

3.1 VHDL 基本术语

首先介绍一些常见的 VHDL 术语,它们是几乎在每一个 VHDL 程序中都会使用的基本 VHDL 构建块。

1. 实体 ENTITY

实体是设计中最基本的构建块,所有的设计都会有实体。整个设计中位于最顶层的实体叫顶层实体(Top-Level Entity)。如果设计是分层递阶的,则顶层实体描述会包含更低层的描述。这些更低层描述就是包含在顶层描述中的更低层实体。

2. 结构体 ARCHITECTURE

所有能仿真的实体都具有结构体描述。结构体描述实体的行为。一个实体可能具有多个结构体。一个设计的结构体可以从行为角度描述,也可以从结构角度描述。

3. 配置 CONFIGURATION

配置语句用于将一个实体结构体对和一个组件实例绑定起来。配置可以被当作设计中的元件清单表。它描述的是对于每个实体,使用的是哪个行为,更像一个元件清单表,用来描述设计中每个器件具体使用的哪个元件。

4. 包 PACKAGE

包中包含了设计中公用的数据类型和子程序的集合。可以把包想象成包含构建设计所需要工具的一个工具箱。

5. 驱动器 DRIVER

驱动器就是信号的驱动源,如果一个信号被两个源所驱动,那么当两个驱动源都有效时,则这个信号具有两个驱动源。

6. 总线 BUS

提到总线,通常会想到是一组信号或者是硬件设计中所使用的一种特定通信方法。在 VHDL 中,总线是一类特殊类型的信号,它能够将其驱动源关掉,即呈现高阻态。

7. 属性 ATTRIBUTE

属性是附属于 VHDL 对象的数据。比如,缓冲器的并行驱动能力,器件的最大工作温度。

8. 类属说明 GENERIC

类属参数是用于表示向实体传递信息的参数。比如,如果一个实体是具有上升延时和下降延时的门级模型,则上升延时和下降延时的具体数值可以使用 GENERIC 参数向实体传送。

9. 进程 PROCESS

进程是 VHDL 中基本的执行单元。VHDL 描述的仿真中,所有的操作都可以被分解成一个或多个进程。

10. 库 LIBRARY

库是经编译后的数据集合。它存放包括集合定义、实体定义、结构体定义和配置定义。库的功能类似于操作系统中的目录,库中存放设计的数据。库通常放在设计的最前面,意味着库中的内容可以供设计者共享。

3.2 VHDL 的三种不同描述风格

VHDL 包括主级设计单元和次级设计单元。其中主级设计单元包括实体 ENTITY 和包 PACKAGE,次级设计单元是结构体 ARCHITECTURE 和包体 PACKAGE BODY。次级设计单元通常是和主级设计单元相关的。库则是主级设计单元和次级设计单元的集合。

3.2.1 实体

以四选一的数据选择器,其逻辑符号描述如图 3-1 所示,其中 a、b、c、d 是四个数据输入端,s1、s0 是地址选择端,x 是输出端。

其所对应的实体描述如下:

```
ENTITY mux IS
    PORT ( a, b, c, d : IN BIT;
          s0, s1 : IN BIT;
          x : OUT BIT);
END mux;
```

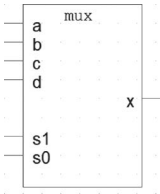


图 3-1 四选一数据选择器符号

其中 VHDL 预留的关键词都用大写表示: 关键词 ENTITY 及 IS 标明是实体描述开始; END 标明实体描述结束; PORT 表示开始描述端口信号信息; IN 表示端口信号的方向为输入信号; OUT 表示端口信号的方向为输出信号; BIT 表示端口信号的信号类型为位类型(0、1 两种取值)。实体描述中小写的 mux 是设计者定义的实体名称,a、b、c、d、s0、s1、x 是设计者定义的端口信号名称。

实体的端口方向(也称端口模式)包括四类,如表 3-1 所示。

表 3-1 实体端口模式

端 口 模 式	说 明
IN	数据只能从端口流入实体
OUT	数据只能从实体流出端口
INOUT	双向,数据可以从端口流入/流出实体
BUFFER	数据从端口流出实体,同时被内部反馈

由此可见,实体描述包括实体名称和器件端口情况,包括哪些端口信号,信号的方向以及信号类型。总之,实体描述的就是器件的逻辑符号,即器件封装起来对外呈现的样子。上

面的实体描述是所有复杂设计的基础框架,在此基础上,实体还可以包含更多的信息。比如类属说明语句 GENERIC,它通常用来定义端口的总线宽度,实体中子元件的数目,以及实体参数等静态信息。它本质上就是定义一个常数,并且将其传递到实体描述内部。

比如,

```
GENERIC(w: INTEGER:= 16);
PORT(abus: OUT BIT_VECTOR(w-1 DOWNT0 0));
```

上面两句就相当于

```
PORT(abus: OUT BIT_VECTOR(15 DOWNT0 0));
```

语法说明: 这个端口定义了一个端口信号 abus,信号方向为输出,信号类型为 BIT_VECTOR。BIT_VECTOR 是位矢量,可以理解成是一维数组,数组的每一位都是 BIT 类型,数组元素个数和元素排序则由括号里内容决定:(15 DOWNT0 0)表示数组元素个数为 16 个,元素排序从 15 到 0;而如果是 BIT_VECTOR(0 to 15),则仅仅是元素排序从 0 到 15。

3.2.2 结构体

结构体则描述了实体的基本功能,包含了建模实体行为的具体描述。基于 VHDL 描述数字系统,有三种不同的描述风格:(以下三种不同风格的结构体描述均基于前面 4 选 1 数据选择器的实体描述)。

1. 数据流描述风格(也称作并行语句描述风格)

```
ARCHITECTURE dataflow OF mux IS
    SIGNAL select : INTEGER;
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 AFTER 0.5 NS WHEN select = 0 ELSE
        b AFTER 0.5 NS WHEN select = 1 ELSE
        c AFTER 0.5 NS WHEN select = 2 ELSE
        d AFTER 0.5 NS;
END dataflow;
```

其中关键词 ARCHITECTURE 标明结构体描述的开始,该结构体的名字为 dataflow,其所匹配的实体为 mux。关键词 ARCHITECTURE 和 BEGIN 之间是结构体中所使用的信号声明和组件声明。在上例中,声明了一个 select 信号,并且其信号类型为 INTEGER。需要强调:VHDL 是强数据类型的语言,不管常量、信号、还是变量,声明都要说清楚数据类型。在 ARCHITECTURE 后面第一个 BEGIN 之后一直到关键词 END 之间是并行语句,即所有语句是并行执行,和语句书写的先后顺序无关,也就说这个区段的语句可以互相颠倒顺序,而不影响设计的行为功能。

条件信号赋值

```
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;
```

符号“<=”代表信号赋值。这里是给信号 select 进行赋值,具体取值可能为 0、1、2、3 四种。当信号 s0、s1 取 00 时,select 信号为 0;当 s0、s1 取 10 时,select 信号为 1;当 s0、s1 取 01 时,select 信号为 2;其他情况,select 信号为 3(关于赋值语句的更多内容请参见信号赋值语句部分)。

接下来对信号 x 的赋值也是类似的理解方法,根据 select 信号的取值情况给信号赋值,不同之处在于信号 x 的赋值并不是立即发生的,而是需要往后延时 0.5ns 才发生。

这两条信号赋值语句是并行语句,也就是二者是被同时执行的,其书写顺序可以颠倒并不影响功能。但是信号赋值是否被激活,取决于赋值符号右侧的信号,如果右侧的信号至少有一个有事件,则信号赋值被激活。所谓事件就是信号从 0 到 1 或者从 1 到 0 的一次变化。也就是说赋值符号右侧的信号中,至少有一个信号有变化,信号赋值才真正被激活,否则赋值语句挂起。

结构体中的语句是并行语句,究其本质是因为它描述的数字电路中,各个节点的逻辑值向后传递是同时发生,与节点所在的位置无关。而节点则对应 VHDL 中的信号,节点的这种并行传输信号特性,也就决定了结构体中多条信号赋值语句必然也是并行执行的。进而推广结论:结构体中的语句都是并行语句。

2. 结构化描述风格(元件例化,也称作图形网表设计)

结构化描述是从网表的角度,描述各元件的引脚连接。上例中,在结构体的 BEGIN 之前,声明了待设计的数据选择器使用三个元件(COMPONENT),分别为 andgate、inverter、orgate。而这三个元件应该都有属于自己的 VHDL 完整描述,也即有以 andgate、inverter、orgate 命名的实体和实体对应的结构体。

```
ARCHITECTURE netlist OF mux IS
    COMPONENT andgate
        PORT(a, b, c : IN BIT; x : OUT BIT);
    END COMPONENT;
    COMPONENT inverter
        PORT(in1 : IN BIT; x : OUT BIT);
    END COMPONENT;
    COMPONENT orgate
        PORT(a, b, c, d : IN BIT; x : OUT BIT);
    END COMPONENT;
    SIGNAL s0_inv, s1_inv, x1, x2, x3, x4 : BIT;
    BEGIN
        U1 : inverter PORT MAP(s0, s0_inv);
        U2 : inverter PORT MAP(s1, s1_inv);
        U3 : andgate PORT MAP(a, s0_inv, s1_inv, x1);
        U4 : andgate PORT MAP(b, s0, s1_inv, x2);
        U5 : andgate PORT MAP(c, s0_inv, s1, x3);
        U6 : andgate PORT MAP(d, s0, s1, x4);
        U7 : orgate PORT MAP(b => x2, a => x1, d => x4, c => x3, x => x);
    END netlist;
```

元件声明语句的格式如下:

```
COMPONENT 元件名 IS
    PORT(端口名表)
END COMPONENT;
```

在数据选择器的顶层描述中,这三个元件的调用和连接关系是用元件例化语句实现的。元件例化语句的格式如下:

例化名: 元件名 PORT MAP ([端口名 =>] 连接端口名, ...);

在本例中,比如 U1 : inverter(s0,s0_inv); 表示调用元件模型 inverter,在 mux 的顶层设计中被称作 U1,inverter 的 in1 在顶层中连接的是 s0 信号,x 在顶层中连接的是 s0_inv 信号。这种按照元件端口名顺序依次匹配关联顶层设计中的连接端口名的方式称作位置映射法,上例中 U1-U6 都是采用这种关联方式。此时,端口的位位置必须严格与元件声明中的完全相同,不可调整; U7 则是采用端口名映射法,即明确这里的符号“=>”仅代表连接映射关系,不代表信号流动的方向,即不限制信号的流动方向。符号“=>”左边是元件的内部端口名,右边则是元件外部需要连接的端口名或信号名。此时直接描述的是端口名的关联关系,因此端口名的描述位置是可以调整的。比如,上例中 U7,先描述的是端口 b 的关联关系,后描述的是端口 a 的关联关系。这与元件声明中的端口顺序不同。上面结构化描述对应的网表连接图如图 3-2 所示。

