第 5 章　VHDL 深入

本章仍将沿用第 3 章中通过一些实例引出了相关的 VHDL 语法结构和语言现象，并给予一定说明的方法，介绍一些新的实例及相关的 VHDL 语法知识，使读者进一步深入了解 VHDL 语言现象和语句规则的特点，以及应用 VHDL 表达与设计电路的方法。

下面首先对第 3 章出现的一些语法现象作更为深入的讨论，然后再引出一些新的实例，帮助读者对相关的语法现象作更详细的了解。

5.1　数据对象及其示例说明

在 VHDL 中，数据对象有三类，即变量(VARIABLE)、常量(CONSTANT)和信号(SIGNAL)。如前所述，数据对象类似于一种容器，它接受不同数据类型的赋值。变量和常量可以从软件语言中找到对应的类型，然而信号的表现较特殊，它具有更多的硬件特征，是 VHDL 中最有特色的语言要素之一。尽管信号和变量已在前面一些示例中出现过，但都没有作更详细的解释，这势必影响对 VHDL 程序的更好理解，以下将针对数据对象给出一些示例，以作进一步的说明。

5.1.1　常数

常数的定义和设置主要是为了使程序更容易阅读和修改。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地改变宽度，从而改变硬件结构。在程序中，常量是一个恒定不变的值，一旦作了数据类型和赋值定义后，在程序中不能再改变，因而具有全局性意义。常数定义的一般表述如下：

```
CONSTANT  常数名：数据类型 := 表达式 ;
```

例如：

```
CONSTANT  FBT : STD_LOGIC_VECTOR := "010110" ; -- 标准位矢类型
CONSTANT  DATAIN : INTEGER := 15 ;              -- 整数类型
```

第 1 句定义常数 FBT 的数据类型是 STD_LOGIC_VECTOR，它等于"010110"；第 2 句定义常数 DATAIN 的数据类型是整数 INTEGER，它等于 15 。

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。

常数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。

常数的可视性，即常数的使用范围取决于它被定义的位置。如果在程序包中定义，常

数具有最大的全局化特征，可以用在调用此程序包的所有设计实体中；常数如果定义在设计实体中，其有效范围为这个实体定义的所有的结构体(含多结构体时)；如果常数定义在设计实体的某一结构体中，则只能用于此结构体；如果常数定义在结构体的某一单元，如一个进程中，则这个常数只能用在这一进程中。这就是常数的可视性规则。这一规则与信号的可视性规则是完全一致的。

5.1.2 变量

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作出定义的当前结构中。变量的赋值是一种理想化的数据传输，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。

定义变量的一般表述如下：

```
VARIABLE 变量名 : 数据类型 := 初始值 ;
```

例如：

```
VARIABLE  a : INTEGER RANGE 0 TO 15 ;--变量 a 定义为常数，取值范围是 0 到 5
VARIABLE  d : STD_LOGIC := '1' ;--变量 a 定义为标准逻辑位数据类型，初始值是 1
```

分别定义 a 的取值范围从 0 到 15 的整数型变量；d 为标准位类型的变量，初始值是 1。

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，同一变量的值将随变量赋值语句前后顺序的运算而改变，因此，变量赋值语句的执行与软件描述语言中的完全顺序执行的赋值操作十分类似。

在变量定义语句可以定义初始值，这是一个与变量具有相同数据类型的常数值，这个表达式的数据类型必须与所赋值的变量一致，初始值的定义不是必需的。此外，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。变量赋值的一般表述如下：

```
目标变量名 := 表达式 ;
```

由此式可见，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，如：

```
VARIABLE  x，y : INTEGER RANGE 15 DOWNTO 0 ;--分别定义变量x和y为整数类型
VARIABLE  a，b : STD_LOGIC_VECTOR(7 DOWNTO 0) ;
x := 11 ;
y := 2 + x ;                    -- 运算表达式赋值，y 也是实数变量
a := b                          --b向a赋值
a(0 TO 5) :=  b(2 TO 7) ;
```

5.1.3 信号

信号是描述硬件系统的基本数据对象，它的性质类似于连接线。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。

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（这决定于语句的表达方式）。这一属性与触发器的记忆功能有很好的对应关系，只是不必注明信号上数据流动的方向。信号定义的语句格式与变量相似，信号定义也可以设置初始值，定义格式是：

```
SIGNAL 信号名： 数据类型 := 初始值 ;
```

同样，信号初始值的设置也不是必需的，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，在实体中定义的信号，在其对应的结构体中都是可见的，即在整个结构体中的任何位置，任何语句结构中都能获得同一信号的赋值。

事实上，除了没有方向说明以外，信号与实体的端口(Port)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(设计芯片内部)的端口。反之，实体的端口只是一种隐形的信号，在实体中对端口的定义实质上是作了隐式的信号定义，并附加了数据流动的方向，而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是一个信号，并加以使用，而不必另作定义。

此外还需要注意，信号的使用和定义范围是实体、结构体和程序包，在进程和子程序的顺序语句中不允许定义信号。此外，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，这是因为只有信号才能把进程外的信息带入进程内部，或将进程内的信息带出进程。

当信号定义了数据类型和表达方式后，在 VHDL 设计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  AFTER 时间量;
```

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，如 AFTER 3 ns。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(不作任何显式的延时设置，即等效于 AFTER 0 ns)，也要经历一个特定的延时，即 ？ 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，因此这与变量的赋值过程有很大差别。

信号的赋值可以出现在一个进程中，也可以直接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，并且允许对同一目标信号进行多次赋值；后者属并行信号赋值，其赋值操作是各自独立并行地发生的，且不允许对同一目标信号进行多次赋值。

即在进程中，可以允许同一信号有多个驱动源(赋值源)，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, y, z: INTEGER ;
  ...
PROCESS (a, b, c)
BEGIN
  y <= a + b ;
  z <= c – a ;
```

```
    y <= b ;
END PROCESS ;
```

上例中，a、b、c 被列入进程敏感表，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 y 的最后一项驱动源是 b，因此 y 被赋值 b。但在并行赋值语句中，不允许如上例所示的同一信号有多个驱动源的情况。

5.1.4 进程中的信号与变量赋值

准确理解和把握一个进程中的信号和变量赋值行为的特点以及它们功能上的异同点，对利用 VHDL 进行正确地设计电路十分重要。以下对信号和变量这两种数据对象在赋值上的异同点作一些比较分析，使读者对它们有更深刻的理解。

一般地，从硬件电路系统来看，变量和信号相当于逻辑电路系统中的连线和连线上的信号值；常量相当于电路中的恒定电平，如 GND 或 VCC 接口。从行为仿真和 VHDL 语句功能上看，信号与变量具有比较明显的区别，其差异主要表现在接受和保持信号的方式和信息保持与转递的区域大小上。例如信号可以设置传输延迟量，而变量则不能；变量只能作为局部的信息载体，如只能在所定义的进程中有效，而信号则可作为模块间的信息载体，如在结构体中各进程间传递信息。变量的设置有时只是一种过渡，最后的信息传输和界面间的通信都靠信号来完成。综合后的 VHDL 文件中信号将对应更多的硬件结构。

但对于信号和变量的认识单从行为仿真和纯语法的角度去认识是不完整的。事实上，在许多情况下，综合后所对应的硬件电路结构中信号和变量并没有什么区别。例如在满足一定条件的进程中，综合后它们都能引入寄存器，如例 3-23。其关键在于，它们都具有能够接受赋值这一重要的共性，而 VHDL 综合器并不理会它们在接受赋值时存在的延时特性(只有 VHDL 仿真器才会考虑这一特性差异)。

表 5-1 就基本用法、适用范围和行为特性方面对信号与变量作了比较。

表 5-1 信号与变量赋值语句功能的比较

	信号 SIGNAL	变量 VARIABLE
基本用法	用于作为电路中的信号连线	用于作为进程中局部数据存储单元
适用范围	在整个结构体内的任何地方都能适用	只能在所定义的进程中使用
行为特性	在进程的最后才对信号赋值	立即赋值

为了更具体地了解信号与变量的特性，以下给出示例说明。

试比较例 5-1 和例 5-2。例 5-2 与例 3-6 是相同的，是典型的 D 触发器的 VHDL 描述，综合后的电路如图 5-2 所示。例 5-1 仅是将例 5-2 中定义的信号换成了变量，其综合的结果与例 5-2 完全一样，也是图 5-2 所示的 D 触发器。此 2 例表明，在不完整的条件语句中，单独的变量赋值语句与信号赋值语句都能产生相同的时序电路，此时变量已不是简单的数据临时储存结构。

【例 5-1】

```
. . .
ARCHITECTURE bhv OF DFF3 IS
```

```
  BEGIN
  PROCESS (CLK)
   VARIABLE QQ : STD_LOGIC ;
   BEGIN
    IF  CLK'EVENT AND CLK = '1'  THEN QQ := D1 ;
    END IF;
 END PROCESS ;
     Q1 <= QQ;
 END ;
```

【例 5-2】

```
. . .
ARCHITECTURE bhv OF DFF3 IS
   SIGNAL QQ : STD_LOGIC ;
  BEGIN
  PROCESS (CLK)
   BEGIN
    IF  CLK'EVENT AND CLK = '1'  THEN  QQ <= D1 ;
    END IF;
 END PROCESS ;
     Q1 <= QQ;
END ;
```

再较例 5-3 和例 5-4，它们惟一的区别是对进程中的 A 和 B 定义了不同的数据对象，前者定义为信号而后者定义为变量。然而，它们的综合结果却有很大的不同：前者的电路是图 5-1，后者的电路是图 5-2，与例 5-2 的结果完全一样。

【例 5-3】

```
LIBRARY IEEE ;
USE IEEE.STD_LOGIC_1164.ALL ;
ENTITY DFF3 IS
PORT ( CLK,D1 : IN STD_LOGIC ;
          Q1  : OUT STD_LOGIC  ) ;
END ;
 ARCHITECTURE bhv OF DFF3 IS
   SIGNAL A,B : STD_LOGIC ;
  BEGIN
  PROCESS (CLK) BEGIN
    IF  CLK'EVENT AND CLK = '1'  THEN
      A <= D1 ;
      B <= A ;
     Q1 <= B ;
    END IF;
 END PROCESS ;
END ;
```

