# 第二章 硬件描述语言VHDL

# 2.1 VHDL语言简介

## 2.1.1 VHDL的特点

20世纪80年代以来，采用计算机辅助设计(CAD)技术设计硬件电路在全世界范围得到了普及和应用。一开始，仅用 CAD 来实现印刷板的布线，以后慢慢地才实现了插件板级规模的设计和仿真，其中最具代表性的设计工具是OrCAD和Tango，它们的出现使电子电路设计和印刷板布线工艺实现了自动化。但这种设计方法就其本身而言仍是自下而上的设计方法，即利用已有的逻辑器件来构成硬件电路，它没有脱离传统的硬件设计思路。

随着集成电路规模与复杂度的进一步提高，特别是大规模、超大规模集成电路的系统集成，使得电路设计不断向高层次的模块式的设计方向发展，原有的电原理图输入方式显得不够严谨规范，过多的图纸和底层细节不利于从总体上把握和交流设计思想。再者，自下而上的设计方法使仿真和调试通常只能在系统硬件设计后期才能进行，因而系统设计时存在的问题只有在后期才易发现。这样，一旦系统设计存在较大缺陷，就有可能要重新设计系统，使得设计周期大大增加。

基于以上电原理图输入方式的缺陷，为了提高开发效率，增加已有成果的可继承性并缩短开发时间，大规模 ASIC研制和生产厂家相继开发了用于各自目的的硬件描述语言。 其中最具代表性的就是美国国防部开发的VHDL语言和 Verilog公司开发的Verilog HDL以及日本电子工业振兴协会开发的UDL/I语言。

从工程的角度看，VHDL独立于各种电子CAD手段之外，它便于学习、 继承、 管理。美国国防部曾硬性规定： 所有牵涉ASIC设计的电子系统合同项目都要用VHDL语言设计和存档， VHDL的地位和作用由此可见一斑。

1987年12月10日，IEEE标准化组织发布IEEE标准的VHDL，定为IEEE 1076 - 1987标准(该标准是从1983年8月美国空军支持并开发的VHDL 7.2版发展而来的)。这使得VHDL成为当时惟一被IEEE标准化的HDL语言，标志着VHDL被电子系统设计 行业普 遍接收并推广为标准的HDL语言。许多公司因而纷纷使自己的开发工具与VHDL 兼容。

利用VHDL语言设计数字系统具有以下4个优点：

(1) 设计技术齐全、方法灵活、支持广泛。VHDL语言可以支持自上而下和基于库的设计方法， 还支持同步电路、异步电路、 FPGA以及其它随机电路的设计。目前大多数EDA工具几乎在不同程度上都支持VHDL语言，这给VHDL语言进一步推广和应用创造了良好的环境。

(2) 系统硬件描述能力强。VHDL具有多层次描述系统硬件功能的能力，其描述的对象可以从系统的数学模型直到门级电路。

(3) VHDL语言可以与工艺无关编程。VHDL设计硬件系统时，可以编写与工艺有关的信息，但是，与大多数HDL语言不同的是，当门级或门级以上层次的描述通过仿真验证后，可以用相应的工具将设计映射成不同的工艺(如MOS、 CMOS等)。这样，工艺更新时，就无须修改程序，只须修改相应的映射工具，所以，在VHDL中，电路设计的编程可以与工艺相互独立。

(4) VHDL语言标准、规范，易于共享和复用。VHDL语言的语法较严格，给阅读和使用都带来了极大的好处。再者，VHDL作为一种工业标准，设计成果便于复用和交流，反过来也能进一步推动VHDL语言的推广和普及。

## 2.1.2 Verilog HDL和VHDL的比较

Verilog最初只是一家普通的民间公司的产品，后被当今第一大EDA公司Cadence收购，并推出了现今广为流传的Verilog HDL。 Verilog HDL是一种类似C语言风格的硬件描述语言，1995年正式成为IEEE标准， 在工业界使用很普遍。而VHDL是类似Pascal语言(脱胎于Ada语言)的硬件描述语言，严谨得有点呆板，但绝对不出错， 所以在国内外教学上用得非常多。现在大部分仿真器都支持Verilog HDL、VHDL混合仿真。 同时使用两种语言也没有什么问题，它们也正在相互向对方的优势学习，相互靠拢。