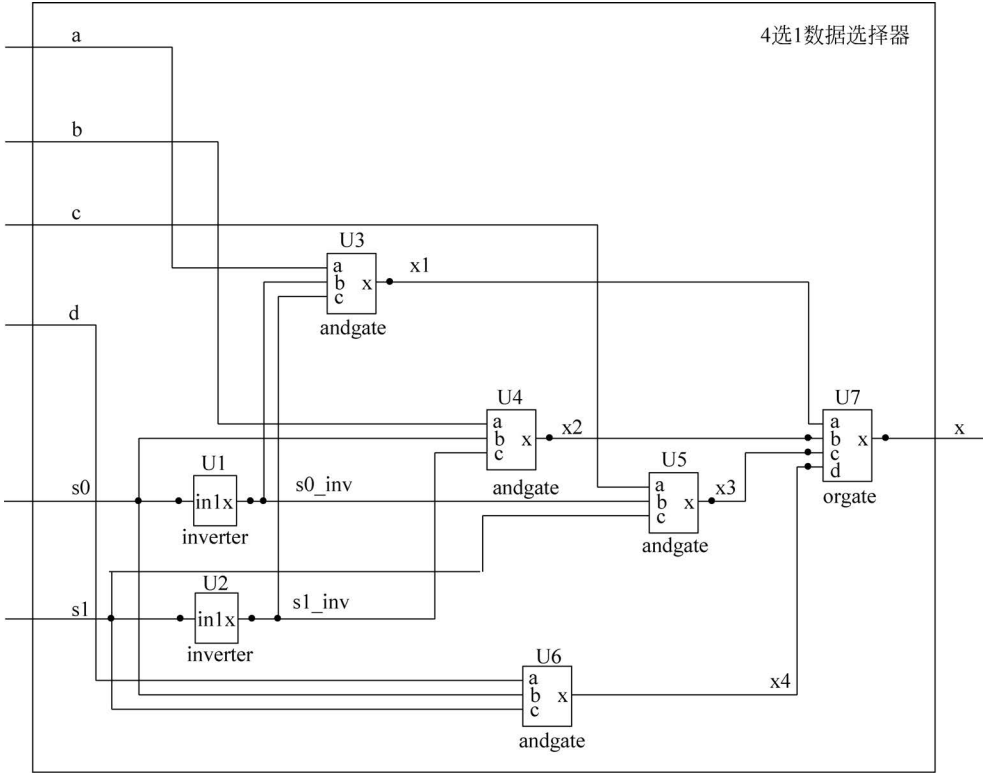


图 3-2 4 选 1 数据选择器原理结构图

3. 行为描述风格(也称作时序描述风格,要使用 PROCESS 语句)

```
ARCHITECTURE sequential OF mux IS
  BEGIN
    PROCESS(a, b, c, d, s0, s1 )
      VARIABLE sel : INTEGER;
      BEGIN
        IF s0 = '0' and s1 = '0' THEN
```

```

        sel := 0;
    ELSIF s0 = '1' AND s1 = '0' THEN
        sel := 1;
    ELSIF s0 = '0' AND s1 = '0' THEN
        sel := 2;
    ELSE
        sel := 3;
    END IF;
    CASE sel IS
        WHEN 0 =>
            x <= a;
        WHEN 1 =>
            x <= b;
        WHEN 2 =>
            x <= c;
        WHEN OTHERS =>
            x <= d;
    END CASE;
END PROCESS;
END sequential;

```

行为描述风格采用 PROCESS 语句去描述数字选择器的行为功能。进程 PROCESS 关键词后面括号内是进程的敏感信号表,只有当敏感信号表中至少一个信号有事件,即信号有从 0 到 1 或者从 1 到 0 的变化,进程才被执行,称作进程被“激活”;反之,如果敏感信号表中的信号都没有事件,进程不会被执行,称作进程被“挂起”。进程 PROCESS 关键词到 BEGIN 之间是声明区,可以声明进程中使用的变量,此例中声明了一个 sel 变量。变量在进程内使用,出了进程,则变量不再存在。进程下面的 BEGIN 标识到 END PROCESS 之间是进程的详细描述,这之间的语句是顺序执行语句,即按照语句出现的先后顺序执行。本例中使用了 IF 结构语句和 CASE 语句。CASE 语句根据 sel 变量的值选择不同的分支执行。比如变量 sel 为 1,则 CASE 语句选择执行 $x \leq b$ 。

3.2.3 配置

前面针对同一个 4 选 1 数据选择器实体,有三种不同描述风格的结构体。使用配置语句,设计者可以选择究竟使用哪个结构体。另外,配置语句也可以在结构体中选择不同的元件。

```

CONFIGURATION muxcon1 OF mux IS
    FOR netlist
        FOR U1,U2 :
            inverter USE ENTITY WORK.myinv(version1);
        END FOR;
        FOR U3,U4,U5,U6 : andgate USE ENTITY WORK.myand(version1);
        END FOR;
        FOR U7 : orgate USE ENTITY WORK.myor(version1);
        END FOR;
    END FOR;
END muxcon1;

```

前面的配置描述具体含义如下:这是实体 mux 的配置方案,称作 muxcon1。对于最顶层实体 mux,使用结构体 netlist;而对于结构体 netlist 中被例化的两个 inverter 类型 U1、

U2,使用 WORK 库中的实体为 myinv,结构体为 version1 的设计描述;对于结构体 netlist 中被例化的四个 andgate 类型 U3、U4、U5、U6,使用 WORK 库中的实体为 myand,结构体为 version1 的设计描述;对于结构体 netlist 中被例化的 orgate 类型 U7,使用 WORK 库中的实体为 myor,结构体为 version1 的设计描述。如果需要更换实体 mux 的结构体,则只需要更换配置,不需要重新进行结构体描述。下面配置为实体 mux 的第二个配置方案:对于最顶层实体 mux,使用结构体 dataflow。

```
CONFIGURATION muxcon2 OF mux IS
    FOR dataflow
        END FOR;
END muxcon2;
```

由此可见,利用配置语句,可以非常方便地实现在设计的各个级别上(包括结构体和其内部元件等不同级别)进行灵活选择不同风格的设计描述。

3.3 数据对象

数据对象是 VHDL 程序设计中进行各种运算与操作的对象。VHDL 使用的数据对象主要有四种:常量、变量、信号和文件。

3.3.1 常量

常量(CONSTANT)是用来存储指定数据类型的数值。虽然常量与数值在程序的功能上没有什么差别,但它能有效提高 VHDL 的可读性与安全性。通俗地讲,常量就是有了名字的固定数值,常量在声明时需要指定相应的数据类型与初始值。常量的数据类型可以是标量或复合类型,但不能是文件类型或存取类型。常量的声明格式如下:

CONSTANT 常量名: 常量的数据类型: = 值

例如:

```
CONSTANT PI: REAL: = 3.14;
CONSTANT bus_width: INTEGER: = 16;
CONSTANT led: STD_LOGIC_VECTOR(7 DOWNTO 0): = "10101010";
```

常量的作用域要视常量的声明位置而定,在 VHDL 程序不同位置声明的常量,作用域也不一样。VHDL 程序中以下的程序位置可进行常量的声明:

- (1) 程序包中声明的常量,引用该程序包的 VHDL 程序都可以使用该常量。
- (2) 实体声明部分的 generic 语句声明的常量,则该实体的结构体可以使用该常量。
- (3) 结构体声明的常量,则该结构体内的所有语句都可以使用该常量。
- (4) 进程、子程序和函数中声明的常量,则只能在其内部使用该常量。

代码 3-1 和代码 3-2 详细说明了常量的声明与使用的规则。

【代码 3-1】 简单的程序包。

```
PACKAGE mylib IS
    CONSTANT bus_length: INTEGER: = 8;
END mylib;
```

代码 3-1 是一个简单的 VHDL 库文件,它声明了一个整型常量 bus_length,它的值为 8。

【代码 3-2】 常量应用的程序。

```

LIBRARY IEEE;                                -- ieee 库的引用
USE IEEE. STD_LOGIC_1164. ALL;
USE IEEE. STD_LOGIC_UNSIGNED. ALL;
LIBRARY mylib;                                -- 自定义库 mylib 的引用
USE mylib.all;
ENTITY counter IS                               -- 实体声明
    GENERIC(                                    -- 类属性常量定义
        period: STD_LOGIC_VECTOR(bus_length-1 DOWNT0 0) := "01100100";
    );
    PORT(clk: IN STD_LOGIC;
        reset_n: IN STD_LOGIC;
        cntout: OUT STD_LOGIC_VECTOR(bus_length-1 DOWNT0 0));
END counter;
ARCHITECTURE behave OF counter IS
    CONSTANT addone: STD_LOGIC_VECTOR(bus_length-1 DOWNT0 0) := "00010100";
    SIGNAL cnt: STD_LOGIC_VECTOR(bus_length-1 DOWNT0 0) := "00000000";
BEGIN
    PROCESS(clk, reset_n)
    BEGIN
        IF reset_n = '0' THEN
            cntout <= "00000000";
        ELSIF clk'EVENT AND clk = '1' THEN
            cntout <= cnt;
        END IF;
    END PROCESS;
    PROCESS(clk, reset_n)
        CONSTANT addtwo: STD_LOGIC_VECTOR(bus_length-1 DOWNT0 0) := "00101000";
    BEGIN
        IF reset_n = '0' THEN
            cnt <= "00000000";
        ELSIF clk'EVENT AND clk = '1' THEN
            IF cnt > period THEN
                cnt <= "00000000";
                -- 当计数器大于计数周期时,计数器归零
            ELSIF cnt > addtwo THEN
                cnt <= cnt + "00000100";
                -- 当计数器大于 addtwo 时计数器作加 4 计数
            ELSIF cnt > addone THEN
                cnt <= cnt + "00000010";
                -- 当计数器大于 addone 时计数器作加 2 计数
            ELSE
                cnt <= cnt + "00000001";
                -- 否则计数器作加 1 计数
            END IF;
        END IF;
    END PROCESS;
END behave;

```

代码 3-2 分别从程序包、实体类属性、结构体声明和进程声明四个 VHDL 程序不同的位置介绍常量的声明。其中,程序包声明的常量,整个 VHDL 程序都可以使用;实体类属性中声明的常量,在声明该常量以下 VHDL 程序都可以使用;结构体声明中声明的常量,在该结构体中可以使用;在进程、子程序和函数中声明的常量,则只能在内部使用。

3.3.2 变量

变量(VARIABLE)也是用来存储指定数据类型的数值,但与常量不同的是,可以通过对变量赋值来改变变量的数值。变量的声明格式如下:

VARIABLE 变量名: 数据类型: = 初始值;

例如:

```
VARIABLE cnt: INTEGER: = 0;
VARIABLE temp: BIT: = '0';
VARIABLE flag: STD_LOGIC_VECTOR(7 DOWNTO 0);
```

其中,初始值是可选的,变量的初始值只在仿真时起作用,综合过程一般忽略变量的初始值。可以进行声明的 VHDL 程序位置是进程、子程序和函数,而且变量只在声明该变量的进程、子程序和函数中可见。

变量被声明后,就可以对变量进行赋值来更改变量数值。变量的赋值是理想化的数据传输,是即时的,不需要延时,这一点与信号不一样。变量赋值的格式如下:

变量名: = 表达式;

例如:

```
cnt: = 10;
temp: = '1';
```

由于变量的赋值是即时的,因此在变量赋值时加上时间延时是非法的,例如:

```
temp: = 10 after 10ns; —非法语句
```

变量赋值只出现在进程、子程序和函数中。将代码 3-2 修改为代码 3-3 来介绍变量的声明与使用。

【代码 3-3】 变量应用的程序。

```
LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164. ALL;
USE IEEE. STD_LOGIC_UNSIGNED. ALL;
ENTITY counter IS
    GENERIC(
        period: STD_LOGIC_VECTOR(7 DOWNTO 0): = "01100100"
    );
    PORT(clk: IN STD_LOGIC;
         reset_n: IN STD_LOGIC;
         cntout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END counter;
ARCHITECTURE behave OF counter IS
    CONSTANT addone: STD_LOGIC_VECTOR(7 DOWNTO 0): = "00010100"
    BEGIN
        PROCESS(clk, reset_n)
            CONSTANT addtwo: STD_LOGIC_VECTOR(7 DOWNTO 0): = "00101000"
            VARIABLE cnt: STD_LOGIC_VECTOR(7 DOWNTO 0): = "00000000"
            BEGIN
                IF reset_n = '0' THEN
                    cnt: = "00000000";
```

```

        cntout <= "00000000";
    ELSIF clk'EVENT AND clk = '1' THEN
        cntout <= cnt;
        IF cnt > period THEN
            cnt: = "00000000";
        ELSIF cnt > addtwo THEN
            cnt: = cnt + "00000100";
        ELSIF cnt > addone THEN
            cnt: = cnt + "00000010";
        ELSE
            cnt: = cnt + "00000001";
        END IF;
    END IF;
END PROCESS;
END behave;