【例 5-4】

```
LIBRARY IEEE ;
 USE IEEE.STD_LOGIC_1164.ALL ;
 ENTITY DFF3 IS
```

```
PORT ( CLK,D1 : IN STD_LOGIC ;
            Q1 : OUT STD_LOGIC  ) ;
END ;
 ARCHITECTURE bhv OF DFF3 IS
  BEGIN
  PROCESS (CLK)
   VARIABLE A,B : STD_LOGIC ;
   BEGIN
    IF  CLK'EVENT AND CLK = '1'  THEN
      A := D1 ;
      B := A ;
     Q1 <= B ;
    END IF;
  END PROCESS ;
END ;
```

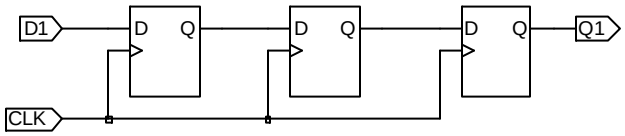

图 5-1 例 5-3 的 RTL 电路

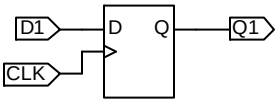

图 5-2 D 触发器电路

在此，例 5-3 和例 5-4 对信号与变量在延时特性的差别上作了很好的注解。对于例 5-3 中信号赋值行为的理解必须把握表 5-1 中的信号行为特性的三个重要方面：

(1) 信号的赋值需要有一个 ？延时，例如当执行到例 5-3 表式 A<=D1 时，D1 向 A 的赋值是在一个？延时后发生的，此时 A 并未得到更新，即 A 并未获得 D1 的值，只是刚刚启动了一个延时为?的模拟定时器，只有在延时为?后，A 才能被更新，获得 D1 的赋值。

(2) 在进程中，所有赋值语句，包括变量赋值，都必须在一个？延时中完成（变量在?延时前即已按顺序完成赋值）。一方面，在进程中的所有信号赋值语句在进程启动的一瞬间立即顺序启动各自的延时为?的定时器，预备在定时结束后分别执行赋值操作，但另一方面，这种顺序启动的间隔几乎为 0，而在顺序执行到“ END PROCESS ”语句时，？延时才结束，因此这时，在进程中的所有信号赋值操作几乎在同时完成赋值（即令赋值对象的值发生更新），即在进程中的顺序赋值是以近乎并行的方式“ 同时 ”完成的，并且是在执行到“ END PROCESS ”语句时才发生。因此不难理解，执行赋值操作和完成赋值是两个不同的概念，对于类似于 C 的软件语言，执行或完成一条语句的赋值是没有区别的，但对于 VHDL 的信号的赋值有很大的不同。“ 执行赋值 ”只是一个过程，它具有顺序的特征；而“ 完成赋值 ”，是一种结果，它的发生具有硬件描述语言最本质的并行的特征。

(3) 当在进程中存在同一信号有多个赋值源(即对同一信号发生多次赋值)时，实际完成赋值，即赋值对象的值发生更新的信号是最接近“ END PROCESS ”语句的信号！

为了更好地理解，首先考察例 5-5。

如上所述，由于进程中的顺序赋值部分没有时间的流逝，所以在顺序语句部分，无论有多少语句，都必须在到达 END PROCESS 语句时，?延迟才能发生，模拟器时钟才能向前

推进。设例 5-5 的进程在 2ns+?时刻被启动，在此后的?时间进程中，所有行为都必须执行完。在 2ns+?时刻，信号 e1 被赋值为 1010（注意并没有即刻获得），变量 c1 被赋值为 0011，但信号 e1 的值在 2ns+2?时刻才被更新，而变量 c1 在赋值的瞬间即被更新，即在 2ns+?时刻，其值就变成 0011。尽管 c1 的赋值语句排在 e1 之后(假定是第 30+m 行)，但 c1 获得 0011 值的时刻比 e1 获得 1010 值的时刻早一个?。

从行为仿真的角度看，对 e1 和 c1“执行赋值”是有顺序性的，即先执行 e1 的赋值操作，后执行 c1 的赋值操作；但“完成赋值”的情况并非一致，实际情况是 c1 在前 e1 在后！

【例 5-5】

```
SIGNAL  in1, in2, e1,  ... : STD_LOGIC ;
...
PROCESS(in1, in2,  . . .)
VARIABLE c1, . . . : STD_LOGIC_VECTOR(3 DOWNTO 0) ;
  BEGIN
    IF in1 = '1' THEN ...           -- 第 1 行
       e1 <= "1010" ;               -- 第 2 行
          ...
    IF in2 = '0' THEN . . .         -- 第 15+n 行
          ...
       c1 := "0011" ;               -- 第 30+m 行
          ...
    END IF;
 END PROCESS;
```

根据以上的讨论就比较好理解例5-3的信号赋值现象了。

由于例5-3中的3个赋值语句都必须在遇到END PROCESS后的?时刻内执行，所以它们具有了近乎并行执行的特性，即语句A<=D1中的A和语句 B<=A 中的A并非是同一时刻的值，B<=A与Q1<=B中的B也非同时刻的B，它们都相差一个?时间。因此在同一时刻中，D1不可能将值传到Q1，使Q1得到更新。在实际运行中，A被更新的值是上一时钟周期的D1(即当前时钟上升沿以前的值)，B被更新的值是上一时钟周期的A，而Q1被更新的值也是上一时钟周期的B1。显然此程序的综合结果只能是图5-1所示的电路。

例 5-4 就不同了。由于 A、B 是变量，它们具有临时保留数据的特性，而且它们的赋值更新是立即发生的，因而有了明显的顺序性。当 3 条赋值语句顺序执行时，变量 A 和 B 就有了传递数据的功能。语句执行中，先将 D1 的值传给 A，再通过 A 传给 B，最后在一个? 时刻后，由 B 传给 Q1。在这些过程中，A 和 B 只担当了 D1 数据的暂存单元。Q1 最终被更新的值是上一时钟周期的 D1。由此来看，例 5-4 的综合结果确实应该是图 5-2 所示的单个 D 触发器了。

下面通过比较例 5-6 和 5-7 可以进一步了解顺序语句中信号与变量之间的差别。

【例 5-6】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux4 IS
PORT (i0, i1, i2, i3, a, b : IN STD_LOGIC;
```

```
     q : OUT STD_LOGIC);
END mux4;
ARCHITECTURE body_mux4 OF mux4 IS
signal muxval : integer range 7 downto 0;
BEGIN
process(i0,i1,i2,i3,a,b)
begin
                    muxval <= 0;
if (a = '1') then   muxval <= muxval + 1; end if;
if (b = '1') then   muxval <= muxval + 2; end if;
case muxval is
     when 0 => q <= i0;
     when 1 => q <= i1;
     when 2 => q <= i2;
     when 3 => q <= i3;
     when others => null;
end case;
end process;
END body_mux4;
```

【例 5-7**】**

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux4 IS
PORT (i0, i1, i2, i3, a, b : IN STD_LOGIC;
     q : OUT STD_LOGIC);
END mux4;
ARCHITECTURE body_mux4 OF mux4 IS
BEGIN
process(i0,i1,i2,i3,a,b)
variable muxval : integer range 7 downto 0;
begin
                     muxval := 0;
if (a = '1') then   muxval := muxval + 1;  end if;
if (b = '1') then   muxval := muxval + 2;  end if;
case muxval is
     when 0 => q <= i0;
     when 1 => q <= i1;
     when 2 => q <= i2;
     when 3 => q <= i3;
     when others => null;
end case;
end process;
END body_mux4;
```

从以上二例的结构看，作者的意图是要设计一个 4 **选** 1 **多路选择器，对应的电路理应是一个纯组合电路，其中的** a **和** b **是通道选通控制信号。**

例 5-6 **和例** 5-7 **的主要不同在于，前者将标识符** muxval **定义为信号，后者将其定义为变量。结果综合出了完全不同的电路，因而产生了迥异的时序波形。它们综合后的电路分**

别示于图 5-3 和图 5-4；它们的工作时序分别示于图 5-5 和图 5-6。

不难发现，图 5-3 含有时序电路模块，而对应的工作时序图图 5-5 表明，仿真未获得输出结果，显然此例的设计是错误的！而图 5-4 是纯组合电路，仿真波形图图 5-6 中显示出了 Q 的时序结果，电路设计是正确的。那么，问题出在哪里呢？

