型和标准逻辑矢量（STD_LOGIC_VECTOR）数据类型。

1）标准逻辑位（STD_LOGIC）数据类型

以下是定义在 IEEE 库程序包 STD_LOGIC_1164 中的 STD_LOGIC 数据类型。

```
TYPE STD_LOGIC IS('U','X','0','1','Z','W','L','H','-');
```

各值的含义是：‘U’——未初始化的，‘X’——强未知的，‘0’——强 0，‘1’——强 1，‘Z’——高阻态，‘W’——弱未知的，‘L’——弱 0，‘H’——弱 1，‘－’——忽略。

由定义可见，STD_LOGIC 是标准的 BIT 数据类型的扩展，共定义了 9 种值，这意味着，对于定义为数据类型是标准逻辑位 STD_LOGIC 的数据对象，其可能的取值已非传统的 BIT 那样只有 0 和 1 两种取值，而是如上定义的那样有 9 种可能的取值。目前在设计中一般只使用 IEEE 的 STD_LOGIC 标准逻辑的位数据类型，BIT 型则很少使用。

由于标准逻辑位数据类型的多值性，在编程时应当特别注意。因为在条件语句中，如果未考虑到 STD_LOGIC 的所有可能的取值情况，综合器可能会插入不希望的锁存器。

在程序中使用此数据类型前，需加入下面的语句。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

程序包 STD_LOGIC_1164 中还定义了 STD_LOGIC 型逻辑运算符 AND、NAND、OR、NOR、XOR 和 NOT 的重载函数及多个转换函数用于不同数据类型间的相互转换。

在仿真和综合中，STD_LOGIC 值是非常重要的，它可以使设计者精确模拟一些未知和高阻态的线路情况。对于综合器，高阻态和“－”忽略态可用于三态的描述。但就综合而言，STD_LOGIC 型数据能够在数字器件中实现的只有其中的 4 种值，即“－”“0”“1”和“Z”。当然，这并不表明其余的 5 种值不存在。这 9 种值对于 VHDL 的行为仿真都有重要意义。

2）标准逻辑矢量（STD_LOGIC_VECTOR）数据类型

STD_LOGIC_VECTOR 类型定义如下。

```
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
```

显然，STD_LOGIC_VECTOR 是定义在 STD_LOGIC_1164 程序包中的标准一维数组，数组中的每一个元素的数据类型都是以上定义的标准逻辑位 STD_LOGIC。

STD_LOGIC_VECTOR 数据类型的数据对象赋值的原则是：同位宽、同数据类型的矢

量间才能进行赋值。

使用 STD_LOGIC_VECTOR 描述总线信号是很方便的，但需注意的是总线中的每一根信号线都必须定义为同一种数据类型 STD_LOGIC。

3. 其他预定义标准数据类型

VHDL 综合工具配带的扩展程序包中，定义了一些有用的类型。如 Synopsys 公司在 IEEE 库中加入的程序包 STD_LOGIC_ARITH 中定义了如下的数据类型：无符号型(UNSIGNED)、有符号型(SIGNED)、小整型(SMALL_INT)等。

如果将信号或变量定义为这几个数据类型，就可以使用本程序包中定义的运算符。在使用之前，请注意必须加入下面的语句。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_ARITH.ALL;
```

UNSIGNED 类型和 SIGNED 类型是用来设计可综合的数学运算程序的重要类型，UNSIGNED 用于无符号数的运算，SIGNED 用于有符号数的运算。在实际应用中，大多数运算都需要用到它们。

在 IEEE 程序包中，NUMERIC_STD 和 NUMERIC_BIT 程序包中也定义了 UNSIGNED 型及 SIGNED 型，NUMERIC_STD 是针对 STD_LOGIC 型定义的，而 NUMERIC_BIT 是针对 BIT 型定义的。在程序包中还定义了相应的运算符重载函数。有些综合器没有附带 STD_LOGIC_ARITH 程序包，此时只能使用 NUMBER_STD 和 NUMERIC_BIT 程序包。

在 STANDARD 程序包中没有定义 STD_LOGIC_VECTOR 的运算符，而整数类型一般只在仿真的时候用来描述算法，或作数组下标运算，因此 UNSIGNED 和 SIGNED 的使用率是很高的。

1) 无符号数据类型(UNSIGNED TYPE)

UNSIGNED 数据类型代表一个无符号的数值，在综合器中，这个数值被解释为一个二进制数，这个二进制数的最左位是其最高位。例如，十进制的 8 可以做如下表示。

```
UNSIGNED("1000")
```

如果要定义一个变量或信号的数据类型为 UNSIGNED，则其位矢长度越长，所能代表的数值就越大。如一个 4 位变量的最大值为 15，一个 8 位变量的最大值则为 255，0 是其最小值，不能用 UNSIGNED 定义负数。以下是两个无符号数据定义的示例。

```
VARIABLE var: UNSIGNED(0 TO 10):
SIGNAL sig: UNSIGNED(5 TO 0):
```

其中，变量 var 有 11 位数值，最高位是 var(0)，而非 var(l0)；信号 sig 有 6 位数值，最高位是 sig(5)。

2) 有符号数据类型(SIGNED TYPE)

SIGNED 数据类型表示一个有符号的数值，综合器将其解释为补码，此数的最高位是符号位，用 0 代表正数，1 代表负数。例如：

SIGNED("0101")代表 +5，SIGNED("l0ll")代表 -5。

若将上例的 var 定义为 SIGNED 数据类型，则数值意义就不同了，例如：

```
VARIABLE var: SIGNED(0 TO 10);
```

其中，变量 var 有 11 位，最左位 var(0)是符号位。

4. 用户自定义数据类型

除了上述一些标准的预定义数据类型外，VHDL 还允许用户自行定义新的数据类型。由用户定义的数据类型可以有多种，如枚举类型（Enumeration Types）、整数类型（Integer Types）、数组类型（Array Types）、记录类型（Record Types）、时间类型（Time Types）、实数类型（Real Types）等。用户自定义数据类型是用类型定义语句 TYPE 来实现的。

TYPE 语句语法结构如下。

```
TYPE 数据类型名 IS 数据类型定义 OF 基本数据类型;
```

或

```
TYPE 数据类型名 IS 数据类型定义;
```

利用 TYPE 语句进行数据类型自定义有两种不同的格式，但方法是相同的。其中，数据类型名由设计者自定，此名将作为数据类型定义之用，方法与以上提到的预定义数据类型的用法一样；数据类型定义部分用来描述所定义的数据类型的表达方式和表达内容；关键词 OF 后的基本数据类型是指数据类型定义中所定义的元素的基本数据类型，一般都是已有的预定义数据类型，如 BIT、STD_LOGIC 或 INTEGER 等。

例如：

```
TYPE state0 IS ARRAY(0 TO 15) OF STD_LOGIC;
```

句中定义的数据类型 state0 是一个具有 16 个元素的数组型数据类型，数组中的每个元素的数据类型都是 STD_LOGIC 型。

下面对常用的几种用户定义的数据类型进行具体介绍。

1）枚举类型

VHDL 中的枚举数据类型是一种特殊的数据类型，它们是用文字符号来表示一组实际的二进制数。例如，状态机的每一状态在实际电路中是以一组触发器的当前二进制数位的组合来表示的，但设计者在状态机的设计中，为了更利于阅读、编译和 VHDL 综合器的优化，往往将表征每一状态的二进制数组用文字符号来代表，即所谓状态符号化。

枚举类型数据的定义格式如下。

```
TYPE 枚举数据类型名 IS(枚举元素 1,枚举元素 2,…);
```

在综合过程中，枚举类型文字元素的编码通常是自动设置的，综合器根据优化情况、优化控制的设置或设计者的特殊设定来确定各元素具体编码的二进制位数、数值及元素间编码顺序。一般情况下，编码顺序是默认的，如一般将第一个枚举量（最左边的量）编码为‘0’或“O000”等，以后的编码值依次加 1。综合器在编码过程中自动将每一枚举元素转变成位矢量，位矢量的长度根据实际情况决定。

例如：

```
TYPE state1 IS(st0,st1,st2,st3);
```

该例中用于表达4个状态的位矢量长度可以为2,编码默认值为st0="00",st1="01",st2="10",st3="11"。

一般地,编码方式也会因综合器及综合控制方式不同而不同,为了某些特殊的需要,编码顺序也可以人为设置。

2) 整数与实数类型

这里说的是用户所定义的整数类型,而不是在VHDL中已存在的整数类型。实际上这里介绍的是整数的一个子类。

整数或实数用户定义数据类型的格式为:

```
TYPE 数据类型名 IS 数据类型定义 约束范围
```

由于标准程序包中预定义的整数和实数的取值范围太大,在综合过程中,综合器很难或者无法进行综合。因此对于需要定义的整数或实数必须由用户根据需要重新定义其数据类型,限定取值范围,从而提高芯片资源的利用率。

3) 数组类型

数组类型是将一组具有相同数据类型的元素集合在一起,作为一个数据对象来处理的数据类型。数组可以是每个元素只有一个下标的一维数组,也可以是每个元素有多个下标的多维数组。VHDL仿真器支持多维数组,但VHDL综合器只支持一维数组,故在此不讨论多维数组。

数组的定义格式如下。

```
TYPE 数组类型名 IS ARRAY 约束范围 OF 数据类型;
```

VHDL允许定义两种不同类型的数组,即限定性数组和非限定性数组。它们的区别是,限定性数组下标的取值范围在数组定义时就被确定了,而非限定性数组下标的取值范围需留待随后确定。

限定性数组定义语句的格式如下。

```
TYPE 数组名 IS ARRAY 约束范围 OF 数据类型;
```

其中,数组名是新定义的限定性数组类型的名称,可以是任何标识符,约束范围明确指出数组元素的定义数量和排序方式,以整数来表示其数组的下标,数据类型即指数组各元素的数据类型。

以下是限定性数组定义示例。

```
TYPE stb IS ARRAY (7 DOWNTO 0) OF STD_LOGIC;
```

这个数组类型的名称是stb,它有8个元素,数组元素是STD_LOGIC型的,各元素的排序是stb(7)、stb(6)、…、stb(0)。

非限定性数组定义语句的格式如下。

```
TYPE 数组名 IS ARRAY(数组下标名 RANGE <>)OF 数据类型;
```

其中,数组名是定义的非限定数组类型的取名,数组下标名是以整数类型设定的一个数

组下标名称，其中符号“<>”是下标范围待定符号，用到该数组类型时，再填入具体的数值范围。注意符号“<>”间不能有空格。数据类型是数组中每一元素的数据类型。

以下是非限定性数组的例子。

```
TYPE word IS ARRAY (NATURAL RANGE <>) OF BIT;
VARIABLE va: word(1 to 6);          -- 将数组取值范围定在 1～6
```

对数组赋值可以按照下标对每一个数组元素进行赋值，也可以对整个数组做一次性赋值。例如，进行如下的定义：

```
TYPE example IS ARRAY (0 TO 7) OF BIT;
SIGNAL a: example;
```

那么在源代码中，对信号 a 进行赋值可以采用以下两种方法。

可以对整个数组进行一次赋值：

```
a <= "01000111";
```

也可以按照下标对每一个数组元素进行赋值：

```
a(7)<= '0';
a(6)<= '1';
…
a(0)<= '1';
```

在引用数组时也有两种方法：引用数组元素和引用整个数组。仍以上面的数组为例，假设 b 也是 example 类型的信号，c、d 都为位类型的信号。

可以引用整个数组：

```
b <= a;
```

也可以引用数组元素：

```
c <= a(0);
d <= a(7);
```

4）记录类型

记录类型与数组类型都属于数组，由相同数据类型的元素构成的数组称为数组类型，由不同数据类型的元素构成的数组称为记录类型。构成记录类型的各种不同的数据类型可以是任何一种已定义过的数据类型，也包括数组类型和已定义的记录类型。显然具有记录类型的数据对象的数值是一个复合值，这些复合值是由这个记录类型的元素决定的。

定义记录类型的语句格式如下。

```
TYPE    记录类型名          IS    RECORD
        元素名: 元素数据类型;
        元素名: 元素数据类型;
…
END   RECORD  [记录类型名];
```

记录类型定义示例如下。

```
TYPE example IS RECORD
  Year: INTEGER RANGE 0 TO 3000;
  Month: INTEGER RANGE 1 TO 12;
  Data: INTEGER RANGE 1 TO 31;
  Addr: STD_LOGIC_VECTOR(7 DOWNTO 0);
  Data: STD_LOGIC_VECTOR(15 DOWNTO 0);
END RECODE;
```

一个记录的每一个元素要由它的记录元素名来进行访问。对于记录类型的对象的赋值与数组类似，可以对记录类型的对象进行整体赋值，也可以对它的记录元素进行分别赋值。

5. 用户定义的子类型

在用 VHDL 对硬件电路进行描述的时候，有时一个对象可能取值的范围是某个类型说明定义范围的子集，那么就要用到子类型的概念。

子类型 SUBTYPE 只是由 TYPE 所定义的原数据类型的一个子集，它满足原数据类型的所有约束条件，原数据类型称为基本数据类型。子类型 SUBTYPE 的语句格式如下。

```
SUBTYPE 子类型名 IS 基本数据类型 RANGE 约束范围;
```

子类型的定义只在基本数据类型上做一些约束，并没有定义新的数据类型，这是与 TYPE 最大的不同之处。子类型定义中的基本数据类型必须在前面已有过 TYPE 定义的类型，包括已在 VHDL 预定义程序包中用 TYPE 定义过的类型。例如：

```
SUBTYPE digits IS INTEGER RANGE 0 to 9;
```

上例中，INTEGER 是标准程序包中已定义过的数据类型，子类型 digits 只是把 INTEGER 约束到只含 10 个值的数据类型。

事实上，在程序包 STANDARD 中，已有两个预定义子类型，即自然数类型(Natural Type)和正整数类型(Positive Type)，它们的基本数据类型都是 INTEGER。

由于子类型与其基本数据类型属同一数据类型，因此属于子类型的和属于基本数据类型的数据对象间的赋值和被赋值可以直接进行，不必进行数据类型的转换。

利用子类型定义数据对象可以提高程序可读性，而且其实质性的好处还在于有利于提高综合的优化效率，这是因为综合器可以根据子类型所设的约束范围，有效地推出参与综合的寄存器的最合适的数目。

6. 数据类型的转换

在 VHDL 中，数据类型的定义是相当严格的，不同类型的数据是不能进行运算和直接代入的。为了实现正确的代入操作，必须将要代入的数据进行类型转换。数据类型的转换有三种方法：函数转换法、类型标记转换法和常数转换法。下面分别进行介绍。

1) 函数转换法

变换函数通常由 VHDL 的程序包提供。例如，在 STD_LOGIC_1164、STD_LOGIC_ARITH 和 STD_LOGIC_UNSIGNED 的程序包中提供了如表 5-1 所示的数据类型变换函数。引用时，先打开库和相应的程序包。例 5-1 就是由 STD_LOGIC_VECTOR 变换成 INTEGER 的实例。

表 5-1 类型变换函数

程 序 包	函 数 名	功 能
STD_LOGIC_1164	TO_STDLOGICVECTOR(A)	由 BIT_VECTOR 转换为 STD_LOGIC_VECTOR
	TO_BITVECTOR(A)	由 STD_LOGIC_VECTOR 转换为 BIT_VECTOR
	TO_STDLOGIC(A)	由 BIT 转换为 STD_LOGIC
	TO_BIT(A)	由 STD_LOGIC 转换为 BIT
STD_LOGIC_ARITH	CONV_STD_LOGIC_VECTOR(A,位长)	由 INTEGER、UNSIGNED、SIGNED 转换成 STD_LOGIC_VECTOR
	CONV_INTEGER(A)	由 UNSIGNED、SIGNED 转换成 INTEGER
STD_LOGIC_UNSIGNED	CONV_INTEGER(A)	由 STD_LOGIC_VECTOR 转换成 INTEGER

【例 5-1】 *数据类型转换*

```
LIBRARY IEEE;
USE IEEE STD_LOGIC_1164.ALL;
USE IEEE STD_LOGIC_UNSIGNED.ALL;
ENTITY zhh IS
PORT(num: IN STD_LOGIC_VECTOR(2 DOWNTO 0);
…
);
END zhh;
ARCHITECTURE behave OF zhh IS
SIGNAL in_num: INTEGER RANGE 0 TO 5;
…
BEGIN
  In_num <= CONV_INTEGER(num);          -- 变换式
…
END behave;
```

此外，由“BIT_VECTOR”变换成“STD_LOGIC_VECTOR”也非常方便。代入“STD_LOGIC_VECTOR”的值只能是二进制数，而代入“BIT_VECTOR”的值除二进制数以外，还可能是十六进制及八进制数。不仅如此，“BIT_VECTOR”还可以用“_”来分隔数值位。下面给出“BIT_VECTOR”和“STD_LOGIC_VECTOR”的赋值语句。

```
SIGNAL a: BIT_VECTOR( 11 DOWNTO 0);
SIGNAL b: STD_LOGIC_VECTOR(11 DOWNTO 0);
a <= X"A8";                             -- 十六进制值可赋予位矢量
b <= X"A8";                             -- 语法错误。十六进制值不能赋予逻辑矢量
b <= TO_STDLOGICVECTOR(X"AF7");
b <= TO_STDLOGICVECTOR(O"5177");        -- 八进制变换
b <= TO_STDLOGICVECTOR(B"1010_1111_0111");
```

2）类型标记转换法

在 VHDL 中的类型标记转换法是直接使用类型名进行数据类型的转换，这与高级语言中的强制类型转换类似。

类型标记就是类型的名称。类型标记转换法是那些关系密切的标量类型之间的类型转换，即整数和实数类型的转换。其语句格式如下。

```
数据类型标志符(表达式);
```

下面几个语句说明了标记类型转换的例子。

```
VARIABLE a: INTEGER;
VARIABLE b: REAL;
a: = INTEGER(b);
b: = REAL(a);
```

在上面的语句中。当把浮点数转换为整数时会发生舍入现象。如果某浮点数的值恰好处于两个整数的正中间,转换的结果可能向任意方向靠拢。

类型标记转换法必须遵循以下原则。

(1) 所有的抽象数据类型是可以互相转换的类型(如整型、浮点型),如果浮点数转换为整数,则转换结果是最接近的一个整型数。

(2) 如果两个数组有相同的维数,且两个数组的元素是同一种类型,并且在各自的下标范围内索引是同一种类型或者是非常接近的类型,那么,这两个数组是可以进行类型转换的。

(3) 枚举类型不能被转换。

3) 常数转换法

常数转换法是指在程序中用常数将一种数据类型转换成另一种数据类型。就转换效率而言,该方法是比较高的,但由于这种方法不经常使用,这里就不详细介绍了。

7. 数据类型的限定

在 VHDL 中,有时可以用所描述的文字的上下文关系来判断某一数据的数据类型。例如:

```
SIGNAL a: STD_LOGIC_VECTOR(7 DOWNTO 0);
a < = "01101010";
```

联系上下文关系,可以断定“01101010”不是字符串,也不是位矢量,而是“STD_LOGIC_VECTOR”。但是,有时也有判断不出来的情况,例如:

```
CASE(a & b & c)IS
  WHEN "001" = > y < = "01111111";
  WHEN "010" = > y < = "10111111";
  …
END CASE;
```

在该例中,a&b&c 的数据类型如果不确定就会发生错误。在这种情况下,就要对数据进行类型限定,这类似于 C 语言中的强制方式。数据类型限定的方式是在数据前加上“类型名”。例如:

```
a < = STD_LOGIC_VECTOR'("01101010");
SUBTYPE STD3BIT IS STD_LOGIC_VECTOR(0 TO 2);
CASE STD3BIT'(a&b&c)IS
  WHEN "000" = > y < = "01111111";
  WHEN "001" = > y < = "10111111";
  …
```

类型限定方式与数据类型变换很相似,这一点要引起注意。

5.1.4 VHDL 操作符

VHDL 的各种表达式由操作数和操作符组成，其中，操作数是各种运算的对象，而操作符则规定运算的方式。

1. 操作符种类及对应的操作数类型

在 VHDL 中有四类操作符，即逻辑操作符（Logical Operator）、关系操作符（Relational Operator）、算术操作符（Arithmetic Operator）和符号操作符（Sign Operator），此外还有重载操作符（Overloading Operator）。前三类操作符是完成逻辑和算术运算的最基本的操作符的单元，重载操作符是对基本操作符做了重新定义的函数型操作符。各种操作符所要求的操作数的类型详见表 5-2。操作符是有优先级的，各操作符之间的优先级别见表 5-3。

表 5-2 VHDL 操作符列表

类　型	操作符	功　能	操作数数据类型
算术操作符	＋	加	整数
	－	减	整数
	&	并置	一维数组
	*	乘	整数和实数（包括浮点数）
	/	除	整数和实数（包括浮点数）
	MOD	取模	整数
	REM	取余	整数
	SLL	逻辑左移	BIT 或布尔型一维数组
	SRL	逻辑右移	BIT 或布尔型一维数组
	SLA	算术左移	BIT 或布尔型一维数组
	SRA	算术右移	BIT 或布尔型一维数组
	ROL	逻辑循环左移	BIT 或布尔型一维数组
	ROR	逻辑循环右移	BIT 或布尔型一维数组
	**	乘方	整数
	ABS	取绝对值	整数
	＋	正	整数
	－	负	整数
关系操作符	＝	等于	任何数据类型
	/＝	不等于	任何数据类型
	＜	小于	枚举与整数类型及对应的一维数组
	＞	大于	枚举与整数类型及对应的一维数组
	＜＝	小于或等于	枚举与整数类型及对应的一维数组
	＞＝	大于或等于	枚举与整数类型及对应的一维数组
逻辑操作符	AND	与	BIT，BOOLEAN，STD_LOGIC
	OR	或	BIT，BOOLEAN，STD_LOGIC
	NAND	与非	BIT，BOOLEAN，STD_LOGIC
	NOR	或非	BIT，BOOLEAN，STD_LOGIC
	XOR	异或	BIT，BOOLEAN，STD_LOGIC
	XNOR	异或非	BIT，BOOLEAN，STD_LOGIC
	NOT	非	BIT，BOOLEAN，STD_LOGIC

表 5-3 VHDL 操作符优先级

运 算 符	优先级
NOT,ABS,**	最高优先级
*,/,MOD,REM	↑
+(正号),-(负号)	↑
+,-,&	↑
SLL,SLA,SRL,SRA,ROL,ROR	↑
=,/=,<,<=,>,>=	↑
AND,OR,NAND,NOR,XOR,XNOR	最低优先级

2. 各种操作符的使用说明

(1) 必须严格遵循在基本操作符之间的操作数是相同数据类型的规则；操作数的数据类型也必须与操作符所要求的数据类型完全一致。

(2) 注意操作符之间的优先级别。当一个表达式中有两个以上的运算符时,可使用括号将这些运算分组。

(3) VHDL 共有 7 种基本逻辑操作符,对于数组类型(如 STD_LOGIC_VECTOR)数据对象的相互作用是按位进行的。一般情况下,信号或变量在这些操作符的直接作用下,可构成组合电路。逻辑操作符所要求的操作数的基本数据类型有三种,即 BIT、BOOLEAN 和 STD_LOGIC。操作数的数据类型也可以是一维数组,其基本数据类型则必须为 BIT_VECTOR 或 STD_LOGIC_VECTOR。

通常,在一个表达式中有两个以上的逻辑运算符时,需要使用括号将这些运算分组。如果一串运算中的算符相同,且是 AND、OR、XOR 这三个算符中的一种,则不需要使用括号；如果一串运算中的算符不同或有除这三种算符之外的算符,则必须使用括号。例 5-2 是一组逻辑运算操作示例,请注意它们的运算表达方式和不加括弧的条件。

【例 5-2】 逻辑运算 VHDL 描述