```

由于变量只有在声明该变量的进程中使用,所以不能像代码 3-2 在另外一个进程对 cntout 进行赋值,而只能在同一进程进行赋值。

2002 版 VHDL 标准支持共享变量,共享变量可以在进程、子程序和函数之外声明,也可以在结构体或程序包中声明。相应地,共享变量的使用域相对普通变量要广,与信号应用相类似,但没有信号相应的属性。代码 3-4 介绍了共享变量的使用方法。

【代码 3-4】 共享常量应用的程序。

```

LIBRARY IEEE;
USE IEEE. STD_LOGIC_1164. ALL;
USE IEEE. STD_LOGIC_UNSIGNED. ALL;
ENTITY sharevar IS
PORT(clk: IN STD_LOGIC;
     sel: IN STD_LOGIC;
     dout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END sharevar;
ARCHITECTURE behave OF sharevar IS
CONSTANT dcon: STD_LOGIC_VECTOR(7 DOWNTO 0) := "10101010";
-- 在结构体的声明中声明一个共享变量 cnt
SHARED VARIABLE cnt: STD_LOGIC_VECTOR(7 DOWNTO 0) := "00000000";
BEGIN
    PROCESS(clk)          -- 在此进程中对共享变量进行加 1 计数
    BEGIN
        IF clk'EVENT AND clk = '1' THEN
            cnt: = cnt + "00000001";
        END IF;
    END PROCESS;
    PROCESS(sel)          -- 在另外一个进程对共享变量进行使用
    BEGIN
        IF sel = '1' THEN
            dout <= cnt;
        ELSE
            dout <= dcon;
        END IF;
    END PROCESS;
END behave;

```

3.3.3 信号

信号(SIGNAL)是 VHDL 作为硬件描述语言的一个主要特征。信号既可以描述硬件内部之间的连线;也可以描述一种数值寄存器,可以保留历史值。在 VHDL 中,信号是具有最多属性的数据对象,有关信号的属性将在后面章节进行讨论。信号的声明的格式如下:

SIGNAL 信号名: 数据类型: = 初始值;

例如:

```
SIGNAL temp: STD_LOGIC;  
SIGNAL cnt: STD_LOGIC_VECTOR(7 DOWNTO 0);  
SIGNAL data_bus: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000";
```

其中,信号声明中的初始值是可选的,而且信号的初始值只在仿真时起作用,综合过程一般会被忽略。信号的声明位置可以是实体的声明部分、结构体和程序包,但是不可在进程、子程序和函数中声明。其中,在程序包声明中声明的信号在引用该程序包的 VHDL 程序中都是可见的,使用时需要特别注意。在实体声明部分声明的信号即端口信号,除了有方向限制外,其他与在结构体声明中声明的信号一样的。

信号声明后就可以对信号进行赋值操作以改变信号数值。信号赋值的格式如下:

信号名<= 表达式;

例如:

```
temp <= '1';  
cnt <= cnt + "00000001";  
data_bus <= "00001000";
```

同样,由于 VHDL 语言是强类型的程序设计语言,因此信号赋值语句右边表达值的类型必须与信号的类型一致,如果不一致则需要进行类型转化。信号的赋值操作与变量的赋值操作除了操作符不一样外(信号赋值为“<=”,变量赋值为“:=”),信号赋值有一个延时过程,而变量赋值没有延时过程是即时的。信号赋值可以直接出现在结构体作为并行语句,也可出现在进程、子程序和函数中作为顺序语句,代码 3-5 介绍了信号的使用方法。

【代码 3-5】 信号应用的程序。

```
LIBRARY IEEE;  
USE IEEE. STD_LOGIC_1164. ALL;  
USE IEEE. STD_LOGIC_UNSIGNED. ALL;  
ENTITY usesignal IS  
PORT(clk: IN STD_LOGIC;  
      sel: IN STD_LOGIC;  
      dcon: IN STD_LOGIC_VECTOR(7 DOWNTO 0);  
      dout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);  
      cntout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  
END usesignal;  
ARCHITECTURE bebase OF usesignal IS  
  -- 结构体声明中声明信号 cnt  
  SIGNAL cnt: STD_LOGIC_VECTOR(7 DOWNTO 0):= "00000000";  
BEGIN  
PROCESS (clk)          -- 信号在进程中使用  
BEGIN
```

```

    IF clk'EVENT AND clk = '1' THEN
        cnt <= cnt + "00000001";
    END IF;
END PROCESS;
    cntout <= cnt;      -- 信号在进程外使用
PROCESS(sel, dcon, cnt)
BEGIN
    IF sel = '1' THEN
        dout <= cnt;
    ELSE
        dout <= dcon;
    END IF;
END PROCESS;
END behave;

```

代码 3-5 中实体声明部分的端口名前面虽然没有保留字 SIGNAL,但实体端口隐含了信号的意义,默认也为信号对象。'EVENT 是信号属性之一,它表示信号的值是否有事件(EVENT)发生,如果有事件发生则返回真,否则返回假。信号是 VHDL 的一大特色,VHDL 支持以下的信号属性,假设 sig 为一信号,t 为一时间值。其中前 4 条属性返回为特殊的信号,可用于正常信号可使用的位置,包括敏感信号表,但不能用于子程序中。

- sig'DELAYED(t)——结果是创建一个类型与 sig 相同的信号,且相对 sig 信号具有 t 时间的延时。
- sig'STABLE(t)——结果是创建一个 BOOLEAN 类型的信号,如果 sig 在时间 t 内没有事件发生则结果为真,否则为假。
- sig'QUIET(t)——结果是创建一个 BOOLEAN 类型的信号,如果 sig 在时间 t 内没有事件/事项处理(TRANSACTION)发生则结果为真,否则为假。
- sig'TRANSACTION——当 sig 有事件/事项处理(TRANSACTION)发生时,则所创建的 BIT 类型信号发生翻转。
- sig'EVENT——如果 sig 有事件发生则结果为真,否则为假。
- sig'ACTIVE——如果 sig 有事件/事项处理(TRANSACTION)发生则结果为真,否则为假。
- sig'LAST_EVENT——sig 最后一次事件发生的时间间隔。
- sig'LAST_ACTIVE——sig 最后一次事件/事项处理发生的时间间隔。
- sig'LAST_VALUE——sig 最后一次事件发生前的值。

信号属性应用举例:

- clk'EVENT AND clk='1'——时钟信号 clk 上升沿的表示方法。
- clk'EVENT AND clk='0'——时钟信号 clk 下降沿的表示方法。

3.3.4 别名

别名是指向一个数据对象或数据对象的部分可替换标识符,对这个标识符的操作相当于对被替换数据对象的操作。别名的声明格式如下:

ALIAS 别名名称: 别名的数据类型 IS 数据对象;

对别名的应用举例如下：

```
VARIABLE addr: STD_LOGIC_VECTOR(7 DOWNTO 0);  
VARIABLE data: STD_LOGIC_VECTOR(15 DOWNTO 0);  
ALIAS myaddr STD_LOGIC_VECTOR(7 DOWNTO 0) IS addr; -- 声明一个别名 myaddr 指向 addr  
ALIAS udata: STD_LOGIC_VECTOR(7 DOWNTO 0) IS data(15 DOWNTO 8);  
-- 声明一个别名 udata 指向 data 的高 8 位  
ALIAS ldata: STD_LOGIC_VECTOR(7 DOWNTO 0) IS data(7 DOWNTO 0); -- 声明一个别名 ldata 指向 data 低 8 位  
myaddr: = "01001101"; -- 相当于对 addr 赋值  
udata: = "11110000"; -- 相当于对 data 的高 8 位赋值  
ldata: = "00001111"; -- 相当于对 data 的低 8 位赋值
```

3.3.5 常量、变量和信号的比较

从硬件电路系统看,常量相当于电路中的恒定电平,如 GND 或 VCC 接口;而信号则相当于组合电路系统中门/模块间的连接及其连线上的信号值;变量则用来暂存中间结果。

从行为仿真和 VHDL 语句功能上看,变量和信号的区别主要表现在接收和保持信息的方式、信息保持与传递的区域大小上。例如,信号可以设置延时量,而变量则不能;变量只能作为局部的信息载体,而信号则可作为模块间的信息载体。变量的设置有时只是一种过渡,最后的信息传输和接口间的通信都靠信号来完成。

从综合后对应的硬件电路结构看,信号一般对应特定的硬件结构,但在许多情况下,信号和变量没有什么区别。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。这时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。

虽然 VHDL 仿真器允许变量和信号设置初始值,但在实际运用中,VHDL 综合器并不会把这些信息综合进去。这是因为实际的 FPGA/CPLD 芯片在上电后,并不能确保其初始状态的取值。因此,对于时序仿真来说,设置的初始值在综合时是没有实际意义的。

信号和变量是 VHDL 中重要的客体,主要区别有:

- (1) 信号赋值至少要有 delta 延时;而变量赋值没有。
- (2) 信号除当前值外有许多相关的信息,如历史信息;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值和连接子元件,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,用于硬件特性的高层次建模所需要的计算中。

信号相对变量来说,具有更多的硬件属性,信号与电路节点相对应,而变量则仅仅是为临时存储中间结果而设置的。

3.4 数据类型

数据类型是程序设计语言的重要特征之一,VHDL 又是强类型的程序设计语言,因此对数据类型是否了解在进行 VHDL 程序设计起到关键的作用。之所以称 VHDL 为强类型的程序设计语言,是因为每一种数据类型决定了它声明的数据对象所能进行的操作,即不同

数据类型数据对象的操作是不同的。VHDL 支持的常用数据类型主要有四类：

(1) 标量(Scalar)：标量类型的数据只有一个存储单元,有具体的数值大小,主要包括枚举类型、整数类型、物理类型和浮点类型。

(2) 复合(Composite)：复合类型的数据是由相同类型的元素(数组类型)或不同类型的元素(记录类型)组成。

(3) 存取(Access)：存取类型提供对指定类型对象的存取,本书不展开讨论。

(4) 文件(File)：文件提供对一系列指定类型数据的读取、写入和文件的检查,本书也不展开讨论。

VHDL 语言中的数据类型总览图如图 3-3 所示。

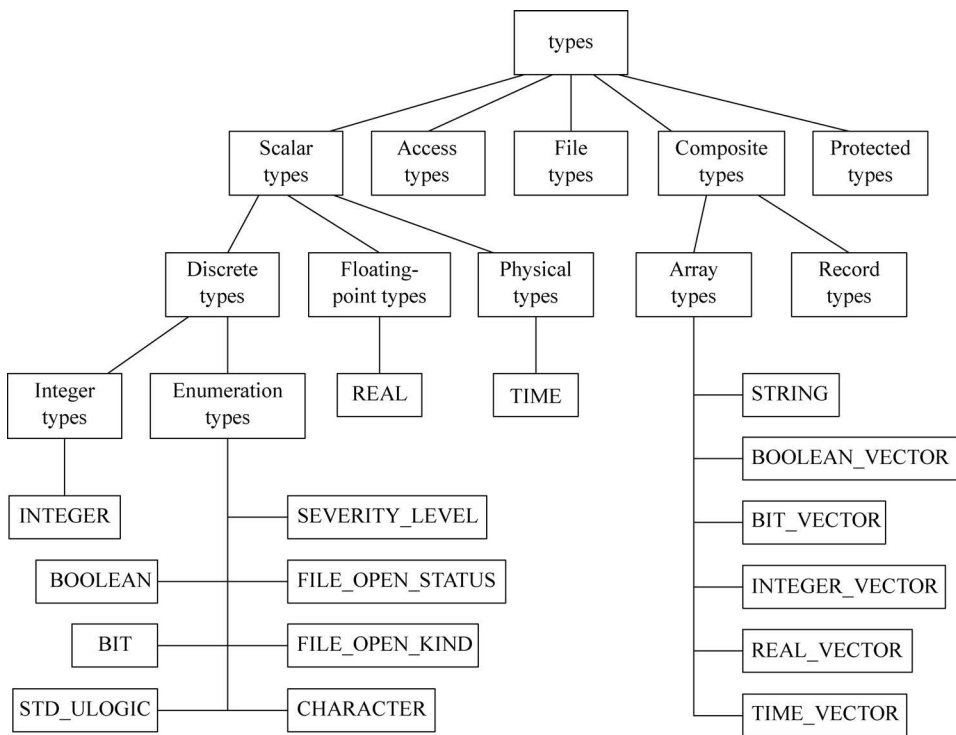


图 3-3 VHDL 中的数据类型总览图

3.4.1 标量

标量类型的数值是有大小之分的,关系操作符可以操作标量的数据对象。标量类型的数据具体包括:整型(INTEGER)、实型(REAL)、物理型(Physical)、枚举型(Enumeration)。枚举型又包括字符(CHARACTER)、位(BIT)和布尔型(BOOLEAN)等,物理型常用的有时间型(Time)。这里仅介绍 VHDL 中比较独特的部分数据类型,而关于整型和实型请大家自行参考相关说明。