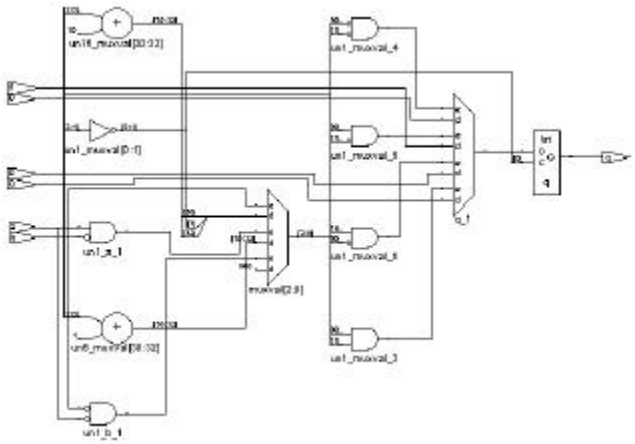

图 5-3　例 5-6 的 RTL 电路（Synplify 综合）

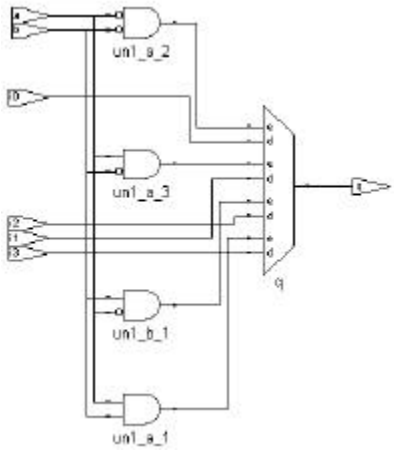

图 5-4　例 5-7 的 RTL 电路（Synplify 综合）

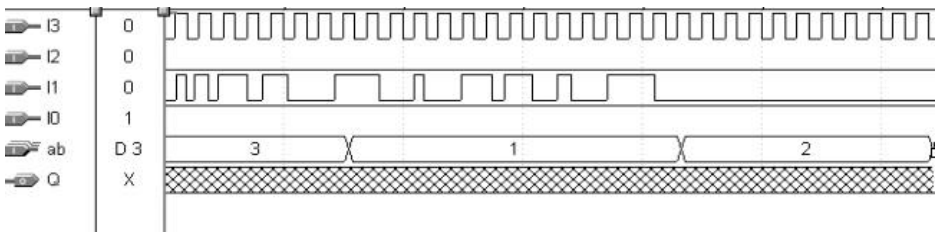

图 5-5　例 5-6 中错误的工作时序

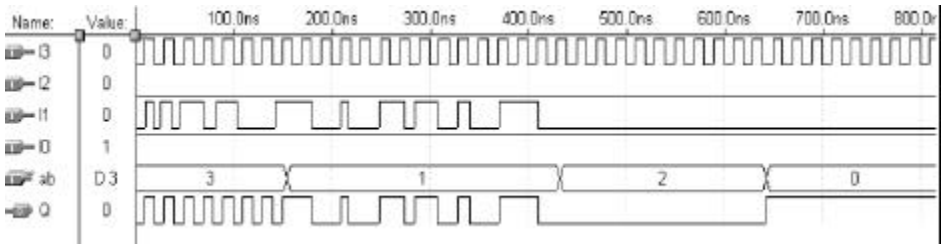

图 5-6　例 5-7 中正确的工作时序

例 5-6 中，信号 muxval 在进程中出现了 3 次赋值操作，即有 3 个赋值源：muxval<=0、muxval<=muxval+1 和 muxval<=muxval+2，但根据进程中信号的赋值规则，前两个赋值语句中的赋值目标信号 muxval 都不可能得到更新，只有最后的 muxval<=muxval+2 语句中的 muxval 的值得到更新，然而传输符号右边的 muxval 始终未得到任何确定的初始值，即语句 muxval<=0 并未完成赋值。结果只能被综合成随 b 和 a(a 和 b 被综合成了时钟输入信号)变动的时序电路，导致 muxval 成为一个不确定的信号。结果在进程最后的 CASE 语句中，

无法通过判断 muxval 的值来确定选通输入，即对 q 的赋值。

例 5-7 就不一样了。程序首先将 muxval 定义为变量，根据变量顺序赋值以及暂存数据的规则，首先执行了语句 muxval:=0（muxval 即刻被更新），从而使两个 IF 语句中的 muxval 都能得到确定的初值。另一方面，当 IF 语句不满足条件时，即当 a 或 b 不等于 1 时，由于 muxval 已经在第一条赋值语句中被更新为确定的值(即 0)了，所以尽管两个 IF 语句从表面上看都属于不完整的条件语句，但都不可能被综合成时序电路了。显然从图 5-4 中只能看到一个纯组合电路了，它们(图 5-6)也就有了正确的波形输出。

例 5-8 是 8 位 CPU 设计中常用的移位寄存器模块，是用 CASE 语句设计的并行输入输出的移位寄存器。我们不妨通过此例再一次体会信号赋值的特性。

例5-8利用进程的顺序语句构成了时序电路，同时又利用了信号赋值的“并行”特性实现了移位。以带进位循环左移操作为例，当MD="001"时，虽然此项WHEN语句中含有的3个赋值语句是顺序语句，但他们并不会发生原数据的覆盖情况。例如，顺序执行REG(0) <= C0 和REG(7 DOWNTO 1) <= REG(6 DOWNTO 0)后并不会发生REG(1)=C0的情况，因为它们是被同时更新的。

图5-7是此情况下的仿真波形。图中，在CLK的第一个上升沿处，MD="101"，此边沿将并行口D的数据（D=″10011010″）加载于移位寄存器中；在此后的3个上升沿处，MD都等于"001"，即执行带进位循环左移操作。在第2个上升沿后，将进位输入的C0的'1' 移入寄存器最低位，其余左移一位，最高位被移出进入CN。另外注意，最后一个上升沿后执行的是自循环右移，即将此前寄存器中的数据右移一位，期间将移出的最低位补入右移结果的最高位。

【例5-8】

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY SHIFT IS
    PORT (CLK,C0 :  IN STD_LOGIC;  --时钟和进位输入
            MD    :  IN STD_LOGIC_VECTOR(2 DOWNTO 0); --移位模式控制字
            D     :  IN STD_LOGIC_VECTOR(7 DOWNTO 0); --待加载移位的数据
            QB    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0); --移位数据输出
            CN    : OUT STD_LOGIC);                    --进位输出
END ENTITY;
ARCHITECTURE BEHAV OF SHIFT IS
      SIGNAL REG  : STD_LOGIC_VECTOR(7 DOWNTO 0);
      SIGNAL   CY : STD_LOGIC ;
 BEGIN
 PROCESS (CLK,MD,C0)
  BEGIN
   IF CLK'EVENT AND CLK = '1' THEN
     CASE MD IS
      WHEN "001" => REG(0) <= C0 ;
        REG(7 DOWNTO 1) <= REG(6 DOWNTO 0); CY <= REG(7);--带进位循环左移
      WHEN "010" => REG(0) <= REG(7);
        REG(7 DOWNTO 1) <= REG(6 DOWNTO 0);               --自循环左移
      WHEN "011" => REG(7) <= REG(0);
```

```
        REG(6 DOWNTO 0) <= REG(7 DOWNTO 1);                    --自循环右移
      WHEN "100" =>  REG(7) <= C0 ;
        REG(6 DOWNTO 0) <= REG(7 DOWNTO 1); CY <= REG(0); --带进位循环右移
      WHEN "101" =>  REG(7 DOWNTO 0) <=   D(7 DOWNTO 0);     --加载待移数
       WHEN OTHERS => REG <= REG ;  CY <= CY ;                 --保持
    END CASE;
  END IF;
 END PROCESS;
    QB(7 DOWNTO 0) <= REG(7 DOWNTO 0); CN <= CY;              --移位后输出
END BEHAV;
```

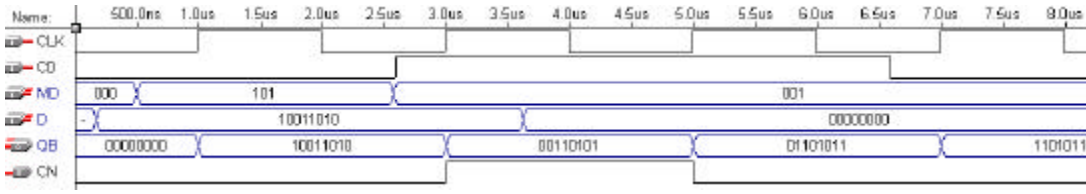

图 5-7　例 5-8 中带进位循环左移仿真波形（MD="001"）

5.2　双向和三态电路信号赋值例解

引入三态门有许多实际的应用，如 CPU 设计中的数据和地址总线的构建，RAM 或堆栈的数据端口的设计等。利用 VHDL 在 FPGA 开发设计中引入三态控制电路是完全可以实现的。在设计中，如果用 STD_LOGIC 数据类型的 'Z' 对一个变量赋值，即会引入三态门，并在控制下可使其输出呈高阻态，这等效于使三态门禁止输出。

5.2.1　三态门设计

例 5-9 是一个 8 位三态控制门电路的描述，当使能控制信号为‘ 1 ’时，8 位数据输出；为‘ 0 ’时输出呈高阻态，语句中将高阻态数据“ ZZZZZZZZ ”向输出端口赋值，其综合结果示于图 5-8。

【例 5-9】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY tri_s IS
    port (     enable : IN STD_LOGIC;
            datain : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
           dataout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );
END tri_s ;
ARCHITECTURE bhv OF tri_s IS
BEGIN
PROCESS(enable,datain)
   BEGIN
   IF enable = '1' THEN dataout <= datain ;
```

```
    ELSE dataout <="ZZZZZZZZ" ;
  END IF ;
END PROCESS;
END bhv;
```

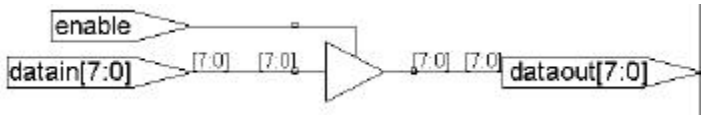

图 5-8　8位 3 态控制门电路（Synplify 综合）

一般地，可以首先将某信号定义为 STD_LOGIC 数据类型，将 'Z' 赋给这个变量来获得 3 态控制门电路，一个 'Z' 表示一个逻辑位。这是由于 STD_LOGIC 数据类型中含有元素'Z'。但应注意，由于'Z'在综合中是一个不确定的值，不同的综合器可能会给出不同的结果，因而对于 VHDL 综合前的行为仿真与综合后功能仿真结果也可能是不同的，有时虽然能通过综合，但却不能获得正确的时序仿真结果，所以建议尽可能不要将'Z'用作比较值，或用作表达式或操作数。此外还应注意，虽然对于关键词，VHDL 语法规定不区分大小写，但当把表示高阻态的'Z'值赋给一个数据类型为 STD_LOGIC 的变量或信号时，'Z'必须是大写，这是因为在 IEEE 库中对数据类型 STD_LOGIC 的预定义已经将高阻态确定为大写'Z'。

无论是设计外围端口上的三态门还是设计器件内部的三态门，在 VHDL 表达上都是一样的，在综合与仿真上也不会有问题。但如果最终实现的目标器件是 FPGA/CPLD 器件，是否能被适配进去，则必须根据具体器件系列来确定，因为不同的器件结构，对三态的综合适配会有不同的结果，现代大多数 FPGA/CPLD 器件内部都无法构成三态门，而只能在端口上形成三态控制端口。因此，诸如 CPU 内部设计中的各路总线的控制选通只能用多路选择器的结构来实现。有的器件甚至在端口上也难以形成可以任意方式控制的三态输出结构，如 ispLSI1000 系列器件。

