第 1 部分

VHDL 代码编写

第1章 绪 言

1.1 传统设计方法——原理图输入

自从20世纪80年代，原理图输入这种设计方法被引入VLSI设计领域之后，它就是一种一直被广泛采用的设计形式。时至今日，工业界仍有很多设计机构沿用这种设计思路。

原理图输入的设计思路，就是采用计算机辅助设计(computer aided design，CAD)的手段，手工绘制电路中的各个逻辑门。手工绘制的原理图绘制完毕后，将保存成数据库文件。数据库的格式一般是电子设计交换格式(electronic design interchange format，EDIF)。

根据此数据库，设计者使用仿真工具对设计进行仿真，并根据仿真结果确定设计对象的功能是否正确。如果功能不正确，设计者就必须编辑、修改手工绘制的原理图并重新进行仿真。这一过程不断地重复，直至设计对象的功能正确得让设计者满意为止。

在20世纪80年代，原理图输入是一种不错的设计方法。但是到了90年代，当设计所采用的逻辑门上升成数十万门之际，这种设计方法就遭遇到了惨重的失败。当IC芯片日益复杂、逻辑门数目随着这种复杂度的增长而惊人地增长时，原理图输入的设计方法对于设计者来说已不再是一种用来协助设计的手段，而越来越成为一种障碍。

况且，在当今的技术世界，新款IC芯片在市场中将很快就会被淘汰掉。IC芯片销售和盈利的“市场窗口”越变越窄。正是这种不断变窄的“市场窗口”迫使设计者在更短的时间期限内设计出产品，以便以更快的速度将更好的产品投入市场。所以设计者需要更快、更有效地进行工作。

在这种情形下，硬件描述语言应运而生了。

1.2 硬件描述语言

随着技术的迅猛发展，电子设计自动化(electronic design automation，EDA)软件在最近几年中正以难以置信的速度发展着。这种发展不断带来更快、更有效的IC逻辑芯片设计手段，而这都是通过使用硬件描述语言实现的。

当今，在使用中最受欢迎的硬件描述语言是VHDL(超高速集成电路硬件描述语言)和Verilog。本书将从代码编写和逻辑综合两方面来介绍VHDL。

采用VHDL进行设计，是为设计对象编写VHDL代码，而不是采用传统的原理图输入

方法。随后,设计人员对代码使用 VHDL 综合工具加以综合。综合出的电路可以保存在网表数据库中。

一般用来实现 VHDL 综合的工具有:Synopsys(新思)公司的 Design Compiler、Mentor Graphics(明导电子)公司的 Autologix、Exemplar 公司的 Exemplar、Synplicity 公司的 Synplify、Cadence 公司的 Ambit 以及其他一些软件。本书将只讨论如何使用 Synopsys 公司的 Design Compiler 去综合和改进 VHDL 代码。

1.3 VHDL 设计的结构

在接下去讨论 VHDL 综合之前,有必要对 VHDL 设计的结构有一个基本认识。

VHDL 设计由实体(*entity*)、结构体(*architecture*)和配置说明(*configuration*)三部分组成:

(1) **实体**——此部分用于声明设计对象的输入端口、输出端口和双向端口;

(2) **结构体**——此部分中编写的代码用于描述设计对象的内部结构;

(3) **配置说明**——此部分用于声明设计对象内各个子模块所使用的实体和结构体。

一般而言,VHDL 源文件看上去类似于以下这个模板:

```
LIBRARY <library_name> ;
USE <library_name> .<sub_library_name> .ALL;

ENTITY <entity_name>  IS
    PORT (
          <port_name>  : <port_direction>  <port_type> ;
          <port_name>  : <port_direction>  <port_type>
    );
END <entity_name> ;

ARCHITECTURE <architecture_name>  OF <entity_name>  IS
SIGNAL <signal_name>  : <signal_type>  :="<starting_value> ";
BEGIN
......自己编写的 VHDL 代码......
END <architecture_name> ;

CONFIGURATION <configuration_name>  OF <entity_name>  IS
    FOR <architecture_name> inputs
    ......各个元件的配置说明......
    END FOR;
END <configuration_name> ;
```