1. 物理型

物理型的值是物理量的度量单元,比如长度、质量、电流、时间等。物理型数据类型一般是不可综合的,它们只应用在仿真过程中。它们的声明是基本单元和二级单元的组合,在默认的 STANDARD 程序包中预定义的物理型是时间 TIME 数据类型。TIME 数据类型在

STANDARD 程序包的定义如下：

```
TYPE TIME IS RANGE      0 TO 2147483647
UNITS
fs;                      -- 飞秒 VHDL 中的最小时间单位
ps = 1000 fs;           -- 皮秒
ns = 1000 ps;           -- 纳秒
us = 1000 ns;           -- 微秒
ms = 1000 us;           -- 毫秒
sec = 1000 ms;          -- 秒
min = 60 sec;           -- 分
hr = 60 min;            -- 时
END UNITS;
```

2. 枚举型

枚举型定义了一种拥有用户定义数值的类型,把属于该类型的所有元素都列出来。一大优点是直观,在元素数量不多的情况下可以有效提高 VHDL 程序的可读性。用户定义的数值包括标识符和字符,枚举型声明举例如下：

```
TYPE traclight IS (green, yellow, red);
TYPE op IS (add, sub, mul, div, mod);
TYPE logic IS ('0', '1', 'z', 'x');
```

和其他标量数据类型一样,枚举型的元素也是有大小之分的。枚举型值的大小是通过声明时的顺序来确定的,左边的值总是小于右边的值,如上面声明的 op 枚举型 add < sub < mul < div < mod。VHDL 预定义的枚举类型有 STD_ULOGIC、CHARACTER、BIT、BOOLEAN、SEVERITY_LEVEL、FILE_OPEN_KIND 和 FILE_OPEN_STATUS。其中,FILE_OPEN_KIND 和 FILE_OPEN_STATUS 本书暂不涉及。

1164 程序包定义的 std_ulogic 是指以下的字符集：

```
TYPE STD_ULOGIC IS ('U',    -- 未初始化状态
                    'X',    -- 强未知
                    '0',    -- 强 0
                    '1',    -- 强 1
                    'Z',    -- 高阻
                    'W',    -- 弱未知
                    'L',    -- 弱 0
                    'H',    -- 弱 1
                    '-' ,   -- 无关状态
                    );
```

STANDARD 程序包定义的 BIT 是指以下的枚举类型定义：

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

STANDARD 程序包定义的 BOOLEAN 是指以下的枚举类型定义：

```
TYPE BOOLEAN IS (FALSE, TRUE); -- 要么是 FALSE,要么是 TRUE
```

STANDARD 程序包定义的 SEVERITY_LEVEL 是指以下的枚举类型定义：

```
TYPE SEVERITY_LEVEL IS (NOTE, WARNING, ERROR, FAILURE); -- 有 4 个状态
```

3. 标量类型的属性

VHDL 中标量类型除了相应的数值,还有一组相应的属性,这些属性丰富了标量类型

的功能。VHDL 支持的预定义属性如下(t 代表标量类型, x 代表该类型的一个值, s 代表一个字符串, n 代表位置号):

- t 'LEFT —— t 类型的第一个值或是最左边的值;
- t 'RIGHT —— t 类型的最后一个值或是最右边的值;
- t 'LOW —— t 类型的最小的值;
- t 'HIGH —— t 类型的最大的值;
- t 'ASCENDING ——如果 t 类型的值是递增的,则为 TRUE,否则为 FALSE;
- t 'IMAGE(x) ——返回 t 类型 x 值所对应的字符串;
- t 'VALUE(s) ——返回字符串 s 表示的 t 类型中的值;
- t 'POS(x) —— t 类型中 x 的位置号,0 为第一个;
- t 'VAL(n) —— t 类型中位置为 n 的值;
- t 'SUCC(x) —— t 类型中位置号比 x 的位置号大 1 处的值;
- t 'PRED(x) —— t 类型中位置号比 x 的位置号小 1 处的值;
- t 'LEFTOF(x) —— t 类型中在 x 的左边的值;
- t 'RIGHTOF(x) —— t 类型中在 x 的右边的值。

下面的代码举例说明标量类型的属性:

```
TYPE length IS RANGE 0 TO 2147483647
UNITS
nm;          -- 纳米
um = 1000 nm; -- 微米
mm = 1000 um ; -- 毫米
m = 1000 mm ; -- 米
km = 1000 m ; -- 千米
END UNITS ;
TYPE tractled IS (red, green, yellow) ;
TYPE smallnum RANGE 255 DOWNT0 0;
length'LEFT = 0 nm;
length'RIGHT = 2147483647 nm;
length'LOW = 0 nm;
length'HIGH = 2147483647 nm;
length'ASCENDING = TRUE;
length'IMAGE(2 um) = "2000 nm";
length'VALUE("1 um") = 1000 nm;
tractled'LEFT = red;
tractled'RIGHT = yellow;
tractled'LOW = red;
tractled'HIGH = yellow;
tractled'ASCENDING = TRUE ;
tractled'IMAGE(green) = "green" ;
tractled'VALUE("yellow") = yellow;
smallnum'LEFT = 255 ;
smallnum'RIGHT = 0 ;
smallnum'LOW = 0 ;
smallnum'HIGH = 255 ;
smallnum'ASCENDING = FALSE;
smallnum'IMAGE(100) = "100";
smallnum'VALUE("15") = 15;
tractled'POS(green) = 1;
tractled'VAL(2) = yellow;
```

```

tractled'SUCC(green) = yellow;
tractled'PRED(green) = red;
tractled'LEFTOF(green) = red;
tractled'RIGHTOF(green) = yellow;

```

3.4.2 复合类型

复合类型是一种类型或是多种类型的组合。一种类型的组合称为数组类型 ARRAY, 多种数据类型的组合称为记录类型 RECORD。本书仅介绍数组类型, 记录类型大家自行查阅文献。

数组类型属于复合类型的一种, 是将一组具有相同数据类型的元素集合在一起组成一个数据对象的数据类型。VHDL 支持一维数组和二维数组。一维数组拥有一个索引下标, 二维数组拥有两个索引下标。VHDL 支持的数组根据索引的确定性分为约束性数组和非约束性数组。

约束性数组在定义的时候就指定了数组索引下标的范围, 定义格式如下:

```
TYPE 数组名 IS ARRAY (数组范围) OF 数据类型;
```

举例如下:

```

TYPE data IS ARRAY(7 DOWNT0 0) OF BIT;
TYPE bus IS ARRAY(0 TO 7) OF BIT;

```

约束性数组使用举例如下:

```

SIGNAL datain: data;      -- 声明一个 data 数组类型的信号 datain, 它包含 8 位
SIGNAL addr_bus: bus;     -- 声明一个 bus 数组类型的信号 addr_bus, 它包含 8 位

```

DOWNT0 与 TO 一般用于范围约束, 用 DOWNT0 描述的数组左边的索引下标值大于右边的索引下标值, 用 TO 描述的数组右边的索引下标值大于左边的索引下标值。举例如下:

```

TYPE data IS ARRAY(7 DOWNT0 0) OF BIT;
TYPE bus IS ARRAY(0 TO 7) OF BIT;
SIGNAL datain: data := "11110000"; -- 声明一个 data 数组类型的信号 datain, 它包含 8 位
SIGNAL addr_bus: bus := "1111000"; -- 声明一个 bus 数组类型的信号 addr_bus, 它包含 8 位
-- datain(7) = '1', datain(0) = '0';
-- addr_bus(7) = '0', addr_bus(0) = '1';

```

在进行 VHDL 程序开发中, 如无特别要求, 关于范围约束就统一使用 DOWNT0 或 TO, 以保持整个开发程序的一致性, 有利于提高 VHDL 程序的可读性与利用性。

非约束性数组在定义的时候就没有指定数组索引下标的范围, 而是用 <> 来代替, 定义格式如下:

```
TYPE 数组名 IS ARRAY(索引下标子类型 RANGE <>) OF 数据类型;
```

举例如下:

```

TYPE data IS ARRAY (INTEGER <>) OF BIT;
TYPE bus IS ARRAY (INTEGER <>) OF BIT;

```

约束性数组使用举例如下:

```

-- 声明一个 data 数组类型的信号 datain 时需要加上具体的范围约束条件, 它包含 8 位
SIGNAL datain: data(7 DOWNT0 0);

```

```
-- 声明一个 bus 数组类型的信号 addr_bus 时需要加上具体的范围约束条件,它包含 8 位
SIGNAL addr_bus: bus(0 TO 7);
```

常用的非约束数组类型有: STRING 是字符元素的集合,BIT_VECTOR 是 BIT 的集合;STD_LOGIC_VECTOR 是 STD_LOGIC 的集合。它们的声明如下:

```
TYPE STRING IS ARRAY (POSITIVE RANGE <>) OF CHARACTER;
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT;
TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;
-- 以上索引范围没有确定的数组是非约束型的数组类型
-- 用上面的数据类型声明信号的格式如下
SIGNAL str: STRING(1 TO 12):="I love FPGA!";
SIGNAL data: BIT_VECTOR(7 DOWNTO 0);
SIGNAL address: BIT_VECTOR(7 DOWNTO 0);
```

二维数组在硬件中一般对应一块存储块,其定义举例如下:

```
-- 定义二维数组方式一: 非约束方式
TYPE readdata IS ARRAY (NATURAL RANGE <>, NATURAL <>) OF BIT;
-- 定义二维数组方式: 约束方式
TYPE writedata IS ARRAY(7 DOWNTO 0) OF BIT_VECTOR(7 DOWNTO 0);
-- 定义二维数组方式一: 定义的数组类型声明一个二维数组需要指定约束范围
SIGNAL rdata: readdata(7 DOWNTO 0,7 DOWNTO 0);
-- 定义二维数组方式二: 定义的数组类型声明一个二维数组不需要指定约束范围
SIGNAL wdata: writedata;
```

VHDL 中数组的属性提供了数组对象或类型索引下标值信息。VHDL 支持的预定义属性如下(a 代表数组对象存取类型变量,n 代表数组标号):

- a'LEFT(n)——n 维数组索引下标范围中最左边的值;
- a'RIGHT(n)——n 维数组索引下标范围中最右边的值;
- a'LOW(n)——n 维数组索引下标范围中最小的值;
- a'HIGH(n)——n 维数组索引下标范围中最大的值;
- a'RANGE(n)——n 维数组索引下标范围;
- a'REVERSE_RANGE(n)——与原来 n 维数组的方向和边界相反的数组的范围;
- a'LENGTH(n)——n 维数组索引下标范围的长度;
- a'ASCENDING(n)——如果 n 维数组索引下标范围为升序,则为 TRUE,否则为 FALSE。