5.2.2　双向端口设计

用 INOUT 模式设计双向端口也必须考虑三态的使用，因为双向端口的设计与三态端口的设计十分相似，都必须考虑端口的三态控制。这是由于双向端口在完成输入功能时，必须使原来呈输出模式的端口呈高阻态，否则，待输入的外部数据势必会与端口处原有电平发生“线与”，导致无法将外部数据正确地读入，从而实现“双向”的功能。

下面来考察比较例 5-10 和例 5-11 两个双向端口的 VHDL 设计实例。

【例 5-10】

```
library ieee;
use ieee.std_logic_1164.all;
entity tri_state is
port (control : in std_logic;
      in1: in std_logic_vector(7 downto 0);
       q : inout std_logic_vector(7 downto 0);
       x : out std_logic_vector(7 downto 0));
end tri_state;
```

```
architecture body_tri of tri_state is
begin
process(control,q,in1)
begin
    if (control = '0') then   x <= q  ;
    else     q <= in1; x<="ZZZZZZZZ" ;
    end if;
end process;
end body_tri;
```

【例 5-11】
(以上部分同上例)

```
process(control,q,in1)
begin
    if (control='0') then  x <= q ; q <= "ZZZZZZZZ";
                else  q <= in1; x <="ZZZZZZZZ";
    end if;
end process;
end body_tri;
```

例 5-10 和例 5-11 都将 q 定义为双向端口，而 x 定义为三态控制输出口。它们的区别仅在于，前者当利用 q 的输入功能将 q 端口的数据读入并传输给 x(即执行 x<=q)时，没有将 q 的端口设置成高阻态输出，即执行语句：X<=“ ZZZZZZZZ ”，从而导致了如图 5-11 所示的综合结果，这是一个错误的逻辑电路。从它的仿真波形图(图 5-9)可以清楚地看到，当 control 为'0'时，x 无法得到正确的输出结果。从图 5-11 还发现，尽管在程序的实体部分已经明确地定义了 q 为双向端口，但显示在电路中的综合结果却只是一个输出端口，而且在电路中还被插入了一个锁存器。例 5-10 变成时序电路的原因十分简单：表面上看，其中的 IF 语句是一个完整的条件语句，但这仅是对 x 而言的，对 q 并非如此，即在 control 的两种不同条件下('1'和'0')都给出了 x 的输出数据，而 q 只在 control 为'1'时，执行赋值命令；而当为'0'时，没有给出 q 的操作说明，这是一个非完整条件语句。后者（例 5-11）的情况就不同了，例中仅增加了语句 q<=“ ZZZZZZZZ ”，就解决了两个重要的问题：

(1) 使 q 在 IF 语句中有了完整的条件描述，从而克服了时序元件的引入；

(2) 在 q 履行输入功能时，将其设定为高阻态输出，使 q 成为真正的双向端口(图 5-12)。从电路 5-12 的仿真波形图(图 5-10)可见，无论 control 为'1'或'0'，q 和 x 都能得到正确的输出结果。

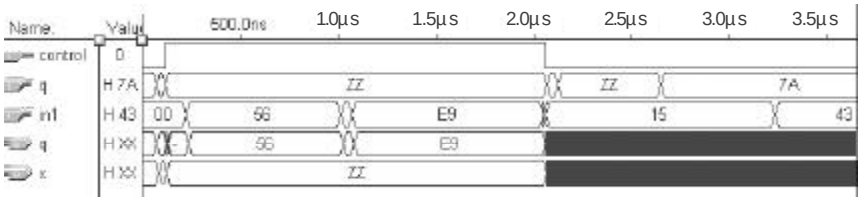

图 5-9 例 5-10 的仿真波形图

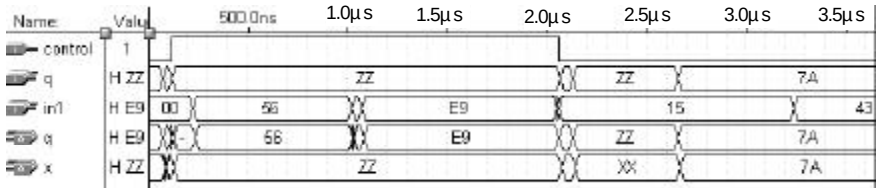

图 5-10 例 5-11 的仿真波形图

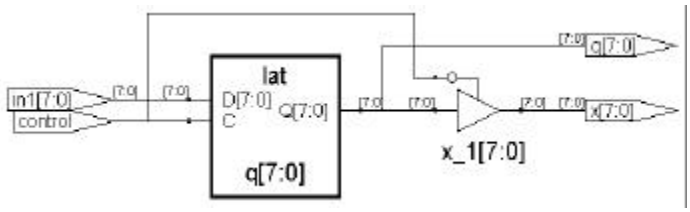

图 5-11 例 5-10 的综合结果（Synplify 综合）

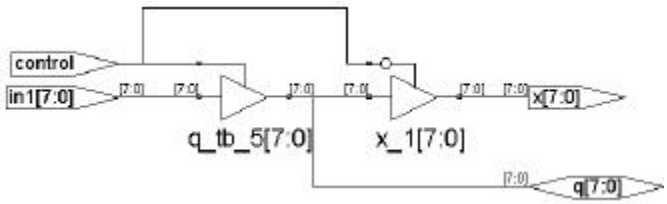

图 5-12 例 5-11 的综合结果（Synplify 综合）

5.2.3 三态总线电路设计

为构成芯片内部的总线系统，必须设计三态总线驱动器电路(注意，大部分 FPGA 中无法形成三态电路)，这可以有多种表达方法，但必须注意信号多驱动源的处理问题。例 5-12 和 5-13 都试图描述一个 8 位 4 通道的三态总线驱动器，但其中一个程序是错误的。

例 5-12 在一个进程结构中放了 4 个顺序完成的 IF 语句，并且是完整的条件描述句。一般认为，会产生 4 个 8 位的三态控制通道，且输出只有一个信号端 output。但如果考虑到上一节对进程语句中信号赋值特点的讨论，就会发现例 5-12 中的输出信号 output 在任何条件下都有 4 个激励源，即赋值源，它们不可能被顺序赋值更新。这是因为在进程中，顺序等价的语句，包括赋值语句和 IF 语句等，当它们列于同一进程敏感表中的输入信号同时变化时(即使不是这样，综合器对进程也自动考虑这一可能的情况)，只可能对进程结束前的那一条赋值语句(含 IF 语句等)进行赋值操作，而忽略其上的所有的等价语句。这就是说，例 5-12 虽然能通过综合，却不能实现原有的设计意图。显然，这是一个错误的设计方案。

此外建议一般情况下,同一进程中最好只放一个 IF 语句结构(可以包含嵌入的 IF 语句)，无论描述组合逻辑还是时序逻辑都一样。

【例 5-12】

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

```
ENTITY tristate2 IS
    port ( input3, input2, input1, input0 :
              IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    enable : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
    output : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END tristate2 ;
ARCHITECTURE multiple_drivers OF tristate2 IS
BEGIN
PROCESS(enable,input3, input2, input1, input0 )
   BEGIN
   IF enable = "00" THEN output <= input3 ;
       ELSE output <=(OTHERS => 'Z');
   END IF ;
   IF enable = "01" THEN output <= input2 ;
       ELSE output <=(OTHERS => 'Z');
   END IF ;
   IF enable = "10" THEN output <= input1 ;
       ELSE output <=(OTHERS => 'Z');
   END IF ;
   IF enable = "11" THEN output <= input0 ;
       ELSE output <=(OTHERS => 'Z');
   END IF ;
END PROCESS;
END multiple_drivers;
```

【例 5-13**】　（注：**MaxplusII **不支持本例）**

```
library ieee;
use ieee.std_logic_1164.all;
entity tri2 is
port (ctl : in  std_logic_vector(1 downto 0);
        datain1， datain2，datain3， datain4 :
             in std_logic_vector(7 downto 0);
       q : out std_logic_vector(7 downto 0)   );
end tri2;
architecture body_tri of tri2 is
begin
 q <= datain1   when ctl="00" else  (others =>'Z') ;
 q <= datain2   when ctl="01" else  (others =>'Z') ;
 q <= datain3   when ctl="10" else  (others =>'Z') ;
 q <= datain4   when ctl="11" else  (others =>'Z') ;
end body_tri;
```

从例 5-12 **的综合电路(图** 5-13)**可以看出，除了** input0 **外，其余** 3 **个** 8 **位输入端都悬空没能用上，显然是因为恰好将** input0 **安排作为进程中** output **的最后一个激励信号的原因！**

例 5-13 **由于在结构体中使用了** 4 **个并列的** WHEN-ELSE **并行语句，因此能综合出图** 5-14 **中的正确电路结构。这是因为，在结构体中的每一条并行语句都等同于一个独立运行的进程，它们独立监测各并行语句中作为敏感信号的输入值** ctl**。例** 5-13 **表明，设计出能产生独立控制的多通道的电路结构，必须使用并行语句结构。不难理解，如果将例** 5-12 **中的**

4 个 IF 语句放在 4 个并列的进程语句中，就一定能综合出与例 5-13 相同的正确结果来。

事实上，初一看，例 5-13 的结构是错误的，因为对于同一信号有并行 4 个赋值源，在实际电路上完全可能发生“线与”，所以在一般语法上显然是错误的。但应注意，此例中，在 else 后使用了高阻态赋值 (others =>'Z') (注，此句等效于″ZZZZZZZZ″)，不可能发生“线与”；但如果将(others =>'Z')改成(others =>'0')就不一样了，综合必定无法通过！