```
…
SIGNAL a,b,c: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL d,e,f,g: STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL h,i,j,k: STD_LOGIC;
SIGNAL l,m,n,o,p: BOOLEAN;
…
a <= b AND c;                    -- b、c 相与后向 a 赋值
d <= e OR f OR g;                -- 两个操作符 OR 相同,不需要括号
h <= (i NAND j) NAND k;          -- NAND 不属于上述三种算符中的一种,必须加括号
l <= (m XOR n)AND(o XOR p);      -- 操作符不同,必须加括号
h <= i AND j AND k;              -- 操作符相同,不必加括号
h <= i AND j OR k;               -- 两个操作符不同,未加括号,表达错误
a < b AND e;                     -- 操作数 b 和 e 的位矢长度不一致,表达错误
h <= i OR l;                     -- 不同数据类型不能相互作用,表达错误
…
```

(4) 关系操作符的作用是将相同数据类型的数据对象进行数值比较(=、/=)或关系排序判断(<、<=、>、>=),并将结果以布尔类型的数据表示出来,即 TRUE 或 FALSE 两

种。对于数组类型的操作数，VHDL 编译器将逐位比较对应位置各位数值的大小而进行比较或关系排序。

就综合而言，简单的比较运算（＝和/＝）在实现硬件结构时，比排序操作符构成的电路芯片资源利用率要高。

同样是对 4 位二进制数进行比较，例 5-3 使用了“＝”操作符，例 5-4 使用了“＞＝”操作符，除了这两个操作符不同外，两个程序是完全相同的。综合结果表明，例 5-4 所耗用的逻辑门比例 5-3 多出近三倍。

【例 5-3】 4 位二进制数比较程序 1

```
ENTITY relational_ops_1 IS
  PORT(a,b: IN BIT_VECTOR(0 TO 3);
       output: OUT BOOLEAN);
END relational_ops_1;
ARCHITECTURE behave OF relational_ops_1 IS
BEGIN
  output <= (a = b);
END behave;
```

【例 5-4】 4 位二进制数比较程序 2

```
ENTITY relational_ops_2 IS
  PORT(a,b: IN BIT_VECTOR(0 TO 3);
       output: OUT BOOLEAN);
END relational_ops_2;
ARCHITECTURE behave OF relational_ops_2 IS
BEGIN
  output <= (a >= b);
END behave;
```

（5）算术操作符可以分为求和操作符、求积操作符、符号操作符、混合操作符、移位操作符五类。

求和操作符包括加减操作符和并置操作符。加减操作符的运算规则与常规的加减法是一致的，VHDL 规定它们的操作数的数据类型是整数。对于大于位宽为 4 的加法器和减法器，VHDL 综合器将调用库元件进行综合。

在综合后，由加减运算符（＋，－）产生的组合逻辑门电路所耗费的硬件资源的规模都比较大，但当加减运算符的其中一个操作数或两个操作数都为整型常数，则运算只需很少的电路资源。例 5-5 就是一个整数加法运算电路的 VHDL 描述。

【例 5-5】 整数加法运算电路

```
ENTITY arithmetic IS
  PORT(a,b: IN INTEGER;
       c: OUT INTEGER);
END arithmetic;
ARCHITECTURE behave OF arithmetic IS
BEGIN
  c <= a + b;
END behave;
```

并置运算符(&)的操作数的数据类型是一维数组,可以利用并置符将普通操作数或数组组合起来形成各种新的数组。例如,“VH”&“DL”的结果为“VHDL”;“0”&“1”的结果为“01”,连接操作常用于字符串。但在实际运算过程中,要注意并置操作前后的数组长度应一致。

求积操作符包括*(乘)、/(除)、MOD(取模)和 REM(取余)四种操作符。VHDL 规定,乘与除的数据类型是整数和实数(包括浮点数)。在一定条件下,还可对物理类型的数据对象进行运算操作。但需注意的是,虽然在一定条件下,乘法和除法运算是可综合的,但从优化综合、节省芯片资源的角度出发,最好不要轻易使用乘除操作符。对于乘除运算可以用其他变通的方法来实现。

操作符 MOD 和 REM 的本质与除法操作符是一样的,因此,可综合的取模和取余的操作数必须是以 2 为底数的幂。MOD 和 REM 的操作数数据类型只能是整数,运算操作结果也是整数。

取余运算(a rem b)的符号与 a 相同,其绝对值为小于 b 的绝对值。例如:

```
(-5) rem 2 = (-1);    5 rem (-2) = 1
```

取模运算(a mod b)的符号与 b 相同,其绝对值为小于 b 的绝对值。例如:

```
(-5) mod 2 = 1; 5 mod (-2) = (-1)
```

符号操作符“+”和“-”的操作数只有一个,操作数的数据类型是整数,操作符“+”对操作数不做任何改变,操作符“-”作用于操作数后的返回值是对原操作数取负,在实际使用中,取负操作数需加括号。例如“Z:=X*(-Y);”。

混合操作符包括乘方“**”操作符和取绝对值“ABS”操作符两种。VHDL 规定,它们的操作数数据类型一般为整数类型。乘方运算的左边可以是整数或浮点数,但右边必须为整数,而且只有在左边为浮点数时,其右边才可以为负数。一般地,VHDL 综合器要求乘方操作符作用的操作数的底数必须是 2。

六种移位操作符号 SLL、SRL、SLA、SRA、ROL 和 ROR 都是 VHDL93 标准新增的运算符。VHDL93 标准规定移位操作符作用的操作数的数据类型应是一维数组,并要求数组中的元素必须是 BIT 或 BOOLEAN 的数据类型,移位的位数则是整数。在 EDA 工具所附的程序包中重载了移位操作符以支持 STD_LOGIC_VECTOR 及 INTEGER 等类型。移位操作符左边可以是支持的类型,右边则必定是 INTEGER 型。如果操作符右边是 INTEGER 型常数,移位操作符实现起来比较节省硬件资源。

其中,SLL 是将位矢量向左移,右边跟进的位补零;SRL 的功能恰好与 SLL 相反;ROL 和 ROR 的移位方式稍有不同,它们移出的位将用于依次填补移空的位,执行的是自循环式移位方式;SLA 和 SRA 是算术移位操作符,其移空位用最初的首位来填补。

移位操作符的语句格式是:

```
标识符  移位操作符  移位位数;
```

例如:

```
"1011" SLL 1 = "0110"    "1011" SRL 1 = "0101"
"1011" SLA 1 = "0111"    "1011" SRA 1 = "1101"
```

```
"1011" ROL 1 = "0111"     "1011" ROR 1 = "1101"
```

操作符可以用以产生电路。就提高综合效率而言，使用常量值或简单的一位数据类型能够生成较紧凑的电路，而表达式复杂的数据类型（如数组）将相应地生成更多的电路。如果组合表达式的一个操作数为常数，就能减少生成的电路。

3. 重载操作符

为了方便各种不同数据类型间的运算，VHDL 允许用户对原有的基本操作符重新定义，赋予新的含义和功能，从而建立一种新的操作符，这就是重载操作符，定义这种操作符的函数称为重载函数。事实上，在程序包 STD_LOGIC_UNSIGNED 中已定义了多种可供不同数据类型间操作的运算符重载函数。

Synopsys 的程序包 STD_LOGIC_ARITH、STD_LOGIC_UNSIGNED 和 STD_LOGIC_SIGNED 中已经为许多类型的运算重载了算术运算符和关系运算符，因此只要引用这些程序包，SIGNED、UNSIGNED、STD_LOGIC 和 INTEGER 之间即可混合运算；INTEGER、STD_LOGIC 和 STD_LOGIC_VECTOR 之间也可以混合运算。在第 4 章已举过相应的例子。

5.2 VHDL 顺序语句

顺序语句和并行语句是 VHDL 程序设计中两类基本描述语句。在逻辑系统的设计中，这些语句从多侧面完整地描述了数字系统的硬件结构和基本逻辑功能，其中包括通信的方式、信号的赋值、多层次的元件例化以及系统行为等。

顺序语句是相对于并行语句而言的，其特点是每一条顺序语句的执行（指仿真执行）顺序是与它们的书写顺序基本一致的，但其相应的硬件逻辑工作方式未必如此。顺序语句只能出现在进程（Process）和子程序中，子程序又包括函数和过程。在 VHDL 中，一个进程是由一系列顺序语句构成的，而进程本身属于并行语句，这就是说，在同一设计实体中，所有的进程是并行执行的。然而任一给定的时刻内，在每一个进程内，只能执行一条顺序语句。一个进程与其设计实体的其他部分进行数据交换的方式只能通过信号或端口。如果要在进程中完成某些特定的算法和逻辑操作，也可以通过依次调用子程序来实现，但子程序本身并无顺序和并行语句之分。利用顺序语句可以描述逻辑系统中的组合逻辑、时序逻辑或它们的综合体。

VHDL 有如下六类基本顺序语句：赋值语句、转向控制语句、等待语句、子程序调用语句、返回语句和空操作语句等。

5.2.1 赋值语句

赋值语句的功能就是将一个值或一个表达式的运算结果传递给某一数据对象，如信号或变量，或由此组成的数组。VHDL 设计实体内的数据传递以及对端口界面外部数据的读写都必须通过赋值语句的进行来实现。

1. 信号和变量的赋值

赋值语句有两种，即信号赋值语句和变量赋值语句。每一种赋值语句都由三个基本部分组成，即赋值目标、赋值符号和赋值源。赋值目标是所赋值的受体，它的基本元素只能是

信号或变量，但表现形式可以有多种，后面将详细介绍。赋值源是赋值的主体，它可以是一个数值，也可以是一个逻辑或运算表达式。VHDL 规定，赋值目标与赋值源的数据类型必须严格一致。

变量赋值语句和信号赋值语句的语法格式如下。

```
变量赋值目标 := 赋值源;
信号赋值目标 <= 赋值源;
```

变量赋值与信号赋值的区别在于，变量具有局部特征，它的有效范围只局限于所定义的一个进程中，或一个子程序中，它是一个局部的、暂时性的数据对象。对于它的赋值是立即发生的（假设进程已启动），即是一种时间延迟为零的赋值行为。

信号则不同，信号具有全局性特征，它不但可以作为一个设计实体内部各单元之间数据传送的载体，而且可通过信号与其他的实体进行通信（端口本质上也是一种信号）。信号的赋值并不是立即发生的，它发生在一个进程结束时。赋值过程总是有某种延时的，它反映了硬件系统的重要特性，综合后可以找到与信号对应的硬件结构，如一根传输导线、一个输入/输出端口或一个 D 触发器等。

但是，必须注意，在某些条件下变量赋值行为与信号赋值行为所产生的硬件结果是相同的，如都可以向系统引入寄存器等。

在信号赋值中，需要注意的是，当在同一进程中，可以允许同一信号有多个驱动源（赋值源），当同一信号赋值目标有多个赋值源时，信号赋值目标获得的是最后一个赋值源的赋值，其前面相同的赋值目标不做任何变化。

例 5-6 说明了信号与变量赋值的特点及它们的区别。当在同一赋值目标处于不同进程中时，其赋值结果就比较复杂了，这可以看作多个信号驱动源连接在一起，可以发生线与、线或或者三态等不同结果。

【例 5-6】 信号与变量的赋值

```
SIGNAL s1,s2: STD_LOGIC;
SIGNAL svec: STD_LOGIC_VECTOR(0 TO 3);
…
PROCESS(s1,s2)
VARIABLE v1,v2: STD_LOGIC;
BEGIN
  v1: = '1';          -- 立即将变量 v1 置位为 1
  v2: = '1';          -- 立即将变量 v2 置位为 1
  s1 <= '1';          -- 信号 s1 被赋值为 1
  s2 <= '1';          -- 由于在本进程中,这里的 s2 不是最后一个赋值语句故不做任何赋值操作
  svec(0)<= v1;       -- 将变量 v1 在上面的赋值 1,赋给 svec(0)
  svec(1)<= v2;       -- 将变量 v2 在上面的赋值 1,赋给 svec(1)
  svec(2)<= s1;       -- 将信号 s1 在上面的赋值 1,赋给 svec(2)
  svec(3)<= s2;       -- 将最下面的赋予 s2 的值'0',赋给 svec(3)
  v1: = '0';          -- 将变量 v1 置入新值 0
  v2: = '0';          -- 将变量 v2 置入新值 0
  s2: <= '0';         -- 由于这是信号 s2 最后一次赋值,赋值有效,此'0'将
                      -- 上面准备赋入的'1'覆盖掉
END PROCESS;
```

2. 赋值目标

赋值语句中的赋值目标有两大类四种类型。

1）标识符赋值目标及数组单元素赋值目标

标识符赋值目标是以简单的标识符作为被赋值的信号或变量名。

数组单元素赋值目标的表达形式为：

数组类信号或变量名(下标名)

下标名可以是一个具体的数字，也可以是一个文字表示的数字名，它的取值范围在该数组元素个数范围内。下标名若是未明确表示取值的文字即为不可计算值，则在综合时，将耗用较多的硬件资源，且一般情况下不能被综合。例 5-6 即为标识符赋值目标及单元素赋值目标的使用示例。

2）段下标元素赋值目标及集合块赋值目标

段下标元素赋值目标可用以下方式表示：数组类信号或变量(下标 1 TO/DOWNTO 下标 2)，括号中的两个下标必须用具体数值表示，并且其数值范围必须在所定义的数组下标范围内，两个下标的排序方向要符合方向关键词 TO 或 DOWNTO，具体用法如下。

```
VARIABLE A,B: STD_LOGIC_VECTOR(0 TO 3);
A(1 TO 2): ="10";                          -- 等效于 A(1): = '1',A(2): = '0'
A(3 DOWNTO 0): ="1011";
```

集合块赋值目标是以一个集合的方式来赋值的。对目标中的每个元素进行赋值的方式，即位置关联赋值方式和名字关联赋值方式，具体用法如下。

```
SIGNAL a,b,c,d: STD_LOGIC;
SIGNAL s: STD_LOGIC_VECTOR(0 TO 3);
VARIABLE e,f: STD_LOGIC;
VARIABLE g: STD_LOGIC_VECTOR(0 TO 1);
VARIABLE h: STD_LOGIC_VECTOR(0 TO 3);
s<="0100";
(a,b,c,d)<=s;                              -- 位置关联方式赋值,结果等效为:
                                           -- a<='0'; b<='1'; c<='0'; d<='0';
(2=>e,3=>f,1=>g(0),0=>g(1): =h);           -- 名字关联方式赋值,结果等效为:
                                           -- g(1): =h(0); g(0): =h(1); e: =h(2); f: =h(3);
```

5.2.2 转向控制语句

转向控制语句通过条件控制，决定是否执行一条或几条语句，或重复执行一条或几条语句，或跳过一条或几条语句等。转向语句共有五种：IF 语句、CASE 语句、LOOP 语句、NEXT 语句和 EXIT 语句。

1. IF 语句

IF 语句是一种条件语句，它根据语句中所设置的一种或多种条件，有选择地执行指定的顺序语句，常见的 IF 语句有以下 3 种形式。

1）
```
IF 条件 THEN
   语句
END IF;
```

2）IF 条件 THEN
　　语句
　ELSE
　　语句
　END IF;

3）IF 条件 THEN
　　语句
　ELSIF　条件 THEN
　　语句
　ELSE
　　语句
　END IF;

IF 语句中至少应有一个条件句，条件句由布尔表达式构成，IF 语句根据条件句产生的判断结果 TRUE 或 FALSE，有条件地选择执行其后的顺序语句。如果布尔条件判断为 TRUE，关键词 THEN 后面的顺序语句则执行；如果条件判断为 FALSE，则 ELSE 后面的顺序语句则执行。举例如下。

例 5-7 就是使用 IF 语句来描述一个 4 位等值比较器功能的实例。

【例 5-7】 4 位等值比较器描述方式 1

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY eqcomp4 IS
  PORT(
       a,b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
       equals: OUT STD_LOGIC);
END eqcomp4;
ARCHITECTURE behave OF eqcomp4 IS
 BEGIN
comp: PROCESS(a,b)
BEGIN
  equals <= '0';
  IF a = b THEN                          -- 第 1 种 IF 语句，也称为门闩控制语句
    equals <= '1';
  END IF;
END PROCESS comp;
END behave;
```

这个例子的描述过程指出，作为一个默认值，equals 被赋值为‘0’；但是，当 a＝b 时，equals 就应该被赋值为‘1’。

例 5-8 是用 IF-THEN-ELSE 语句描述 4 位等值比较器的功能的结构体。

【例 5-8】 4 位等值比较器描述方式 2

```
ARCHITECTURE behave OF eqcomp4 IS
 BEGIN
comp: PROCESS(a,b)
BEGIN
  IF a = b THEN                          -- 第 2 种 IF 语句，实现二选一功能
    equals <= '1';
```

```
  ELSE
    equals <= '0';
  END IF;
END PROCESS comp;
END behave;
```

例 5-9 是使用 IF-THEN-ELSIF-ELSE 语句描述 4 位宽的 4 选 1 多路选择器功能的实例。

【例 5-9】 4 选 1 多路选择器描述方式 1

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux4 IS
PORT(
       a,b,c,d: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
       s: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
       X: OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
END mux4;
ARCHITECTURE behave OF mux4 IS
BEGIN
   Mux4: PROCESS(a,b,c,d)
BEGIN
  IF s = "00" THEN                          -- 第 3 种 IF 语句,实现多选 1 功能
     X <= a;
  ELSIF s = "01" THEN
     X <= b;
  ELSIF s = "10" THEN
     X <= c;
  ELSE
     X <= d;
  END IF;
END process mux4;
END behave;
```

2. CASE 语句

CASE 语句是 VHDL 提供的另一种形式的条件控制语句，它根据所给表达式的值选择执行语句集。CASE 语句与 IF 语句的相同之处在于：它们都根据某个条件在多个语句中集中进行选择。CASE 语句与 IF 语句的不同之处在于：CASE 语句根据某个表达式的值来选择执行体。CASE 语句的一般形式为：

```
CASE 表达式 IS
   WHEN 值 1 =>  语句 A;
   WHEN 值 2 =>  语句 B;
   …
   WHEN OTHERS =>  语句 C;
END CASE
```

根据以上 CASE 语句的形式可知，如果表达式的值等于某支路的值，那么该支路所选择的语句就要被执行。表达式可以是一个整数类型或枚举类型的值，也可以是由这些数据类型的值构成的数组。条件句中的“=>”不是操作符，它只相当于“THEN”的作用。在

CASE 语句中的选择必须是唯一的，即计算表达式所得的值必须且只能是 CASE 语句中的一支。CASE 语句中支路的个数没有限制，各支的次序也可以任意排列，但关键词 OTHERS 的分支例外，一个 CASE 语句最多只能有一个 OTHERS 分支，而且，如果使用了 OTHERS 分支，那么该分支必须放在 CASE 语句的最后一个分支的位置上。

CASE 语句使用中应注意以下几点。

(1) WHEN 条件句中的选择值或标识符所代表的值必须在表达式的取值范围内。

(2) 除非所有条件句中的选择值能完整覆盖 CASE 语句中表达式的取值，否则最后一个条件句中的选择必须用关键词 OTHERS 表示以上已列的所有条件句中未能列出的其他可能的取值。使用 OTHERS 的目的是为了使条件句中的所有选择值能涵盖表达式的所有取值，以免综合器会插入不必要的锁存器。关键词 NULL 表示不做任何操作。

(3) CASE 语句中的选择值只能出现一次，不允许有相同选择值的条件语句出现。

(4) CASE 语句执行中必须选中，且只能选中所列条件语句中的一条。

例如，使用 CASE-WHEN 语句描述一个 4 选 1 多路选择器如例 5-10 所示。其中，s1、s2 为控制信号，a、b、c、d 为 4 个输入端口，z 为输出端口。通过 s1 与 s2 的取值来选择输出哪一个端口。

【例 5-10】 4 选 1 多路选择器描述方式 2

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY test_case IS
PORT(
s1,s2: IN STD_LOGIC;
a,b,c,d: IN STD_LOGIC;
z: OUT STD_LOGIC
);
END test_case;
ARCHITECTURE behave OF test_case IS
SIGNAL s: STD_LOGIC_VECTOR(1 DOWNTO 0);
  BEGIN
  S <= s1 & s2;
    PROCESS(s1,s2,a,b,c,d)
      BEGIN
        CASE s IS                          -- CASE - WHEN 语句
          WHEN "00" => z <= a;
          WHEN "01" => z <= b;
          WHEN "10" => z <= c;
          WHEN "11" => z <= d;
          WHEN  OTHERS => z <= 'x';
        END CASE;
    END PROCESS;
END behave;
```

注意本例的 WHEN OTHERS 语句是必需的，因为对于定义 STD_LOGIC_VECTOR 数据类型的 s，在 VHDL 综合过程中，它可能的选择值除了 00、01、10 和 11 以外，还可以有其他定义于 STD_LOGIC 的选择值。

与 IF 语句相比，CASE 语句组的程序可读性比较好，这是因为它把条件中所有可能出

现的情况全部列出来了，可执行条件比较清晰。而且CASE语句的执行过程不像IF语句那样有一个逐项条件顺序比较的过程。CASE语句中条件句的次序是不重要的，它的执行过程更接近于并行方式。但是在一般情况下，经过综合后，对相同的逻辑功能，CASE语句比IF语句的描述耗用更多的硬件资源，而且有的逻辑功能，CASE语句无法描述，只能用IF语句来描述。

3. LOOP语句

LOOP语句就是循环语句，用于实现重复的操作，由FOR循环或WHILE循环组成。FOR语句的执行根据控制值的规定数目重复；WHILE语句将连续执行操作，直到控制逻辑条件判断为TRUE。下面给出FOR循环语句和WHILE循环语句的一般形式。

1）FOR循环

FOR循环语句的一般形式为：

```
[循环标号：] FOR 循环变量 IN 循环次数范围 LOOP
             顺序处理语句
             END LOOP[循环标号];
```

FOR循环语句中的循环变量的值在每次循环中都将发生变化，而IN后面的循环次数范围则表示循环变量在循环过程中依次取值的范围。

例5-11就是利用FOR LOOP语句实现8位奇偶校验电路的VHDL程序。

【例5-11】 8位奇偶校验电路

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY p_check IS
  PORT(a: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       y: OUT STD_LOGIC);
END p_check;
ARCHITECTURE behave OF p_check IS
  SIGNAL tmp: STD_LOGIC;
  BEGIN
  PROCESS(a)
  BEGIN
    tmp <= '0';
    FOR n IN 0 TO 7 LOOP                    -- FOR循环语句
     tmp <= tmp XOR a(n);
    END LOOP;
    y <= tmp;
  END PROCESS;
END behave;
```

FOR LOOP语句中的n无论在信号说明和变量说明中都未涉及，是一个循环变量，它是一个整数变量，当然也可以是其他类型，只要保证数值是离散的即可。

2）WHILE循环

WHILE循环语句的一般形式为：