3.4.3 数据类型转换

由于 VHDL 是强类型的程序设计语言,当不同数据类型的数据对象进行运算时,就需要做数据类型转换。STD_LOGIC_ARITH 程序包定义了多个数据类型转换函数,引用它就可以在 VHDL 程序中使用 STD_LOGIC_ARITH 提供的函数实现数据类型的转换。STD_LOGIC_ARITH 提供的转换函数有如下:

```
FUNCTION CONV_INTEGER(arg: INTEGER) RETURN INTEGER;
FUNCTION CONV_INTEGER(arg: UNSIGNED) RETURN INTEGER;
FUNCTION CONV_INTEGER(arg: SIGNED) RETURN INTEGER;
FUNCTION CONV_INTEGER(arg: STD_ULOGIC) RETURN INTEGER;
FUNCTION CONV_UNSIGNED(arg: INTEGER; size: INTEGER) RETURN UNSIGNED;
FUNCTION CONV_UNSIGNED(arg: UNSIGNED; size: INTEGER) RETURN UNSIGNED;
FUNCTION CONV_UNSIGNED(arg: SIGNED; size: INTEGER) RETURN UNSIGNED;
```

```

FUNCTION CONV_UNSIGNED(arg: STD_ULOGIC; size: INTEGER) RETURN UNSIGNED;
FUNCTION CONV_SIGNED(arg: INTEGER; size: INTEGER) RETURN SIGNED;
FUNCTION CONV_UNSIGNED(arg: UNSIGNED; size: INTEGER) RETURN SIGNED;
FUNCTION CONV_SIGNED(arg: SIGNED; size: INTEGER) RETURN SIGNED;
FUNCTION CONV_UNSIGNED(arg: STD_ULOGIC; size: INTEGER) RETURN SIGNED;
FUNCTION CONV_STD_LOGIC_VECTOR(arg: INTEGER; size: INTEGER)
    RETURN STD_LOGIC_VECTOR;
FUNCTION CONV_STD_LOGIC_VECTOR(arg: UNSIGNED; size: INTEGER)
    RETURN STD_LOGIC_VECTOR;
FUNCTION CONV_STD_LOGIC_VECTOR(arg: SIGNED; size: INTEGER)
    RETURN STD_LOGIC_VECTOR;
FUNCTION CONV_STD_LOGIC_VECTOR(arg: STD_ULOGIC; size: INTEGER)
    RETURN STD_LOGIC_VECTOR;

```

STD_LOGIC_1164 程序包提供的类型转换如下：

```

FUNCTION TO_BIT (s:STD_ULOGIC; xmap: BIT = '0') RETURN BIT;
FUNCTION TO_BITVECTOR(s:STD_LOGIC_VECTOR;xmap: BIT = '0') RETURN BIT_VECTOR;
FUNCTION TO_BITVECTOR(s:STD_ULOGIC_VECTOR; xmap: BIT = '0') RETURN BIT_VECTOR;
FUNCTION TO_STDULOGIC (b:BIT ) RETURN STD_ULOGIC;
FUNCTION TO_STDLOGICVECTOR (b: BIT_VECTOR ) RETURN STD_LOGIC_VECTOR;
FUNCTION TO_STDLOGICVECTOR (s:STD_ULOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
FUNCTION TO_STDULOGICVECTOR(b:BIT_VECTOR )RETURN STD_ULOGIC_VECTOR;
FUNCTION TO_STDULOGICVECTOR(s:STD_LOGIC_VECTOR ) RETURN STD_ULOGIC_VECTOR;

```

代码 3-6 介绍了使用类型转换函数的 VHDL 程序。

【代码 3-6】 函数转换应用的程序。

```

-- 库的引用
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
ENTITY typeconv IS
PORT(clk: IN STD_LOGIC;
      reset_n: IN STD_LOGIC;
      dout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END typeconv;
ARCHITECTURE behave OF typeconv IS
    SIGNAL cnt: INTEGER RANGE 0 TO 255;
BEGIN
PROCESS(clk, reset_n)
BEGIN
    IF reset_n = '0' THEN
-- 应用类型转换函数,把 INTEGER 类型转换为 STD_LOGIC_VECTOR
        dout <= CONV_STD_LOGIC_VECTOR(0,8);
        ELSEIF clk'EVENT AND clk = '1' THEN
-- 应用类型转换函数,把 INTEGER 类型转换为 STD_LOGIC_VECTOR
            dout <= CONV_STD_LOGIC_VECTOR(cnt,8);
        END IF;
    END PROCESS;
END behave;

```

3.4.4 子类型

子类型是某一数据类型的一个子集。有时为了提高程序的可读性,一些数据对象的取

值只是某一数据类型的一个子集,这时可以定义一个子类型来专门为这些数据对象进行声明。子类型的基本格式如下:

```
SUBTYPE 子类型名 IS 数据类型 RANGE <约束范围>;
```

举例如下:

```
-- 定义一个子类型 smallnum,它是整型的子集,取值范围为 0~255
SUBTYPE smallnum IS INTEGER RANGE 0 TO 255;
-- 用子类型 smallnum 声明一个变量,这样 cnt 的取值范围就是 0~255
VARIABLE cnt: smallnum := 0;
```

3.5 行为描述

3.5.1 信号赋值

信号赋值分为顺序信号赋值和并行信号赋值两种情况。

1. 顺序信号赋值

顺序信号赋值发生在进程中,前面例子中的信号只是简单形式的信号赋值,另外还有条件信号赋值和选择信号赋值。

条件信号赋值语句:

```
reg : PROCESS (clk) IS
BEGIN
    IF RISING_EDGE(clk) THEN
        q <= (OTHERS => '0') WHEN reset ELSE d;
    END IF;
END PROCESS reg;
```

上面进程中的选择信号赋值语句相当于下面这个 if 结构。

```
IF reset THEN
    q <= (OTHERS => '0');
ELSE
    q <= d;
END IF;
```

选择信号赋值语句:

如果在一个进程中使用选择信号赋值语句去实现一个数据选择器,如下。

```
WITH d_sel SELECT
    q <= source0 WHEN "00",
        source1 WHEN "01",
        source2 WHEN "10",
        source3 WHEN "11",
        source4 WHEN OTHERS;
```

它可以用下面的 case 语言来等效:

```
CASE d_sel IS
    WHEN "00" =>
        q <= source0;
    WHEN "01"
        q <= source1;
    WHEN "10"
```

```

        q <= source2;
    WHEN "11"
        q <= source3;
    WHEN OTHERS
        q <= source4;
END CASE;

```

进程的信号赋值并不是按照出现的先后顺序赋值的,而是在进程结束时同时被赋值的。如果在进程存在多条信号赋值语句,则这些信号赋值都是在进程结束时同时被赋值的。如果在进程中有多条赋值语句对同一个信号赋值,则该信号被赋予的值是最后一条赋值语句的数值。而如果结构体中有多个进程,则认为各个进程是同时结束的。

【代码 3-7】

```

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
    SIGNAL muxval : INTEGER RANGE 7 DOWNT0 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;

```

【代码 3-8】

```

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 DOWNT0 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;

```

代码 3-7 中,信号 muxval 在进程中出现了三次赋值操作,即有三个赋值源: $\text{muxval} \leq 0$, $\text{muxval} \leq \text{muxval} + 1$ 和 $\text{muxval} \leq \text{muxval} + 2$; 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 $\text{muxval} \leq \text{muxval} + 2$ 语句中的 muxval 的值得到更新。然而,由于传输符号右边的 muxval 始终未得到任何确定的初始值,即语句 $\text{muxval} \leq 0$ 并未完成赋值,所以 muxval 始终是个未知值。可以证明,代码 3-7 的设计只能被综合成随 b 和 a (a 和 b 被综合成了时钟输入信号)变动的时序电路,从而导致 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。这在图 3-4 中十分清晰。

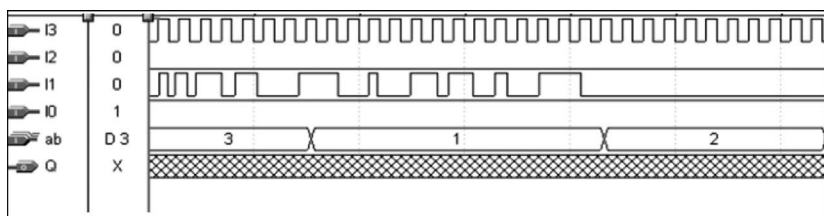


图 3-4 代码 3-7 的错误工作时序

代码 3-8 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 $\text{muxval} := 0$ (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 语句从表面上看很像不完整的条件语句,但都不可能综合成时序电路。显然代码 3-8 是一个纯组合电路,它们也就有了图 3-5 的正确波形输出。

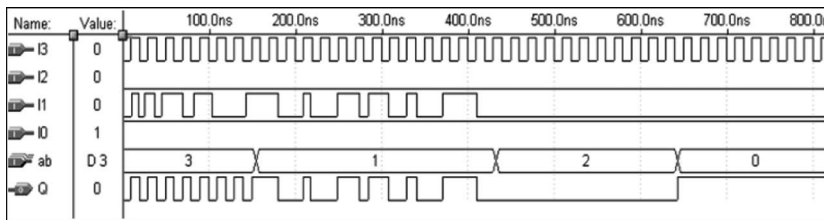


图 3-5 代码 3-8 的正确工作时序

2. 并行信号赋值

所谓并行信号赋值,是指发生在结构体中的信号赋值。同样也存在简单信号赋值、条件信号赋值、选择信号赋值三种情况。由于结构体的语句都是并行语句,因此如果有多个信号赋值,则它们之间是并行执行的,即都是在结构体结束的瞬间同时被赋值的。但是如果对同一个信号进行多次并行赋值,则 VHDL 原则是不允许的,即此时认为发生了线与,需要进行决断处理,否则系统会报错。

在结构体中的一次信号赋值被称为信号的一次驱动,等号右边的信号被称作驱动源,而信号赋值语句则构成了一个驱动器。比如,

```
ARCHITECTURE test OF test IS
BEGIN
    a <= b AFTER 10 ns;
    a <= c AFTER 10 ns;
END test;
```

上面例子中,信号 a 被赋值两次,即被驱动两次,这两次的驱动源分别为经过延时的信号 b 和信号 c。由于在结构体中存在两次对同一个信号 a 的并行赋值,因此相当于对两个驱动源实现线与,这是不允许的,需要进行决断处理。

再比如,下面代码 3-9 例子中,对于信号 q 相当于存在 4 个驱动源,这显然是不允许的,因此需要复杂的信号决断处理。如果能够改成代码 3-10,用一条条件信号赋值语句去实现,信号 q 相当于只有一个驱动源,则可以免去信号决断处理,并且能够实现正确的数据选择功能。

【代码 3-9】

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux IS
    PORT (i0, i1, i2, i3, a, b: IN STD_LOGIC;
          q : OUT STD_LOGIC);
END mux;
ARCHITECTURE bad OF mux IS
BEGIN
    q <= i0 WHEN a = '0' AND b = '0' ELSE '0';
    q <= i1 WHEN a = '1' AND b = '0' ELSE '0';
    q <= i2 WHEN a = '0' AND b = '1' ELSE '0';
    q <= i3 WHEN a = '1' AND b = '1' ELSE '0';
END bad;
```

【代码 3-10】