Verilog HDL和VHDL作为描述硬件电路设计的语言， 不同点在于：Verilog HDL在行为级抽象建模的覆盖范围比VHDL稍差一些，而在门级描述方面比VHDL强一些，建模能力比较如图2 - 1所示。

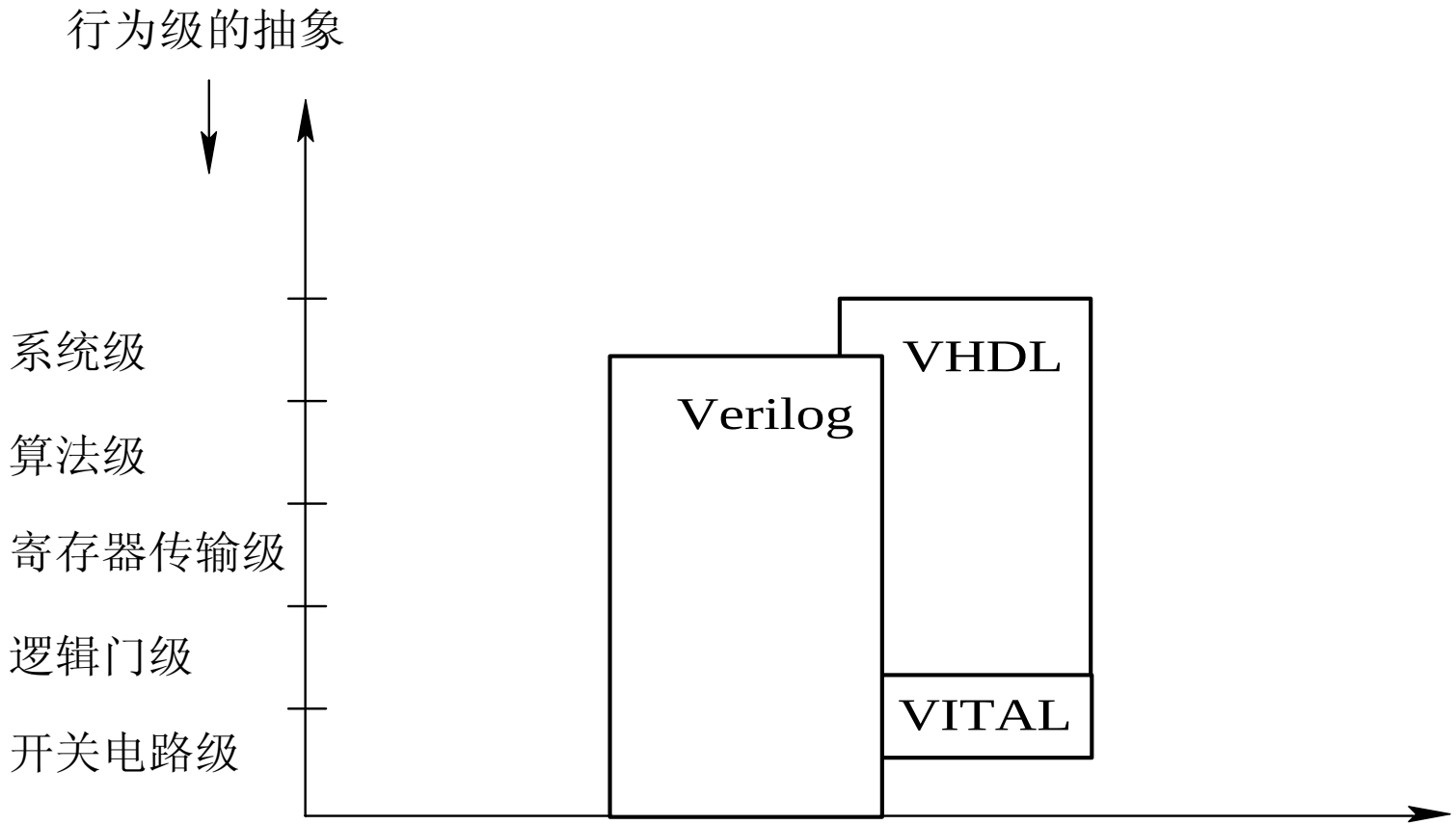



图 2-1 Verilog HDL与VHDL建模能力比较示意图

Verilog HDL和VHDL共同特点在于：都能形式化地抽象表示电路的结构和行为；都支持逻辑设计中层次与领域的描述；具有电路仿真和验证机制，以保证设计的正确性；支持电路描述由高层到低层的综合转换， 便于文档管理，易于理解与设计重用。从设计流程看，都利用层次化、结构化的设计方法， 自顶向下进行设计， 其HDL设计流程图如图2-2所示。