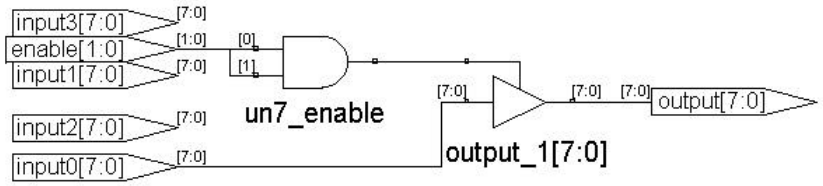

图 5-13 例 5-12 错误的综合结果（Synplify 综合结果）

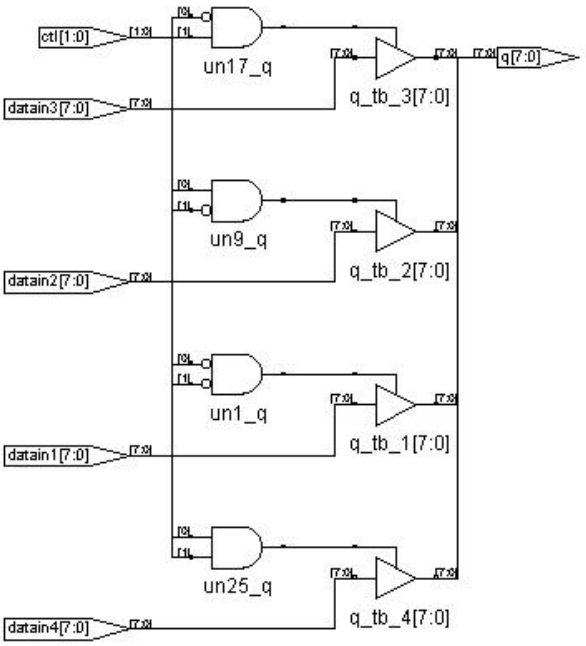

图 5-14 例 5-13 正确的综合结果（Synplify 综合结果）

5.3 IF 语句概述

在前面出现的许多示例中都含有不同表达方式的 IF 语句。IF 语句是 VHDL 设计中最重要和最常用的顺序条件语句，以下对 IF 语句用法及表述作一概述。

IF 语句作为一种条件语句，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。IF 语句的语句结构有以下 4 种：

```
(1) IF   条件句  Then
         顺序语句
    END IF ;
(2) IF  条件句 Then
      顺序语句
    ELSE
      顺序语句
    END IF ;
(3) IF  条件句 Then
       IF  条件句 Then
         ...
       END IF
    END IF
(4) IF  条件句 Then
     顺序语句
    ELSIF  条件句 Then
      顺序语句
    ...
    ELSE
     顺序语句
    END IF
```

IF 语句中至少应有一个条件句，“条件句”可以是一个 Boolean 类型的标识符，如 IF a1 THEN..，或者是一个判别表达式，如 IF a<b+1 THEN..；判别表达式输出的值，即判断结果的数据类型是 Boolean。IF 语句根据条件句产生的判断结果是 true 或是 false，有条件地选择执行其后的顺序语句。注意例 5-14 中对端口数据类型的定义：

【例 5-14】

```
 LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 ENTITY control_stmts IS
PORT (a, b, c: IN BOOLEAN;
       output: OUT BOOLEAN);
 END control_stmts;
 ARCHITECTURE example OF control_stmts IS
 BEGIN
   PROCESS (a, b, c)
     VARIABLE n: BOOLEAN;
   BEGIN
     IF  a  THEN   n := b;   ELSE   n := c;
     END IF;
     output <= n;
   END PROCESS;
 END example;
```

以下简要介绍此四类条件语句：

第(1)种条件语句的执行情况是，当执行到此句时，首先检测关键词 IF 后的条件表达式的布尔值是否为真，如果条件为真，于是(THEN)将顺序执行条件句中列出的各条语句，直到 END IF，即完成全部 IF 语句的执行。如果条件检测为假，则跳过以下的顺序语句不

予执行，直接结束 IF 语句的执行。这种语句形式在以上已出现过多次，是一种非完整性条件语句，通常用于产生时序电路。

与第(1)种语句相比，第(2)种 IF 语句差异仅在于当所测条件为 FALSE 时，并不直接跳到 END IF 结束条件句的执行，而是转向 ELSE 以下的另一段顺序语句进行执行。所以第(2)种 IF 语句具有条件分支的功能，就是通过测定所设条件的真假以决定执行哪一组顺序语句，在执行完其中一组语句后，再结束 IF 语句的执行。这是一种完整性条件语句，它给出了条件句所有可能的条件，因此通常用于产生组合电路。

第(3)种 IF 语句是一种多重 IF 语句嵌套式条件句，可以产生比较丰富的条件描述，既可以产生时序电路，也可以产生组合电路，或是两者的混合。如例 3-23 的移位寄存器。

该语句在使用中应注意，END IF 结束句应该与嵌入条件句数量一致。

第(4)种 IF 语句与第(3)种语句一样，也可以实现不同类型电路的描述。在以上的许多示例中都用到了这种语句。该语句通过关键词 ELSIF 设定多个判定条件，以使顺序语句的执行分支可以超过两个。这一类型的语句有一个重要特点，就是其任一分支顺序语句的执行条件是以上各分支所确定条件的相与(即相关条件同时成立)，即语句中顺序语句的执行条件具有向上相与的功能。有的逻辑设计恰好需要这种功能。

例 5-15 正是利用了 IF 语句中各条件向上相与这一功能，以十分简洁的描述完成了一个 8 线-3 线优先编码器的设计，表 5-2 是此编码器的真值表。

表 5-2　8 线-3 线优先编码器真值表

输　入								输　出		
din0	din1	din2	din3	din4	din5	din6	din7	output0	output1	output2
x	x	x	x	x	x	x	0	0	0	0
x	x	x	x	x	x	0	1	1	0	0
x	x	x	x	x	0	1	1	0	1	0
x	x	x	x	0	1	1	1	1	1	0
x	x	x	0	1	1	1	1	0	0	1
x	x	0	1	1	1	1	1	1	0	1
x	0	1	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	1	1	1	1

注：表中的“x”为任意，类似 VHDL 中的“－”值。

【例 5-15】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY coder IS
  PORT (    din : IN STD_LOGIC_VECTOR(0 TO 7);
         output : OUT STD_LOGIC_VECTOR(0 TO 2)  );
END coder;
ARCHITECTURE behav OF coder IS
  SIGNAL SINT : STD_LOGIC_VECTOR(4 DOWNTO 0);
  BEGIN
```

```
  PROCESS (din)
BEGIN
    IF (din(7)='0') THEN  output <= "000" ;
  ELSIF (din(6)='0') THEN  output <= "100" ;
  ELSIF (din(5)='0') THEN  output <= "010" ;
  ELSIF (din(4)='0') THEN  output <= "110" ;
  ELSIF (din(3)='0') THEN  output <= "001" ;
  ELSIF (din(2)='0') THEN  output <= "101" ;
  ELSIF (din(1)='0') THEN  output <= "011" ;
                     ELSE  output <= "111" ;
END IF ;
  END PROCESS ;
 END behav;
```

显然，程序的最后一项赋值语句 output <= **“** 111 **”的执行条件(相与条件)是：**
(in(7)='1')AND(in(6)='1')AND(in(5)='1')AND(in(4)='1')AND(in(3)='1')AND(in(2)='1')AND(in(1)='1')AND(in(0)='0')；

这恰好与表 5-2 **最后一行吻合。**

5.4　进程语句归纳

本节对进程语句结构及其使用特点作一归纳总结。

在一个结构体中，允许放置任意多个进程语句结构，而每一进程的内部是由一系列顺序语句来构成的。PROCESS **语句结构包含(或者说引导)了一个代表着设计实体中部分逻辑行为的、独立的顺序语句描述的进程。与并行语句的同时执行方式不同，顺序语句可以根据设计者的要求，利用顺序可控的语句完成逐条执行的功能。顺序语句与** C **或** PASCAL **等软件编程语言中语句功能有类似的顺序执行的功能。但必须注意，在** VHDL **中，所谓顺序仅仅是指语句按序执行上的顺序性，但这并不意味着** PROCESS **语句结构在综合后所对应的硬件逻辑行为也具有相同的顺序性。**PROCESS **结构中的顺序语句，及其所谓的顺序执行过程只是相对于计算机中的软件行为仿真的模拟过程而言的，这个过程与硬件结构中实现的对应的逻辑行为是不完全相同的。**PROCESS **结构中既可以有时序逻辑的描述，也可以有组合逻辑的描述，它们都可以用顺序语句来表达。**

5.4.1　进程语句格式

PROCESS **语句结构的一般表达格式如下**

```
[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];
```

每一个 PROCESS 语句结构可以赋予一个进程标号。进程说明部分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当它的某个敏感信号(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，这个行为由进程语句中的顺序语句定义，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，执行过程将自动返回到进程的第一个语句，以等待下一次敏感信号变化。

一个结构体中可以含有多个 PROCESS 结构，每一 PROCESS 结构对于其敏感信号参数表中定义的任一敏感参量的变化，可以在任何时刻被激活或者称为启动。而在一结构体中，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。PROCESS 语句必须以语句 END PROCESS [进程标号]；结尾，对于目前常用的综合器来说，其中进程标号不是必须的，敏感表旁的[IS]也不是必须的。

5.4.2 PROCESS 组成

如上所述，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、变量、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，它们包括：

- 信号赋值语句：在进程中将计算或处理的结果向信号赋值。
- 变量赋值语句：在进程中以变量的形式存储计算的中间值。
- 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 语句可以看成是一种隐式的敏感信号表。
- 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。
- 顺序描述语句：包括 IF 语句、CASE 语句、LOOP 语句、NULL 语句等。
- 进程跳出语句：包括 NEXT 语句、EXIT 语句，用于控制进程的运行方向。

(3) 多数 VHDL 综合器要求敏感信号表必须列出本进程中所有输入信号名。

5.4.3 进程要点

从设计者的认识角度看，VHDL 程序与普通软件语言构成的程序有很大的不同，普通软件语言中的语句的执行方式和功能实现十分具体和直观，在编程中几乎可以立即作出判断。但 VHLD 程序，特别是进程结构，设计者要从三个方面去判断它的功能和执行情况：

- 基于 CPU 的纯软件的 VHDL 行为仿真运行方式；
- 基于 VHDL 综合器的综合结果所可能实现的运行方式；

✍✍ 基于最终实现的硬件电路的运行方式。

与其他语句相比，进程语句结构具有更多的特点，对进程的认识和进行进程的设计需要注意以下几方面的问题。

1. PROCESS 为一无限循环语句

在同一结构体中的任一进程是一个独立的无限循环程序结构，但进程中却不必放置诸如软件语言中的返回语句，它的返回是自动的。进程只有两种运行状态，即执行状态和等待状态。进程是否进入执行状态，取决于是否满足特定的条件，如敏感变量是否发生变化。如果满足条件，即进入执行状态，当遇到 END PROCESS 语句后即停止执行，自动返回到起始语句 PROCESS，进入等待状态。

2. PROCESS 中的顺序语句具有明显的顺序/并行运行双重性

严格地说，VHDL 程序不能称之为“程序”，而应称之为 VHDL 代码(CODE)，而其语句的运行也不能称之为“执行”，而应称之为“实现”。因为“程序”和“执行”的概念只能适用于纯软件语言。PROCESS 中的顺序语句的“执行”方式与通常的软件语言中的语句的顺序执行方式有很大的不同。软件语言中每一条语句的执行是按 CPU 的机器周期的节拍顺序执行的，每一条语句执行的时间是确定的，它与 CPU 的主频频率、工作方式、状态周期、机器周期及指令周期的长短紧密相连。但在 PROCESS 中，一个执行状态的运行周期，即从 PROCESS 的启动执行到遇到 END PROCESS 为止所花的时间与任何外部因素都无关(从综合结果来看)，甚至与 PROCESS 语法结构中的顺序语句的多少都没有关系，其执行时间从行为仿真的角度看（如果没有设置任何显式的惯性或传输延时)，只有一个 VHDL 模拟器的最小分辨时间，即一个 ？时间；但从综合和硬件运行的角度看，其执行时间是 0，这与信号的传输延时无关，与被执行的语句的实现时间也无关。即在同一 PROCESS 中，10 条语句和 1000 条语句的执行时间是一样的，显然，从效果上看，PROCESS 中的顺序语句具有并行执行的性质。例如下式的 CASE 语句中：