```
ARCHITECTURE better OF mux IS
BEGIN
    q <= i0 WHEN a = '0' AND b = '0' ELSE
    i1 WHEN a = '1' AND b = '0' ELSE
    i2 WHEN a = '0' AND b = '1' ELSE
    i3 WHEN a = '1' AND b = '1' ELSE
    'X'; -- unknown
END better;
```

3.5.2 延时

在 VHDL 中,关于行为建模有两种类型的延时:惯性延时和传输延时,两类延时特性不同。除此之外,还有一个专为仿真设置的 delta 延时。

1. 惯性延时 INERTIAL Delay

真实的电子电路通常不具有无限频率响应,由于分布寄生电容、电阻等的存在,使得输出值随输入值的改变通常需要延时特定的时间,这个时间通常被称作 T_{pd} 。只有输入值维持足够长的时间,输出值才能有相应改变,类似于器件具有一定的惯性,因此称作惯性延时

(INERTIAL Delay)。惯性延时机制使得较小的输入脉冲不会引发器件输出变化。通常信号赋值中默认的延时机制就是惯性延时,用关键词 INERTIAL 表示。

如图 3-6 所示的缓冲器,其惯性延时时间为 20ns,在 10ns 时刻输入信号从 0 变成 1,则预期其将导致在 30ns 时刻输出发生变化。输入信号 A 在 20ns 时刻从 1 变到 0,则预期其将导致在 40ns 时刻输出发生变化。由于器件的惯性延时为 20ns,输出在 30ns 的变化还没有来得及传过来,就被输入信号新的变化所冲掉,导致输出在 30ns 的变化根本不会发生,而直接取代发生的是输出信号在 40ns 的变化。因此,在输出端,看起来输入信号的脉冲并未影响到输出端信号,其根本原因是输入信号的脉冲过窄,小于器件的惯性延时时间 20ns,相当于将输入的窄小脉冲自动过滤掉了。

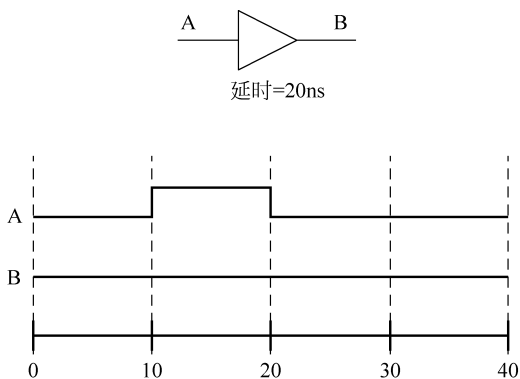


图 3-6 惯性延时模型

2. 传输延时

传输延时(TRANSPORT Delay)机制用来建模具有无限频率响应的理想器件,所谓无限频率响应,是指不管输入脉冲多么短暂,都会产生一个输出脉冲。理想的传输线是这类器件的典型实例,它会传送所有的输入变化,只是需要延时信号赋值语句中规定的时间。传输延时关键词是 TRANSPORT,不能省略。

仍然以缓冲器为例,如果将其延时机制换成传输延时,延时时间仍然为 20ns,则输入信号波形 A 的任何变化都会导致如图 3-7 中器件输出信号 B 的相应变化。传输延时不会过滤掉窄小的输入脉冲,而是会将输入信号上的任何变化都反应在输出端。而实际器件中往往不存在这样的情况,因此这种延时机制较少使用,仅仅在讨论导线延时,通常会使用它。

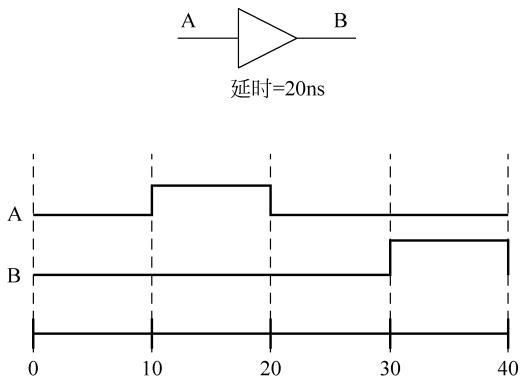


图 3-7 传输延时模型

具有 500ps 延时的传输线建模如下所示:

```
transmission_line : PROCESS (line_in) IS
BEGIN
    line_out <= TRANSPORT line_in AFTER 500 ps;
END PROCESS transmission_line;
```

3. delta 仿真延时

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 δ 延时,或称仿真 δ 。它是 VHDL 仿真器的最小分辨时间,并不代表器件实际的惯性延时情况。在 VHDL 程序语句中,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 δ 延时。

要理解 δ 延时,首先要弄清楚 VHDL 仿真的过程。仿真初始化阶段,所有的进程均被执行一次,直到进程被挂起,然后就周而复始地完成多个仿真周期,直到没有新的信号更新值,不会激活新的进程执行为止。每个仿真周期包括两个阶段:信号更新阶段和进程执行阶段。进程中,通常会对信号进行赋值,然而信号赋值并不是立即发生,信号赋值语句仅仅是信号的赋值计划(Schedules),这些赋值计划均需要等到进程被执行完,再次挂起时才能真正被赋值。也就说进程执行完,被挂起,则进入信号更新阶段。然而在信号更新阶段,往往存在多个信号需要同时被赋值。如果某些信号赋值之后有事件发生,即有从 0 到 1 或者从 1 到 0 的变化,则会激活相应的进程,进而转入下一个时刻的进程执行阶段。如此信号更新和进程执行周而复始,完成整个 VHDL 仿真波形的绘制。而在信号更新阶段,虽然可以理解成是瞬间完成的,各个信号同时被赋值,但是由于 EDA 工具本质上是在微机系统中完成的,而微机系统核心工作原理就是程序存储执行原理,即冯·诺依曼原理,其程序执行是按照顺序执行的。如果将信号更新的瞬间无限放大,就会发现,各个信号的赋值过程其实是按照逻辑电路的连接顺序先后被赋值的,此时各个信号被赋值的先后顺序会导致电路的各节点的仿真波形不同,违背特定赋值顺序即会导致电路功能错误。假设每个信号赋值需要 δ 延时完成,于是信号更新的瞬间就可以被细化成若干个 δ 延时,而每个 δ 延时均为无穷小的量,若干个(只要不是无限多)信号赋值时长在数学上也仍然是无穷小的量。

因此, δ 延时设置仅为了仿真(计算)正确,由于数字电路中各个节点传送信号是并行的,为了实现并行信号赋值的特点,同时又保证能得到逻辑功能正确的仿真波形。

3.5.3 进程

进程(PROCESS)是描述系统行为的最重要并行语句之一,进程本身是并行语句,多个进程语句是并行执行的,但它内部却是顺序语句组成的。它可以实现组合电路的行为描述,也可以实现时序逻辑电路的行为描述。

进程的敏感信号列表至少需要一个敏感信号。敏感信号的变化决定着进程是否执行,如果进程敏感信号列表没有信号,那进程永远挂起而不执行;但是,如果进程敏感信号列表没有信号,则可以使用 WAIT 语句来代替敏感信号列表的功能。

关于在敏感信号列表中需要列出哪些信号,经验规则是:如果进程是实现组合逻辑功能,则赋值操作符右边的信号就应该都列入敏感信号列表(除非有特殊考虑);如果进程是实现时序逻辑功能且没有一个异步信号,则只需要把时钟信号列入敏感信号列表就可以了;

如果进程实现时序逻辑功能但有异步信号,则需要把时钟信号与异步信号列入敏感信号列表。

进程的声明语句可以为进程定义一些变量供进程内部使用,举例如下:

```
PROCESS( clk)
    VARIABLE temp: INTEGER RANGE 0 TO 127 := 0;    -- 进程内部声明变量的位置
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        temp := temp + 1;
        dout <= temp;
    END IF;
END PROCESS;
```

代码 3-11 介绍了各种进程的使用方法。

【代码 3-11】 进程应用的程序。

```
LIBRARY IEEE;                                -- 库的引用
USE IEEE. STD_LOGIC_1164. ALL;
USE IEEE. STD_LOGIC_UNSIGNED. ALL;
ENTITY useprocess IS                          -- 实体声明
PORT(clk: IN STD_LOGIC;
     da: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
     db: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
     rst: IN STD_LOGIC;
     dand: OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
     isequal: OUT STD_LOGIC;
     isless: OUT STD_LOGIC;
     cntout: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END useprocess;
ARCHITECTURE behave OF useprocess IS
BEGIN
    -- 纯组合逻辑功能:敏感信号列表必须列出全部的右操作信号
    andout: PROCESS(da, db)
    BEGIN
        dand <= da AND db;
    END PROCESS andout;
    -- 纯同步的时序逻辑功能:敏感信号列表只需要列出时钟信号即可
    PROCESS(clk)
    BEGIN
        IF clk'EVENT AND clk = '1' THEN
            IF da = db THEN
                isequal <= '1';
            ELSE
                isequal <= '0';
            END IF;
        END IF;
    END PROCESS;
    -- 没有敏感信号列表:需要使用 WAIT 语句来控制进程的启动
    PROCESS
    BEGIN
        IF da < db THEN
```

```

        isless<= '1';
    ELSE
        isless<= '0';
    END IF;
-- WAIT 语句控制进程的启动;UNTIL 条件为真,启动进程
    WAIT UNTIL (clk'EVENT AND clk = '1');
END PROCESS;
-- 带有异步信号的时序逻辑功能: 敏感信号列表需要列出时钟信号与所有的异步信号
PROCESS(clk, rst)
-- 进程声明部分声明一个变量 cnt
    VARIABLE cnt: STD_LOGIC_VECTOR(7 DOWNTO 0) := "00000000";
BEGIN
-- 异步信号操作部分
    IF rst = '1' THEN
        cntout<= "00000000";
        cnt:= "00000000";
-- 同步信号操作部分
    ELSIF clk'EVENT AND clk = '1' THEN
        cnt:= cnt + "00000001";
        cntout<= cnt;
    END IF;
END PROCESS;
END behave;

```

3.5.4 WAIT 语句

WAIT 语句使设计者可以暂停进程或者子程序(函数和过程)的顺序执行过程。被暂停的进程或者子程序恢复执行的条件可以用下面三种形式的 WAIT 语句来规定。

1. WAIT ON signal [,signal]

如果 WAIT ON 后面的信号列表中至少一个信号有事件,则启动执行该 WAIT 语句后面的语句。所谓有事件,即指信号有从 0 到 1 的变化或者从 1 到 0 的变化。

下面例子中,WAIT 语句是放在进程结束之前的,其效果等效于在进程开始时的敏感信号表。虽然进程不包含敏感信号表,但是等效其敏感信号为 WAIT ON 后面的信号。之所以把 WAIT ON 语句放在进程最后,是由于在仿真器初始化时所有进程都会被执行一次,为了确保复位信号为异步复位,必须把 WAIT ON 放在进程结束之前。下面例子中,当 reset 或者 clock 信号中至少有一个发生事件,进程才会再次被执行。进程中的复位信号是异步复位。

```

PROCESS
BEGIN
    IF (reset = '1') THEN
        q<= '0';
    ELSIF clock'EVENT AND clock = '1' THEN
        q<= d;
    END IF;
    WAIT ON reset, clock;
END PROCESS;

```

等效成

```
PROCESS (reset, clock)
BEGIN
    IF (reset = '1') THEN
        q <= '0';
    ELSIF clock'EVENT AND clock = '1' THEN
        q <= d;
    END IF;
END PROCESS;
```

另外, WAIT ON 可以用来动态地改变进程的敏感信号表, 比如下例中, 位置 1 处之后的语句敏感信号为 a, 位置 2 处之后的语句敏感信号为 b。

```
PROCESS
BEGIN
    .
    WAIT ON a; -- 1.
    .
    .
    WAIT ON b; -- 2.
    .
    .
END PROCESS;
```

2. WAIT UNTIL boolean_expression

如果 WAIT UNTIL 后面的布尔表达式为 TRUE, 则启动执行 WAIT 后面的语句。下面例子中, 直到 clock 信号为 1, 而且有事件, 也就是 clock 出现一个上升沿, 才执行 WAIT 后面的语句。因此, 此例中描述的是一个具有同步复位功能的 D 触发器。