```
[循环标号：]WHILE 条件 LOOP
            顺序处理语句
```

```
END LOOP[循环标号];
```

在 WHILE 循环中，如果条件为“真”，则进行循环；如果条件为“假”，则结束循环。

例 5-12 描述的仍然是 8 位奇偶校验电路，但是以 WHILE LOOP 循环语句来进行描述的。

【例 5-12】 8 位奇偶校验电路

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY p_check2 IS
  PORT(a: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       y: OUT STD_LOGIC);
END p_check2;
ARCHITECTURE behave OF p_check2 IS
  SIGNAL tmp: STD_LOGIC;
  BEGIN
  PROCESS(a)
  VARIABLE i: INTEGER: = 0;
  BEGIN
    tmp < = '0';
    WHILE i < 8 LOOP                          -- WHILE 循环
     tmp < = tmp XOR a(i);
     i: = i + 1;
    END LOOP;
    y < = tmp;
  END PROCESS;
END behave;
```

WHILE 循环语句在这里可用于替代 FOR 循环语句，但需要有附加的说明、初始化和递增循环变量的操作。

请注意：一般的综合工具可以对 FOR LOOP 循环语句进行综合；而对 WHILE LOOP 循环语句来说，只有一些高级的综合工具才能对它进行综合，所以，一般使用 FOR LOOP 循环语句，而很少使用 WHILE LOOP 循环语句。

4. NEXT 语句

有时由于某种情况需要跳出循环，而去执行另外的操作，这就需要采用跳出循环的操作。VHDL 提供了两种跳出循环的操作，一种是 NEXT 语句，另一种是 EXIT 语句。NEXT 语句主要用于在 LOOP 语句执行中有条件的或无条件的转向控制。它的语句格式有以下三种。

(1) NEXT；

(2) NEXT LOOP 标号；

(3) NEXT LOOP 标号 WHEN 条件表达式。

对于第一种格式，当 LOOP 内的顺序语句执行到 NEXT 语句时，即可无条件终止当前的循环，跳回到本次循环 LOOP 语句处，开始下一次循环。

对于第二种语句格式，即在 NEXT 旁加“LOOP 标号”后的语句功能，与未加 LOOP 标号的功能是基本相同的，只是当有多重 LOOP 语句嵌套时，前者可以跳转到指定标号的

LOOP 语句处，重新开始执行循环操作。

第三种语句格式中，分句“WHEN 条件表达式”是执行 NEXT 语句的条件，若条件表达式的值为 TRUE，则执行 NEXT 语句，进入跳转操作，否则继续向下执行。但当只有单层 LOOP 循环语句时，关键词 NEXT 与 WHEN 之间的“LOOP 标号”可以如例 5-13 那样省去。

【例 5-13】 NEXT 语句的应用情况 1

```
…
L1: FOR cnt_value IN 1 TO 8 LOOP
S1: a(cnt_value): = '0';
NEXT WHEN (b = c);
S2: a(cnt_value + 8): = '0';
END LOOP L1;
```

本例中，当程序执行到 NEXT 语句时，如果条件判断式(b=c)的结果为 TRUE，将执行 NEXT 语句，并返回到 L1，使 cnt_value 加 1 后执行 S1 开始赋值语句，否则将执行 S2 开始的赋值语句。

在多重循环中，NEXT 语句必须如例 5-14 那样，加上跳转标号。

【例 5-14】 NEXT 语句的应用情况 2

```
…
L_X: FOR cnt_value IN 1 TO 8 LOOP
  S1: a(cnt_value): = '0';
    K: = 0;
L_Y: LOOP
  S2: b(k): = '0';
    NEXT L_X WHEN (e > f);
  S3: b(k + 8): = '0';
    K: = k + 1;
      NEXT LOOP L_Y;
      NEXT LOOP L_X;
        …
```

当 e > f 为 TRUE 时执行语句 NEXT L_X，跳转到 L_X，使 cnt_value 加 1，从 S1 处开始执行语句，若为 FALSE，则执行 S3 后使 k 加 1。

5. EXIT 语句

EXIT 语句与 NEXT 语句具有十分相似的语句格式和跳转功能，它们都是 LOOP 语句的内部循环控制语句。EXIT 的语句格式也有三种：

(1) EXIT；

(2) EXIT LOOP 标号；

(3) EXIT LOOP 标号 WHEN 条件表达式。

这里，每一种语句格式与对应的 NEXT 语句格式和操作功能非常相似，唯一的区别是 NEXT 语句跳转的方向是 LOOP 标号指定的 LOOP 语句处，当没有 LOOP 标号时，跳转到当前的 LOOP 语句的循环起始点，而 EXIT 语句跳转的方向是 LOOP 标号指定的 LOOP 循环结束处，即完全跳出指定的循环，并开始执行循环外的语句。这就是说，NEXT 语句是

转向 LOOP 语句的起始点，而 EXIT 语句则是转向 LOOP 语句的终点。

例 5-15 是一个两元素位矢量值比较程序。在程序中，当发现比较值 a 和 b 不同时，由 EXIT 语句跳出循环比较程序，并报告比较结果。

【例 5-15】 EXIT 语句应用实例

```
SIGNAL a,b: STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL a_less_then_b: BOOLEAN;
...
a_less_then_b <= FLASE;                    -- 设初始值
FOR i IN DOWNTO 0 LOOP
  IF(a(i) = '1'AND b(i) = '0')THEN
    a_less_then_b <= FALSE;                -- a > b
    EXIT;
  ELSIF(a(i) = '0'AND b(i) = '1')THEN
    A_less_then_b <= TRUE;                 -- a < b
    EXIT;
  ELSE NULL;
  END IF;
END LOOP;                                  -- 当 i = 1 时返回 LOOP 语句继续比较
```

NULL 为空操作语句，是为了满足 ELSE 的转换。此程序先比较 a 和 b 的高位，高位是 1 者为大，输出判断结果 TRUE 或 FALSE 后中断比较程序，当高位相等时，继续比较低位，这里假设 a 不等于 b。

5.2.3 WAIT 语句

在进程中(包括过程中)，当执行到 WAIT(等待)语句时，运行程序将被挂起，直到满足此语句设置的结束挂起条件后，将重新开始执行进程(或过程)中的程序。但 VHDL 规定，已列出敏感量的进程中不能使用任何形式的 WAIT 语句。WAIT 语句的格式如下。

```
WAIT[ON 信号表][UNTIL 条件表达式][FOR 时间表达式];
```

WAIT 语句有以下几种形式。

(1) 单独的 WAIT，未设置停止挂起的条件，表示永远挂起。

(2) WAIT ON 信号表，即敏感信号等待语句，当敏感信号变化时，结束挂起。例如：

```
WAIT ON  a,b;
```

表示当 a 或 b 信号中任一信号变化时，就结束挂起，继续执行此语句后面的语句。

(3) WAIT UNTIL 条件表达式，即条件等待语句，当条件表达式中所含的信号发生了变化，并且条件表达式为真时，进程才能脱离挂起状态，继续执行此语句后面的语句。例如：

```
WAIT UNTIL((x * 10)< 100);
```

表示当信号量 x 的值大于或等于 10 时，进程执行到该语句，将被挂起，当 x 的值小于 10 时，进程再次被启动，继续执行此语句后面的语句。

(4) WAIT FOR 时间表达式，直到指定的时间到时，挂起才结束。

例如，语句 WAIT FOR 20ns；表示执行到该语句时需等待 20ns 后再继续执行下一条

语句。

（5）多条件 WAIT 语句，即上述条件中有多个同时出现，此时只要多个条件中有一个成立，则终止挂起。

例 5-16 所描述的两个进程的描述是等效的。

【例 5-16】 WAIT 语句应用情况 1

```
PROCESS(a,b)                              -- 进程 1
    BEGIN
      Y <= a AND b;
END PROCESS;
PROCESS                                   -- 进程 2
    BEGIN
     Y <= a AND b;
    WAIT ON a,b;
END PROCESS;
```

注意：已列出敏感信号的进程中不能使用任何形式的 WAIT 语句，一般情况下，只有 WAIT UNTIL 格式的等待语句可以被综合器所接受，其余语句格式只能在 VHDL 仿真器中使用。

例 5-17 描述的一个进程中，有一个无限循环的 LOOP 语句，其中用 WAIT 语句描述了一个具有同步复位功能的电路。

【例 5-17】 WAIT 语句应用情况 2

```
PROCESS
BEGIN
  rst_loop: LOOP
  WAIT UNTIL clock = '1' AND clock'EVENT;        -- 等待时钟信号
  NEXT rst_loop WHEN (rst = '1');                -- 检测复位信号
  x <= a;                                        -- 无复位信号，执行赋值操作
  WAIT UNTIL clock = '1' AND clock'EVENT;        -- 等待时钟信号
  NEXT rst_loop WHEN (rst = '1');                -- 检测复位信号
  y <= b;                                        -- 无复位信号，执行赋值操作
  END LOOP rst_loop;
END PROCESS;
```

例 5-17 中每一时钟上升沿的到来都将结束进程的挂起，继而检测电路的复位信号“rst”是否为高电平。如果是高电平，则返回循环的起始点；如果是低电平，则执行正常的顺序语句操作。

一般情况下，在一个进程中使用了 WAIT 语句后，经综合即产生时序逻辑电路。

5.2.4 子程序调用语句

子程序包括过程和函数，可以在 VHDL 的结构体或程序包中的任何位置对子程序进行调用。从硬件角度讲，一个子程序的调用类似于一个元件模块的例化，也就是说，VHDL 综合器为子程序的每一次调用都生成一个电路逻辑块。所不同的是，元件的例化将产生一个新的设计层次，而子程序调用只对应于当前层次的一部分。子程序的结构详见 5.4 节，它包括子程序首和子程序体。

1. 过程调用

过程调用就是执行一个给定名字和参数的过程。调用过程的语句格式如下。

```
过程名[([形参名 =>]实参表达式
      {,[形参名 =>]实参表达式})];
```

其中,形参为欲调用过程中已说明的参数名,实参是当前调用程序中过程形参的接受体。被调用中的形参与调用语句中的实参可以采用位置关联法和名字关联法进行对应,位置关联可以省去形参名。一个过程的调用有以下三个步骤。

(1) 将 IN 和 INOUT 模式的实参值赋给欲调用的过程中与它们对应的形参。

(2) 执行这个过程。

(3) 将过程中 IN 和 INOUT 模式的形参值返回给对应的实参。

实际上,一个过程对应的硬件结构中,其标识形参的输入/输出是与其内部逻辑相连的。在例 5-18 中定义了一个名为 swap 的局部过程(没有放在程序包中的过程),这个过程的功能是对一个数组中的两个元素进行比较,如果发现这两个元素的排列不符合要求,就进行交换,使得左边的元素值总是大于右边的元素值。连续调用三次 swap 后,就能将一个三元素的数组元素从左至右按序排列好,最大值排在左边。

【例 5-18】 过程的应用

```
PACKAGE data_types IS                              -- 定义程序包
SUBTYPE data_element IS INTEGER RANGE 0 TO 3;      -- 定义数据类型
TYPE data_array IS array(1 TO 3) OF data_element;
END data_types;
USE WORK.data_types.ALL;                    -- 打开以上建立在当前工作库的程序包 data_types
ENTITY sort IS
   PORT( in_array: IN data_array;
         out_array: OUT data_array);
END sort;
  ARCHITECTURE behave OF sort IS
  BEGIN
  PROCESS(in_array)                         -- 进程开始,设 data_types 为敏感信号
    PROCEDURE swap(data: INOUT data_array;  -- swap 的形参名为 data、low、high
                   low,high: IN INTEGER) IS
    VARIABLE  temp: data_element;
    BEGIN                                   -- 开始描述本过程的逻辑功能
      IF (data(low)> data(high)) THEN       -- 检测数据
          temp: = data(low);
          data(low): = data(high);
          data(high): = temp;
      END IF;
     END swap;                              -- 过程 swap 定义结束
  VARIABLE my_array: data_array;            -- 在本进程中定义变量 my_array
  BEGIN                                     -- 进程开始
    my_array: = in_array;                   -- 将输入值读入变量
    swap(my_array,1,2);                     -- my_array、1、2 是对应于 data、low、high 的实参
    swap(my_array,2,3);                     -- 位置关联调用,第 2、第 3 元素交换
    swap(my_array,1,2);                     -- 位置关联调用,第 1、第 2 元素再次交换
    out_array < = my_array;
```

```
  END PROCESS;
END behave;
```

2. 函数调用

函数调用与过程调用十分相似，不同之处是，调用函数将返回一个指定数据类型的值，函数的参量只能是输入值。

5.2.5 返回语句

返回语句只能用于子程序中，并用来结束当前子程序的执行。其语句有以下两种格式。

(1) RETURN。

(2) RETURN 表达式。

第一种语句格式只能用于过程，它只是结束过程，并不返回任何值；第二种语句格式只能用于函数，并且必须返回一个值。每一个函数必须至少包含一个返回语句，并可以拥有多个返回语句，但是在函数调用时，只有其中一个返回语句可以将值返回。

例 5-19 是一个过程定义语句，它将完成一个 RS 触发器的功能。注意其中的时间延迟语句和 REPORT 语句是不可综合的。

【例 5-19】 过程的返回

```
PROCEDURE rsff (SIGNAL s,r: IN STD_LOGIC;
                SIGNAL q,nq: INOUT STD_LOGIC) IS
BEGIN
  IF(s = '1' AND r = '1')THEN
    REPORT "Forbidden state: s and r quual to '1'";
    RETURN;
  ELSE
    q <= s NAND nq AFTER 5 ns;
    nq <= r NAND q AFTER 5 ns;
  END IF;
END PROCEDURE rsff;
```

当信号 r 和 s 同时为 1 时，在 IF 语句中的 RETURN 语句将中断过程。

例 5-20 是在一个函数体中使用 RETURN 语句的示例。

【例 5-20】 函数的返回

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY max21 IS
  PORT(a,b: IN INTEGER;
       q: OUT INTEGER);
END max21;
ARCHITECTURE behave OF max21 IS
BEGIN
  PROCESS(a,b)
  FUNCTION max(a,b: INTEGER) RETURN INTEGER IS
  VARIABLE temp: INTEGER;
  BEGIN
    IF(a > b)THEN
```

```
      temp: = a;
    ELSE
      temp: = b;
    END IF;
    RETURN(temp);
  END max;
BEGIN
  q < = max(a,b);
END PROCESS;
END behave;
```

例 5-20 实现的是对两个输入整数取最大值,在结构体的进程中定义了一个取最大值的函数。在函数体中,通过 RETURN 语句将比较得到的最大值返回,而且结束该函数体的执行。

5.2.6 NULL 语句

空操作语句不完成任何操作,它唯一的功能就是使程序执行下一个语句。NULL 常用于 CASE 语句中,利用 NULL 来表示所有的不用的条件下的操作行为,以满足 CASE 语句对条件值全部列举的要求。

空操作语句格式如下。

```
NULL;
```

在例 5-21 的 CASE 语句中,NULL 语句用于排除一些不用的条件。

【例 5-21】 NULL 语句的应用

```
CASE opcode IS
WHEN "001" = > tmp: = rega AND regb;
WHEN "101" = > tmp: = rega OR regb;
WHEN "110" = > tmp: = NOT rega;
WHEN OTHERS = > NULL;
END CASE;
```

此例类似于一个 CPU 内部的指令译码器的功能,“001”“101”“110”分别代表指令操作码,对于它们所对应寄存器中的操作数的操作算法,CPU 只对这三种指令码做反应,当出现其他码时,不做任何操作。

5.2.7 其他语句

1. 属性描述与定义语句

属性描述与定义语句有许多实际的应用。VHDL 中具有属性的项目有:类型、子类型、过程、函数、信号、变量、常量、实体、结构体、配置、程序包、元件和语句标号等。

属性就是这些项目的特性,某一项目的属性可以通过一个值或一个表达式来表示,通过 VHDL 的预定义属性描述语句就可以加以访问。

属性的值与对象(信号、变量和常量)的值完全不同,在任一给定的时刻,一个对象只能有一个值,但却可以有多个属性,VHDL 还允许设计者自己定义属性,即用户自定义属性。

综合器支持的属性有 LEFT、RIGHT、HIGH、LOW、RANGE、REVERS_RANGE、

LENGTH、EVENT 及 STABLE 等。

预定义属性描述语句实际上是一个内部预定义函数，其语句格式是：

```
属性测试项目名'属性标识符
```

其中，属性测试项目即属性对象，可由相应的标识符表示，属性标识符即属性名。以下对可以综合的属性项目的使用方法做一说明。

1）信号类属性

信号类属性中，最常用的当属 EVENT，这在第 4 章已进行了详细说明。

属性 STABLE 的测试功能恰与 EVENT 相反，它是信号在 δ 时间内无事件发生，则返回 TRUE 值。以下两个语句的功能是一样的。

```
NOT(clock'STABLE AND clock = '1')
(clock'EVENT AND clock = '1')
```

注意：语句“NOT(clock'STABLE AND clock＝'1')”的表达方式是不可综合的。因为对于 VHDL 综合器来说，括号中的语句等效于一条时钟信号边沿测试专用语句，它已不是普通的操作数，所以不能以操作数方式来对待。

在实际使用中，'EVENT 比 'STABLE 更常用。对于目前常用的 VHDL 综合器来说，EVENT 只能用于 IF 和 WAIT 语句中。

2）数据区间类属性

数据区间类属性有'RANGE[(n)]和'REVERSE_RANGE[(n)]。这类属性函数主要是对属性项目取值区间进行测试，返回的内容不是一个具体值，而是一个区间。对于同一属性项目，'RANGE 和'REVERSE_RANGE 返回的区间次序相反，前者与原项目次序相同，后者相反。例如：

```
…
SIGNAL rangel: IN STD_LOGIC_VECTOR(0 TO 7);
…
FOR I IN rangel'RANGE LOOP
…
```

此例中的 FOR LOOP 语句与语句“FOR I IN 0 TO 7 LOOP”的功能是一样的，这说明 rangel'RANGE 返回的区间即为位矢量 rangel 定义的元素范围。如果用 'REVERSE RANGE，则返回的区间正好相反，是(7 DOWNTO 0)。

3）数值类属性

在 VHDL 中的数值类属性测试函数主要有 'LEFT、'RIGHT、'HIGH 及 'LOW。这些属性函数主要用于对属性测试目标一些数值特性进行测试。例如：

```
…
PROCESS(clk,a,b);
TYPE obj IS ARRAY(0 TO 15) OF BIT;
SIGNAL s1,s2,s3,s4: INTEGER;
BEGIN
  S1 <= obj'RIGHT;
  S2 <= obj'LEFT;
```

```
  S3 < = obj'HIGH;
  S4 < = obj'LOW;
…
```

信号 s1、s2、s3 和 s4 获得的赋值分别为 0、15、0 和 15。

4) 数组属性

数组属性'LENGTH 的用法同前,只是对数组的宽度或元素的个数进行测定。例如:

```
…
TYPE arry1 ARRAY(0 TO 7) OF BIT;
VARIABLE wth: INTEGER;
…
wth1: = arry1'LENGTH;                          -- wth1 = 8
…
```

5) 用户自定义属性

属性与属性值的定义格式如下。

```
ATTRIBUTE 属性名: 数据类型;
ATTRIBUTE 属性名 OF 对象名: 对象类型 IS 值;
```

VHDL 综合器和仿真器通常使用自定义的属性实现一些特殊的功能,由综合器和仿真器支持的一些特殊的属性一般都包含在 EDA 工具厂商的程序包里,例如,Synplify 综合器支持的特殊属性都在 synplify. attributes 程序包中,使用前加入以下语句即可。

```
LIBRARY synplify;
USE synplicity.attributes.all;
```

2. 文本文件操作

在 VHDL 中提供了一个预先定义的包集合是文本输入/输出包集合(TEXTIO),在该 TEXTIO 中包含对文本文件进行读写的过程和函数。文件操作只能用于 VHDL 仿真器中,VHDL 综合器将忽略程序中所有与文件操作有关的部分。在完成较大的 VHDL 程序的仿真时,由于输入信号很多,输入数据复杂,这时可以采用文件操作的方式设置输入信号。将仿真时输入信号所需要的数据用文本编辑器写到一个磁盘文件中,然后在 VHDL 程序的仿真驱动信号生成模块中调用 STD. TEXTIO 程序包中的子程序,读取文件中的数据,经过处理后或直接驱动输入信号端。

仿真的结果或中间数据也可以用 STD. TEXTIO 程序包中提供的子程序保存在文本文件中,这对复杂的 VHDL 设计的仿真尤为重要。

VHDL 仿真器 ModelSim 支持许多操作子程序,附带的 STD. TEXTIO 程序包源程序是很好的参考文件。

下面简要说明一下 TEXTIO 中读、写文件的语句的书写格式。

1) 从文件中读一行

```
READLINE(文件变量,行变量);
```

READLINE 用于从指定的文件中读一行的语句。

2）从一行中读一个数据

```
READ(行变量,数据变量);
```

利用 READ 语句可以从一行中取出一个字符，放到所指定的数据变量（信号）中。

3）写一行到输出文件

```
WRITELINE(文件变量,行变量);
```

该行写语句与行读语句相反，将行变量中存放的一行数据写到文件变量所指定的文件中去。

4）写一个数据至行

```
WRITE(行变量,数据变量);
```

该写语句将一个数据写到某一行中。

5）文件结束检查

```
ENDFILE(文件变量);
```

该语句检查文件是否结束，如果检查出文件结束标志，则返回“真”值，否则返回“假”值。

TEXTIO 常用于测试图的输入和输出。在使用 TEXTIO 的包集合时，首先要进行必要的说明，例如：

```
LIBRARY STD;
USE STD.TEXTIO.ALL;
```

在 VHDL 的标准格式中，TEXTIO 只能使用“BIT”和“BIT_VECTOR”两种数据类型。如果要使用“STD_LOGIC”和“STD_LOGIC_VECTOR”，就要调用“STD_LOGIC_TEXTIO”，即：

```
USE IEEE.STD_LOGIC_TEXTIO.ALL;
```

3. ASSERT 语句

断言语句主要用子程序仿真、调试中的人机对话，它可以给出一个文字串作为警告和错误信息。其一般格式为：

```
ASSERT 条件表达式 [REPORT 信息][SEVERITY 级别];
```

其中：条件表达式为布尔表达式，如果表达式值是真，ASSERT 语句任何事不做；如果表达式值是假，则输出错误信息和错误严重程度的级别。信息是文字串，通常用以说明错误的原因。文字串应用双引号“”括起来。级别是指错误严重程度的级别。在 VHDL 中错误严重程度分为四个级别：失败（FAILURE）、出错（ERROR）、警告（WARING）和注意（NOTE）。

例如：