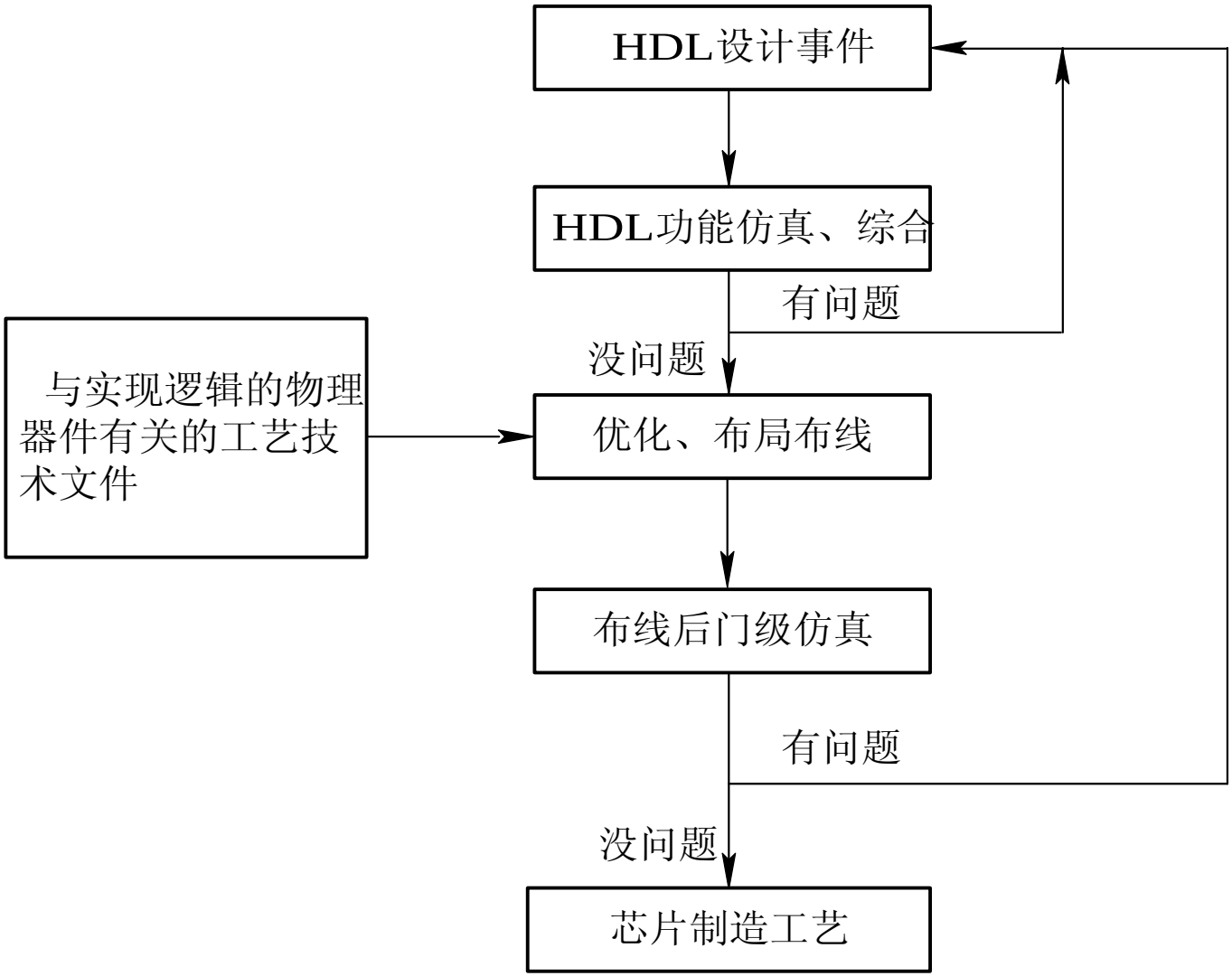



图 2-2 HDL设计流程图

## 2.1.3 传统设计与VHDL设计对照

我们把以原理图(Schematic)输入设计为主的方式称之为传统设计， 而把以VHDL语言进行设计描述的方式称之为高层次设计。

由于VHDL代码完全由文字组成，而传统的设计往往是一张张的原理图，这两者之间存在一定的对应关系。我们知道，传统的原理图总是由线和一些符号相互连接而构成，实体是与符号相对应的。而且它规定了一个设计单元对外的接口信号结构体则是与某一层的原理图相对应，而且它总是与某个实体相关， 并对该实体的结构和行为进行描述。图2-3是一个RS触发器传统设计的实例。

图2-3所示rsFF的符号为设计者描述了下面几部分信息：器件输入脚的数目(本例为Set和Reset)、 器件输出脚的数目(本例为Q和QB)以及器件的功能(本例由符号名描述器件的功能)。这个RS触发器的简单线路图如图2-4所示。