```
PROCESS(abc)
  BEGIN
   CASE abc IS
    WHEN "0000" => so<="010" ;
    WHEN "0001" => so<="111" ;
    WHEN "0010" => so<="101" ;
    . . .
    WHEN "1110" => so<="100" ;
    WHEN "1111" => so<="000" ;
    WHEN OTHERS => NULL ;
   END CASE;
 END PROCESS;
```

当 abc 发生改变时首先“执行”语句：CASE abc IS，即顺序执行最前面的语句，若此时 abc=“1110”，则立即执行语句 WHEN “1110” => so<=“100”，而不像软件语言那样必须逐条语句进行比较，直到遇到满足条件的语句为止。从仿真执行的角度看，执行一

条 WHEN 语句和执行十条 WHEN 语句的时间是一样的，都是一个? 。这就是为什么顺序语句同样能生成并行结构的组合电路的道理。又如例 5-3 和例 5-4 中的进程语句，前者有明显的并行运行特征，而后者有明显的顺序运行特征，然而它们的运行周期都是一个?。

3. 进程必须由敏感信号的变化来启动

进程的启动必须由敏感信号表中定义的任一敏感信号的变化来启动，否则必须有一个显式的 WAIT 语句来激励。这就是说，进程既可以通过敏感信号的变化来启动，也可以由满足条件的 WAIT 语句而激活；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，就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。

4. 进程语句本身是并行语句

虽然进程语句引导语句属于顺序语句，但同一结构体中的不同进程是并行运行的，或者说是根据相应的敏感信号独立运行的。如例 5-16 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的，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 5-16】

```
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;
```

事实上，任何一条信号的并行赋值语句都是一个简化的进程语句，其输入表式中的各信号都是此“进程语句”的敏感信号。

5. 信号是多个进程间的通信线

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信号来实现的。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。

6. 一个进程中只允许描述对应于一个时钟信号的同步时序逻辑

如前所述，时序电路必须由进程中的顺序语句描述，而此顺序语句必须由不完全的条件语句构成。然而尽管在同一进程中可顺序放置多个条件语句(如 IF 语句)，但是只能放置一个含有时钟边沿检测语句的条件语句。即一个进程中只能描述针对于同一时钟的同步时序逻辑，而异步时序逻辑必须由多个进程来表达。

5.5　并行语句例解

大部分的并行语句的执行和实现是同时发生的，但也有一些语句并非如此，如进程语句和并行赋值语句，前者的行为在以上已作了说明，即除非几个进程都拥有完全相同的敏感信号，否则不可能出现完全并行运行的情况；后者的情况与进程十分类似。

设以下是在同一结构体中的两个赋值语句，尽管它们属于并行语句，但在绝大多数情况下并不可能同时执行，除非 a 或 b，和 c 同时发生一个事件，即它们的值同时发生变化。

```
data1 <=  a AND b ;
data2 <=  c ;
```

并行赋值与顺序赋值都会有一个?延时。以以上第一个语句为例，当 a 或 b 发生一个事件时立即进行 a、b 间的逻辑运算（此运算的仿真时间为 0），同时启动延时为?的模拟定时器，当?时间结束后，data1 才获得（a AND b）的运算值，即在此时 data1 才被更新。

再来考察另一个示例。例 5-17 中有两个赋值语句，它们的赋值目标信号分别是 x 和 select，它们对应的语句被执行的条件是各自赋值符号右边的信号 s0、s1 或 a、b、c、d 发生一个事件。假设在某一稳态条件下 s0=0，s1=0；a、b、c、d 都为 0，这时 x 为 a。

现在假设 a 发生了一个事件，变为 1，这时上面的 select 赋值语句并不会被执行，因为此句并不对 a 敏感。这时只会执行第 2 句赋值语句，因为此句对 a 敏感，这时会将 a 的值 1 赋值给 x，使 x 从原来的 0 更新为 1。

如果这时发生了一个事件，s0 从 0 变为 1。由于第 1 个赋值语句是对 s0 敏感的，所以即被执行。当此并行语句被执行时，所有条件表达式中含有 s0 的都将被更新为新值，并算出新值作为条件判断依据。于是，随着 s0 为 1，s1 为 0，信号 select 被赋予‘1’，这个新值将导致处于第 2 个赋值语句中的 select 发生一个事件，于是这条语句将使用 select 的新值去选择 x 的赋值，即将输入端口 b 的值赋给 x，这时 x 从 0 变为 1。

【例 5-17】

```
ARCHITECTURE dataflow OF mux IS
  SIGNAL seiect : INTEGER RANGE 15 DOWNTO 0;
BEGIN
   Select <= 0 WHEN s0='0' AND s1='0' ELSE
             1 WHEN s0='1' AND s1='0' ELSE
             2 WHEN s0='0' AND s1='1' ELSE
             3 ;
```

```
    x <= a WHEN select=0  ELSE
         b WHEN select=1  ELSE
         c WHEN select=2  ELSE
         d ;
. . .
```

5.6 仿真延时

延时是 VHDL 仿真需要的重要特性设置，为设计建立精确的延时模型，可以使用 VHDL 仿真器得到接近实际的精确结果。在 FPGA/CPLD 设计过程中，源文件一般不需要建立延时模型，因为 EDA 软件可以使用门级仿真器对选定的 FPGA 或 CPLD 适配所得的时序仿真文件进行精确仿真。本节讨论的问题主要是针对行为仿真延时建模的。VHDL 中有两类延时模型能用于行为仿真建模，即固有延时和传输延时。本节介绍对 VHDL 进行行为仿真时必须考虑的信号延时情况，同时对前面多次出现的?延时的概念作进一步的解释。

5.6.1 固有延时（Inertial Delay）

固有延时也称为惯性延时，是任何电子器件都存在的一种延时特性。固有延时的主要物理机制是分布电容效应。分布电容产生的因素很多，分布电容具有吸收脉冲能量的效应。当输入器件的信号脉冲宽度小于器件输入端的分布电容对应的时间常数时，或者说小于器件的惯性延时宽度时，即使脉冲有足够高的电平，也无法突破数字器件的阈值电平实现信号输出的目的，从而在输出端不会产生任何变化。这就类似于用一外力推动一静止物体时，无论瞬间外力有多大，如果它持续的时间过短，仍将无法克服物体的静止惯性而将其推动。

不难理解，在惯性延时模型中，器件的输出都有一个固有的延时。当信号的脉宽(或者说信号的持续时间)小于器件的固有延时时，器件将对输入的信号不作任何反应，即有输入而无输出。为了使器件对输入信号的变化产生响应，就必须使信号维持的时间足够长，即信号的脉冲宽度必须大于器件的固有延时。

在 VHDL 仿真和综合器中，有一个默认的固有延时量，它在数学上是一个无穷小量，被称为 ？延时，或称仿真 ？。这个延时小量的设置仅为了仿真，它是 VHDL 仿真器的最小分辨时间，并不能完全代表器件实际的惯性延时情况。在 VHDL 程序的语句中如果没有指明延时的类型与延时量，就意味着默认采用了这个固有延时量 ？延时。在大多数情况下，这一固有延时量近似地反映了实际器件的行为。

在所有当前可用的仿真器中，固有模式是最通用的一种，为了在行为仿真中比较逼真地模仿电路的这种延时特性，如，VHDL 中含有固有延时模型的赋值语句可由下式表达，

```
z <= x XOR y AFTER 5ns ;
```

表示此赋值电路的惯性延时为 5ns，即要求信号值 x XOR y变化的稳定时间不能少于 5ns，换句话说，x XOR y的值在发生变化 5ns 后才被赋给 z，此前 x 或 y 的任何变化都是

无效的。对于下句：

```
z <= x XOR y ；
```

则表明，x XOR y 的值在 ？ 时间段后才被赋给 z 。

下式表示一赋值延时模型的固有延时是 20ns，即表明，当 A 有一个事件后，其导致这一事件的电平的维持时间必须大于 20ns，在固有延时定时器被启动 20ns 后，B 端口才有可能输出信号（获得更新）。