```
ASSERT NOT (reset = '0') AND (preset = '0')
REPORT" Control error" SEVERITY Error;      -- 断言语句,检查是否有置位和清零同时
                                            -- 作用的错误
```

ASSERT 语句可以作为顺序语句使用，也可以作为并行语句使用。作为并行语句时，

ASSERT 语句可看作一个被动进程。

4. REPORT 语句

REPORT 语句类似于 ASSERT 语句,区别是它没有条件。其语句格式如下。

```
REPORT 信息[SEVERITY 级别];
```

例如:

```
WHILE COUNTER < = 100 LOOP
  IF COUNTER > 50
     THEN REPORT "THE COUNTER OVER 50";
  END IF;
  …
END LOOP;
```

在 VHDL'93 标准中,REPORT 语句相当于前面省略了 ASSERT FALSE 的 ASSERT 语句,而在 1987 标准中不能单独使用 REPORT 语句。

5.3 VHDL 并行语句

相对于传统的软件描述语言,并行语句结构是最具 VHDL 特色的。在 VHDL 中,并行语句具有多种语句格式,各种并行语句在结构体中的执行是同步进行的,或者说是并行运行的,其执行方式与书写的顺序无关。在执行中,并行语句之间可以有信息交流,也可以是互为独立、互不相关、异步运行的(如多时钟情况)。每一并行语句内部的语句运行方式可以有两种不同的方式,即并行执行方式(如块语句)和顺序执行方式(如进程语句)。结构体中的并行语句主要有进程语句、并行信号赋值语句、块语句、元件例化语句、生成语句、并行过程调用语句等。

并行语句在结构体中的使用格式如下。

```
ARCHITECTURE 结构体名 OF  实体名 IS
  说明语句
  BEGIN
   并行语句
END ARCHITECTURE 结构体名
```

5.3.1 进程语句

进程(PROCESS)语句是最具 VHDL 特色的语句。因为它提供了一种算法(顺序语句)描述硬件行为的方法。进程实际上是用顺序语句描述的一种进行过程,也就是说,进程用于描述顺序事件。一个结构体中可以有多个并行运行的进程结构,而每一个进程的内部结构却是由一系列顺序语句来构成的。

1. PROCESS 语句格式

PROCESS 语句的表达格式如下。

```
[进程标号: ]PROCESS[(敏感信号参数表)][IS]
[进程说明部分]
```

```
BEGIN
  顺序描述语句
END PROCESS[进程标号];
```

进程说明部分用于定义该进程所需的局部数据环境。

顺序描述语句部分是一段顺序执行的语句，描述该进程的行为。PROCESS 中规定了每个进程语句在它的某个敏感信号（由敏感信号参量表列出）的值改变时都必须立即完成某一功能行为。这个行为由进程顺序语句定义，行为的结果可以赋给信号，并通过信号被其他的 PROCESS 或 BLOCK 读取或赋值。当进程中定义的任一敏感信号发生更新时，由顺序语句定义的行为就要重复执行一次，当进程中最后一个语句执行完成后，执行过程将返回到第一个语句，以等待下一次敏感信号变化，如此循环往复以至无限。但当遇到 WAIT 语句时，执行过程将被有条件地终止，即所谓的挂起（Suspention）。

一个结构体中可含有多个 PROCESS 结构，每个进程可以在任何时刻被激活或者称为启动。而所有被激活的进程都是并行运行的，这就是为什么 PROCESS 结构本身是并行语句的道理。

2. PROCESS 组成

PROCESS 语句结构是由三个部分组成的，即进程说明部分、顺序描述语句部分和敏感信号参数表。

（1）进程说明部分主要定义一些局部量，可包括数据类型、常数、属性、子程序等。但需注意，在进程说明部分中不允许定义信号和共享变量。

（2）顺序描述语句部分可分为赋值语句、进程启动语句、子程序调用语句、顺序描述语句和进程跳出语句等。

① 信号赋值语句：即在进程中将计算或处理的结果向信号赋值。

② 变量赋值语句：即在进程中以变量的形式存储计算的中间值。

③ 进程启动语句：当 PROCESS 的敏感信号参数表中没有列出任何敏感量时，进程的启动只能通过进程启动语句 WAIT 语句。这时可以利用 WAIT 语句监视信号的变化情况，以便决定是否启动进程。WAIT 语句可以看作一种隐式的敏感信号表。

④ 子程序调用语句：对已定义的过程和函数进行调用，并参与计算。

⑤ 顺序描述语句：包括 IF 语句、CASE 语句、LOOP 语句和 NULL 语句等。

⑥ 进程跳出语句：包括 NEXT 语句和 EXIT 语句。

（3）敏感信号参数表需列出用于启动本进程可读入的信号名（当有 WAIT 语句时例外）。

3. 进程设计要点

进程的设计需要注意以下几方面的问题。

（1）虽然同一结构体中的进程之间是并行运行的，但同一进程中的逻辑描述语句则是顺序运行的，因而在进程中只能设置顺序语句。

（2）进程的激活必须由敏感信号表中定义的任一敏感信号的变化来启动，否则必须有一显式的 WAIT 语句来激活。这就是说，进程既可以由敏感信号的变化来启动，也可以由满足条件的 WAIT 语句来激活。反之，在遇到不满足条件的 WAIT 语句后，进程将被挂起。因此，进程中必须定义显式或隐式的敏感信号。如果一个进程对一个信号集合总是敏感的，那么，可以使用敏感表来指定进程的敏感信号。但是，在一个使用了敏感表的进程（或

者由该进程所调用的子程序)中不能含有任何等待语句。

(3) 结构体中多个进程之所以能并行同步运行,一个很重要的原因是进程之间的通信是通过传递信号和共享变量值来实现的。所以相对于结构体来说,信号具有全局特性。它是进程间进行并行联系的重要途径。因此,在任一进程的进程说明部分不允许定义信号(共享变量是 VHDL'93 增加的内容)。

(4) 进程是重要的建模工具。进程结构不但为综合器所支持,而且进程的建模方式将直接影响仿真和综合结果。需要注意的是,综合后对应于进程的硬件结构,对进程中的所有可读入信号都是敏感的,而在 VHDL 行为仿真中并非如此,除非将所有的读入信号列为敏感信号。

进程语句是 VHDL 程序中使用最频繁和最能体现 VHDL 特点的一种语句,其原因是由于它的并行和顺序行为的双重性,以及行为描述风格的特殊性。为了使 VHDL 的软件仿真与综合后的硬件仿真对应起来,应当将进程中的所有输入信号都列入敏感表中。不难发现,在对应的硬件系统中,一个进程和一个并行赋值语句确实有十分相似的对应关系,并行赋值语句就相当于一个将所有输入信号隐性地列入结构体监测范围的(即敏感表的)进程语句。

综合后的进程语句所对应的硬件逻辑模块,其工作方式可以是组合逻辑方式的,也可以是时序逻辑方式的。例如,在一个进程中,一般的 IF 语句,在一定条件下综合出的多为组合逻辑电路;若出现 WAIT 语句,在一定条件下,综合器将引入时序元件,如触发器。

例 5-22 中有两个进程:p_a 和 p_b,它们的敏感信号分别为 a、b、selx 和 temp、c、sely。除 temp 外,两个进程完全独立运行,除非两组敏感信号中的一对同时发生变化,两个进程才被同时启动。

【例 5-22】 进程的应用

```
ENTITY mul IS
PORT(a,b,c,selx,sely: IN BIT;
     data_out: OUT BIT);
END mul;
ARCHITECTURE ex OF mul IS
  SIGNAL temp: BIT;
BEGIN
p_a: PROCESS(a,b,selx)
    BEGIN
      IF(SELX = '0')THEN temp < = a;
      ELSE temp < = b;
      END IF;
END PROCESS p_a;
p_b: PROCESS(temp,c,sely)
    BEGIN
     IF (sely = '0') THEN data_out < = temp;
     ELSE data_out < = c;
     END IF;
END PROCESS p_b;
END ex;
```

5.3.2 并行信号赋值语句

并行信号赋值语句有三种形式：简单信号赋值语句、条件信号赋值语句和选择信号赋值语句。这三种信号赋值语句的共同点是赋值目标必须都是信号，所有赋值语句与其他并行语句一样，在结构体内的执行是同时发生的，与它们的书写顺序和是否在块语句中没有关系。每一信号赋值语句都相当于一条缩写的进程语句，而这条语句的所有输入信号都被隐性地列入此过程的敏感信号表中。因此，任何信号的变化都将启动相关并行语句的赋值操作，而这种启动完全是独立于其他语句的，它们都可以直接出现在结构体中。

1. 简单信号赋值语句

简单信号赋值语句是 VHDL 并行语句结构的最基本的单元，它的语句格式如下。

```
赋值目标<= 表达式
```

式中赋值目标的数据对象必须是信号，它的数据类型必须与赋值符号右边表达式的数据类型一致。例 5-23 结构体中的五条信号赋值语句的执行是并行发生的。

【例 5-23】 简单信号赋值语句

```
ARCHITECTURE curt OF bcl IS
SIGNAL s,e,f,g,h: STD_LOGIC;
BEGIN
    Output1 <= a AND b ;
    Output2 <= c + d;
    g <= e OR f ;
    h <= e XOR f ;
    s1 <= g;
END ARCHITECTURE curt;
```

2. 条件信号赋值语句

作为另一种并行赋值语句，条件信号赋值语句的表达方式如下。

```
赋值目标<= 表达式 1 WHEN 赋值条件 1 ELSE
           表达式 2 WHEN 赋值条件 2   ELSE
           …
           表达式 n;
```

在结构体中条件信号赋值语句的功能与在进程中的 IF 语句相同，在执行条件信号语句时，每一赋值条件是通过书写的先后关系逐项测定的，一旦发现赋值条件为 TRUE，立即将表达式的值赋给目标变量。从这个意义上讲，条件赋值语句与 IF 语句具有十分相似的顺序性（注意，条件赋值语句中的 ELSE 不可省略），这意味着，条件信号赋值语句将第一个满足关键词 WHEN 后的赋值条件所对应的表达式中的值，赋给赋值目标信号，这里的赋值条件的数据类型是布尔量，当它为真时表示满足赋值条件，最后一项表达式可以不跟条件子句，用于表示以上各条件都不满足时，则将此表达式赋予赋值目标信号。由此可知，条件信号语句允许有重叠现象，这与 CASE 语句有很大的不同，应注意辨别。

例 5-24 就是条件信号赋值语句的应用例子。应该注意，由于条件测试的顺序性，第一子句具有最高赋值优先级，第二句其次，第三句最后。这就是说，当 P1 和 P2 同时为 1 时，Z

获得的赋值是 a。

【例 5-24】 条件信号赋值

```
ENTITY mux IS
  PORT(a,b,c: IN BIT;
       p1,p2: IN BIT;
          z: OUT BIT);
END mux;
ARCHITECTURE behave OF mux IS
  BEGIN
   z <= a WHEN p1 = '1' ELSE
      b WHEN p2 = '1' ELSE
      c;
END;
```

3. 选择信号赋值语句

选择信号赋值语句的语句格式如下。

```
WITH 选择表达式 SELECT
赋值目标信号<= 表达式 1 WHEN 选择值 1,
               表达式 2 WHEN 选择值 2,
               ...
               表达式 n  WHEN 选择值 n;
```

选择信号赋值语句本身不能在进程中应用,但其功能却与进程中的 CASE 语句的功能相似。CASE 语句的执行依赖于进程中敏感信号的改变,而且要求 CASE 语句中各子句的条件不能有重叠,必须包容所有的条件。

选择信号语句中也有敏感量,即关键词 WITH 旁的选择表达式,每当选择表达式的值发生变化时,就将启动此语句对各子句的选择值进行测试对比,当发现有满足条件的子句时,就将此子句表达式中的值赋给赋值目标信号。与 CASE 语句相类似。选择赋值语句对子句各选择值的测试具有同期性,不像条件信号赋值语句那样是按照子句的书写顺序从上至下逐条测试的,因此,选择赋值语句不允许有条件重叠的现象,也不允许存在条件涵盖不全的情况。

例 5-25 是一个简化的指令译码器,对应有 a、b、c 三个位构成的不同指令码,由 datal 和 data2 输入的两个值将进行不同的逻辑操作,并将结果从 dataout 输出,当不满足所列的指令时,将输出高阻态。

【例 5-25】 选择信号赋值

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY decoder IS
  PORT(a,b,c: IN STD_LOGIC;
       data1,data2: IN STD_LOGIC;
       dataout: OUT STD_LOGIC);
END decoder;
ARCHITECTURE concunt OF decoder IS
  SIGNAL instruction: STD_LOGIC_VECTOR(2 DOWNTO 0);
```

```
BEGIN
  instruction <= c&b&a;
  WITH instruction SELECT
  dataout <= data1 AND data2 WHEN "000",
            data1 OR data2 WHEN "001",
            data1 NAND data2 WHEN "010",
            data1 NOR data2 WHEN "011",
            data1 XOR data2 WHEN "100",
            data1 XNOR data2 WHEN "101",
                        'Z' WHEN  OTHERS;
END concunt;
```

注意：选择信号赋值语句的每一个子句结尾是逗号，最后一句是分号；而条件赋值语句每一子句的结尾没有任何标点，只有最后一句为分号。

5.3.3 块语句结构

块(BLOCK)的应用类似于利用 PROTEL 画电路原理时，可将一个总的原理图分成多个子模块，则这个总的原理图成为一个由多个子模块原理连接而成的顶层模块图，而每一个模块可以是一个具体的电路原理图。但是，如果子模块的原理图仍然太大，还可将它变成更低层次的原理图模块的连接图(BLOCK 嵌套)。显然，按照这种方式划分结构体仅是形式上的，而非功能上的改变。事实上，将结构体以模块方式划分的方法有多种，使用元件例化语句也是一种将结构体并行描述分成多个层次的方法，其区别只是后者涉及多个实体和结构体，且综合后硬件结构的逻辑层次有所增加。

实际上，结构体本身就等价于一个 BLOCK，或者说是一个功能块。BLOCK 是 VHDL 中具有的一种划分机制，这种机制允许设计者合理地将一个模块分为数个区域，在每个块都能对其局部信号、数据类型和常量加以描述和定义。任何能在结构体的说明部分进行说明的对象都能在 BLOCK 说明部分中进行说明。BLOCK 语句应用只是一种将结构体中的并行描述语句进行组合的方法，客观存在的主要目的是改善并行语句及其结构的可读性，或是利用 BLOCK 的保护表达式关闭某些信号。BLOCK 语句的表达式如下。

```
块标号: BLOCK[(块保护表达式)]
        说明语句
        BEGIN
        并行语句
END BLOCK 块标号;
```

作为一个 BLCOK 语句结构，在关键词“BLOCK”的前面必须设置一个块标号，并在结尾语句“END BLOCK”右侧也写上此标号(此处的块标号不是必需的)。

其中说明语句又包括类属说明语句和端口说明语句，类属语句主要用于参数的定义，而端口说明语句主要用于信号的定义，它们通常是通过 GENERIC 语句、GENERIC MAP 语句、PORT 语句和 PORT MAP 语句来实现的。说明语句主要是对 BLOCK 的接口设置以及外界信号的连接状态加以说明。

块的类属说明部分和接口说明部分的适用范围仅限于当前 BLOCK。所以，所有这些在 BLOCK 内部的说明对于这个块的外部来说是完全不透明的，即不能适用于外部环境，或

由外部环境所调用，但对于嵌套于更内层的块却是透明的，即可将信息向内部传递。块的说明部分可以定义的项目主要有：USE 语句、子程序、数据类型、子类型、常数、信号和元件。

块中的并行语句部分可包含结构体中的任何并行语句结构。BLOCK 语句本身属于并行语句，BLOCK 语句中所包含的语句也是并行语句。BLOCK 的应用可使结构体层次鲜明，结构明确。利用 BLOCK 语句可以将结构体中的并行语句划分为多个并列方式的 BLOCK，每一个 BLOCK 都像一个独立的设计实体，具有自己的类属参数说明和界面端口，以及与外部环境的衔接描述。在较大的 VHDL 程序的编程中，恰当的块语句的应用对于技术交流、程序移植、排错和仿真都是有益的。

例 5-26 是一个广泛使用的微处理器的 VHDL 源代码，在其中就使用了多层块嵌套。

【例 5-26】 块的应用

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE bit32 IS
  TYPE tw32 IS ARRAY(31 DOWNTO 0)OF STD_LOGIC;
END bit32;
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE WORK.bit32.ALL;
ENTITY cpu IS
  PORT(clk,interrupt: IN STD_LOGIC;
       add: OUT tw32;
       data: INOUT tw32);
END cpu;
ARCHITECTURE behave OF cpu IS
SIGNAL ibus,dbus: tw32;
BEGIN
  alu: BLOCK
  SIGNAL qbus: tw32;
  BEGIN
    -- ALU 行为描述
  END BLOCK alu;
  reg8: BLOCK
  SIGNAL zbus: tw32;
  BEGIN
    reg1: BLOCK
    SIGNAL qbus: tw32;
    BEGIN
     -- REG 行为描述
    END BLOCK reg1;
     -- 其他 REG 行为描述语句
  END BLOCK reg8;
END behave;
```

在例 5-26 中，CPU 模块有四个端口；输入端口 clk 和 interrupt；输出端口 add；双向端口 data。在 CPU 的结构体中使用 BLOCK 语句描述了 ALU 模块和 REG 模块，相对于这些块，以上四个端口是可见的，可以在块内使用。

在结构体中定义了 ibus 和 dbus，它们是该结构体的内部信号。在此结构体内部的块都可以使用这两个信号。

在 reg8 中说明了信号 zbus，这个信号可以在 reg8 内部使用，包括其中的 regl。但是对 reg8 块外部是不透明的，即对于 ALU 块是不能使用的。

内层嵌套块 regl 中说明了信号 qbus，虽然它与 ALU 中说明的信号是同名的，但是它属于内部信号，仅在此块范围内有效，但以后为了避免不必要的错误，在编程中尽量不要出现这种命名。

5.3.4 并行过程调用语句

并行过程调用语句可以作为一个并行语句直接出现在结构体或块语句中。并行过程调用语句的功能等效于一个只有一个过程调用的进程，过程参数的模式只能为 IN、OUT 或 INOUT，当参数之一改变时，过程调用就会被激活。并行过程调用语句的语句调用格式与顺序过程调用语句是相同的，即：

```
[过程标号: ]过程名(关联参量名);
```

例 5-27 就是并行过程调用的应用实例，它的主要功能是取出三个输入中值最大的那一个。

【例 5-27】 并行过程的调用

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY mft IS
  PORT(a: IN STD_LOGIC;
       b: IN STD_LOGIC;
       C: IN STD_LOGIC;
       q: OUT STD_LOGIC);
END mft;
ARCHITECTURE behave OF mft IS
  PROCEDURE max(ina,inb: IN STD_LOGIC;      -- 定义过程 max
                SIGNAL ouc: OUT STD_LOGIC)IS
  VARIABLE temp: STD_LOGIC;
  BEGIN
    IF(ina < inb) THEN
      temp: = inb;
    ELSE
      temp: = ina;
    END IF;
    ouc < = temp;
  end max;
SIGNAL temp1,temp2: STD_LOGIC;
BEGIN
  max(a,b,temp1);                          -- 调用过程 max
  max(temp1,c,temp2);                      -- 调用过程 max
  q < = temp2;
END behave;
```

5.3.5 元件例化语句

在第4章曾对此语句做了简要介绍，本节将做进一步介绍。元件例化是可以多层次的，在一个设计实体中被调用安插的元件本身也可以是一个低层次的当前设计实体，因而可以调用其他的元件，以便构成更低层次的电路模块。因此元件例化就意味着在当前结构体内定义了一个新的设计层次，这个设计层次的总称叫作元件，但它可以以不同的形式出现，这个元件可以是已设计好的一个VHDL设计实体，可以是来自FPGA元件库中的元件，它们可能是以别的硬件描述语言，如Verilog设计的实体；元件还可以是IP核，或者是FPGA中的嵌入式硬IP核。

元件例化语句由两部分组成，前一部分是把一个现成的设计实体定义为一个元件，第二部分则是此元件与当前设计实体中的连接说明，它们的完整的语句格式如下。

```
COMPONENT   元件名   IS                          -- 元件定义语句
GENERIC   (类属表);
PORT(端口名表);
END COMPONENT   元件名;
例化名: 元件名   PORT MAP(                       -- 元件例化语句
        [端口名 =>]连接端口名, …);
```

以上两部分语句在元件例化时都是必须存在的，第一部分语句是元件定义语句，相当于对一个现成的设计实体进行封装，使其只留出对外的接口界面，就像一个集成芯片只留几个引脚在外面一样，它的类属表可列出端口的数据类型和参数，端口名表可列出对外通信的各端口名。元件例化的第二部分语句即为元件例化语句，其中的例化名是必须存在的，它类似于标在当前系统(电路板)中的一个插座名，而元件名则是准备在此插座上插入的、已定义好的元件名。PORT MAP是端口映射的意思，其中的端口名是在元件定义语句中的端口名表中已定义好的元件端口的名字，连接端口名则是当前系统与准备接入的元件对应端口相连的通信端口，相当于插座上各插针的引脚名。

元件例化语句中所定义的元件的端口名与当前系统的连接端口名的接口表达有两种方式，一种是名字关联方式。在这种关联方式下，例化元件的端口名和关联符号“=>”两者都是必须存在的。这时，端口名与连接端口名的对应式，在PORT MAP句中的位置可以是任意的。另一种是位置关联方式，在使用这种方式时，端口名和关联连接符号都可省去，在PORT MAP子句中，只要列出当前系统中的连接端口名就行了，但要求连接端口名的排列方式与所需例化的元件端口定义中的端口名一一对应。

例5-28以一个4位移位寄存器为例，进一步说明元件例化语句应用，它由4个相同的D触发器组成。

【例5-28】 元件例化语句

```
ENTITY shifter IS
  PORT(din,clk: IN BIT;
       dout: OUT BIT);
END shifter;
ARCHITECTURE a OF shifter IS
  COMPONENT dff
```

```
    PORT(d,clk: IN BIT;
         q: OUT BIT);
  END COMPONENT;
SIGNAL d: BIT_VECTOR(0 TO 4);
  BEGIN
    d(0)<=din;                                          --并行信号赋值
U0: dff PORT MAP(d(0),clk,d(1));                        --位置关联方式
U1: dff PORT MAP(d(1),clk,d(2));
U2: dff PORT MAP(d=>d(2),clk=>clk,q=>d(3));             --名字关联方式
U3: dff PORT MAP(d=>d(3),clk=>clk,q=>d(4));
dout<=d(4);
END a;
```