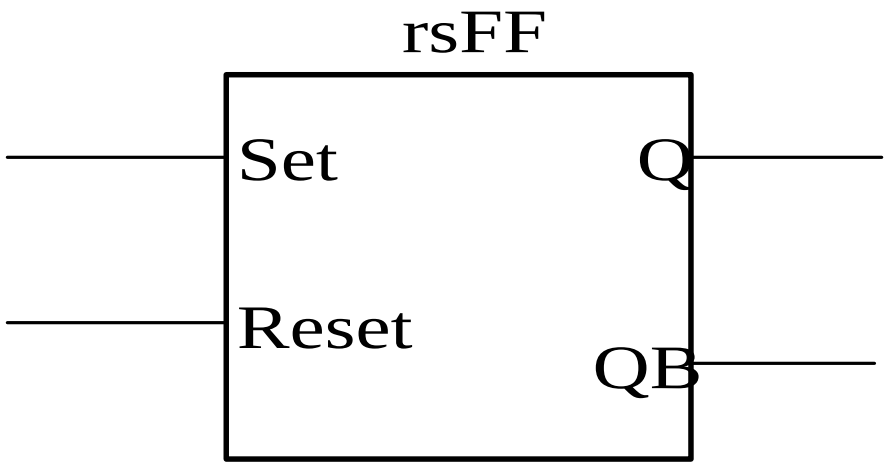



图2-3 rsFF符号

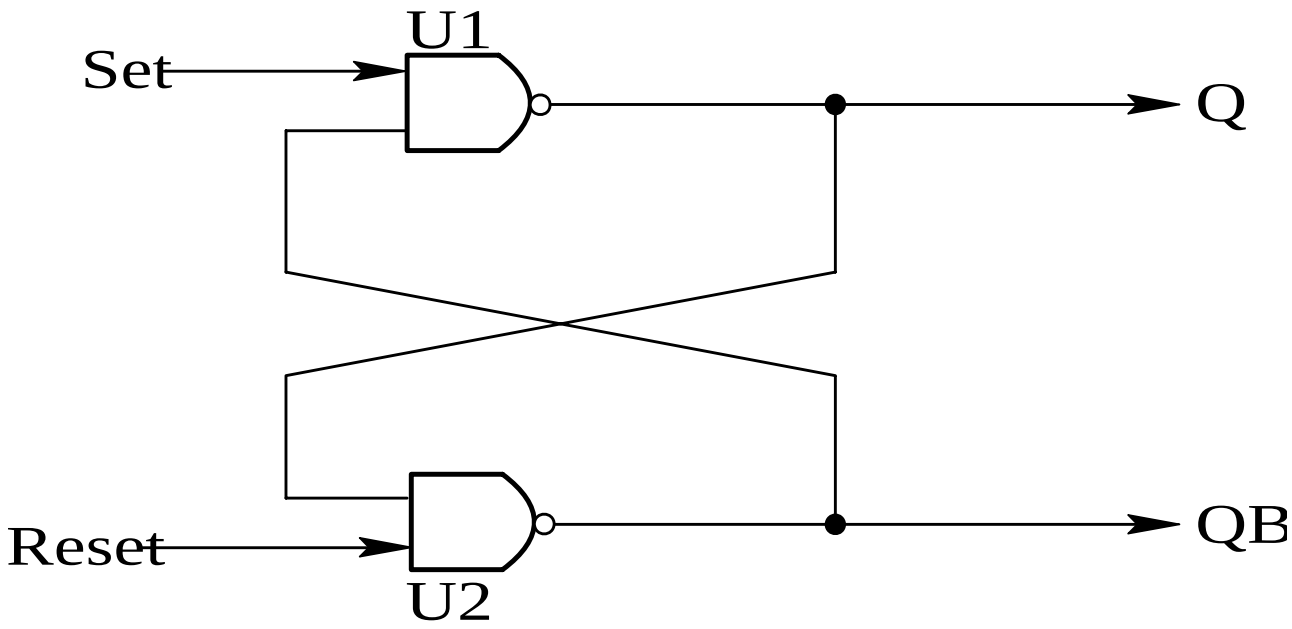



图2-4 rsFF线路图

同样的设计在VHDL中将如何完成? 在VHDL中将如何表示符号呢? 在VHDL中将看到什么样的原理图呢?

(1) 实体Entity与符号对应。由实体建立所有的设计，在VHDL中实体直接和传统设计中的符号相对应，图2-3所表示的这个rsFF符号对应于如下的VHDL代码：

```
ENTITY rsFF IS

PORT (set, reset:IN BIT;

        Q, QB:BUFFER BIT);

END rsFF;
```

注意： 在VHDL语言中，库的声明与包的使用总是放在设计单元的最前面。

```
LIBRARY IEEE;

USE IEEE.STD.LOGIC.1164.ALL;
```

这里，IEEE是库名，STD.LOGIC.1164是IEEE库中的一个包。有了上面这两条语句，下面的设计单元(实体和结构体)就可以使用STD.LOGIC.1164包中定义的数据类型和各种函数了。在图2-3的VHDL代码中，关键词ENTITY是实体语句的开头，即由程序包STD.LOGIC.1164提供该语言的关键词与数据类型，它们全部用大写表示，以便于熟记， 例如IS、PORT、IN、BIT等，提供的标准数据类型是BIT。 VHDL编辑器并不区分大、小写。