其中：

(1) <library_name> 是将使用的库的名字，如 IEEE；

(2) <sub_library_name> 是将使用的子库的名字，如 std_logic_1164；

(3) <entity_name> 是实体名；

(4) <port_name> 是设计对象的端口名；

(5) <port_direction> 是端口方向声明，其取值可以是 IN、OUT、INOUT 和 BUFFER；

(6) <port_type> 是端口类型声明，其取值可以是 std_logic、std_logic_vector、bit、bit_vector 以及其他在 IEEE std_logic_1164 库中所定义的类型；

(7) <architecture_name> 是结构体名；

(8) <signal_name> 是信号名；

(9) <signal_type> 是信号类型声明，其取值可以是 std_logic、std_logic_vector、bit、bit_vector 以及其他在 IEEE std_logic_1164 库中所定义的类型；

(10) <starting_value> 是信号的初始值，它是个可选参数；

(11) <configuration_name> 是设计对象配置说明的名字。

每一个 VHDL 设计都只能具有一个实体声明，但可以具有多种结构体声明和多种配置说明。

大多数设计只具有一个实体、一种结构体和一种配置说明。图 1 是只有一个结构体的设计。这种编码方式既简单、又直观，因此提倡采用这种方式。此外，在当今很多流行的 VHDL 仿真和综合工具中，如果设计对象只具有一种结构体声明而没有任何元件实例，则不必总是对其进行配置说明。

如果打算对代码的构成作更多的控制，那么指定的实体可以具有多种结构体声明(如图 2 所示)。但是，每一种结构体声明都必须有与其相对应的配置说明。不过，在可综合 VHDL 代码中极少使用多种结构体声明。

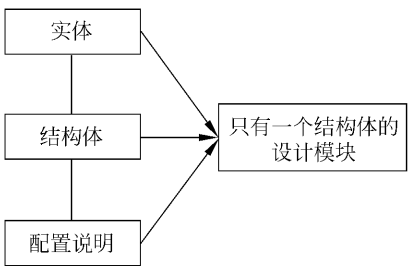

图 1　只有一个结构体的设计

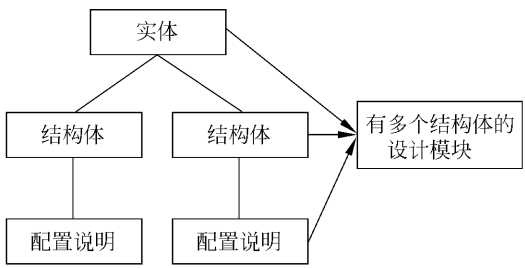

图 2　具有多个结构体的设计

例 1 表明实体 ***multiple_arch_ent*** 具有两种结构体声明：***multiple_arch_ent_arch1*** 和 ***multiple_arch_ent_arch2***；而 ***config_1*** 和 ***config_2*** 分别是与这两种结构体声明相对应的配置说明。

例 1 具有多种结构体的 VHDL 代码

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;


ENTITY multiple_arch_ent IS
    PORT (
         input : IN std_logic;
        output : OUT std_logic
    );
END multiple_arch_ent;


ARCHITECTURE multiple_arch_ent_arch1 OF multiple_arch_ent IS
BEGIN

  --自己编写的结构体代码

END multiple_arch_ent_arch1;

ARCHITECTURE multiple_arch_ent_arch2 OF multiple_arch_ent IS
BEGIN

  --自己编写的另一个结构体代码

END multiple_arch_ent_arch2;


CONFIGURATION config_1 OF multiple_arch_ent IS
    FOR multiple_arch_ent_arch1
    END FOR;
END config_1;