例 5-28 所描述的 4 位移位寄存器实际电路如图 5-1 所示。

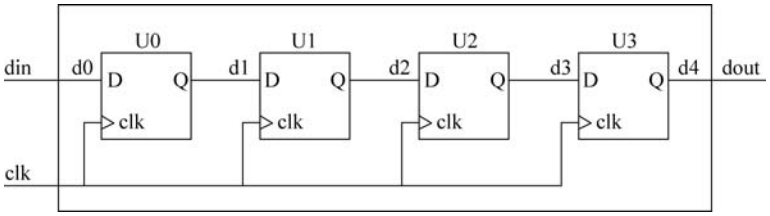


图 5-1　4 位移位寄存器结构图

元件例化语句是一种应用十分广泛的 VHDL 语句，它使得在进行 VHDL 描述时可以使用以前建立的 VHDL 模块，避免大量的重复工作。

5.3.6　生成语句

生成语句可以简化有规则设计结构的逻辑描述，适用于高重复性的电路设计。生成语句有一种复制作用，在设计中，只要根据某些条件，设定好某一元件或设计单位，就可以利用生成语句复制一组完全相同的并行元件或设计单元电路结构，生成语句的语句格式有如下两种形式。

（1）[标号：]FOR 循环变量　IN 取值范围 GENERATE
　　说明
　　BEGIN
　　并行语句
　　END GENERATE [标号];

（2）[标号：] IF 条件 GENERATE
　　说明
　　BEGIN
　　并行语句
　　END GENERATE [标号];

这两种语句格式都是由如下四部分组成的。

（1）生成方式：由 FOR 或 IF 语句结构构成，用于规定并行语句的复制方式。

（2）说明部分：包括对元件数据类型、子程序、数据对象做一些局部说明。

（3）并行语句：对被复制的元件的结构和行为进行描述。主要包括元件、进程语句、块语句、并行过程用语句、并行信号赋值语句，甚至生成语句，这表示生成语句允许存在嵌套结

构，因而可用于生成元件的多维阵列结构。它是用来进行“Copy”的基本单元。

(4) 标号：其中的标号并非必需，但如果在嵌套式生成语句结构中就是十分重要的。

1. FOR 格式的生成语句

FOR 格式的生成语句主要是用来描述设计中一些有规律的单元结构，其生成参数及其取值范围的含义和运行方式与 LOOP 语句十分相似，但是在生成语句中使用的是并行处理语句，因此在结构内部的语句不是按书写顺序执行的，而是并发执行的。FOR_GENERATE 语句结构中不能使用 EXIT 语句和 NEXT 语句。

生成参数(循环变量)是自动产生的，它是一个局部变量，根据取值范围自动递增或递减。取值范围的语句格式与 LOOP 语句是相似的，有以下两种形式。

```
表达式    TO   表达式;          -- 递增方式,如 1  TO   5
表达式  DOWNTO   表达式;       -- 递减方式,如 5  DOWNTO  1
```

其中的表达式必须是整数。

例 5-29 和例 5-30 将利用元件例化语句和 FOR GENERATE 生成语句完成一个 8 位三态锁存器的设计。示例仿照 74373 的工作逻辑进行设计。例 5-29 是一个一位的锁存器，例 5-30 为顶层文件，端口信号 d 为数据输入端，q 为数据输出端，ena 为输出使能端，若 ena=1，则 q8～q1 的输出为高阻态，若 ena=0，则输出保存在锁存器中；g 为数据锁存控制端，若 g=1，d8～d1 输入端的信号进入 74373 中的 8 位锁存器中，若 g=0，74373 中的 8 位锁存器将保持原先锁入的信号值不变。

【例 5-29】 1 位锁存器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY latch IS
  PORT(d: IN STD_LOGIC;
       ena: IN STD_LOGIC;
       q: OUT STD_LOGIC);
END ENTITY latch;
ARCHITECTURE one OF latch IS
  SIGNAL sig_save: STD_LOGIC;
  BEGIN
    PROCESS(d,ena)
      BEGIN
        IF ena = '1' THEN sig_save <= d;
        END IF;
        q <= sig_save;
    END PROCESS;
END one;
```

【例 5-30】 顶层文件

```
LIBRARY IEEE;
  USE IEEE.STD_LOGIC_1164.ALL;
  ENTITY SN74373 IS
    PORT(d: IN STD_LOGIC_VECTOR(8 DOWNTO 1);
         oen,g: IN STD_LOGIC;
```

```
        q: OUT STD_LOGIC_VECTOR(8 DOWNTO 1));
END ENTITY SN74373;
ARCHITECTURE two OF SN74373 IS
  SIGNAL sigvec_save: STD_LOGIC_VECTOR(8 DOWNTO 1);
   BEGIN
   PROCESS(d,oen,g,sigvec_save)
     BEGIN
       IF oen = '0' THEN q < = sigvec_save;
       ELSE  q < = "ZZZZZZZZ";
       END IF;
       IF g = '1' THEN sigvec_save < = d;
       END IF;
   END PROCESS;
 END ARCHITECTURE two;
ARCHITECTURE one OF SN74373 IS
  COMPONENT latch
   PORT(d,ena: IN STD_LOGIC;
         q: OUT STD_LOGIC);
   END COMPONENT;
   SIGNAL sig_mid: STD_LOGIC_VECTOR(8 DOWNTO 1);
   BEGIN
     gelatch: FOR inum IN 1 to 8 GENERATE
     latchx: latch PORT MAP(d(inum),g,sig_mid(inum));
   END GENERATE;
     q < = sig_mid WHEN oen = '0' ELSE
        "ZZZZZZZZ";
 END ARCHITECTURE one;
```

由例 5-30 可以看出：

(1) 程序中安排了两个结构体，以不同的电路来实现相同的逻辑，即一个实体可以对应多个结构体，每个结构体对应一种实现方案。在例化这个器件的时候，需要利用配置语句指定一个结构体，即指定一种实现方案，否则 VHDL 综合器会自动选择最新编译的结构体，在本例中即为结构体 one。

(2) COMPONENT 语句对将要例化的器件进行了接口声明，它对应一个已设计好的实体(例 5-29)。VHDL 综合器根据 COMPONENT 指定的器件名和接口信息来装配器件。

(3) 在 FOR GENERATE 语句使用中，gelatch 为标号，inum 为变量，从 1 到 8 共循环了 8 次。

(4) 语句“latchx：latch PORT MAP(d(inum)，g，sig_mid(inum))；”是一条含有循环变量 inum 的例化语句，且信号的连接方式采用的是位置关联方式，安装后的元件标号是 latchx。latch 的引脚 d 连在信号线 d(inum)上，引脚 ena 连在信号线 g 上，引脚 q 连在信号线 sig_mid(inum)上。inum 的值为 1～8，latch 从 1 到 8 共例化了 8 次，即共安装了 8 个 latch。信号线 d(1)～d(8)，sig_mid(1)～sig_mid(8)都分别连在这 8 个 latch 上。

2. IF 格式的生成语句

IF 格式的生成语句主要是用来描述产生例外的情况。当执行到该语句时，首先进行条件的判断，如果条件为 Ture，则执行生成语句中的并行处理语句，否则不执行该语句。

IF 格式的生成语句与普通的 IF 语句有着很大的不同，普通的 IF 语句中的处理语句是

顺序执行的，而 IF 格式的生成语句中的并行语句却是并行执行的。另外，IF 格式的生成语句中是不能出现 ELSE 语句的。现以例 5-28 介绍过的移位寄存器为例来介绍 IF 格式的生成语句的使用。首先描述 D 触发器，如例 5-31 所示。

【例 5-31】 D 触发器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY d_ff IS
  PORT(clk,d: IN STD_LOGIC;
       q: OUT STD_LOGIC);
END d_ff;
ARCHITECTURE behave OF d_ff IS
  SIGNAL q_in: STD_LOGIC;
  BEGIN
    q <= q_in;
    PROCESS(clk)
    BEGIN
      IF(clk'EVENT AND clk = '1')THEN
        q_in <= d;
      END IF;
    END PROCESS;
END behave;
```

然后生成 4 位移位寄存器，如例 5-32 所示。

【例 5-32】 4 位移位寄存器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY shift_reg IS
  PORT(d1: IN STD_LOGIC;
       cp: IN std_LOGIC;
       d0: OUT STD_LOGIC);
END shift_reg;
ARCHITECTURE behave OF shift_reg IS
  COMPONENT d_ff
    PORT(d: IN STD_LOGIC;
         clk: IN STD_LOGIC;
         q: OUT STD_LOGIC);
  END COMPONENT;
SIGNAL q: STD_LOGIC_VECTOR(3 DOWNTO 1);
BEGIN
  l: FOR i IN 0 TO 3 GENERATE                    -- FOR 格式的生成语句
     m: IF(i = 0) GENERATE                       -- IF 格式的生成语句
       dffx: d_ff PORT MAP(d1,cp,q(i + 1));
     END GENERATE m;
     n: IF(i = 3) GENERATE                       -- IF 格式的生成语句
       dffx: d_ff PORT MAP(q(i),cp,d0);
     END GENERATE n;
     o: IF((i/ = 0) AND (i/ = 3)) GENERATE       -- IF 格式的生成语句
       dffx: d_ff PORT MAP(q(i),cp,q(i + 1));
```

```
    END GENERATE o;
  END GENERATE l;
END behave;
```

在例 5-32 中，FOR 格式的生成语句中使用了 IF 格式的生成语句。IF 格式的生成语句首先进行条件判断，判断所使用的 D 触发器是第一个还是最后一个。可以使用 IF 格式的生成语句来解决硬件电路中输入/输出端口的不规则问题。

在实际应用中可以把两种格式混合使用，设计中，可以根据电路两端的不规则部分形成的条件用 IF GENERATE 语句来描述，而用 FOR GENERATE 语句描述电路内部的规则部分。使用这种描述方法的好处是，使设计文件具有更好的通用性、可移植性和易改性。实用中，只要改变几个参数，就能得到任意规模的电路结构。

5.4 子程序

子程序是一个 VHDL 程序模块，它是利用顺序语句来定义和完成算法的，应用它能更有效地完成重复性的设计工作。子程序不能像进程那样可以从所在结构体的其他块或进程结构中读取信号值或者向信号赋值，而只能通过子程序调用及与子程序的界面端口进行通信。

子程序有两种类型，即过程（PROCEDURE）和函数（FUNCTION）。过程和函数的区别在于：过程的调用可通过其界面获得多个返回值，而函数只能返回一个值；在函数入口中，所有参数都是输入参数，而过程有输入参数、输出参数和双向参数；过程一般被看作一种语句结构，而函数通常是表达式的一部分；过程可以单独存在，而函数通常作为语句的一部分调用。

子程序可以在 VHDL 程序的三个不同的位置进行定义，即在程序包、结构体和进程中定义。但由于只有在程序包中定义的子程序可被其他不同的设计所调用，所以一般应该将子程序放在程序包中。VHDL 子程序有一个非常有用的特性，就是具有可重载性的特点，即允许有许多重名的子程序，但这些子程序的参数类型及返回值数据类型是不同的。

在实用中必须注意，综合后的子程序将映射于目标芯片中的一个相应的电路模块，且每一次调用都将在硬件结构中产生具有相同结构的不同模块，这一点与在普通的软件中调用子程序有很大的不同，因此，在 VHDL 的编程过程中，要密切关注和严格控制子程序的调用次数，每调用一次子程序都意味着增加了一个硬件电路模块。

5.4.1 函数

在 VHDL 中有多种函数（FUNCTION）形式，如在库中现成的具有专用功能的预定义函数和用于不同目的的用户自定义函数。函数的表达式如下。

```
FUNCTION   函数名(参数表)  RETURN  数据类型;    -- 函数首
FUNCTION   函数名(参数表)   RETURN  数据类型 IS -- 函数体开始
[说明部分];
BEGIN
顺序语句;
END  FUNCTION     函数名;
```

一般地，函数定义由两部分组成，即函数首和函数体。

1. 函数首

函数首是由函数名、参数表和返回值的数据类型三部分组成的。函数首的名称即为函数的名称，需放在关键词 FUNCTION 之后，它可以是普通的标识符，也可以是运算符，这时必须加上双引号，这就是所谓的运算符重载。函数的参数表是用来定义输出值的，它可以是信号或常数。参数名需放在关键词 CONSTANT 或 SIGNAL 之后，若没有特别说明，则参数被默认为常数。如果要将一个已编制好的函数并入程序包，函数首必须在程序包的说明部分，而函数体需放在程序包的包体内。如果只是在一个结构体中定义并调用函数，则仅需函数体即可。由此可见，函数首的作用只是作为程序包的有关此函数的一个接口界面。下面是四个不同的函数首，它们都放在某一程序包的说明部分。

```
FUNCTION  max(a,b: IN STD_LOGIC_VECTOR)
          RETURN STD_LOGIC_VECTOR;
FUNCTION func1(a,b,c: REAL)
          RETURN  REAL;
FUNCTION "*"(a,b: INTEGER)
          RETURN  INTEGER;
FUNCTION  as2 (SIGNAL in1,in2: REAL)
         RETURN  REAL;
```

2. 函数体

函数体包括对数据类型、常数、变量等的局部说明，以及用以完成规定算法或转换顺序语句，并以关键词 END FUNCTION 以及函数名结尾。一旦函数被调用，就将执行这部分语句。

例 5-33 在一个结构体中定义了一个函数 sam，功能是完成输入总线各位的运算操作，然后将结果输出到输出总线的各位上。在进程 PROCESS 中调用了此函数，这个函数没有函数首。在进程中，输入端口信号位矢 a 被列为敏感信号，当 a 的 3 个输入元素 a(0)、a(1)和 a(2)中的任何一位有变化时，将启动对函数 sam 的调用，并将函数的返回值赋给 m 输出。

【例 5-33】 *函数的应用*

```
LIBRARY IEEE;
USE IEEE.std_LOGIC_1164.ALL;
ENTITY func IS
  PORT(a: IN STD_LOGIC_VECTOR(0 TO 2);
       m: OUT STD_LOGIC_VECTOR(0 TO 2));
END ENTITY func;
ARCHITECTURE demo OF func IS
  FUNCTION sam(x,y,z: STD_LOGIC) RETURN STD_LOGIC IS
  --定义函数 sam,该函数无函数首
  BEGIN
    RETURN(x AND y) OR z;
  END FUNCTION sam;
  BEGIN
    PROCESS(a)
    BEGIN
```

```
    m(0)<=sam(a(0),a(1),a(2));      --当3个位输入元素a(0),a(1),a(2)中的任何
    m(1)<=sam(a(2),a(0),a(1));      --一位有变化时,将启动对函数sam的调用,并
    m(2)<=sam(a(1),a(2),a(0));      --将函数的返回值赋给m输出
  END PROCESS;
END ARCHITECTURE demo;
```

例5-33中是在结构体中定义函数的例子。在通常情况下，函数是定义在程序包中的。在程序包的说明和包体中，可以分别描述函数的说明和函数的定义，这样可以将各种实用函数写入一个程序包中，并将其编译到库中以便在其他设计中使用。

5.4.2 重载函数

VHDL允许以相同的函数名定义函数，即重载函数（OVERLOADED FUNCTION）。但这时要求函数中定义的操作数具有不同的数据类型，以便调用时用以分辨不同功能的同名函数，即同样名称的函数可以用不同的数据类型作为此函数的参数定义多次，以此定义的函数称为重载函数。函数还可以允许用任意位矢长度来调用。在具有不同数据类型操作数构成的同名函数中，以运算符重载函数最为常用。这种函数为不同数据类型间的运算带来极大的方便，例5-34中以加号“+”为函数名的函数即为运算符重载函数。VHDL的IEEE库中的STD_LOGIC_UNSIGNED程序包中预定义的操作符如+、-、*、=>、<=、>、<、/=、AND和MOD等，对相应的数据类型INTEGRE、STD_LOGIC和STD_LOGIC_VECTOR的操作做了重载，赋予了新的数据类型操作功能，即通过重新定义运算符的方式，允许被重载的运算符能够对新的数据类型进行操作，或者允许不同的数据类型之间用此运算符进行运算。例5-34是程序包STD_LOGIC_UNSIGNED中的部分函数结构，其说明部分只列出了四个函数的函数首，在程序包体部分只列出了对应的部分内容，程序包体部分的UNSIGNED函数是从IEEE.STD_LOGIC_ARITH库中调用的，在程序包体中的最大整型数检出函数MAXIUM只有函数体，没有函数首，这是因为它只是在程序包内调用。

【例5-34】 程序包STD_LOGIC_UNSIGNED中的部分函数结构

```
LIBRARY IEEE;                          --程序包首
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
PACKAGE STD_LOGIC_UNSIGNED IS
function "+"(l: STD_LOGIC_VECTOR; r: INTEGER)
                RETURN STD_LOGIC_VECTOR;
function "+"(l: INTEGER; r: STD_LOGIC_VECTOR)
                RETURN STD_LOGIC_VECTOR;
function "+"(l: STD_LOGIC_VECTOR; r: STD_LOGIC)
                RETURN STD_LOGIC_VECTOR;
function  SHR(arg: STD_LOGIC_VECTOR;
                 count: STD_LOGIC_VECTOR)
                RETURN STD_LOGIC_VECTOR;
...
END STD_LOGIC_UNSIGNED;

LIBRARY IEEE;                          --程序包体
USE IEEE.STD_LOGIC_1164.ALL;
```

```
USE IEEE.STD_LOGIC_ARITH.ALL;
PACKAGE body STD_LOGIC_UNSIGNED IS
function maximum(l,r: INTEGER) RETURN INTEGER IS
BEGIN
  IF l > r THEN RETURN l;
  ELSE          RETURN r;
  END IF;
END;
function " + "(l: STD_LOGIC_VECTOR; r: INTEGER)
RETURN STD_LOGIC_VECTOR IS
VARIABLE result: STD_LOGIC_VECTOR(l'range);
BEGIN
  result: = UNSIGNED(L) + r;
  RETURN STD_LOGIC_VECTOR(result);
end;
…
END STD_LOGIC_UNSIGNED:
```

通过此例,不但可以从中看到在程序包中完整的函数置位形式,而且还将注意到,在函数首的三个函数名都是同名的,即都是以加法运算符“+”作为函数名。以这种方式定义函数即所谓运算符重载。对运算符重载(即对运算符重新定义)的函数称作重载函数。

实用中,如果已用“USE”语句打开了程序包 STD_LOGIC_VECTOR 位和一个整数相加,程序就会自动调用第一个函数,并返回位类型的值。若是一个位与 STD_LOGIC 数据相加,则调用第三个函数,并以位矢类型的值返回。例 5-35 为重载函数使用实例,其功能是实现 4 位二进制加法计数器。

【例 5-35】 重载函数使用

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY CNT4 IS
PORT(clk: IN STD_LOGIC;
     q: BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
END cnt4;
ARCHITECTURE one OF cnt4 IS
  BEGIN
  PROCESS(clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      IF q = 15 THEN          -- q 两边的数据类型不一致,程序自动调用了重载函数
         q < = "0000";
      ELSE
         q < = q + 1;         -- 程序自动调用了加号" + "的重载函数
      END IF;
    END IF;
  END PROCESS;
END ARCHITECTURE;
```

5.4.3 过程

VHDL 中，子程序的另外一种形式是过程(PROCEDURE)，过程的语句格式是：

```
PROCEDURE 过程名(参数表);                    -- 过程首
PROCEDURE 过程名(参数表)IS                   -- 过程体开始
[说明部分];
BEGIN
顺序语句;
END PROCEDURE 过程名;                        -- 过程体结束
```

与函数一样，过程由过程首和过程体两部分组成，过程首不是必需的，过程体可以独立存在和使用。

1. 过程首

过程首由过程名和参数表组成。参数表用于对常数、变量和信号三类数据对象目标做出说明，并用关键词 IN、OUT 和 INOUT 定义这些参数的工作模式，即信息的流向。如果没有指定模式，则默认为 IN。以下是三个过程首的定义示例。

```
PROCEDURE pro1(VARIABLE  a,b: INOUT  REAL);
PROCEDURE pro2(CONSTANT  a1: IN  INTEGER;
               VARIABLE  b1: OUT INTEGER);
PROCEDURE pro3(SIGNAL  sig : INOUT  BIT);
```

过程 pro1 定义了两个实数双向变量 a 和 b；过程 pro2 定义了两个参量。第一个是常数，它的数据类型为整数，信号模式是 IN，第二个参量是变量，信号模式和数据类型分别是 OUT 和整数；过程 pro3 中只定义了一个信号参量，即 sig，它的信号模式是双向 INOUT，数据类型是 BIT。一般情况下，可在参数表中定义三种信号模式，即 IN、OUT 和 INOUT。如果只定义了 IN 模式而未定义目标参数类型，则默认为常数量；若只定义了 INOUT 或 OUT，则默认目标参数类型是变量。

2. 过程体

过程体是由顺序语句组成的，过程的调用即启动了对过程体的顺序语句的执行，过程体中的说明部分只是局部的，其中的各种定义只能适用于过程体内部，过程体的顺序语句部分可以包含任何顺序执行的语句，包括 WAIT 语句，但如果一个过程是在进程中调用的，且这个进程已列出了敏感参量表，则不能在此过程中使用 WAIT 语句。

根据调用环境的不同，过程调用有两种方式，即顺序语句方式和并行语句方式。在一般的顺序语句自然执行过程中，一个过程被执行，则属于顺序语句方式，当某个过程处于并行语句环境中时其过程体中定义的任一 IN 或 INOUT 的目标参量发生改变时，将启动过程的调用，这时的调用是属于并行语句方式的，过程与函数一样可以重复调用或嵌套式调用，综合器一般不支持含有 WAIT 语句的过程，例 5-36 和例 5-37 是两个过程体的使用示例。

【例 5-36】 过程体使用示例 1

```
PROCEDURE shift (din,s: IN STD_LOGIC_VECTOR;
          SIGNAL dout: OUT STD_LOGIC_VECTOR) IS
VARIABLE sc: INTEGER;
BEGIN
```

```
  sc: = conv_integer(s);                    -- 确定左移的位数
  FOR i IN din'range LOOP
    IF(sc + i < = din'left) THEN            -- 完成循环左移
      dout(sc + i)< = din(i);
    ELSE
      dout(sc + i - din'left)< = din(i);
    END IF;
  END LOOP
END shift;
```

此过程将根据输入 S 值完成循环左移的功能。

【例 5-37】 过程体使用示例 2

```
PROCEDURE comp(a,r: IN REAL;
               m: IN INTEGER;
               v1,v2: OUT REAL)IS
VARIABLE cnt: INTEGER;
BEGIN
v1: = 1.6 * a;                              -- 赋初始值
v2: = 1.0;
q1: FOR cnt IN 1 TO m LOOP
    v2: = v2 * v1;
    EXIT q1 WHEN v2 > v1;                   -- 如果 v2 > v1,则跳出循环 LOOP
 END LOOP q1;
ASSERT (v2 < v1);
    REPORT "OUT OF RANGE"                   -- 输出错误报告
      SEVERITY ERROR;
 END PROCEDURE comp;
```

在以上过程 comp 的参量表中,定义 a 和 r 为输入模式,数据类型为实数; m 为输入模式,数据类型为整数。这三个参量都没有显式定义它们的目标参量类型,显然它们的默认类型都是常数。由于 v1、v2 定义为输入模式的实数,因此默认类型是变量。在过程 comp 的 LOOP 语句中,对 v2 进行循环计算到 v2 大于 r,EXIT 语句中断运算,并由 REPORT 语句给出错误报告。