(2) ARCHITECTURE与线路图对应。在VHDL中也有rsFF元件线路图的相对应部分，它叫结构体， 结构体总是对应实体而言并描述那个实体的行为。对应rsFF线路图的结构体的VHDL代码如下所示：

```
ARCHITECTURE netlist OF rsFF IS
    COMPONENT nand2
        PORT (A, B:IN BIT;
            C:OUT BIT);
    END COMPONENT;
BEGIN
  U1:nand2
      PORT MAP (set, QB, Q);
  U2:nand2
      PORT MAP (reset, Q, QB);
END netlist;
```

这里，在关键词ARCHITECTURE标明的语句中描述实体的结构， 结构体名字是netlist，正描述的结构体叫rsFF实体的结构体。

VHDL语言的语法比较严格。一段完整的VHDL代码通常由实体语句、结构体语句、配置说明语句以及库、包说明语句组成。其中实体语句可用于描述设计单元的外部接口信号，结构体语句可用于描述设计单元内部的结构和行为。一般来说，结构体对设计单元内部的功能描述有三种描述：行为级描述、寄存器传输级描述和结构描述（本例中采用的是结构方式的描述）；配置说明语句可用于从库中选取不同的元件来构成设计单元的不同版本；设计单元主要用于存放已经编译过的实体结构体、设计单元和配置数据。当前，在VHDL中的库大致可以分为5种：IEEE库、STD库、ASIC矢量库、用户定义和WORK库，包则主要用于罗列程序中用到的各种数据、常量、子程序