```
B <=  A AFTER 20ns ；--固有延时模型
```

如果 A 在稳态时是'0',有事件发生后，A 由'0'变为'1'，并假设 A 的维持时间是 10ns，则以上的固有延时模型的信号波形如图 5-15 所示。按照 20ns 延时量，输出信号 B 的上升沿应该出现在 30ns 处，而其下降沿出现在 40ns 处。但实际上，B 的输出信号没有任何变化，显然，正是由于在固有延时模型的赋值语句中，A 的维持时间小于 20ns 造成的。

5.6.2　传输延时（Transport Delay）

另一种延时模型是传输模型。传输延时与固有延时相比，其不同之处在于传输延时表达的是输入与输出之间的一种绝对延时关系。传输延时并不考虑信号持续的时间，它仅仅表示信号传输推迟或延迟了一个时间段，这个时间段即为传输延时。对于器件来说，传输延时是由半导体延时特性决定的；对于连线来说，传输延时是由连线的介质结构、传输阻抗特性和信号频率特性决定的。传输延时对延时器件、PCB 板上的连线延时和 ASIC 上的通道延时的建模特别有用。表达传输延时的语句可如以下例句所示：

```
B <= TRANSPORT A AFTER 20 ns;-- 传输延时模型
```

其中关键词 TRANSPORT 表示语句后的延时量为传输延时量。

对于此模型，仍然设 A 的持续时间为 10ns，其输入输出波形如图 5-16 所示。图中，与 A 的输入波形相比，B 的输出波形正好延时了 20ns，尽管 A 的脉宽只有 10ns，但在 30 和 40ns 处仍然能看到对应的 B 的输出信号。

应该注意，虽然产生传输延时与固有延时的物理机制不一样，但在行为仿真中，传输延时与固有延时造成的延时效应是一样的。在综合过程中，综合器将忽略 AFTER 后的所有延时设置。

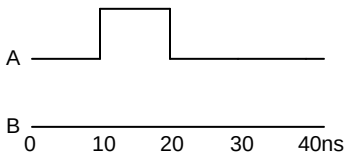

图 5-15 固有延时输入输出波形

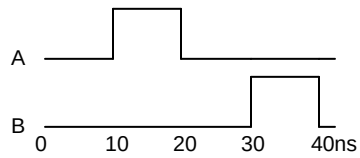

图 5-16 传输延时输入输出波形

5.6.3　仿真 ？（Simulation Delta）

前面曾提到过功能仿真的概念，由于综合器不支持延时语句，在综合后的功能仿真中，仿真器仅对设计的逻辑行为进行了模拟测定，而没有把器件的延时特性考虑进去，仿真器给出的结果也仅仅是逻辑功能。按理说，功能仿真就是假设器件间的延迟时间为零的仿真。然而事实并非如此，由于无论是行为仿真还是功能仿真，都是利用计算机进行软件仿真，即使在并行语句的仿真执行上也是有先后的，在零延时条件下，当作为敏感量的输入信号发生变化时，并行语句执行的先后次序无法确定，而不同的执行次序会得出不同的非唯一的仿真结果，最后将导致矛盾的和错误的仿真结果。这种错误仿真的根本原因在于零延时假设在客观世界中是不可能存在的。

为了解决这一矛盾，VHDL 仿真器和综合器将自动为系统中的信号赋值配置一足够小而又能满足逻辑排序的延时量，即仿真软件的最小分辨时间，这个延时量就称为仿真 ？，或称 ？ 延时，从而使并行语句和顺序语句中的并列赋值逻辑得以正确执行。

由此可见，在行为仿真、功能仿真乃至综合中，引入 ？ 延时是必需的。仿真中，？ 延时的引入由 EDA 工具自动完成，无需设计者介入。

习 题

5-1. 什么是固有延时？什么是惯性延时？

5-2. ？ 是什么？在 VHDL 中，？ 有什么用处？

5-3. 哪些情况下需要用到程序包 STD_LOGIC_UNSIGNED？试举一例。

5-4. 说明信号和变量的功能特点，应用上的异同点。

5-5. 在 VHDL 设计中，给时序电路清 0(复位)有两种方法，它们是什么？

5-6. 哪一种复位方法必须将复位信号放在敏感信号表中？给出这两种电路的 VHDL 描述。

5-7. 什么是重载函数？重载算符有何用处？如何调用重载算符函数？

5-8. 判断下面 3 个程序中是否有错误，若有则指出错误所在，并给出完整程序。

程序 1：

```
Signal A, EN : std_logic;
Process (A, EN)
   Variable B : std_logic;
Begin
if EN = 1 then   B <= A;  end if;
end process;
```

程序 2：

```
Architecture one of sample is
    variable a, b, c : integer;
begin
    c <= a + b;
end;
```

程序 3：

```
library ieee;
```

```
use ieee.std_logic_1164.all;
entity mux21 is
    port ( a, b : in std_logic; sel : in std_logic; c : out std_logic;);
end sam2;
architecture one of mux21 is
begin
if sel = '0' then   c := a; else    c := b;  end if;
end two;
```

5-9. **根据例** 3-23 **设计** 8 **位左移移位寄存器，给出时序仿真波形。**

5-11. **将例** 5-12 **中的** 4 **个** IF **语句分别用** 4 **个并列进程语句表达出来。**

实 验 与 设 计

5-1. 7 **段数码显示译码器设计**

(1) **实验目的：学习** 7 **段数码显示译码器设计；学习** VHDL **的** CASE **语句应用及多层次设计方法。**

(2) **实验原理：**7 **段数码是纯组合电路，通常的小规模专用** IC**，如** 74 **或** 4000 **系列的器件只能作十进制** BCD **码译码，然而数字系统中的数据处理和运算都是** 2 **进制的，所以输出表达都是** 16 **进制的，为了满足** 16 **进制数的译码显示，最方便的方法就是利用译码程序在** FPGA/CPLD **中来实现。例** 5-18 **作为** 7 **段译码器，输出信号** LED7S **的** 7 **位分别接如图** 5-18 **数码管的** 7 **个段，高位在左，低位在右。例如当** LED7S **输出为“** 1101101 **”时，数码管的** 7 **个段：**g、f、e、d、c、b、a **分别接** 1、1、0、1、1、0、1**；接有高电平的段发亮，于是数码管显示“** 5 **”。注意，这里没有考虑表示小数点的发光管，如果要考虑，需要增加段** h**，例** 5-18 **中的** LED7S:OUT STD_LOGIC_VECTOR(6 DOWNTO 0)**应改为** ... (7 DOWNTO 0) **。**

(3) **实验内容** 1**：说明例** 5-18 **中各语句的含义，以及该例的整体功能。在** QuartusII **上对该例进行编辑、编译、综合、适配、仿真，给出其所有信号的时序仿真波形。**

提示：用输入总线的方式给出输入信号仿真数据，仿真波形示例图如图 5-17 **所示。**

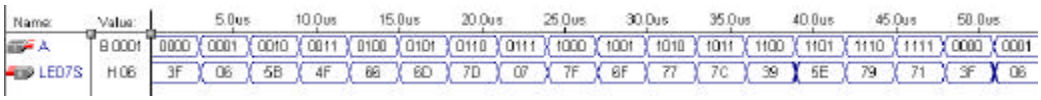

图 5-17　7**段译码器仿真波形**

【例 5-18**】**

```
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 ;
ARCHITECTURE one OF DECL7S IS
BEGIN
 PROCESS( A )
 BEGIN
```

```
CASE  A  IS
 WHEN "0000" =>  LED7S <= "0111111" ;
 WHEN "0001" =>  LED7S <= "0000110" ;
 WHEN "0010" =>  LED7S <= "1011011" ;
 WHEN "0011" =>  LED7S <= "1001111" ;
 WHEN "0100" =>  LED7S <= "1100110" ;
 WHEN "0101" =>  LED7S <= "1101101" ;
 WHEN "0110" =>  LED7S <= "1111101" ;
 WHEN "0111" =>  LED7S <= "0000111" ;
 WHEN "1000" =>  LED7S <= "1111111" ;
 WHEN "1001" =>  LED7S <= "1101111" ;
 WHEN "1010" =>  LED7S <= "1110111" ;
 WHEN "1011" =>  LED7S <= "1111100" ;
 WHEN "1100" =>  LED7S <= "0111001" ;
 WHEN "1101" =>  LED7S <= "1011110" ;
 WHEN "1110" =>  LED7S <= "1111001" ;
 WHEN "1111" =>  LED7S <= "1110001" ;
 WHEN OTHERS =>  NULL ;
 END CASE ;
END PROCESS ;
END ;
```

(4) **实验内容** 2：**引脚锁定及硬件测试。建议选** GW48 **系统的实验电路模式** 6，**用数码** 8 **显示译码输出**(PIO46-PIO40)，**键** 8、**键** 7、**键** 6 **和键** 5 **四位控制输入，硬件验证译码器的工作性能。**

(5) **实验内容** 3：**用第** 3 **章介绍的例化语句，按图** 5-19 **的方式连接成顶层设计电路（用** VHDL **表述），图中的** CNT4B **是一个** 4 **位二进制加法计数器，可以由例** 3-22 **修改获得；模块** DECL7S **即为例** 5-18 **实体元件，重复以上实验过程。注意图** 5-20 **中的** tmp **是** 4 **位总线，**led **是** 7 **位总线。对于引脚锁定和实验，建议选电路模式** 6，**用数码** 8 **显示译码输出，用键** 3 **作为时钟输入(每按** 2 **次键为** 1 **个时钟脉冲)，或直接接时钟信号** clock0。

(8) **实验报告：根据以上的实验内容写出实验报告，包括程序设计、软件编译、仿真分析、硬件测试和实验过程；设计程序、程序分析报告、仿真波形图及其分析报告。**

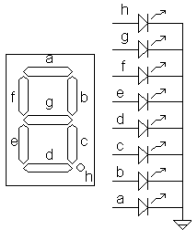

图 5-18 **共阴数码管及其电路**

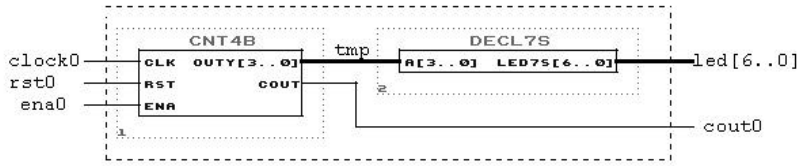

图 5-19 **计数器和译码器连接电路的顶层文件原理图**

5-2. 8 **位数码扫描显示电路设计**

(1) **实验目的：学习硬件扫描显示电路的设计。**

(2) **实验原理：图** 5-20 **所示的是** 8 **位数码扫描显示电路，其中每个数码管的** 8 **个段：**h、g、f、e、d、c、b、a（h **是小数点）都分别连在一起，**8 **个数码管分别由** 8 **个选通信号** k1、k2、…k8 **来选择。被选通的数码管显示数据，其余关闭。如在某一时刻，**k3 **为高电平，其余选通信号为低电平，这时仅** k3 **对应的数码管显示来自段信号端的数据，而其它** 7 **个数码管呈现关闭状态。根据这种电路状况，如果希望在** 8 **个**

数码管显示希望的数据，就必须使得 8 个选通信号 k1、k2、…k8 分别被单独选通，并在此同时，在段信号输入口加上希望在该对应数码管上显示的数据，于是随着选通信号的扫变，就能实现扫描显示的目的。

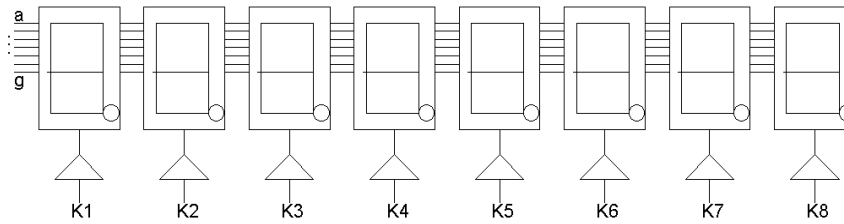

图 5-20　8位数码扫描显示电路

例 5-19 是扫描显示的示例程序，其中 clk 是扫描时钟；SG 为 7 段控制信号，由高位至低位分别接 g、f、e、d、c、b、a 7 个段；BT 是位选控制信号，接图 5-20 中的 8 个选通信号：k1、k2、…k8 。程序中 CNT8 是一个 3 位计数器，作扫描计数信号，由进程 P2 生成；进程 P3 是 7 段译码查表输出程序，与例 5-18 相同；进程 P1 是对 8 个数码管选通的扫描程序，例如当 CNT8 等于"001" 时，K2 对应的数码管被选通，同时，A 被赋值 3，再由进程 P3 译码输出"1001111"，显示在数码管上即为“ 3 ”；当 CNT8 扫变时，将能在 8 个数码管上显示数据：13579BDF 。

【例 5-19】