CONFIGURATION config_2 OF multiple_arch_ent IS
    FOR multiple_arch_ent_arch2
    END FOR;
END config_2;
```

声明使用 *IEEE* 库

声明端口方向和类型（std_logic）

第一种结构体声明

第二种结构体声明

第一种配置声明

第二种配置声明

注意：层次化的设计基于同样的由实体、结构体和配置说明所构成的 VHDL 结构。层次关系中的每一个子模块(包括顶层设计对象)都使用实体、结构体和配置声明来表示。在

顶层 VHDL 代码中实例化出各个子模块。如果子模块由更低层次的子模块所组成，则在这些子模块都要用实体、结构体和配置说明来表示，在子模块中对它们进行实例化，并用(胶黏逻辑)将它们粘接在一起。

在图 3 中，A、B 和 C 都是 D 的子模块，而 D 和 E 又都是顶层模块的子模块。

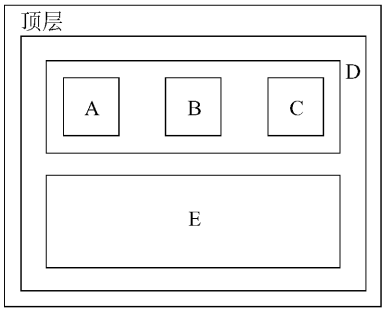

图 3　层次化的设计对象

1.4　VHDL 设计内的元件实例化

对于含有元件实例化的设计，必须为其结构体中的每一种元件声明相应的配置说明。对于图 4 的设计模块，例 2 表明用来实例化出各个元件对应的 VHDL 代码。

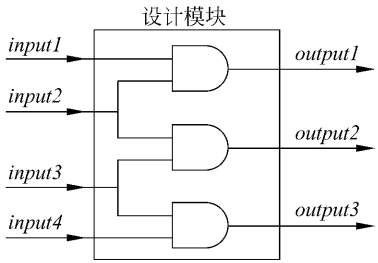

图 4　实例化出 3 个与门元件的设计

例 2　实例化出 3 个与门元件设计的 VHDL 代码

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;


ENTITY design_module_ent IS
    PORT (
```

声明使用 *IEEE* 库

```
        input1 : IN std_logic;
        input2 : IN std_logic;
        input3 : IN std_logic;
        input4 : IN std_logic;
        output1 : OUT std_logic;
        output2 : OUT std_logic;
        output3 : OUT std_logic
    );
END design_module_ent;


ARCHITECTURE design_module_arch OF design_module_ent IS

COMPONENT AND_gate_ent
    PORT (
        AND_input1 : IN std_logic;
        AND_input2 : IN std_logic;
        AND_output : OUT std_logic
    );
END COMPONENT;

BEGIN
    logic_AND_1: AND_gate_ent PORT MAP (input1, input2, output1);
    logic_AND_2: AND_gate_ent PORT MAP (input2, input3, output2);
    logic_AND_3: AND_gate_ent PORT MAP (input3, input4, output3);
END design_module_arch;


CONFIGURATION design_module_config OF design_module_ent IS
    FOR design_module_arch
        FOR logic_AND_1: AND_gate_ent
            USE ENTITY WORK.AND_gate_ent(AND_gate_arch);
        END FOR;
        FOR logic_AND_2: AND_gate_ent
            USE ENTITY WORK.AND_gate_ent(AND_gate_arch);
        END FOR;
        FOR logic_AND_3: AND_gate_ent
            USE ENTITY WORK.AND_gate_ent(AND_gate_arch);
        END FOR;
    END FOR;
END design_module_config;
```

声明端口方向和类型

声明 AND_gate_ent 元件的使用方法

对元件实例作端口映射

元件实例的配置说明

注意：例2中的代码假定在 ***WORK*** 库中存在事先编译出的一种与门，其实体名为 ***AND_gate_ent***、结构体名为 ***AND_gate_arch***。

将实体名为 ***AND_gate_ent***、结构体名为 ***AND_gate_arch*** 的这种与门事先编译至 ***WORK*** 库的步骤如下：

(1) 建立 ***WORK*** 目录，并在所使用的 VHDL 仿真/综合工具中将其链接成 VHDL 库；

(2) 建立另一个名为 ***SOURCE*** 的目录；

(3) 在 ***SOURCE*** 目录中建立〈filename〉文件；

(4) 在〈filename〉文件中，键入以下内容：

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY AND_gate_ent IS
    PORT (
        AND_input1 : IN std_logic;
        AND_input2 : IN std_logic;
        AND_output1 : OUT std_logic
    );
END AND_gate_ent;