(3) VHDL的描述。 VHDL对系统与电路提供的描述主要是两个方面：结构方式和行为方式。从上例中可看出，结构化描述与传统的原理图设计更接近，但VHDL的主要优点是它能从行为上描述系统和电路，而且可以按算法方式或按数据流的方式进行行为描述。如何选用一种具体的描述方式，要视具体的系统和电路而定。 RS触发器对应的行为描述如下：

```
ARCHITECTURE behave OF rsFF IS
BEGIN
    Q<=NOT(QB AND set) AFTER 2ns;
    QB<=NOT (Q AND reset) AFTER 2ns;
END behave;
```

在这个结构体中用并行信号赋值语句，当名字隐含时，含在模块中的语句给信号赋值，该语句的执行是并行的而不是串行的。 在此结构体也不存在元件具体安装语句，也没有进一步的分层层次，而是由并行的两个信号赋值语句来完成的行为方式结构体。

在VHDL中描述rsFF器件的功能，按算法表示方式将采用进程语句。 下面由sequential结构体再给出一个例子。

```
ARCHITECTURE sequential    OF rsFF IS
BEGIN
      PROCESS (set, reset)
      BEGIN
           IF set= ′ 1′   AND reset = ′ 0′    THEN
             Q<= ′ 0′    AFTER 2ns;
             QB<= ′ 1′    AFTER 4ns;
           ELSIF set =′ 0′   AND reset= ′ 1′    THEN
             Q<= ′ 1′    AFTER 4ns;
             QB<= ′ 0′    AFTER 2ns;
           ELSIF set= ′ 0′    AND  reset= ′ 0′    THEN
             Q<= ′ 1′    AFTER  2ns;
             QB<= ′ 1′    AFTER  2ns;
      END IF;
END  PROCESS;
END  sequential;
```

这种结构体只含有一个语句，称为进程语句。它在开始的一行用PROCESS关键字开始，由含有END PROCESS的行结束，在关键字PROCESS和BEGIN之间是语句说明部分，在关键字PROCESS和BEGIN这两行之间的所有语句是进程语句部分。

上面介绍了VHDL语句的一些基本构成和语句，用VHDL语句来进行电子设计的主要优势在于它支持自顶向下的设计方法，设计者可以在设计的每个层次（行为级、RTL级和门级）上进行仿真和验证，这样就避免了在设计进行到后期才发现原来设计中的逻辑性错误或结构性错误，从而减少了设计风险，也缩短了设计周期。另外，VHDL设计与最终的工艺实现无关。

## 2.1.4 VHDL的新发展

当前复杂数字系统设计面临着这样一些问题：设计复用、知识产权和内核插入；综合，特别是高层次综合和混合模型的综合；验证， 包括仿真验证和形式验证等自动验证手段；深亚微米效应。为了进一步提高设计能力，以解决这些问题，近年来众多研究者致力于从更高的抽象层次上开展设计，这方面的工作以OO-VHDL、DE-VHDL为代表。如何拓宽VHDL的应用范围，也是研究的重点，值得注意的有VITAL(VHDL Initiative Towards ASIC Library)。

由于缺乏高效可靠的用VHDL描述的ASIC库，因而一定程度上影响了VHDL的应用。 而建立ASIC库的最大困难在于VHDL没有统一、有效的方法处理时域参数。为此，工业界和IEEE开展了一系列研究，其中VITAL是最重要的结果。为了对ASIC精确建模， VITAL必须解决以下关键问题：