```
PROCESS
BEGIN
    WAIT UNTIL clock = '1' AND clock'EVENT;
    IF (reset = '1') THEN
        q <= '0';
    ELSE
        q <= d;
    END IF;
END PROCESS;
```

3. WAIT FOR time_expression

该语句表示当前的进程或者子程序(函数和过程)暂停 WAIT FOR 后面时间表达式所规定的时间, 当时间结束时, 则启动执行 WAIT 后面的语句。比如下面语句表示暂停 10ns, 之后继续执行 WAIT 之后的语句。

```
WAIT FOR 10 ns;
```

3.5.5 决断信号

在数字电路中, 通常要避免将多个器件的输出直接连接在一起, 即不允许直接实现线与, 这样有时会烧毁器件, 更重要的是这样会造成该节点的逻辑值混乱。同样, 在 VHDL 中, 如果在结构体中, 存在多个驱动同时给同一个信号并行赋值, 则相当于该节点信号发生线与。此时, 需要设计者采用决断信号对此进行处理。

决断信号是常规信号的扩展形式。使用决断信号进行处理时,需要涉及的三要素为:决断子类型、决断信号和决断函数。其中决断函数则具体规定了多个驱动源驱动同一信号时,为确定该信号的最终取值需要的具体处理方式。决断信号的类型为预先定义的决断类型。

决断信号有两种使用方法:

方法①:定义一个决断函数,然后声明一个决断信号,在相应的信号声明中规定其决断所使用的决断函数。

典型实例如下所示:

【代码 3-12】

```
PACKAGE fourpack IS
    TYPE fourval IS (X, L, H, Z);
    TYPE fourvalvector IS ARRAY(natural RANGE <>) OF fourval;
    FUNCTION resolve( s: fourvalvector) RETURN fourval;    -- 决断函数声明
    SUBTYPE resfour IS resolve fourval;                  -- 决断类型
END fourpack;
PACKAGE BODY fourpack IS
    FUNCTION resolve( s: fourvalvector) RETURN fourval IS -- 决断函数体定义
        VARIABLE result : fourval := Z;
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE result IS
                WHEN Z =>
                    CASE s(i) IS
                        WHEN H =>
                            result := H;
                        WHEN L =>
                            result := L;
                        WHEN X =>
                            result := X;
                        WHEN OTHERS =>
                            NULL;
                    END CASE;
                WHEN L =>
                    CASE s(i) IS
                        WHEN H =>
                            result := X;
                        WHEN X =>
                            result := X;
                        WHEN OTHERS =>
                            NULL;
                    END CASE;
                WHEN H =>
                    CASE s(i) IS
                        WHEN L =>
                            result := X;
                        WHEN X =>
                            result := X;
                        WHEN OTHERS =>
                            NULL;
                    END CASE;
                WHEN X =>
                    result := X;
            END CASE;
        END LOOP;
    END FUNCTION;
END PACKAGE BODY fourpack;
```

```

        END CASE;
    END LOOP;
    RETURN result;
END resolve;
END fourpack;
USE WORK.fourpack.ALL;
ENTITY mux2 IS
    PORT( i1, i2, a : IN fourval;
          q : OUT fourval);
END mux2;
ARCHITECTURE different OF mux2 IS
    COMPONENT and2
        PORT( a, b : IN fourval;
              c : OUT fourval);
    END COMPONENT;
    COMPONENT inv
        PORT( a : IN fourval;
              b : OUT fourval);
    END COMPONENT;
    SIGNAL nota : fourval;
    -- resolved signal
    SIGNAL intq : resolve fourval := X;           -- 决断信号
BEGIN
    U1: inv PORT MAP(a, nota);
    U2: and2 PORT MAP(i1, a, intq);
    U3: and2 PORT MAP(i2, nota, intq);
    q <= intq;
END different;

```

需要说明的是,在代码 3-12 的包声明中的四句声明顺序不能颠倒,如果没有前面的声明,则后面的声明不成立。信号 intq 是一个决断信号,因为它同时是两个器件 U2 和 U3 的输出,即有多个源同时驱动信号 intq,因此需要用决断函数 resolve 来进行决断。而信号 nota 则不需要采用决断信号,因为它只是 U1 器件的输出。

方法②: 定义个决断子类型,然后声明一个信号使用该子类型。

代码 3-12 中,可以将信号声明部分做如下改变,效果相同。

```
SIGNAL intq : resolve fourval := X;
```

可以替换成

```
SIGNAL intq : resfour := X;
```

VHDL 中广泛使用的程序包 IEEE STD_LOGIC_1164 中的数据类型 STD_LOGIC,其实就是一个多值决断子类型。该决断子类型对应的基本类型为 STD_ULOGIC,其定义为:

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

所对应的向量类型 STD_ULOGIC_VECTOR 定义如下:

```
TYPE STD_ULOGIC_VECTOR IS ARRAY ( NATURAL RANGE <> ) OF STD_ULOGIC;
```

这两个类型是定义决断子类型和决断函数的基本类型。在此基础上,决断函数和决断子类型的声明如下:

```
FUNCTION resolved ( s : STD_ULOGIC_VECTOR ) RETURN STD_ULOGIC;
```

```
SUBTYPE STD_LOGIC IS resolved STD_ULOGIC;
SUBTYPE STD_LOGIC_VECTOR (resolved) STD_ULOGIC_VECTOR;
```

在程序包声明中声明了决断函数,同时程序包包体中还定义了决断函数的具体实现:

```
TYPE stdlogic_table IS ARRAY (STD_ULOGIC, STD_ULOGIC) OF STD_ULOGIC;
CONSTANT resolution_table : stdlogic_table :=
-----
-- 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
-----
( ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- 'U'
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- 'X'
  ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- '0'
  ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- '1'
  ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- 'Z'
  ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- 'W'
  ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- 'L'
  ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- 'H'
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- '-'
);
FUNCTION resolved ( s : STD_ULOGIC_VECTOR ) RETURN STD_ULOGIC IS
  VARIABLE result : STD_ULOGIC := 'Z'; -- 默认的最弱状态
BEGIN
  IF s'LENGTH = 1 THEN
    RETURN s(s'LOW);
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END FUNCTION resolved;
```

在决断子类型、决断函数均已声明和定义的基础上,可以使用决断子类型来定义信号,如其他部分程序中所使用的 STD_LOGIC 和 STD_LOGIC_VECTOR。

3.6 子程序

VHDL 中子程序(Subprogram)包括过程(PROCEDURE)和函数(FUNCTION)。过程是一系列语句的集合,过程调用相当于一条独立的语句;而函数则是计算某个特定值,相当于定义了一种新的运算操作,其调用通常是放在赋值或者表达式中使用。

VHDL 子程序与其他软件语言程序中的子程序的应用目的是相似的,即能更有效地完成重复性的工作。子程序的使用方式只能通过子程序调用及与子程序的界面端口进行通信。子程序可以在 VHDL 程序的三个不同位置进行定义,即在程序包、结构体和进程中定义。由于只有在程序包中定义的子程序才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 子程序具有可重载的特点,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。子程序的调用分为顺序调用和并行调用,即在进程中调用和在结构体中调用。

3.6.1 过程

过程由过程首和过程体构成。过程首也不是必需的,过程体可以独立存在和使用。过程体如果在进程或结构体中定义,则不必定义过程首,而在程序包中必须定义过程首。过程名后面的参数表可以包括常数、变量和信号三类数据对象,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。对于 IN 参数,默认是常数;对于 OUT 参数,则默认是变量。过程调用可以在进程中,也可以在结构体中。调用方法为:直接写出过程名以及参数传递关系,将其看成是一条独立的语句去执行。

过程 PROCEDURE 的格式:

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

下面是在进程中定义过程的示例,该过程没有形参列表,过程的调用发生在进程中。

```
ARCHITECTURE rtl OF control_processor IS
    TYPE func_code IS (add, subtract);
    SIGNAL op1, op2, dest : INTEGER;
    SIGNAL Z_flag : BOOLEAN;
    SIGNAL func : func_code;
    ...
BEGIN
    alu : PROCESS IS
        PROCEDURE do_arith_op IS
            VARIABLE result : INTEGER;
        BEGIN
            CASE func IS
                WHEN add =>
                    result := op1 + op2;
                WHEN subtract =>
                    result := op1 - op2;
            END CASE;
            dest <= result AFTER Tpd;
            Z_flag <= result = 0 AFTER Tpd;
        END PROCEDURE do_arith_op;
        BEGIN
            ...
            do_arith_op;
        ...
    END PROCESS alu;
    ...
END ARCHITECTURE rtl;
```

多数过程均带有形参列表,假设在结构体的声明部分定义了一个过程 p(s1,s2, val1),其中,s1,s2 为输入信号,val1 为输入常数,那么,并行调用的过程调用发生在结构体中,如下所示:

```
call_proc : p ( s1, s2, val1 );
```

等效成一个进程,这意味着当信号 s1,s1 值发生变化,即有事件时,过程会被再次执行。需要强调的是,只有与 IN,INOUT 模式关联的信号参数才会被放在进程的敏感信号表中。

```
call_proc : PROCESS IS
BEGIN
    p ( s1, s2, val1 );
    WAIT ON s1, s2;
END PROCESS call_proc
```

3.6.2 函数

一般地,函数定义应由两部分组成,即函数首和函数体。

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,如果将所定义的函数组织成程序包入库的话,定义函数首是必需的。函数的参数表是用来定义输入值的,因此不必以显式表示参数的方向,函数参量可以是信号或常数,不能是变量。参数名需放在关键词 CONSTANT 或 SIGNAL 之后。如果没有特别说明,则参数被默认为常数。函数体的输出是用 RETURN 来指定的。如果要将一个已编制好的函数并入程序包,函数首必须放在程序包的说明部分,而函数体需放在程序包的包体内。函数体也可以在结构体或者进程的声明部分进行定义,则此时不需要定义函数首。函数调用可以在进程中,也可以在结构体中。调用方法为:直接写出函数名以及参数传递关系,将其看成是表达式的一部分去执行。

函数定义的一般格式:

```
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
[说明部分]
BEGIN
    顺序语句;
END FUNCTION 函数名
```

函数体定义举例:

```
FUNCTION limit ( value, min, max : INTEGER ) RETURN INTEGER IS
BEGIN
    IF value > max THEN
        RETURN max;
    ELSIF value < min THEN
        RETURN min;
    ELSE
        RETURN value;
    END IF;
END FUNCTION limit;
```

函数调用举例:

```
new_temperature := limit ( current_temperature + increment, 10, 100 );
new_motor_speed := old_motor_speed + scale_factor * limit ( error, -10, +10 );
```

在结构体中调用函数举例:

```
FUNCTION bv_add ( bv1, bv2 : IN BIT_VECTOR ) RETURN BIT_VECTOR IS
BEGIN
```

```
...
```

```
END FUNCTION bv_add;
```

在结构体中假设定义了两个信号：

```
SIGNAL source1, source2, sum : BIT_VECTOR(0 TO 31);
```

结构体完成并行调用函数的语句如下：

```
adder : sum <= bv_add(source1, source2) AFTER T_delay_adder;
```

3.6.3 函数/过程重载

实际应用时,函数/过程名通常是标识函数/过程所完成的操作,而其参数列表则是操作的对象。VHDL 允许针对不同数据类型的相同操作以相同的名字命名函数/过程。即同名函数/过程可以用不同的数据类型为其参数定义多次,以此定义的函数/过程称为重载函数/过程。这样可以确保在调用时,不会造成混淆,究竟调用哪个函数,由函数/过程的参数所对应的数据类型决定。如果函数是以“+”“-”“<”等运算符命名的,则称为运算符重载。

下面是一个完整的重载函数 max 的定义和调用的示例。

【代码 3-13】