5.4.4 重载过程

两个或两个以上有相同的过程名和互不相同的参数量及数据类型的过程为重载过程,对于重载过程,也是靠参数类型来辨别究竟调用哪一个过程。例如:

```
PROCEDURE calcu (v1,v2: IN REAL;
                 SIGNAL out1: INOUT INTEGER);
PROCEDURE calcu (v1,v2: IN INTEGER;
                 SIGNAL out1: INOUT REAL);
…
calcu(20.15,1.42,sign1);                -- 调用第一个重载过程
calcu(23,320,sign2);                    -- 调用第二个重载过程
…
```

此例中定义了两个重载过程,它们的过程名、参量数目及各参量的模式是相同的,但参

量的数据类型是不同的。第一个过程中定义的两个输入参量 v1 和 v2 为实数型常数，out1 为 INOUT 模式的整数信号；而第二个过程中 v1、v2 则为整数常数，out1 为实数信号。

如前所述，在过程结构中的语句是顺序执行的，调用者在调用过程前应将初始值传递给过程的输入参数，一旦调用，即启动过程语句，按顺序自上而下执行过程中的语句，执行结束后，将输出值返回到调用者 OUT 和 INOUT 定义的变量或信号中。

5.5 库、程序包及其配置

在 VHDL 中，除了设计实体和结构体可以独立编译外，还有另外三个可以进行独立编译的源设计单元：库、程序包和配置。其中，库主要用来存放已经编译的实体、结构体、程序包和配置；程序包主要用来存放各个设计都能共享的数据类型、子程序说明、属性说明和元件说明等部分；配置用来从库中选取所需的各个模块来完成硬件电路的描述。

5.5.1 库

在利用 VHDL 进行工程设计时，为了提高设计效率以及使设计遵循某些统一的语言标准或数据格式，有必要将一些有用的信息汇集在一个或几个库中以供调用。这些信息可以是预先定义好的数据类型、子程序等设计单元的集合体（程序包），或预先设计好的各种设计实体（元件库程序包）。因此，可以把库（LIBRARY）看成是一种用来存储预先完成的程序包和数据集合体的仓库。

正如 C 语言中头文件的说明总是放在程序的最前面一样，在 VHDL 中，库的说明总是放在设计单元的最前面。使用库的语句格式如下。

```
LIBRARY 库名;
```

这一语句即相当于为其后的设计实体打开了以此库名命名的库，以便设计实体可以利用其中的程序包，如语句“LIBRARY IEEE;”表示打开了 IEEE 库。

1. 库的种类

VHDL 程序设计中常用的库有 4 种。

1）IEEE 库

IEEE 库是 VHDL 设计中最为常见的库，它包含 IEEE 标准的程序包和其他一些支持工业标准的程序包。IEEE 库中的标准程序包主要包括 STD_LOGIC_1164、NUMERIC_BIT 和 NUMERIC_STD 等。其中的 STD_LOGIC_1164 是最重要且最常用的程序包，大部分基于数字系统设计的程序包都是以此程序包中设定的标准为基础的。

此外，还有一些程序包虽非 IEEE 标准，但由于其已成事实上的工业标准，也都并入了 IEEE 库。在这些程序包中，最常用的是 Synopsys 公司的 STD_LOGIC_ARITH、STD_LOGIC_SIGNED 和 STD_LOGIC_UNSIGNED 程序包。目前流行于我国的大多数 EDA 工具都支持 Synopsys 公司的程序包。一般基于大规模可编程逻辑器件的数字系统设计，IEEE 库中的 4 个程序包 STD_LOGIC_1164、STD_LOGIC_ARITH、STD_LOGIC_SIGNED 和 STD_LOGIC_UNSIGNED 已经足够使用。另外需要注意的是，在 IEEE 库中符合 IEEE 标准的程序包并非符合 VHDL 标准，如 STD_LOGIC_1164 程序包。因此在使

用 VHDL 设计实体的前面必须显式表达出来。

2）STD 库

VHDL 标准定义了两个标准程序包，即 STANDARD 和 TEXTIO 程序包，它们都被收入在 STD 库中。只要在 VHDL 应用环境中，可随时调用这两个程序包中的所有内容，即在编译和综合过程中，VHDL 的每一项设计都自动地将其包含进去了。由于 STD 库符合 VHDL 标准，在应用中不必如 IEEE 库那样显式表达出来。

3）WORK 库

WORK 库是用户进行 VHDL 设计的现行工作库，用于存放用户设计和定义的一些设计单元和程序包，因而是用户自己的仓库，用户设计项目的成品、半成品模块，以及先期已设计好的元件都放在其中。WORK 库自动满足 VHDL 标准，在实际调用中，不必显式预先说明。在计算机上利用 VHDL 进行项目设计，不允许在根目录下进行，而是必须为此设定一个目录，用于保存所有此项目的设计文件，VHDL 综合器将此目录默认为 WORK 库。但是必须注意，工作库并不是这个目录的目录名，而是一个逻辑名。

4）VITAL 库

使用 VITAL 库，可以提高 VHDL 门级时序模拟的精度，因而只在 VHDL 仿真器中使用。库中包含时序程序包 VITAL_TIMING 和 VITAL_PRIMITIVES。VITAL 程序包已经成为 IEEE 标准，在当前的 VHDL 仿真器的库中，VITAL 库中的程序包都已经并到 IEEE 库中。实际上，由于各 FPGA/CPLD 生产厂商的适配工具都能为各自的芯片生成带时序信息的 VHDL 门级网表，用 VHDL 仿真器仿真该网表可以得到精确的时序仿真结果，因此 FPGA/CPLD 设计开发过程中，一般并不需要 VITAL 库中的程序包。

5）用户定义库

用户为自身设计需要所开发的共用包集合和实体等，也可以汇集在一起定义成一个库，这就是用户定义库或称用户库。在使用时同样要首先说明库名。

2. 库的用法

在 VHDL 中，库的说明语句总是放在实体单元前面，而且库语言一般必须与 USE 语句同用。库语言关键词为 LIBRARY，指明所使用的库名。USE 语句指明库中的程序包。一旦说明了库和程序包，整个设计实体都可进入访问或调用，但其作用范围仅限于所说明的设计实体。VHDL 要求一项含有多个设计实体的更大的系统，每一个设计实体都必须有自己完整的库说明语句和 USE 语句。

USE 语句的使用将使所说明的程序包对本设计实体部分全部开放，即可视的。USE 语句的使用有以下两种常用格式。

```
USE 库名.程序包名.项目名;
USE 库名.程序包名.ALL;
```

第一个语句格式的作用是，向本设计实体开放指定库中的特定程序包内所选定的项目。第二个语句格式的作用是，向本设计实体开放指定库中的特定程序包内所有的内容。

例如：

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
```

以上三条语句表示打开 IEEE 库，再打开此库中的 STD_LOGIC_1164 程序包和 STD_LOGIC_UNSIGNED.ALL 程序包的所有内容。

又例如：

```
LIBRARY IEEE:
USE IEEE.STD_LOGIC_1164.STD_ULOGIC;
USE IEEE.STD_LOGIC_1164.RISING_EDGE;
```

此例中向当前设计实体开放了 STD_LOGIC_1164 程序包中的 RISING_EDGE 函数。但由于此函数需要用到数据类型 STD_ULOGIC，所以在上一条 USE 语句中开放了同一程序包中的这一数据类型。

5.5.2 程序包

为了使已定义的常数、数据类型、元件调用说明以及子程序能被更多的 VHDL 设计实体方便地访问和共享，可以将它们收集在一个 VHDL 程序包(PACKAGE)中。多个程序包可以并入一个 VHDL 库中，使之适用于更一般的访问和调用范围。这一点对于大系统开发、多个或多级开发人员并行工作显得尤为重要。

程序包的说明就像 C 语言中的 include 语句一样，要使用程序包中的某些说明和定义，可以用 USE 语句来进行说明。

程序包的内容主要由如下 4 种基本结构组成，因此一个程序包中至少应包含以下结构中的一种。

(1) 常数说明：主要用于预定义系统的宽度，如数据总线通道的宽度。

(2) 数据类型说明：主要用于说明在整个设计中通用的数据类型，例如通用的地址总线数据类型定义等。

(3) 元件定义：主要规定在 VHDL 设计中参与元件例化的文件接口界面。

(4) 子程序说明：用于说明在设计中任一处可调用的子程序。

程序包由两部分组成：程序包首和程序包体。程序包首为程序包定义接口，声明包中的类型、元件、函数和子程序，其方式和实体定义模块接口非常相似。程序包体规定程序包的实际功能，存放说明中的函数和子程序，其方式与结构体语句模块方式相同。一个完整的程序包中，程序包首名与程序包体名是同一个名字。

1. 程序包首

程序包首的说明部分可收集多个不同的 VHDL 设计所需的公共信息，其中包括数据类型说明、信号说明、子程序说明及元件说明等。

定义程序包首的一般语句结构如下。

```
PACKAGE  程序包名  IS                    -- 程序包首
    程序包首说明部分
END  程序包名;
```

程序包结构中，程序包体并非总是必需的，程序包首可以独立定义和使用。例 5-38 就是一个程序包首定义的示例。

【例 5-38】 程序包首定义示例

```
PACKAGE pacl IS
  TYPE byte IS RANGE 0 TO 255;
  SUBTYPE nibble IS byte RANGE 0 TO 15;
CONSTANT byte_ff : byte:  = 255;
SIGNAL addend: nibble;
COMPONENT byte_adder
PORT(a,b: IN byte;
C: OUT byte;
Overflow: OUT BOOLEAN);
END COMPONENT;
FUNCTION my_function(a: IN byte) RETURN byte;
END pacl;
```

例 5-38 中,其程序包名是 pacl,在其中定义了一个新的数据类型 byte 和一个子类型 nibble;接着定义了一个数据类型为 byte 的常数 byte_ff 和一个数据类型为 nibble 的信号 addend;还定义了一个元件和函数。由于元件和函数必须有具体的内容,所以将这些内容安排在程序包体中。如果要使用这个程序包中的所有定义,可利用 USE 语句按如下方式获得访问此程序包的方法。

```
LIBRARY WORK;
USE WORK.pacl.ALL;
ENTITY...
ARCHITECTURE...
...
```

由于 WORK 库是默认打开的,所以可省去 LIBRARY WORK 语句,只要加入相应的 USE 语句即可。

2. 程序包体

程序包体用于定义在程序包首中已定义的子程序的子程序体。程序包体说明部分的组成可以是 USE 语句(允许对其他程序包的调用)、子程序定义、子程序体、数据类型说明、子类型说明和常数说明等。对于没有子程序说明的程序包体可以省去。

定义程序包体的一般语句结构如下。

```
PACKAGE BODY  程序包名  IS              -- 程序包体
    程序包体说明部分以及包体内容
END 程序包名;
```

如上例所示,如果仅仅是定义数据类型或定义数据对象等内容,程序包体是不必要的,程序包首可以独立使用;但在程序包中若有子程序说明时,则必须有对应的子程序包体。这时,子程序体必须放在程序包体中。程序包常用来封装属于多个设计单元分享的信息。常用的预定义的程序包有以下 4 种。

1) STD_LOGIC_1164 程序包

它是 IEEE 库中最常用的程序包,是 IEEE 的标准程序包。其中包含一些数据类型、子类型和函数的定义,这些定义将 VHDL 扩展为一个能描述多值逻辑(即除具有“0”和“1”以外还有其他的逻辑量,如高阻态“Z”、不定态“X”等)的硬件描述语言,很好地满足了实际数

字系统的设计需求。该程序包中用得最多和最广的是定义了满足工业标准的两个数据类型STD_LOGIC 和 STD_LOGIC_VECTOR，它们非常适合于 FPGA/CPLD 器件中逻辑设计结构。

2）STD_LOGIC_ARITH 程序包

STD_LOGIC_ARITH 预先编译在 IEEE 库中，此程序包在 STD_LOGIC_1164 程序包的基础上扩展了三个数据类型 UNSIGNED、SIGNED 和 SMALL_INT，并为其定义了相关的算术运算符和转换函数。

3）STD_LOGIC_UNSIGNED 和 STD_LOGIC_SIGNED 程序包

STD_LOGIC_UNSIGNED 和 STD_LOGIC_SIGNED 程序包都是 SynoPSyS 公司的程序包，都预先编译在 IEEE 库中。这些程序包重载了可用于 INTEGER 型及 STD_LOGIC 和 STD_LOGIC_VECTOR 型混合运算的运算符，并定义了一个由 STD_LOGIC_VECTOR 型到 INTEGER 型的转换函数。这两个程序包的区别是，STD_LOGIC_SIGNED 中定义的运算符考虑到了符号，是有符号数的运算，而 STD_LOGIC_UNSIGNED 则正好相反。

4）STANDARD 和 TEXTIO 程序包

这两个程序包是 STD 库中的预编译程序包。STANDARD 程序包中定义了许多基本的数据类型、子类型和函数。TEXTIO 程序包定义了支持文件操作的许多类型和子程序。在使用本程序包之前，需加语句 USE STD. TEXTIO. ALL。TEXTIO 程序包主要供仿真器使用。

5.5.3 配置

配置(CONFIGURATION)语句描述层与层之间的连接关系以及实体与结构体之间的连接关系。可以利用配置语句来选择不同的结构体，使其与要设计的实体相对应。配置也是 VHDL 设计实体中的一个基本单元，在综合或仿真中，可以利用配置语句为确定整个设计提供许多有用信息。例如，对以元件例化的层次方式构成的 VHDL 设计实体，就可把配置语句的设置看成是一个元件表，以配置语句指定在顶层设计中的每一元件与一特定结构体相衔接，或赋予特定属性。配置语句还能用于对元件的端口连接进行重新安排等。VHDL 综合器允许将配置规定为一个设计实体中的最高层设计单元，但只支持对最顶层的实体进行配置。

配置语句的一般格式如下。

```
CONFIGURATION 配置名 OF 实体名 IS
  配置说明
END 配置名;
```

配置主要为顶层设计实体指定结构体，或为参与例化的元件实体指定所希望的结构体，以层次方式来对元件例化做结构配置。每个实体可以拥有多个不同的结构体，而每个结构体的地位是相同的，在这种情况下，可以利用配置说明为这个实体指定一个结构体。该配置的书写格式如下。

```
CONFIGURATION 配置名 OF 实体名 IS
  FOR 选配结构体名
  END FOR;
```

```
END 配置名;
```

其中,配置名是该默认配置语句的唯一标志,实体名就是要配置的实体的名称,选配结构体名就是用来组成设计实体的结构体名。

例 5-39 是一个配置的简单方式应用,即在一个描述与非门 nand1 的设计实体中有两个以不同的逻辑描述方式构成的结构体,用配置语句来为特定的结构体需求做配置指定。

【例 5-39】 配置的简单应用

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY nand1 IS
PORT(a: IN STD_LOGIC;
     b: IN STD_LOGIC;
     c: OUT STD_LOGIC);
END ENTITY nand1;
ARCHITECTURE one OF nand1 IS
  BEGIN
    c <= NOT(a AND b);
  END ARCHITECTURE one;
ARCHITECTURE two OF nand1 IS
  BEGIN
    c <= '1' WHEN (a = '0')AND (b = '0') ELSE
       '1' WHEN (a = '0')AND (b = '1') ELSE
       '1' WHEN (a = '1')AND (b = '0') ELSE
       '0' WHEN (a = '1')AND (b = '1') ELSE
       '0';
  END ARCHITECTURE two;
CONFIGURATION second OF nand1 IS
  FOR two
  END FOR;
END second;
CONFIGURATION first OF nand1 IS
  FOR one
  END FOR;
END FIRST;
```

在例 5-39 中,若指定配置名为 second,则为实体 nand1 配置的结构体为 two;若指定配置名为 first,则为实体 nand1 配置的结构体为 one。这两种结构体的描述方式是不同的,但具有相同的逻辑功能。

当一个设计的结构体中包含另外的元件时,配置语句应该包含更多的配置信息,此时采用元件配置语句来进行结构体中引用元件的配置。例 5-40 就是利用例 5-39 的文件实现 RS 触发器的实例。最后利用配置语句指定元件实体 nand1 中的第二个结构体 two 来构成 nand1 的结构体。

【例 5-40】 结构体内含有元件的配置

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY rs1 IS
```

```
  PORT(r,s: IN STD_LOGIC;
       q,qf: BUFFER STD_LOGIC);
END rs1;
ARCHITECTURE rsf OF rs1 IS
  COMPONENT nand1
    PORT(a,b: IN STD_LOGIC;
         c: OUT STD_LOGIC);
  END COMPONENT;
  BEGIN
  u1: nand1 PORT MAP (a => s,b => qf,c => q);
  u2: nand1 PORT MAP (a => q,b => r,c => qf);
 END rsf;
CONFIGURATION sel OF rs1 IS
  FOR rsf
    FOR u1,u2: nand1 USE  CONFIGURATION  WORK.first;
    END FOR;
  END FOR;
END sel;
```

在例 5-40 中，假设与非门 nand1 的设计实体已进入工作库 WORK 中，结构体首先对要引用的元件进行说明，然后使用元件例化语句来描述 RS 触发器的功能。正如该实例中用到的配置方式一样，通常在元件配置中采用如下方式。

```
CONFIGURATION 配置名 OF 实体名 IS
  FOR 选配结构体名
    FOR 元件例化标号: 元件名 USE CONFIGURATION 库名.元件配置名;
    END FOR;
    FOR 元件例化标号: 元件名 USE CONFIGURATION 库名.元件配置名;
    END FOR;
    …
  END FOR;
END  配置名;
```

在例 5-40 的元件配置部分中，该配置是对设计实体 rs1 进行配置，该配置的名称为 sel。在配置中采用结构体 rsf 作为最顶层设计实体 rs1 的结构体。结构体 rsf 中例化的两个元件 u1 和 u2，实体是 nand1，并指定元件所用的配置是 first，它来源于 WORK 库。这样就为所有的实体指定了结构体：实体 rs1 的结构体为 rsf；元件 nand1 的实体为 nand1，结构体为低层配置 first 指定的结构体。

5.6 VHDL 描述风格

VHDL 的结构体用于具体描述整个设计实体的逻辑功能，对于所希望的电路功能行为，可以在结构体中用不同的语句类型和描述方法来表达，对于相同的逻辑行为，可以有不同的语句表达方式。在 VHDL 中，这些描述方法称为描述风格，通常可归纳为三种：行为描述、数据流描述和结构描述。这三种描述方式从不同的角度对硬件系统进行行为和功能的描述。

5.6.1 行为描述

行为描述只表示输入与输出间转换的行为，它不包含任何结构信息。行为描述主要使用函数、过程和进程语句，以算法形式描述数据的变换和传送。行为描述方式的优点在于只需要描述清楚输入与输出的行为，而不需要花费更多的精力关注设计功能的门级实现。VHDL的行为描述能力使自顶向下的设计方式成为可能。例如，对于如图5-2所示的1位全加器，其行为描述如例5-41所示。

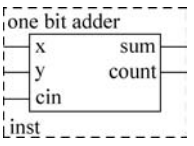


图5-2 1位全加器电路图

【例5-41】 1位全加器行为描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY onebitadder IS
  PORT(x,y,cin: IN STD_LOGIC;
       sum,count: OUT STD_LOGIC);
END onebitadder;
ARCHITECTURE behave OF onebitadder IS
BEGIN
  PROCESS(x,y,cin)
    VARIABLE n: INTEGER;
    BEGIN
      n: =0;
      IF(x='1') THEN
        n: =n+1;
      END IF;
      IF(y='1') THEN
        n: =n+1;
      END IF;
      IF(cin='1') THEN
        n: =n+1;
      END IF;
      IF(n=0) THEN
        sum<='0'; count<='0';
      ELSIF (n=1) THEN
        sum<='1'; count<='0';
      ELSIF (n=2) THEN
        sum<='0'; count<='1';
      ELSE
        sum<='1'; count<='1';
      END IF;
  END PROCESS;
END behave;
```

在例5-41中，端口cin是低位进位的输入端口，count是向高位进位的输出端口。在源代码的描述中，采用的是对全加器的数学模型的描述，没有涉及任何有关电路的组成结构和门级电路。在应用VHDL进行系统执行时，行为描述方式是最重要的逻辑描述方式，是VHDL编程的核心，可以说，没有行为描述就没有VHDL。

5.6.2 数据流描述

数据流描述方式，也称作 RTL 描述方式，主要使用并行的信号赋值语句，既显式地表示了该设计单元的行为，又隐含该设计单元的结构。对于上述的 1 位全加器，其数据流描述如例 5-42 所示。

【例 5-42】 1 位全加器数据流描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY onebitadder1 IS
  PORT(x,y,cin: IN BIT;
       sum,count: OUT BIT);
END onebitadder1;
ARCHITECTURE dataflow OF onebitadder1 IS
BEGIN
    sum <= x XOR y XOR cin;
    count <= (x AND y)OR (x AND cin) OR (y AND cin);
 END dataflow;
```

由例 5-42 可以看到，结构体的数据流描述方式就是按照全加器的逻辑表达式来进行描述的，这要求设计人员对全加器的电路实现要有清楚的认识。实例中是采用并行赋值语句来进行功能描述的，并行赋值语句是并行执行的，与源代码书写顺序无关。

5.6.3 结构描述

结构描述是描述该设计单元的硬件结构，即该硬件是如何构成的。其主要使用元件例化语句及配置语句来描述元件的类型及元件的互连关系。在层次设计中，高层次的设计模块调用低层次的设计模块，或者直接用门电路设计单元来构成一个复杂的逻辑电路的描述方法。例如，仍以 1 位全加器为例，假设已经具有与门、或门和异或门等逻辑电路的设计单元，那么就可以将这些现成的设计单元经适当的连接构成新的设计电路，如例 5-43 所示。

【例 5-43】 1 位全加器结构描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY onebitadder2 IS
  PORT(x,y,cin: IN BIT;
       sum,count: OUT BIT);
END onebitadder2;
ARCHITECTURE structure OF onebitadder2 IS
  COMPONENT xor3
   PORT(a,b,c: IN BIT;
        O: OUT BIT);
  END COMPONENT;
COMPONENT and2
   PORT(a,b: IN BIT;
        O: OUT BIT);
  END COMPONENT;
COMPONENT or3
```

```
    PORT(a,b,c: IN BIT;
         O: OUT BIT);
  END COMPONENT;
SIGNAL s1,s2,s3: BIT;
BEGIN
  G1: xor3 PORT MAP (x,y,cin,sum);
  G2: and2 PORT MAP (x,y,s1);
  G3: and2 PORT MAP (x,cin,s2);
  G4: and2 PORT MAP (y,cin,s3);
  G5: or3 PORT MAP (s1,s2,s3,cout);
END STRUCTURE;
```

在这个电路中,用 component 语句指明了在本结构体中将要调用的已生成的模块电路,用 port map()语句将生成模块的端口与所设计的各模块(g1,g2,g3,g4,g5)的端口联系起来,并定义了相应的信号,以表示所设计的各模块之间的连接关系。从例 5-43 中可以看出,结构描述方式可以将已有的设计成果应用到当前的设计中,因而大大提高了设计效率。对于可分解为若干个子元件的大型设计,结构描述方式是首选。