ARCHITECTURE AND_gate_arch OF AND_gate_ent IS
BEGIN
    output1 <=input1 AND input2;
END AND_gate_arch;
```

(5) 通过使用第1步所定义的 VHDL 库，将〈filename〉文件编译至 ***WORK*** 目录；

(6) 执行完这5步，就得到了所定义的 VHDL 库，并已链接至事先编译出的 ***AND_gate_ent*** 元件所在的 ***WORK*** 目录中；

(7) 还应当注意，对于不同的软件，定义和链接 VHDL 库的步骤不同，因此需要使用者查阅所使用的工具软件的使用手册。

以下是对例2中的模板所作的简要说明：

(1) 前两行代码声明将使用 ***IEEE*** 库中的 ***std_logic_1164*** 子库；

(2) 实体部分声明了 std_logic 类型的输入端口（***input1***、***input2***、***input3*** 和 ***input4***）和输出端口（***output1***、***output2*** 和 ***output3***）；

(3) 结构体部分实例化出三个与门元件，它们的实体名都是 ***AND_gate_ent***、结构体名都是 ***AND_gate_arch***；

(4) 配置说明部分声明了用来实例化出各个元件的配置说明。

1.5 结构式、行为式与可综合 VHDL 设计

VHDL 设计一般可以分为三类，即结构式、行为式和可综合 VHDL 设计，它们各自具有不同特征和代码编写方式。本节将引用图 5 中的电路，分别说明这三类设计在代码编写方式上的差别。

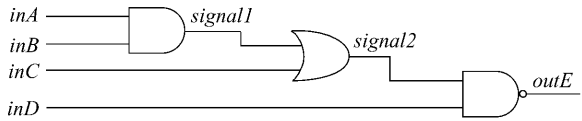

图 5 “某一模块”的原理图

1.5.1 结构式 VHDL 描述

结构式 VHDL 描述是最适合于将设计对象和原理图描述成网表的数据式结构。它声明了设计中所使用的每一种类型的元件以及各个元件间的连接关系。

例 3 “某一模块”的结构式 VHDL 描述

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;


ENTITY structural_code_ent IS
    PORT (
        inA : IN std_logic;
        inB : IN std_logic;
        inC : IN std_logic;
        inD : IN std_logic;
        outE : OUT std_logic
    );
END structural_code_ent;


ARCHITECTURE structural_code_arch OF structural_code_ent IS

COMPONENT AND_gate_ent
    PORT (
```

```
        AND_input1 : IN std_logic;
        AND_input2 : IN std_logic;
        AND_output : OUT std_logic
    );
END COMPONENT;

COMPONENT OR_gate_ent
    PORT (
        OR_input1 : IN std_logic;
        OR_input2 : IN std_logic;
        OR_output : OUT std_logic
    );
END COMPONENT;

COMPONENT NAND_gate_ent
    PORT (
        NAND_input1 : IN std_logic;
        NAND_input2 : IN std_logic;
        NAND_output : OUT std_logic
    );
END COMPONENT;

SIGNAL signal1, signal2 : std_logic;

BEGIN
    logic_AND: AND_gate_ent
        PORT MAP (inA, inB, signal1);
    logic_OR: OR_gate_ent
        PORT MAP (signal1, inC, signal2);
    logic_NAND: NAND_gate_ent
        PORT MAP (signal2, inD, outE);
END structural_code_arch;


CONFIGURATION structural_code_config OF structural_code_ent IS
    FOR structural_code_arch
        FOR logic_AND: AND_gate_ent
            USE ENTITY WORK.AND_gate_ent(AND_gate_arch);
        END FOR;
        FOR logic_OR: OR_gate_ent
            USE ENTITY WORK.OR_gate_ent(OR_gate_arch);
```

```
        END FOR;
        FOR logic_NAND: NAND_gate_ent
            USE ENTITY WORK.NAND_gate_ent(NAND_gate_arch);
        END FOR;
    END FOR;
END structural_code_config;
```

注意：例 3 中的代码假定 ***WORK*** 库具有事先编译出的与门、或门和与非门。

1.5.2 行为式 VHDL 描述

行为式 VHDL 描述就是以行为方式去描述设计对象，从而模仿其行为和功能。行为式编码描述的设计对象就像个黑匣子，让给定的一组输入信号产生指定的各种输出信号。这种形式编写出的代码是不可综合的，正常情况下只用来测试系统。

使用图 5 建立表示功能的真值表(如表 1 所示)。在功能真值表中，输出值被映射到各个输入值上，然后根据这种映射关系去编写行为式代码。

表 1 “某一模块”的真值表

inA	*inB*	*inC*	*inD*	*signal1*	*signal2*	*outE*
0	0	0	0	0	0	1
0	0	0	1	0	0	1
0	0	1	0	0	1	1
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	0	0	1
0	1	1	0	0	1	1
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	0	1	1
1	0	1	1	0	1	0
1	1	0	0	1	1	1
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	0