```
LIBRARY IEEE
USE IEEE.STD_LOGIC_1164.ALL;
PACKAGE packexp IS
    FUNCTION max(a,b: IN STD_LOGIC_VECTOR)
        RETURN STD_LOGIC_VECTOR ;
    FUNCTION max(a,b: IN BIT_VECTOR) -- 定义函数首
        RETURN BIT_VECTOR ;
    FUNCTION max( a,b : N INTEGER ) -- 定义函数首
        RETURN INTEGER ;
END;
PACKAGE BODY packexp IS
    FUNCTION max( a,b : IN STD_LOGIC_VECTOR) -- 定义函数体
        RETURN STD_LOGIC_VECTOR IS
    BEGIN
        IF a > b THEN RETURN a; ELSE RETURN b; END IF;
    END FUNCTION max; -- 结束 FUNCTION 语句
    FUNCTION max ( a,b : IN INTEGER) -- 定义函数体
        RETURN INTEGER IS
    BEGIN
        IF a > b THEN RETURN a; ELSE RETURN b; END IF;
    END FUNCTION max; -- 结束 FUNCTION 语句
    FUNCTION max ( a,b : IN BIT_VECTOR)
        RETURN BIT_VECTOR IS
    BEGIN
        IF a > b THEN RETURN a; ELSE RETURN b; END IF;
    END FUNCTION max;
END; -- 结束 PACKAGE BODY 语句

LIBRARY IEEE ; -- 以下是调用重载函数 max 的程序;
USE IEEE.STD_LOGIC_1164.ALL ;
USE WORK.packexp.ALL
ENTITY axamp IS
```

```

PORT(a1,b1 : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
a2,b2 : IN BIT_VECTOR(4 DOWNTO 0);
a3,b3 : IN INTEGER RANGE 0 TO 15;
c1 : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
c2 : OUT BIT_VECTOR(4 DOWNTO 0);
c3 : OUT INTEGER RANGE 0 TO 15);
END;
ARCHITECTURE bhv OF axamp IS
BEGIN
    c1 <= max (a1,b1);          -- 对函数 max (a,b : IN STD_LOGIC_VECTOR)的调用
    c2 <= max (a2,b2);          -- 对函数 max (a,b : IN BIT_VECTOR)的调用
    c3 <= max (a3,b3);          -- 对函数 max (a,b : IN INTEGER)的调用
END;
```

3.7 设计库和标准程序包

库一般是一些常用 VHDL 代码的集合,包括:数据类型的定义、函数的定义、子程序的定义、元件引用声明、常量的定义等一些可复用或共享的 VHDL 代码。程序引用了库就可以使用该库中的 VHDL 代码。

库的声明格式:

```

LIBRARY 库名;
USE 库名.库中程序包.程序包中的项;
```

例如:

```

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

其中,IEEE 是库名,是 VHDL 设计中使用频率最高的库之一,包括一些常用数据类型的定义及相关的操作。IEEE 库有以下几个常用的程序包:

- (1) STD_LOGIC_1164 库定义了 STD_LOGIC 和 STD_ULOGIC 的数据类型。
- (2) STD_LOGIC_SIGNED 库定义了与 SIGNED 数据类型相关的函数。
- (3) STD_LOGIC_UNSIGNED 库定义了与 UNSIGNED 数据类型相关的函数。
- (4) STD_LOGIC_ARITH 库定义了一些不同类型数据之间相互转换的函数。

此外,IEEE 还包括 MATH_REAL、NUMERIC_BIT、NUMERIC_STD 等程序包,只用于算术运算,读者可根据实际需要进行引用。除了 IEEE 库外,比较常用的库还有标准库 STD 库、工作库 WORK 库。WORK 库是设计者自己定义的。而在标准库 STD 中的 STANDARD 程序包中规定了关于 BIT、BOOLEAN、CHARACTER、INTEGER 等最基础的数据类型,因此几乎每个 VHDL 设计程序都要使用。VHDL 规定标准库 STD 和工作库 WORK 是默认打开的。STD 库和 WORK 库隐含在每个 VHDL 程序中,也就是说不需要显示引用时就可以使用 STD 库和 WORK 库的 VHDL 代码。相当于所有的 VHDL 设计程序前面都默认添加下面两句。VHDL 将当前工程文件夹所在的位置默认为工作库 WORK。

```

LIBRARY STD, WORK;
USE STD. STANDARD. ALL;
```

除了设计库中的程序包之外,用户可以自行设计程序包。程序包 PACKAGE 包括包声

明和包主体两部分。

包声明中的声明区可以声明函数、过程、数据类型、子类型、常量、信号。包声明相当于定义了包的接口,帮助设计者了解包中都包含哪些内容。其中这里声明的信号是全局信号。包声明中声明项对于使用该包的实体而言,都是可见的。对于信号和类型及子类型而言,设计者需要完整的定义。而对于函数和过程,则设计者只需要知道函数首和过程首,以便调用,而不关心其具体实现过程。因此,只需要在此声明函数首和过程首;对于某些常量,如果并不需要将常量的具体值呈现给包的使用者,则可以在这里只声明常量,而将对其的赋值操作延时到包主体中完成。包声明格式如下:

```
PACKAGE 标识 IS
{ 包声明项 }
END [ PACKAGE ] [ 标识 ] ;
```

包主体是包声明的具体实现。包主体中的声明项对于使用该包的实体而言,是不可见的。所有在包声明中声明的函数和过程以及被延时赋值的常量,都需要在包主体中给出具体实现,并且要求其具体实现要包含与包声明中完全一致的函数首和过程首的声明,以便逐项确认。而如果在包声明中不包含函数和过程以及被延时赋值的常量,则可以省略包主体。除此之外,包主体中还可以声明新的类型、子类型、常量和函数与过程,但这些声明项对外不可见,通常它们是实现包声明中所声明的函数和过程所必需的项目。需要强调,在包主体中,是不能声明信号的。包主体的格式如下:

```
PACKAGE BODY 标识 IS
{ 包主体声明项 }
END [ PACKAGE BODY ] [ 标识 ] ;
```

下面是包的一个完整定义示例,这是一个 BIT_VECTOR 类型的有符号算数运算程序包,其中定义了关于 BIT_VECTOR 类型的有符号算数运算的一系列函数。

【代码 3-14】

```
PACKAGE bit_vector_signed_arithmetic IS -- 包声明
    FUNCTION " + " ( bv1, bv2 : BIT_VECTOR )
        RETURN BIT_VECTOR;
    FUNCTION " - " ( bv : BIT_VECTOR )
        RETURN BIT_VECTOR;
    FUNCTION " * " ( bv1, bv2 : BIT_VECTOR )
        RETURN BIT_VECTOR;
    ...
END PACKAGE bit_vector_signed_arithmetic;
PACKAGE BODY bit_vector_signed_arithmetic IS -- 包体
    FUNCTION " + " ( bv1, bv2 : BIT_VECTOR )
        RETURN BIT_VECTOR IS
    ...
    FUNCTION " - " ( bv : BIT_VECTOR )
        RETURN BIT_VECTOR IS ...
    FUNCTION mult_unsigned ( bv1, bv2 : BIT_VECTOR )
        RETURN BIT_VECTOR IS
    ...
BEGIN
    ...
END FUNCTION mult_unsigned;
```

-- 包内的声明区

```

FUNCTION "*" ( bv1, bv2 : bit_vector )
    RETURN BIT_VECTOR IS
BEGIN
    IF NOT bv1(bv1'LEFT) AND NOT bv2(bv2'LEFT) THEN
        RETURN mult_unsigned(bv1, bv2);
    ELSIF NOT bv1(bv1'LEFT) AND bv2(bv2'LEFT) THEN
        RETURN - mult_unsigned(bv1, - bv2);
    ELSIF bv1(bv1'LEFT) AND NOT bv2(bv2'LEFT) THEN
        RETURN - mult_unsigned(- bv1, bv2);
    ELSE
        RETURN mult_unsigned(- bv1, - bv2);
    END IF;
END FUNCTION "*";

...

END PACKAGE BODY bit_vector_signed_arithmetic;

```

代码 3-15 和代码 3-16 则是两个自定义包的典型应用实例。请读者自行阅读并理解。

【代码 3-15】

```

PACKAGE my_pack IS -- 包声明
    TYPE nineval IS (Z0, Z1, ZX, R0, R1, RX, F0, F1, FX);
    TYPE nvector2 IS ARRAY(0 TO 1) OF nineval;
    TYPE fourstate IS (X, L, H, Z);
    FUNCTION convert4state(a : fourstate)
        RETURN nineval;
    FUNCTION convert9val(a : nineval)
        RETURN fourstate;
END my_pack;

PACKAGE body my_pack IS -- 包体
    FUNCTION convert4state(a : fourstate)
        RETURN nineval IS
    BEGIN
        CASE a IS
            WHEN X =>
                RETURN FX;
            WHEN L =>
                RETURN F0;
            WHEN H =>
                RETURN F1;
            WHEN Z =>
                RETURN ZX;
        END CASE;
    END convert4state;

    FUNCTION convert9val(a : nineval)
        RETURN fourstate IS
    BEGIN
        CASE a IS
            WHEN Z0 =>
                RETURN Z;
            WHEN Z1 =>
                RETURN Z;
            WHEN ZX =>
                RETURN Z;
            WHEN R0 =>
                RETURN L;
            WHEN R1 =>

```

```

        RETURN H;
    WHEN RX =>
        RETURN X;
    WHEN F0 =>
        RETURN L;
    WHEN F1 =>
        RETURN H;
    WHEN FX =>
        RETURN X;
    END CASE;
END convert9val;
END my_pack;

USE WORK.my_pack.ALL;
ENTITY trans2 IS
    PORT( a, b : INOUT nvector2;
          enable : IN nineval);
END trans2;
ARCHITECTURE struct OF trans2 IS
    COMPONENT trans
        PORT( x1, x2 : INOUT fourstate;
              en : IN fourstate);
    END COMPONENT;
BEGIN
    U1 : trans PORT MAP(
        convert4state(x1) => convert9val(a(0)),
        convert4state(x2) => convert9val(b(0)),
        en => convert9val(enable) );
    U2 : trans PORT MAP(
        convert4state(x1) => convert9val(a(1)),
        convert4state(x2) => convert9val(b(1)),
        en => convert9val(enable) );
END struct;

```

【代码 3-16】

```

PACKAGE my_std IS
    TYPE fourval IS (X, L, H, Z);
    TYPE fourvalue IS ('X', '0', '1', 'Z');
    TYPE fvector4 IS ARRAY(0 TO 3) OF fourval;
END my_std;
USE WORK.my_std.ALL;
ENTITY reg IS
    PORT(a : IN fvector4;
          clr : IN fourval;
          clk : IN fourval;
          q : OUT fvector4);
    FUNCTION convert4val(S : fourval)
        RETURN fourvalue IS
    BEGIN
        CASE S IS
            WHEN X =>
                RETURN 'X';
            WHEN L =>
                RETURN '0';
            WHEN H =>
                RETURN '1';

```

```

        WHEN Z =>
            RETURN 'Z';
        END CASE;
    END convert4val;
    FUNCTION convert4value(S : fourvalue)
        RETURN fourval IS
    BEGIN
        CASE S IS
            WHEN 'X' =>
                RETURN X;
            WHEN '0' =>
                RETURN L;
            WHEN '1' =>
                RETURN H;
            WHEN 'Z' =>
                RETURN Z;
        END CASE;
    END convert4value;
END reg;
ARCHITECTURE structure OF reg IS
    COMPONENT dff
        PORT(d, clk, clr : IN fourvalue;
            q : OUT fourvalue);
    END COMPONENT;
BEGIN
    U1 : dff PORT MAP(convert4val(a(0)),
        convert4val(clk),
        convert4val(clr),
        convert4value(q) => q(0));
    U2 : dff PORT MAP(convert4val(a(1)),
        convert4val(clk),
        convert4val(clr),
        convert4value(q) => q(1));
    U3 : dff PORT MAP(convert4val(a(2)),
        convert4val(clk),
        convert4val(clr),
        convert4value(q) => q(2));
    U4 : dff PORT MAP(convert4val(a(3)),
        convert4val(clk),
        convert4val(clr),
        convert4value(q) => q(3));
END structure;

```