在实际应用中,为了能兼顾整个设计的功能、资源、性能等几方面的因素,通常混合使用这三种描述方式。

5.7 常用单元的设计举例

为了能更深入地理解使用 VHDL 设计逻辑电路的具体步骤和方法,本节以常用基本逻辑电路设计为例,进一步介绍利用 VHDL 描述基本逻辑电路的方法。

5.7.1 组合逻辑电路设计

组合逻辑电路设计的应用可简单地分成以下几个方面:基本逻辑门、编码器和译码器、多路选择器和分配器、算术运算器等。

1. 基本门电路

门电路是构成所有组合电路的基本电路,常见的门电路有与门、或门、与非门、或非门、异或门及反相器等,例 5-44 用 VHDL 来描述一个二输入的与门。

【例 5-44】 二输入与门

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY andgate IS
  PORT(a,b: IN STD_LOGIC;          --a,b 为输入端口
       c: OUT STD_LOGIC);          --c 为输出端口
END andgate;
ARCHITECTURE and_2 OF andgate IS
BEGIN
 c<=a AND b;
END and_2;
```

其他几种基本逻辑门电路只要布尔方程稍做一点儿改动即可。

例 5-44 的工作时序如图 5-3 所示。从图中可以看出，输出信号 c 实现了输入信号 a 和 b 的“逻辑与”。

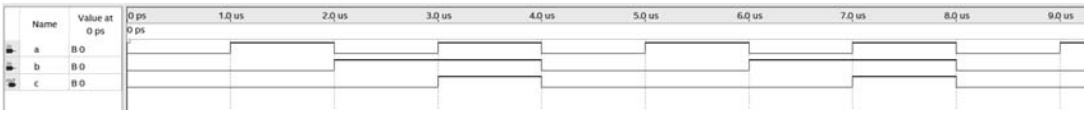


图 5-3　与门工作时序

2. 8-3 线优先编码器

在实际的逻辑电路中，编码器的功能就是把两个输入转换为 N 位编码输出。目前经常使用的编码器主要有两种：普通编码器和优先编码器。其中，普通编码器对于某一给定时刻，只能对一个输入信号进行编码，而优先编码器的输入端允许同一时刻出现两个或两个以上的信号，此时，编码器已经将所有的输入信号按优先顺序排了队，当几个输入信号同时出现时，只对其中优先级最高的一个输入信号进行编码。例 5-45 用三种方法设计 8-3 线优先编码器。其输入信号为 a、b、c、d、e、f、g 和 h，输出信号为 out0、out1 和 out2，输入信号中 a 的优先级别最低，以此类推，h 的优先级别最高。

【例 5-45】 8-3 线优先编码器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY encoder IS
  PORT(a,b,c,d,e,f,g,h: IN STD_LOGIC;
       out0,out1,out2: OUT STD_LOGIC);
END encoder;
```

方法 1　使用条件赋值语句

```
ARCHITECTURE behave1 OF encoder IS
SIGNAL outvec: STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
  outvec(2 downto 0)<="111"WHEN h='1' ELSE      -- 条件赋值语句
  "110" WHEN g='1' ELSE
  "101" WHEN f='1' ELSE
  "100" WHEN e='1' ELSE
  "011" WHEN d='1' ELSE
  "010" WHEN c='1' ELSE
  "001" WHEN b='1' ELSE
  "000" WHEN a='1' ELSE
  "000";
  out0<=outvec(0);
  out1<=outvec(1);
  out2<=outvec(2);
END behave1;
```

方法 2　使用 LOOP 语句

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
```

```
USE IEEE.STD_LOGIC_ARITH.ALL;
ENTITY encoder IS
  PORT(a,b,c,d,e,f,g,h: IN STD_LOGIC;
       out0,out1,out2: OUT STD_LOGIC);
END encoder;
ARCHITECTURE behave2 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              --LOOP 循环语句
        i: =i-1;
      END LOOP;
      (out2,out1,out0)<=CONV_STD_LOGIC_VECTOR(i,3); --将 i 转换成三位的标准信号序列,并
                                                     --赋值给输出
  END PROCESS;
END behave2;
```

方法 3　使用 IF 语句

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY encoder IS
  PORT(in1: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       out1: OUT STD_LOGIC_VECTOR(2 DOWNTO 0));
END encoder;
ARCHITECTURE behave3 OF encoder IS
BEGIN
  PROCESS(in1)
  BEGIN
    IF in1(7)='1' THEN out1<="111";                  --IF 语句
    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 behave3;
```

例 5-45 的工作时序如图 5-4 所示。从图中可以看出,输出信号 out0、out1 和 out2 实现了对输入信号 a、b、c、d、e、f、g 和 h 的优先编码,其中,a 的优先级别最低,h 的优先级别最高。

3. 七段显示译码器

七段显示译码器是对一个 4 位二进制数进行译码,并在七段显示器上显示出相应的十

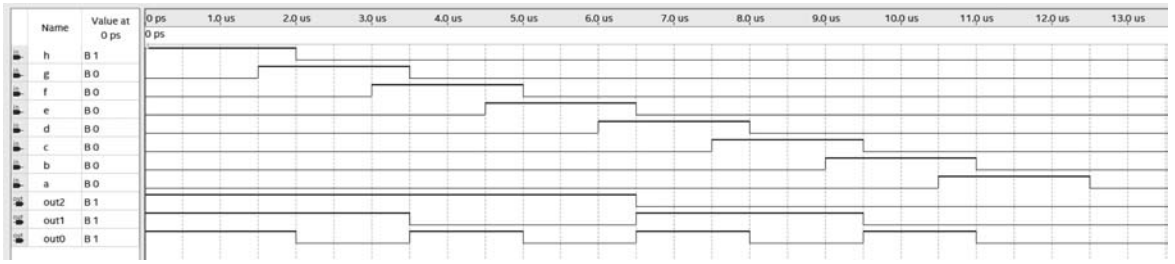


图 5-4　优先编码器工作时序

进制数。一个七段显示译码器的设计方框图如图 5-5 所示。根据图 5-5 可知，输入信号 D3、D2、D1、D0 是二进制 BCD 码的集合，可表示为[D3…D0]。输出信号 a、b、c、d、e、f、g 也是用二进制数表示，为书写代码方便起见，输出信号用 x 的集合来表示。其 VHDL 描述如例 5-46 所示。

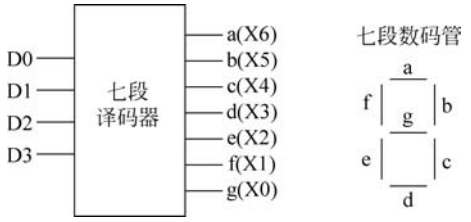


图 5-5　七段显示译码器

【例 5-46】　七段显示译码器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY decoder IS
  PORT( d: IN STD_LOGIC_VECTOR(3 DOWNTO 0);          -- 输入 4 位二进制数据
        x: OUT STD_LOGIC_VECTOR(6 DOWNTO 0));        -- 七段译码输出
END decoder;
ARCHITECTURE a OF decoder IS
BEGIN
  WITH d SELECT
    x <= "1111110"WHEN "0000",
       "0110000"WHEN "0001",
       "1101101"WHEN "0010",
       "1111001"WHEN "0011",
       "0110011"WHEN "0100",
       "1011011"WHEN "0101",
       "1011111"WHEN "0110",
       "1110000"WHEN "0111",
       "1111111"WHEN "1000",
       "1111011"WHEN "1001",
       "0000000"WHEN OTHERS;
END a;
```

例 5-46 的工作时序如图 5-6 所示。从图中可以看出，输出信号 x 为输入信号 d 的显示代码。

图 5-6　七段显示译码器工作时序

4. 多路分配器

多路分配器的作用是为输入信号选择输出，在计算机和通信设备中往往用于信号的分配。一个 1-8 多路分配器如图 5-7 所示，它有 1 根输入信号线 data，3 根选择信号线 S0、S1、S2，1 根使能信号线 enable 和 8 根输出线 y0～y7。其 VHDL 描述如例 5-47 所示。

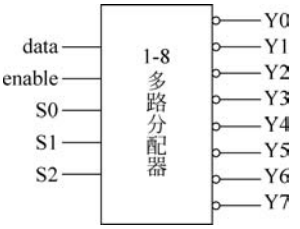


图 5-7　多路分配器

【例 5-47】　多路分配器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY dmux1to8 IS
  PORT(data,enable: IN STD_LOGIC;                    -- 分别为输入和使能端口
       s: IN STD_LOGIC_VECTOR(2 DOWNTO 0);           -- 选择信号端口
       y0,y1,y2,y3,y4,y5,y6,y7: OUT STD_LOGIC); -- 输出端口
END dmux1to8;
ARCHITECTURE a OF dmux1to8 IS
BEGIN
  PROCESS(enable,s,data)
  BEGIN
    IF enable = '0' THEN
    y0 <= '1'; y1 <= '1'; y2 <= '1'; y3 <= '1'; y4 <= '1';
    y5 <= '1'; y6 <= '1'; y7 <= '1';
    ELSIF s = "000" THEN
      y0 <= NOT(data);
    ELSIF s = "001" THEN
      y1 <= NOT(data);
     ELSIF s = "010" THEN
      y2 <= NOT(data);
    ELSIF s = "011" THEN
      y3 <= NOT(data);
    ELSIF s = "100" THEN
      y4 <= NOT(data);
    ELSIF s = "101" THEN
      y5 <= NOT(data);
    ELSIF s = "110" THEN
      y6 <= NOT(data);
    ELSIF s = "111" THEN
      y7 <= NOT(data);
    END IF;
  END PROCESS;
END a;
```

例 5-47 的工作时序如图 5-8 所示。从图中可以看出，根据不同的选择信号 s，可以把输

入信号在不同的输出端输出。

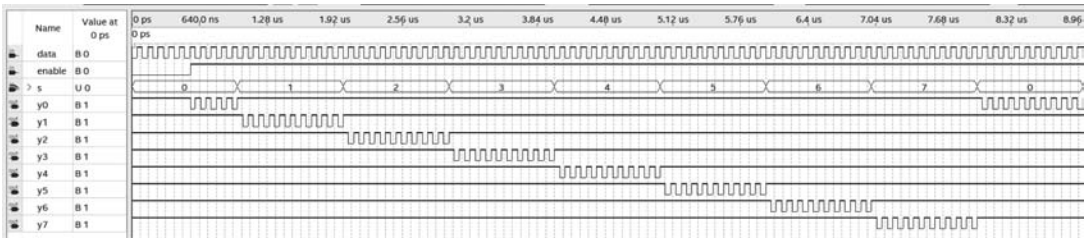


图 5-8　多路分配器工作时序

5. 多位加法运算

例 5-48 的程序实现对输入操作数 a、b 作加法运算。

【例 5-48】　多位加法运算

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY adder IS
  PORT(a,b: IN STD_LOGIC_VECTOR(7 DOWNTO 0);      -- 输入两个 8 位二进制数
       cin: IN STD_LOGIC;                          -- 低位来的进位
       s: OUT STD_LOGIC_VECTOR(8 DOWNTO 0));       -- 输出 8 位结果及产生的进位
END adder;
ARCHITECTURE behave OF adder IS
BEGIN
  s <= ('0'&a) + ('0'&b) + ("0000000"&cin);
END behave;
```

例 5-48 的工作时序如图 5-9 所示。从图中可以看出，输出信号 s 实现了输入信号 a、b 和 cin 的多位加法运算。

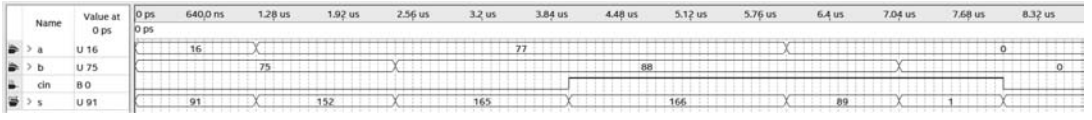


图 5-9　多位加法器工作时序

6. 三态门及总线缓冲器

三态门和总线缓冲器是驱动电路经常用到的器件。

1）三态门电路

三态门是微机应用系统经常要用到的逻辑部件，其 VHDL 描述如例 5-49 所示。

【例 5-49】　三态门电路

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tristate IS
  PORT(en,din: IN STD_LOGIC;                      -- en 为使能端口，din 为输入端口
       dout: OUT STD_LOGIC);                      -- 输出端口
END tristate;
```

```
ARCHITECTURE tri OF tristate IS
BEGIN
  PROCESS(en,din)
  BEGIN
    IF en = '1' THEN
      dout <= din;
    ELSE
      dout <= 'Z';
    END IF;
  END PROCESS;
end tri;
```

例 5-49 的工作时序如图 5-10 所示。从图中可以看出，当使能信号 en 为高电平时，输出信号为输入信号，当使能信号为低电平时，输出信号处在高阻状态。

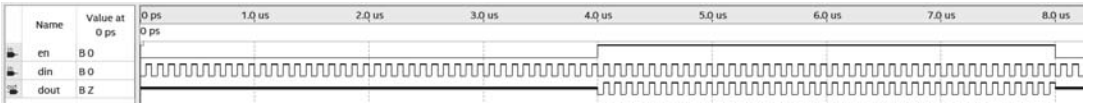


图 5-10　三态门工作时序

2）单向总线驱动器

在微型计算机的总线驱动中经常要用到单向总线缓冲器，它通常由多个三态门组成，用来驱动地址总线和控制总线。一个 8 位的单向总线缓冲器如图 5-11 所示。其对应的 VHDL 描述如例 5-50 所示。

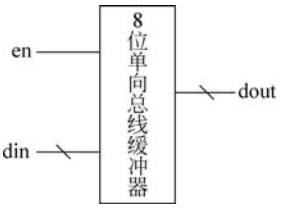


图 5-11　单向总线缓冲器

【例 5-50】　单向总线缓冲器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY trl_buf8 IS
  PORT(din: IN STD_LOGIC_VECTOR(7 DOWNTO 0);          -- 输入 8 位二进制数
       dout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);        -- 输出 8 位二进制数
       en: IN STD_LOGIC);                             -- 使能端口
END trl_buf8;
ARCHITECTURE behave OF trl_buf8 IS
BEGIN
  PROCESS(en,din)
  BEGIN
    IF(en = '1')THEN
      dout <= din;
    ELSE
      dout <= "ZZZZZZZZ";
    END IF;
  END PROCESS;
END behave;
```

例 5-50 的工作时序如图 5-12 所示。该图与图 5-10 类似，只是数据位较多。

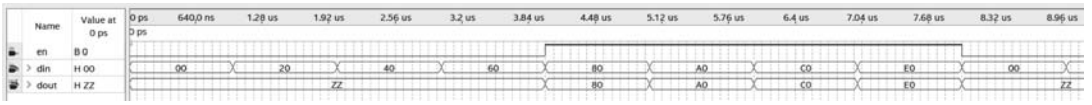


图 5-12　单向总线缓冲器工作时序

3）双向总线驱动器

双向总线缓冲器用于对数据总线的驱动和缓冲，典型的双向总线缓冲器如图 5-13 所示，图中的双向总线缓冲器有两个数据输入/输出端 a 和 b，一个方向控制端 dir 和一个选通端 en。en＝0 时双向总线缓冲器选通，若 dir＝0，则 a＝b，反之则 b＝a。其 VHDL 描述如例 5-51 所示。

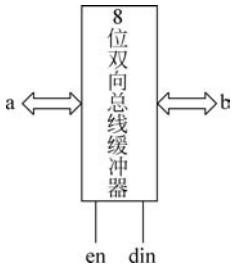


图 5-13　双向总线缓冲器

【例 5-51】　双向总线缓冲器源程序

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY bidir IS
  PORT(a,b: INOUT STD_LOGIC_VECTOR(7 DOWNTO 0); -- 双向端口
       en,dir: IN STD_LOGIC);                   -- 使能和方向端口
END bidir;
ARCHITECTURE bi OF bidir IS
  SIGNAL aout,bout: STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
  PROCESS(a,en,dir)
  BEGIN
    IF((en = '0')and(dir = '1'))THEN bout <= a;
    ELSE
      bout <= "ZZZZZZZZ";
    END IF;
      b <= bout;
   END PROCESS;
   PROCESS(b,dir,en)
   BEGIN
     IF((en = '0')and(dir = '0'))THEN aout <= b;
     ELSE
       aout <= "ZZZZZZZZ";
      END IF;
       a <= aout;
    END PROCESS;
END bi;
```

例 5-51 的工作时序如图 5-14 所示。从图中可以看出，en＝0 时双向总线缓冲器选通，若 dir＝0，则 a＝b，反之则 b＝a。

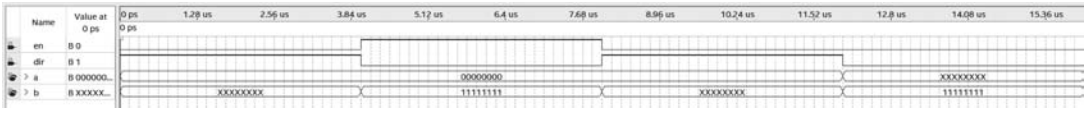


图 5-14　双向总线缓冲器工作时序

5.7.2 时序逻辑电路设计

上面介绍的组合逻辑电路中没有记忆元件，当输入信号发生变化时，输出信号随之变化。而时序电路中含有记忆元件，输出信号与时钟有关，当时钟脉冲到来之前，输出保持原来状态，只有在时钟脉冲到来之时，输出信号才发生改变；输出信号值取决于时钟有效沿来临之时激励端的输入信号值。在时序逻辑电路中，记忆元件为触发器。在 CPLD、FPGA 器件中，常用的触发器为 D 触发器，其他类型的触发器都可由 D 触发器构成。

时序电路可分为同步电路和异步电路。在同步电路中，所有触发器的时钟都接在一个时钟线上；而异步电路各触发器的时钟不接在一起。大多数可编程器件的内部结构是同步电路。

常用的时序电路有触发器、锁存器、计数器和移位寄存器等。D 触发器已在前面做了介绍，下面再介绍一些常见的时序电路。

1. JK 触发器

一个基本的 JK 触发器有两个数据输入端 j 和 k，一个时钟输入端 clk 和两个反相输出端 q 和 nq。JK 触发器的 VHDL 描述如例 5-52 所示。

【例 5-52】 JK 触发器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY jkff1 IS
  PORT(clk,j,k: IN STD_LOGIC;
       q,nq: OUT STD_LOGIC);
END jkff1;
ARCHITECTURE behave OF jkff1 IS
  SIGNAL q_s,nq_s: STD_LOGIC;
  BEGIN
    PROCESS(clk,j,k)
      BEGIN
        IF(clk'EVENT AND clk = '1')THEN
           IF(j = '0')AND(k = '1')THEN          -- j = '0'和 k = '1'状态
              q_s <= '0';
              nq_s <= '1';
           ELSIF(j = '1')AND(k = '0')THEN       -- j = '1'和 k = '0'状态
              q_s <= '1';
              nq_s <= '0';
            ELSIF(j = '1')AND(k = '1')THEN      -- j = '1'和 k = '1'状态
              q_s <= NOT q_s;
              nq_s <= NOT nq_s;
            END IF;
           END IF;
         q <= q_s;
         nq <= nq_s;
     END PROCESS;
END behave;
```

例 5-52 的工作时序如图 5-15 所示。从图中可以看出，当 j＝0、k＝0 时，JK 触发器状态

保持不变；当 j、k 不相同时，触发器状态同 j 状态；当 j＝k＝1 时，每来一个时钟脉冲，触发器状态翻转一次。

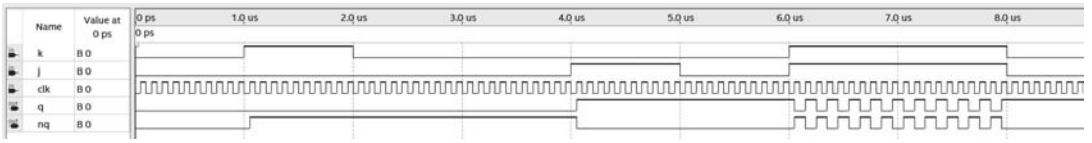


图 5-15　JK 触发器工作时序

2. 8 位锁存器

锁存器仍然是一种触发器，在控制信号有效的时候，输出信号随着数据输入的变化而变化。8 位锁存器的 VHDL 描述如例 5-53 所示。

【例 5-53】 8 位锁存器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY reg_8 IS
  PORT(d: IN STD_LOGIC_VECTOR(0 TO 7);
       clk: IN STD_LOGIC;
       q: OUT STD_LOGIC_VECTOR(0 TO 7));
END reg_8;
ARCHITECTURE behave OF reg_8 IS
BEGIN
  PROCESS(clk)
  BEGIN
    IF(clk'EVENT AND clk = '1')THEN        -- 时钟到来
    q <= d;                                -- 信号锁存
    END IF;
  END PROCESS;
END behave;
```

例 5-53 的工作时序如图 5-16 所示。从图中可以看出，每来一个时钟信号后，就能够把输入信号值锁存住。

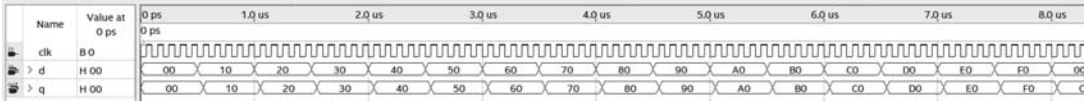


图 5-16　8 位锁存器工作时序

3. 异步复位和置位、同步预置的 4 位计数器

计数器是数字设备中的基本逻辑单元，它不仅可以用在对时钟脉冲的计数上，还可以用于分频、定时产生脉冲序列以及进行数字运算等。具有异步复位和置位、同步预置功能的 4 位计数器的 VHDL 描述如例 5-54 所示。

【例 5-54】 4 位计数器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
```

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY cnt41 IS
  PORT(pst,clk,enable,rst,load: IN STD_LOGIC;     -- pst 为置位信号,clk 为时钟信号,enable 为
                                                  -- 使能信号,rst 为复位信号,load 为预置信号
       date: IN STD_LOGIC_VECTOR(3 DOWNTO 0);     -- 预置输入数据
       cnt: BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));-- 计数器输出
END cnt41;
ARCHITECTURE behave OF cnt41 IS
  BEGIN
   count: PROCESS(rst,clk,pst)
   BEGIN
     IF rst = '1' THEN                            -- 异步复位
       cnt <= (OTHERS => '0');
     ELSIF pst = '1' THEN                         -- 异步置位
       cnt <= (others => '1');
     ELSIF(clk'EVENT AND clk = '1')THEN
       IF load = '1' THEN                         -- 同步预置
          cnt <= date;
       ELSIF enable = '1' THEN
          cnt <= cnt + 1;                         -- 计数
       END IF;
     END IF;
  END PROCESS count;
END behave;
```