例 4 “某一模块”的行为式代码

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

```
ENTITY behavior_ent IS
    PORT (
        inA : IN std_logic;
        inB : IN std_logic;
        inC : IN std_logic;
        inD : IN std_logic;
        outE : OUT std_logic
    );
END behavior_ent;


ARCHITECTURE behavior_arch OF behavior_ent IS
BEGIN①

outE <='0' WHEN (((inA =  '0') AND (inB =  '0') AND (inC =  '1') AND (inD =  '1'))
            OR ((inA ='0') AND (inB ='1') AND (inC ='1') AND (inD ='1'))
            OR ((inA ='1') AND (inB ='0') AND (inC ='1') AND (inD ='1'))
            OR ((inA ='1') AND (inB ='1') AND (inC ='0') AND (inD ='1'))
            OR ((inA ='1') AND (inB ='1') AND (inC ='1') AND (inD ='1')))
  ELSE '1' WHEN(((inA ='0') AND (inB ='0') AND (inC ='0') AND (inD ='0'))
            OR ((inA ='0') AND (inB ='0') AND (inC ='0') AND (inD ='1'))
            OR ((inA ='0') AND (inB ='0') AND (inC ='1') AND (inD ='0'))
            OR ((inA ='0') AND (inB ='1') AND (inC ='0') AND (inD ='0'))
            OR ((inA ='0') AND (inB ='1') AND (inC ='0') AND (inD ='1'))
            OR ((inA ='0') AND (inB ='1') AND (inC ='1') AND (inD ='0'))
            OR ((inA ='1') AND (inB ='0') AND (inC ='0') AND (inD ='0'))
            OR ((inA ='1') AND (inB ='0') AND (inC ='0') AND (inD ='1'))
            OR ((inA ='1') AND (inB ='0') AND (inC ='1') AND (inD ='0'))
            OR ((inA ='1') AND (inB ='1') AND (inC ='0') AND (inD ='0'))
            OR ((inA ='1') AND (inB ='1') AND (inC ='1') AND (inD ='0')))
    ELSE 'H' WHEN ((inA ='X') OR (inB ='X') OR (inC ='X') OR (inD ='X'))
    ELSE 'L' WHEN ((inA ='U') OR (inB ='U') OR (inC ='U') OR (inD ='U'))
    ELSE 'Z' AFTER 1.5 ns; -- 传播延迟为 1.5ns

PROCESS (inA, inB, inC, inD)
BEGIN
    IF ((inA ='U') OR (inB ='U') OR (inC ='U') OR (inD ='U')) THEN
```

这是进程敏感表，请参见代码后的“注意”

① 原书对 outE 的赋值代码错误。它是 4 种情况取值为 0,12 种情况取值为 1,而且赋值条件也错了。从表 1 中可以看出有 5 种情况取值为 0,11 种情况取值为 1,因此作了相应改正。——译者注

```
        ASSERT FALSE
        REPORT "One of the inputs is at 'U'. Output is driven as L."
        SEVERITY WARNING;
    ELSIF ((inA = 'X') OR (inB = 'X') OR (inC = 'X') OR (inD = 'X')) THEN
            ASSERT FALSE
            REPORT "One of the inputs is at 'X'. Output is driven as H."
            SEVERITY WARNING;
    END IF;
END PROCESS;

END behavior_arch;
```

注意：敏感表是由那些值一旦发生变化就会触发执行**时序**进程的信号所组成的表。因此，重要的是在于将**进程**中所用到的每一个信号都在敏感表中列出来。

1.5.3 RTL 级代码

寄存器传输级(register transfer level，RTL)代码是最为复杂的编码形式，这是因为它是在较高(抽象)层次上采用 VHDL 语法子集描述设计对象。这种编码形式介于结构式和行为式之间。相对于结构式 VHDL 代码，它具有更高的抽象层次；而相对于行为式 VHDL 代码，它处于较低的抽象层次。

例 5 “某一模块”的描述式 VHDL 代码

```
LIBRARY IEEE;                                    ← 声明使用 IEEE 库
USE IEEE.std_logic_1164.ALL;



ENTITY descriptive_ent IS
    PORT (
        inA : IN std_logic;
        inB : IN std_logic;
         inC : IN std_logic;                     ← 声明端口方向和类型
        inD : IN std_logic;
        outE : OUT std_logic
    );
END descriptive_ent;



ARCHITECTURE descriptive_arch OF descriptive_ent IS
BEGIN
```

```
    PROCESS (inA, inB, inC, inD)
    BEGIN
        IF (((inA = '0') AND (inB = '0') AND (inC = '1') AND (inD = '1')) OR
           ((inA = '0') AND (inB = '1') AND (inC = '1') AND (inD = '1')) OR
           ((inA = '1') AND (inB = '0') AND (inC = '1') AND (inD = '1')) OR
           ((inA = '1') AND (inB = '1') AND (inC = '0') AND (inD = '1')) OR
           ((inA = '1') AND (inB = '1') AND (inC = '1') AND (inD = '1'))) THEN
               outE <= '0';
        ELSE
            outE <= '1';
        END IF;
    END PROCESS;
END descriptive_arch;
```