```
LIBRARY IEEE;
 USE IEEE.STD_LOGIC_1164.ALL;
 USE IEEE.STD_LOGIC_UNSIGNED.ALL;
 ENTITY SCAN_LED IS
     PORT (   CLK   : IN STD_LOGIC;
              SG    : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);  --段控制信号输出
              BT    : OUT STD_LOGIC_VECTOR(7 DOWNTO 0) );--位控制信号输出
  END;
  ARCHITECTURE one OF SCAN_LED IS
    SIGNAL CNT8  : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL    A  : INTEGER RANGE 0 TO 15;
  BEGIN
  P1：PROCESS( CNT8 )
    BEGIN
       CASE  CNT8  IS
          WHEN "000" =>  BT <= "00000001" ; A <= 1 ;
          WHEN "001" =>  BT <= "00000010" ; A <= 3 ;
          WHEN "010" =>  BT <= "00000100" ; A <= 5 ;
          WHEN "011" =>  BT <= "00001000" ; A <= 7 ;
         WHEN "100" =>  BT <= "00010000" ; A <= 9 ;
          WHEN "101" =>  BT <= "00100000" ; A <= 11 ;
          WHEN "110" =>  BT <= "01000000" ; A <= 13 ;
          WHEN "111" =>  BT <= "10000000" ; A <= 15 ;
          WHEN OTHERS =>  NULL ;
       END CASE ;
     END PROCESS P1;
    P2：PROCESS(CLK)
       BEGIN
         IF CLK'EVENT AND CLK = '1' THEN CNT8 <= CNT8 + 1;
        END IF;
       END PROCESS P2 ;
```

```
P3：PROCESS( A ) --译码电路
 BEGIN
  CASE  A  IS
   WHEN 0  => SG <= "0111111";  WHEN 1  => SG <= "0000110";
   WHEN 2  => SG <= "1011011";  WHEN 3  => SG <= "1001111";
   WHEN 4  => SG <= "1100110";  WHEN 5  => SG <= "1101101";
   WHEN 6  => SG <= "1111101";  WHEN 7  => SG <= "0000111";
   WHEN 8  => SG <= "1111111";  WHEN 9  => SG <= "1101111";
   WHEN 10 => SG <= "1110111";  WHEN 11 => SG <= "1111100";
   WHEN 12 => SG <= "0111001";  WHEN 13 => SG <= "1011110";
   WHEN 14 => SG <= "1111001";  WHEN 15 => SG <= "1110001";
   WHEN OTHERS =>  NULL ;
  END CASE ;
 END PROCESS P3;
 END;
```

(3) **实验内容** 1：**说明例** 5-19 **中各语句的含义，以及该例的整体功能。对该例进行编辑、编译、综合、适配、仿真，给出仿真波形。实验方式：若考虑小数点，**SG **的** 8 **个段分别与** PIO49、PIO48、…、PIO42**（高位在左）、**BT **的** 8 **个位分别与** PIO34、PIO35、…、PIO41**（高位在左）；电路模式不限，将** GW48EDA **系统左下方的拨码开关全部向上拨，这时实验系统的** 8 **个数码管构成图** 5-20 **的电路结构，时钟** CLK **可选择** clock0**，通过跳线选择** 16384Hz **信号。引脚锁定后进行编译、下载和硬件测试实验。将实验过程和实验结果写进实验报告。**

(4) **实验内容** 2：**修改例** 5-19 **的进程** P1 **中的显示数据直接给出的方式，增加** 8 **个** 4 **位锁存器，作为显示数据缓冲器，使得所有** 8 **个显示数据都必须来自缓冲器。缓冲器中的数据可以通过不同方式锁入，如来自** A/D **采样的数据、来自分时锁入的数据、来自串行方式输入的数据，或来自单片机等。**

5-3. **数控分频器的设计**

(1) **实验目的：学习数控分频器的设计、分析和测试方法。**

(2) **实验原理：数控分频器的功能就是当在输入端给定不同输入数据时，将对输入的时钟信号有不同的分频比，数控分频器就是用计数值可并行预置的加法计数器设计完成的，方法是将计数溢出位与预置数加载输入信号相接即可，详细设计程序如例** 5-20 **所示。**

(3) **分析：根据图** 5-21 **的波形提示，分析例** 5-20 **中的各语句功能、设计原理及逻辑功能，详述进程** P_REG **和** P_DIV **的作用，并画出该程序的** RTL **电路图。**

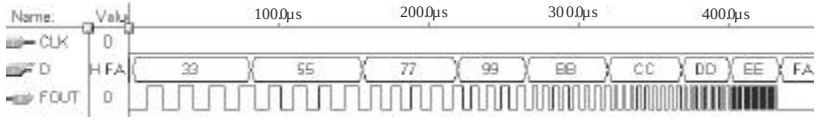

图 5-21 **当给出不同输入值** D **时，**FOUT **输出不同频率**(CLK **周期**=50ns)

(4) **仿真：输入不同的** CLK **频率和预置值** D**，给出如图** 5-21 **的时序波形。**

(5) **实验内容** 1：**在实验系统上硬件验证例** 5-20 **的功能。可选实验电路模式** 1**；键** 2/**键** 1 **负责输入** 8 **位预置数** D(PIO7-PIO0)**；**CLK **由** clock0 **输入，频率选** 65536Hz **或更高(确保分频后落在音频范围)；输出** FOUT **接扬声器**(SPKER)**。编译下载后进行硬件测试：改变键** 2/**键** 1 **的输入值，可听到不同音调的声音。**

(6) **实验内容** 2：**将例** 5-20 **扩展成** 16 **位分频器，并提出此项设计的实用示例，如** PWM **的设计等。**

(7) **思考题：怎样利用** 2 **个例** 5-20 **给出的模块设计一个电路，使其输出方波的正负脉宽的宽度分别**

由可两个 8 位输入数据控制？

(8) **实验报告：根据以上的要求，将实验项目分析设计，仿真和测试写入实验报告。**

【例 5-20】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY DVF IS
    PORT (   CLK  : IN STD_LOGIC;
              D  : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
            FOUT : OUT STD_LOGIC  );
END;
ARCHITECTURE one OF DVF IS
    SIGNAL   FULL : STD_LOGIC;
BEGIN
  P_REG: PROCESS(CLK)
   VARIABLE CNT8 : STD_LOGIC_VECTOR(7 DOWNTO 0);
   BEGIN
      IF CLK'EVENT AND CLK = '1' THEN
           IF CNT8 = "11111111" THEN
           CNT8 := D;      --当CNT8计数计满时，输入数据D被同步预置给计数器CNT8
             FULL <= '1'; --同时使溢出标志信号FULL输出为高电平
               ELSE   CNT8 := CNT8 + 1;  --否则继续作加1计数
                      FULL <= '0';       --且输出溢出标志信号FULL为低电平
           END IF;
      END IF;
    END PROCESS P_REG ;
   P_DIV: PROCESS(FULL)
     VARIABLE CNT2 : STD_LOGIC;
   BEGIN
   IF FULL'EVENT AND FULL = '1' THEN
     CNT2 := NOT CNT2; --如果溢出标志信号FULL为高电平，D触发器输出取反
         IF CNT2 = '1' THEN  FOUT <= '1'; ELSE FOUT <= '0';
        END IF;
   END IF;
    END PROCESS P_DIV ;
END;
```

5-4. 32 位并进/并出移位寄存器设计

仅用例 5-8 **一个** 8 **位移位寄存器，再增加一些电路，如** 4 **个** 8 **位锁存器等，设计成为一个能为** 32 **位二进制数进行不同方式移位的移位寄存器。这个电路模型十分容易用到** CPU **的设计中。**