上述的 4 位计数器也可用整数形式实现,描述如下。

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY cnt41 IS
  PORT(pst,clk,enable,rst,load: IN BIT;
       date: IN INTEGER RANGE 0 to 15;
       q: OUT INTEGER RANGE 0 TO 15);
END cnt41;
ARCHITECTURE behave OF cnt41 IS
  BEGIN
   PROCESS(rst,clk,pst)
     VARIABLE cnt: INTEGER RANGE 0 TO 15;
   BEGIN
     IF rst = '1' THEN
       cnt: = 0;
     ELSIF pst = '1' THEN
       IF load = '1' THEN
          cnt: = date;
       ELSIF enable = '1' THEN
            cnt: = cnt + 1;
       END IF;
     END IF;
     q <= cnt;
  END PROCESS ;
END behave;
```

例 5-54 的工作时序如图 5-17 所示。从图中可以看出，当 rst＝1 时，输出信号清零，当 pst＝1 时，输出信号置 1，当 load＝1 时，实现预置功能，预置完毕，即可实现计数功能。

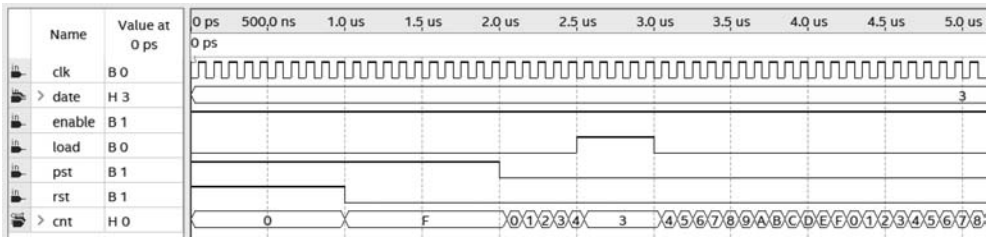


图 5-17　异步复位和置位、同步预置的 4 位计数器工作时序

4. 同步计数器

同步计数器是指在时钟脉冲的控制下，构成计数器的各触发器的状态同时发生变化的一类计数器。例 5-55 是一个模为 60，具有异步复位、同步置数功能的 8421BCD 码计数器。

【例 5-55】　同步计数器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY cntm60 IS
  PORT(ci: IN STD_LOGIC;
       nreset: IN STD_LOGIC;
       load: IN STD_LOGIC;
       d: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
       clk: IN STD_LOGIC;
       co: OUT STD_LOGIC;
       qh: BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0);
       ql: BUFFER STD_LOGIC_VECTOR(3 DOWNTO 0));
END cntm60;
ARCHITECTURE behave OF cntm60 IS
BEGIN
  co<='1' WHEN (qh="0101" AND ql="1001" AND ci='1')ELSE '0';
  PROCESS(clk,nreset)
  BEGIN
   IF(nreset='0') THEN                          --异步清零
     qh<="0000";
     ql<="0000";
   ELSIF(clk'EVENT and clk='1') THEN
       IF(load='1')THEN                         --同步预置
            qh<=d(7 downto 4);
            ql<=d(3 downto 0);
        ELSIF(ci='1')THEN                       --模 60 的实现
           IF(ql=9)THEN
              ql<="0000";                       --低 4 位清零
                IF(qh=5) THEN
                  qh<="0000";                   --高 4 位清零
                ELSE                            --计数功能的实现
                   qh<=qh+1;
```

```
            END IF;
          ELSE
            ql <= ql + 1;                    -- 低 4 位加 1
          END IF;
        END IF;
      END IF;
    END PROCESS;
END behave;
```

例 5-55 的工作时序如图 5-18 所示。从图中可以看出，当 load＝1 时，把输入信号 56 预置到输出端，然后开始计数，计到 60 时，重新从 0 开始计数。

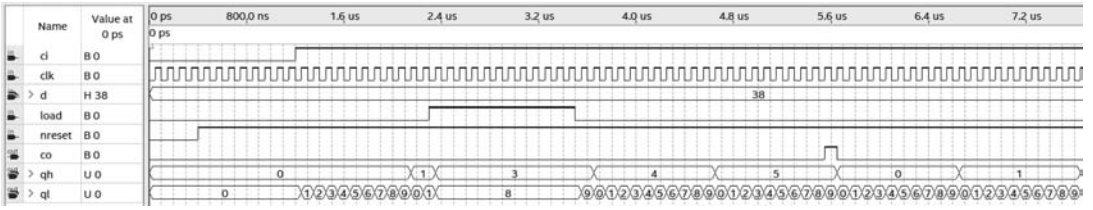


图 5-18　模为 60 的同步计数器工作时序

5. 序列信号发生器

在数字信号的传输和数字系统的测试中，有时需要用到一组特定的串行数字信号，产生序列信号的电路称为序列信号发生器。例 5-56 就是一“01111110”的序列发生器的 VHDL 描述，该电路可由计数器与数据选择器构成。

【例 5-56】 序列信号发生器

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY senqgen IS
  PORT(clk,clr,clock: IN STD_LOGIC;
       zo: OUT STD_LOGIC);
END senqgen;
ARCHITECTURE behave OF senqgen IS
  SIGNAL count: STD_LOGIC_VECTOR(2 DOWNTO 0);
  SIGNAL z: STD_LOGIC: = '0';
  BEGIN
    PROCESS(clk,clr)
      BEGIN
        IF (clr = '1') THEN count <= "000";
        ELSE
          IF(clk = '1' AND clk'EVENT) THEN
            IF(count = "111")THEN count <= "000";
            ELSE count <= count + '1';
            END IF;
          END IF;
        END IF;
```

```
    END PROCESS;
  PROCESS(count)
    BEGIN
      CASE count IS
        WHEN "000" = > z < = '0';
        WHEN "001" = > z < = '1';
        WHEN "010" = > z < = '1';
        WHEN "011" = > z < = '1';
        WHEN "100" = > z < = '1';
        WHEN "101" = > z < = '1';
        WHEN "110" = > z < = '1';
        WHEN "111" = > z < = '0';
        WHEN OTHERS = > z < = '0';
      END CASE;
  END PROCESS;
  PROCESS(clock,z)
     BEGIN
       IF(clock'EVENT AND clock = '1')THEN
       zo < = z;
       END IF;
   END PROCESS;
END behave;
```

例 5-56 的工作时序如图 5-19 所示。从图中可以看出，输出信号实现了“01111110”的序列代码。

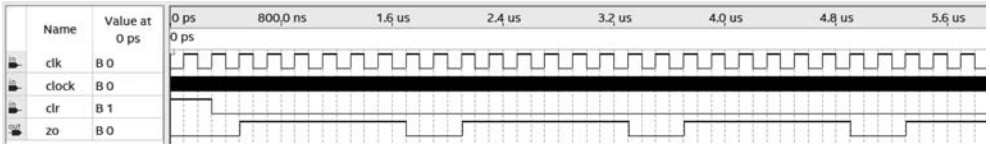


图 5-19　序列信号发生器工作时序

5.8　VHDL 与原理图混合设计方式

前面分别介绍了原理图设计方式和 VHDL 设计方式，实际上，很多较为复杂的设计采用的是两者的结合，即采用 VHDL 与原理图混合方式来进行设计。一般情况下，使用 VHDL 描述底层模块，再应用原理图设计方法设计顶层原理图文件。下面就一个 4 位二进制计数译码显示器的设计来进行说明。

5.8.1　4 位二进制计数器的 VHDL 设计

例 5-57 是 4 位二进制计数器的 VHDL 源程序，该文件名取为 cnt4。

【例 5-57】　4 位二进制计数器 VHDL 描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
```

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY cnt4 IS
PORT(pst,clk,rst,enable,load: IN STD_LOGIC;
     date: IN STD_LOGIC_VECTOR( 3 DOWNTO 0);
     cnt: BUFFER std_logic_vector (3 DOWNTO 0));
END cnt4;
ARCHITECTURE a OF cnt4 IS
  BEGIN
  count: PROCESS(rst,clk,pst)
   BEGIN
    IF rst = '1' THEN          -- 复位
     cnt <= (others =>'0');
    ELSIF pst = '1' THEN       -- 置位
     cnt <= (others =>'1');
    ELSIF (clk'event and clk = '1') THEN
     IF load = '1' THEN
      cnt <= date;             -- 预置
     ELSIF enable = '1' THEN
      cnt <= cnt + 1;          -- 计数
     END IF;
    END IF;
  END PROCESS count;
 END a;
```

文件存盘后，为了能在图形编辑器中调用该计数器，需要为该计数器创建一个元件图形符号。选择 File→Create/Update→Create Symbol Files for Current File 菜单，生成 cnt4 的图形符号。如果源程序有错，要对源程序进行修改，重复上面的步骤，直到此元件符号创建成功。

5.8.2 7段显示译码器的VHDL设计

decl7s.vhd 完成 7 段显示译码器的功能，用来将 4 位二进制数译码为驱动 7 段数码管的显示信号。decl7s.vhd 及其元件符号的创建过程同上，文件放在同一目录 D:\mylx\GUIDE 内，其源程序如例 5-58 所示。

【例 5-58】 7段显示译码器VHDL描述

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
 ENTITY decl7s IS
   PORT ( a     : IN  STD_LOGIC_VECTOR(3 DOWNTO 0) ;
          led7s : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)  ) ;
END decl7s;
ARCHITECTURE one OF decl7s IS
BEGIN
   PROCESS( a )
   BEGIN
       CASE  a(3 DOWNTO 0)  IS
         WHEN "0000" =>  LED7S <= "0111111" ; -- X"3F" 0
         WHEN "0001" =>  LED7S <= "0000110" ; -- X"06" 1
```

```
            WHEN "0010" =>  LED7S <= "1011011" ; --X"5B" 2
            WHEN "0011" =>  LED7S <= "1001111" ; --X"4F" 3
            WHEN "0100" =>  LED7S <= "1100110" ; --X"66" 4
            WHEN "0101" =>  LED7S <= "1101101" ; --X"6D" 5
            WHEN "0110" =>  LED7S <= "1111101" ; --X"7D" 6
            WHEN "0111" =>  LED7S <= "0000111" ; --X"07" 7
            WHEN "1000" =>  LED7S <= "1111111" ; --X"7F" 8
            WHEN "1001" =>  LED7S <= "1101111" ; --X"6F" 9
            WHEN "1010" =>  LED7S <= "1110111" ; --X"77" 10
            WHEN "1011" =>  LED7S <= "1111100" ; --X"7C" 11
            WHEN "1100" =>  LED7S <= "0111001" ; --X"39" 12
            WHEN "1101" =>  LED7S <= "1011110" ; --X"5E" 13
            WHEN "1110" =>  LED7S <= "1111001" ; --X"79" 14
            WHEN "1111" =>  LED7S <= "1110001" ; --X"71" 15
            WHEN OTHERS =>  NULL ;
        END CASE ;
    END PROCESS ;
END one;
```

5.8.3 顶层文件原理图设计

TOP.GDF 是本项示例的最顶层的图形设计文件，调用了前面 1、2 段创建的两个功能元件，将 Cnt4.vhd 和 Decl7s.vhd 两个模块组装起来，成为一个完整的设计。

在主菜单上选择 File→New，或单击工具栏上的 图标，或利用快捷键 Ctrl+N，在弹出的 New 菜单中选择 Block Diagram/Schemetic File 后单击 OK 按钮，绘出如图 5-20 所示的原理图。将此顶层原理图文件取名为 TOP.GDF，或其他名字，并写入 File Name 中，存入同一目录中。

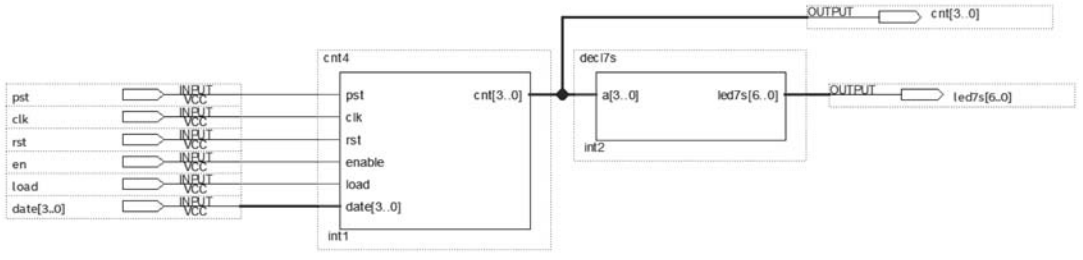


图 5-20 顶层设计原理图

最后开始编译和综合。如前述，在 Quartus Prime 18 菜单中选择 Processing→Start Compilation 项或单击工具栏上的 按钮，在此可运行编译器（此编译器将一次性完成编译、综合、优化、逻辑分割和适配/布线等操作）。如果源程序有错误，用鼠标双击红色的错误信息即可返回图形或文本编辑器进行修改，然后再次编译，直到通过。通过后双击 Fitter 下的 rpt 标记，即可进入适配报告，以便了解适配情况，然后了解引脚的确定情况是否与以上设置一致。关闭编辑器。

编译完成后，还可以进行仿真顶层设计文件，方法与前述有关内容相同。得到的仿真波形图如图 5-21 所示。

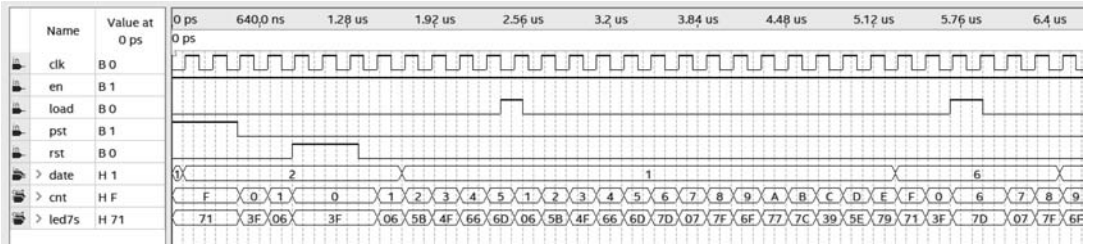


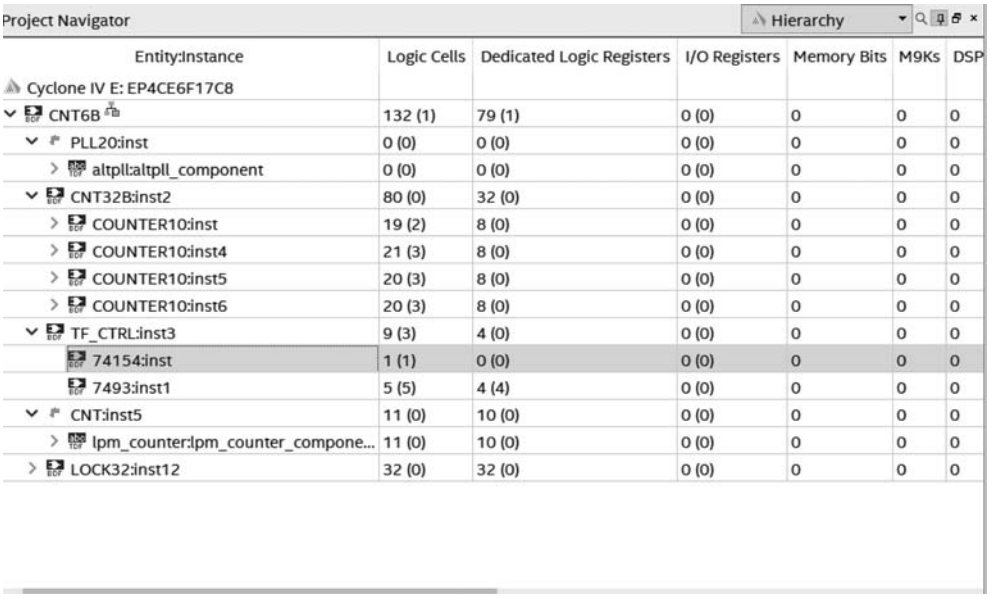
图 5-21　4 位二进制计数器仿真波形图

5.8.4　查看工程的层次结构

Quartus Prime 18 的层次显示工具可以显示当前工程的层次结构，使设计者对工程的组成模块及其之间的关系一目了然，并可方便地穿越层次，根据工程内不同的设计文件自动打开相应的编辑器。

1. 打开层次显示窗口

在菜单栏中选择 View →Utility Windows →Project Navigator 或者按 Alt+0 组合键，选择 Hierarchy 打开工程层次显示窗口，如图 5-22 所示。当前工程的层次树中的每个文件都显示在层次显示窗口内，右侧显示每个模块占用逻辑单元(Logic Cells)个数等信息。



Entity:Instance	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M9Ks	DSP
Cyclone IV E: EP4CE6F17C8						
CNT6B	132 (1)	79 (1)	0 (0)	0	0	0
PLL20:inst	0 (0)	0 (0)	0 (0)	0	0	0
altpll:altpll_component	0 (0)	0 (0)	0 (0)	0	0	0
CNT32B:inst2	80 (0)	32 (0)	0 (0)	0	0	0
COUNTER10:inst	19 (2)	8 (0)	0 (0)	0	0	0
COUNTER10:inst4	21 (3)	8 (0)	0 (0)	0	0	0
COUNTER10:inst5	20 (3)	8 (0)	0 (0)	0	0	0
COUNTER10:inst6	20 (3)	8 (0)	0 (0)	0	0	0
TF_CTRL:inst3	9 (3)	4 (0)	0 (0)	0	0	0
74154:inst	1 (1)	0 (0)	0 (0)	0	0	0
7493:inst1	5 (5)	4 (4)	0 (0)	0	0	0
CNT:inst5	11 (0)	10 (0)	0 (0)	0	0	0
lpm_counter:lpm_counter_compone...	11 (0)	10 (0)	0 (0)	0	0	0
LOCK32:inst12	32 (0)	32 (0)	0 (0)	0	0	0

图 5-22　top 工程的层次显示窗口

2. 打开层次树中的文件

层次显示窗口可以让设计者迅速打开层次中的设计文件和辅助文件，或将其带至前台。当从层次显示窗口打开一个文件时，Quartus Prime 18 会根据文件类型自动打开相应的编辑器。

下面打开 CNT32B. gdf，将其带至前台。双击 CNT32B: inst2，文件名旁边的 gdf 图标，

图形编辑器启动并把 CNT32B.gdf 带至前台，如图 5-23 所示。

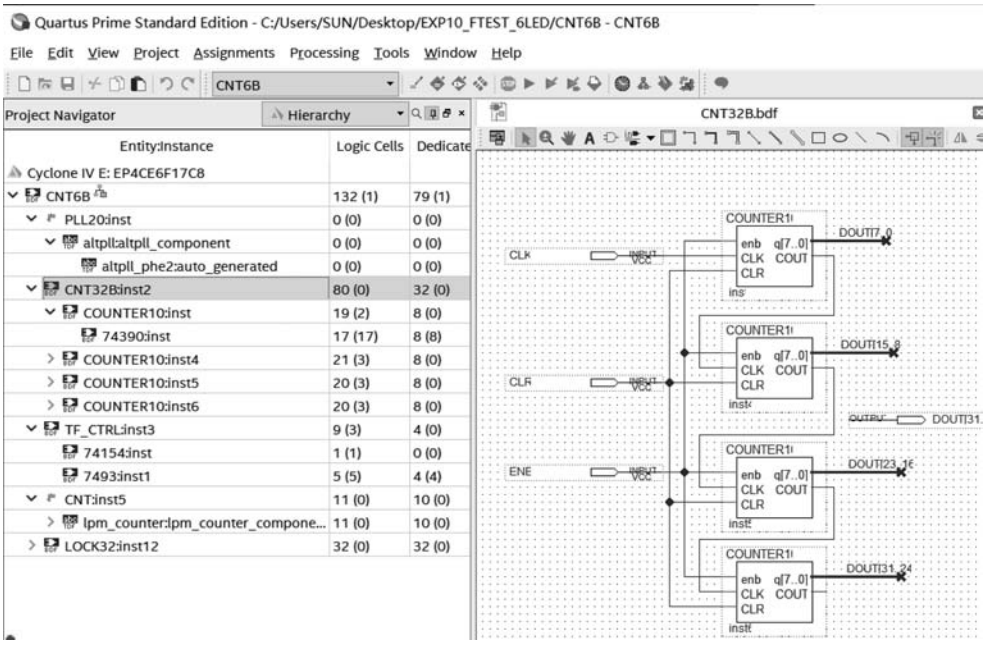


图 5-23　打开层次树中的文件

3. 关闭层次中的文件

在菜单栏中选择 File→Close，或用鼠标左键单击“关闭”按钮，这样就可以关闭已打开的设计文件。

思考题与习题

1. 试说明实体端口模式 BUFFER 和 INOUT 的不同之处。

2. VHDL 的数据对象有哪几种？它们之间有什么不同？

3. 说明下面各定义的意义。

```
SIGNAL a,b,c: BIT : = '0';
CONSTANT TIME1,TIME2: TIME: 20ns;
VARIABLE x,y,z: STD_LOGIC: = 'X';
```

4. 什么是重载函数？重载运算符有何用处？如何调用重载运算符函数？

5. 数据类型 BIT INTEGER BOOLEAN 分别定义在哪个库中？哪些库和程序包总是可见的？

6. 函数和过程有什么区别？

7. 若在进程中加入 WAIT 语句，应注意哪几个方面的问题？

8. 哪些情况下需要用到程序包 STD_LOGIC_UNSIGNED？试举一例。

9. 为什么说一条并行赋值语句可以等效为一个进程？如果是这样的话，怎样实现敏感信号的检测？

10. 比较 CASE 语句与 WITH_SELECT 语句，叙述它们的异同点。

11. 将以下程序段转换为 WHEN ELSE 语句。

```
PROCESS(a,b,c,d)
    BEGIN
        IF a = '0' AND b = '1' THEN next1 < = "1101";
        ELSIF a = '0' THEN next1 < = d;
        ELSIF b = '1' THEN next1 < = c;
        ELSE
        Next1 < = "1011";
        END IF;
END PROCESS;
```

12. 试给出 1 位全减器的算法描述、数据流描述、结构描述和混合描述。

13. 用 VHDL 描述下列器件的功能。

(1) 十进制-BCD 码编码器，输入、输出均为低电平有效。

(2) 时钟(可控)RS 触发器。

(3) 带复位端、置位端、延迟为 15ns 的响应 CP 下降沿的 JK 触发器。

(4) 集成计数器 74161。

(5) 集成移位寄存器 74194。

14. 用 VHDL 描述一个三态输出的双 4 选 1 的数据选择器，其地址信号共用，且各有一个低电平有效的使能端。

15. 试用并行信号赋值语句分别描述下列器件的功能。

(1) 3-8 译码器。

(2) 8 选 1 数据选择器。

16. 利用生成语句描述一个由 n 个一位全减器构成的 n 位减法器，n 的默认值为 4。

17. 用 VHDL 设计实现输出占空比为 50%的 1000 分频器。

18. 用 VHDL 描述一个单稳态触发器。定时时间由类属参数决定。该触发器有 A、B 两个触发信号输入端，A 为上升沿触发(当 B=1 时)，B 为下降沿触发(当 A=0 时)；有 Q 和 $\overline{Q}$ 两个输出端，分别输出正、负两种脉冲信号。

19. 某通信接收机的同步信号为巴克码 1110010。设计一个检测器，其输入为串行码 x，输出为检测结果 y，当检测到巴克码时，输出 1。