以可综合代码描述“某一模块”

注意：编写RTL代码有很多种不同的方式。例5是RTL代码编写方式的示例。但是，编写例5的代码还有更简单、更有效率的方式。不同设计者往往会采用不同的代码编写方式。用来编写例5的代码的另一种方式如下：

```
PROCESS (inA, inB, inC, inD)
BEGIN
    IF ((((inA = '1') AND (inB = '1')) OR (inC = '1')) AND (inD = '1')) THEN
        OutE <= '0';
    ELSE
        OutE <= '1';
    END IF;
END PROCESS;
```

您是否已经注意到以上这种代码编写方式与图5的组合逻辑门很相似？

对例5的代码进行综合就能得到图5所示的组合逻辑门。

为了确认从例5综合出的结果是最优综合结果，可以从表1中推出其卡诺图（如表2所示）。

表2 “某一模块”的卡诺图

inA & inB					
		00	01	11	10
inC&inD	00	1	1	1	1
	01	1	1	0	1
	11	0	0	0	0
	10	1	1	1	1

可优化成
NOT (D(AB+C))

此卡诺图的优化输出表明其具有与图 5 相同的结果。

在实际设计中，人们只需要关心如何编写出可综合 VHDL 代码，然后使用综合工具，如 Synopsys 的 Design Compiler，让工具自动去实现综合优化。第 8 章将详细介绍用来获得优化的综合结果的各种综合方法。

1.6 在 VHDL 设计中使用库声明

您可能会注意到：前几个例子的 VHDL 代码均以下列这两行代码开头：

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
```

这两行代码声明将使用 ***std_logic_1164*** 库中的函数和过程。

std_logic_1164 库还声明了 VHDL 语言中用来声明端口、信号和变量的各种类型。例如：

(1) ***BIT*** ——此类型仅取值 0 和 1；

(2) ***Boolean*** ——此类型仅取值 *TRUE* 和 *FALSE*；

(3) ***Integer*** ——此类型的取值范围是 $-(2^{31}-1)$ 至 $(2^{31}-1)$ 之间的整数；

(4) ***std_ulogic*** ——此类型具有下列 9 种不同的取值：

- 1——强逻辑 1
- 0——强逻辑 0
- H——弱‘1’
- L——弱‘0’
- X——强不定态
- U——未初始化态
- Z——高阻态
- －——无关值
- W——弱不定态

(5) ***std_logic*** ——这是解析式(resolved)的 ***std_ulogic*** 类型，是可综合代码中最常用的类型。建议在编写可综合 VHDL 代码的时候尽量只使用一种类型，这样做能够减少将各个模块集成到一起的时候使用类型转换函数的必要性。

std_logic_1164 库之外，***std_logic_arith*** 是另一个常用的 ***IEEE*** 库。

std_logic_arith 库中含有编写 VHDL 代码时十分有用的各种函数，尤其是在将具有不同类型的端口和信号的各个模块和子模块集成在一起的时候。例如，设计人员在某些场合下需要使用转换端口和信号类型的转换函数。如，***CONV_INTEGER*** 函数可以将信号转

换成 ***INTEGER*** 类型；又如，***CONV_STD_LOGIC_VECTOR*** 函数可以将信号转换成 ***STD_LOGIC_VECTOR*** 类型。

例 6 表明如何在设计中使用 ***CONV_STD_LOGIC_VECTOR*** 函数将 ***internal*** 信号从 ***INTEGER*** 类型转换成 ***STD_LOGIC_VECTOR*** 类型。

例 6　编写可综合 VHDL 代码时使用类型转换函数

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;                    ← 声明使用 IEEE 库
USE IEEE.std_logic_arith.ALL;



ENTITY convert_ent IS
    PORT (
        input : IN std_logic_vector (1 downto 0);
        ......其他输入端口......
        output : OUT std_logic_vector(3 downto 0);   ← 声明端口方向和类型
        ......其他输出端口......
    );
END convert_ent;



ARCHITECTURE convert_arch OF convert_ent
SIGNAL internal : INTEGER;                      ← 将 internal 声明成 integer 类型
BEGIN
......
    PROCESS (......敏感表......)
    BEGIN
......对内部信号进行赋值的 VHDL 代码......
    END PROCESS;
......
    output <=CONV_STD_LOGIC_VECTOR(internal, 4);   ← 将整数转换成 4 位 std_logic_vector 类型
END convert_arch;
```

为了更好地了解 ***IEEE*** 库，请参阅附录 A。该附录含有 ***std_logic_1164*** 库的 VHDL 代码。

第 2 章　VHDL 仿真与综合流程

HDL 仿真与综合的一般流程(见图 6)可以分成三个主要环节：

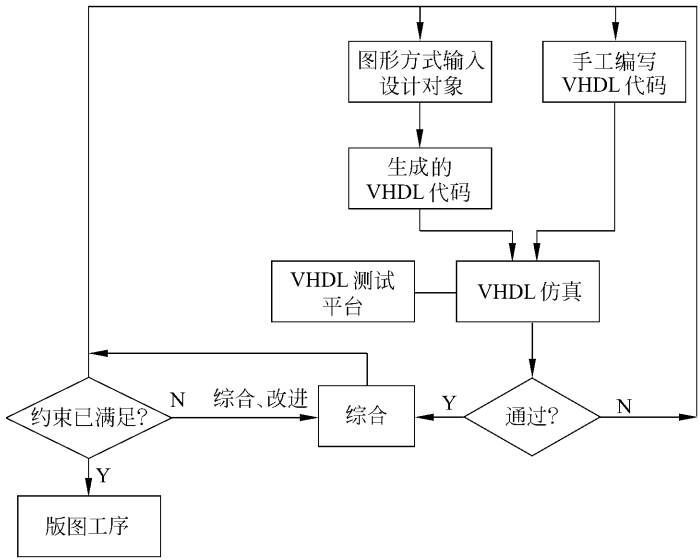

图 6　HDL 仿真与综合的常规流程

(1) **代码编写**　这是设计流程的开端,在此阶段为特定的设计对象编写代码。代码既可以手工编写,也可以由工具软件生成。在用来生成 HDL 代码的软件中,使用较多的是 Summit 公司的 Visual HDL 和 Mentor Graphics 公司的 Renoir。

(2) **仿真**　编写完 HDL 代码之后,设计者准备好仿真设计对象后,就可以对设计进行仿真。设计流程中的这一环节专门核对所编写的代码功能是否正确。在仿真过程中,用来检验设计对象功能的一般作法是使用测试平台(请参见图 7)。测试平台环绕着设计对象,向其注入输入激励,同时监听输出波形是否为预期波形。如果输出信号与预期波形不相符,则表明设计中有错误。出现这种情况之后,就必须返回到 HDL 代码编写阶段,修改其代码以解决输出信号与预期波形不相符的问题。仿真和重新编写代码这两种行为构成一个循环往复的过程,直至设计对象的输出信号与预期波形相符为止。

(3) **综合**　在完成仿真工作、对设计对象满意之后,下一步将是对设计对象进行综合。综合也要经历一个循环往复的过程,直至综合出的设计满足其设计规约。用来评估综合结果的指标中更常用的是性能和面积使用量。如果不能满足这些指标,则必须再次使用综合

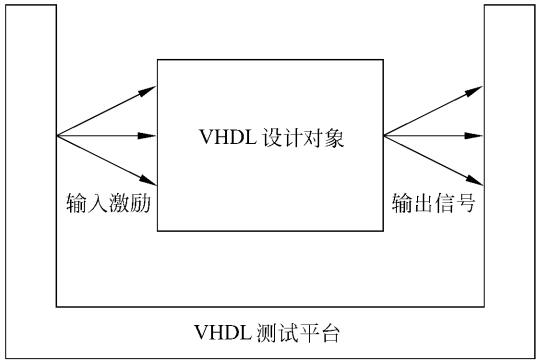

图 7　VHDL 设计对象及其测试平台

工具对综合出的数据库作更进一步优化。如果优化后仍然无法满足所提出的指标，则必须去修改设计所采用的微结构实现方案。此后，设计者接下去将重新编写 HDL 代码。在各项约束都得以满足之后，就可以将综合出的数据库和时序要求交给版图工序。