(1) 精确描述时序关系包括延时模型(可能是引脚到引脚的延时模型或分布式延时模型)、时序检查(如建立、保持时间检查)、尖峰脉冲处理和宏单元间的互连延时等。

(2) 高仿真效率。VITAL主要处理门级逻辑单元，因此必须具备高的仿真效率，否则，常因ASIC的规模很大，导致仿真时间太长。

(3) 具有反向标注能力。VLSI设计是层次式的迭代和提炼的过程，层次越低，就可以得到更精确的延时信息， 因此，需要一种机制把低层次的延时信息反向标注到较高层次。

(4) 通用性。 VITAL模型应该适用于各种VHDL仿真器。

为了同时解决以上这些问题，VITAL没有对VHDL语言本身进行扩展，而是提供了两个预定义的包文件(VITAL Timing、VITAL Primitive)， 还提供了对功能和延时建模所需的子程序库。VITAL-Timing包含与ASIC单元中时间行为有关的各种类型定义、常数、属性和过程。 Cadence 公司的SDF（标准的延时格式）应用证明了VITAL提供了与Verilog HDL相当的仿真精度和仿真性能。 VITAL-Primitive提供了一系列子程序， 描述数字电路建模中常见的各种电路模块的行为，VITAL-Primitive库的做法类似于在VHDL中引入了Verilog 的一些特点。使用VITAL时，功能在模型内部描述，而所有延时的计算则在模型外部完成。VITAL 3.0版规范主要包括：预定义库、 建模指导方针，从SDF获得回注的语法。

# 2.2 VHDL程序的基本结构

## 2.2.1 VHDL程序的基本单元与构成

VHDL程序的基本单元是设计实体(Design Entity)，它对应于硬件电路中的某个基本模块。该模块可以是一个门，也可以是一个微处理器，甚至整个系统。但无论是简单的还是复杂的数字电路， VHDL程序的基本构成都是一样的， 即都由实体和结构体构成(实体描述模块的对外端口，结构体描述模块的内部情况即模块的行为和结构)。

【例 1】 一个半加器(如图2-5)的VHDL描述如下。

注意，关键字在Active　VHDL中自动用蓝色高亮显示，因此不需再用大写字母表示。为了便于记忆和识认，从第2.2小节起统一用斜体英文小写字母来表示VHDL语言的关键字。

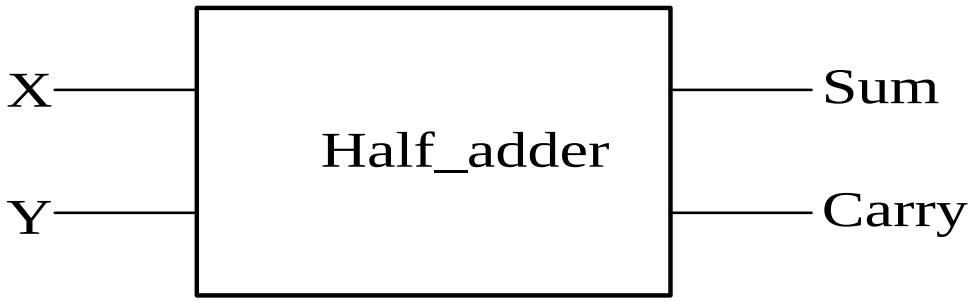



图2-5 半加器

1) 设计实体

---打开标准库

```
library    IEEE;
use    IEEE.std_logic_1164.all;
- - 实体描述
entity    Half_adder    is    - - VHDL不区分大小写
          port (
                    X:    in    STD_LOGIC;
                    Y:    in    STD_LOGIC;
                    SUM: out    STD_LOGIC;
                    CARRY: out    STD_LOGIC
          );
end    Half_adder;
```

2) 结构体描述

```
architecture Half_adder of Half_adder is
begin
    process (x, y)
    begin
        SUM<=x xor y after 5ns;
        CARRY<=X and Y after 5ns;
    end process;
end Half_adder;
```

3) 端口模式

(1) 输入(*in*)。输入模式仅允许数据流由外部向实体内进行。它主要用于时钟输入、控制输入(如复位和使能)和单向的数据输入。

(2) 输出(*out*)。输出模式仅允许数据流从内部流向实体输出端口。它主要用于计数输出。输出模式不能用于反馈，因为这样的端口不能看作实体内可读。

(3) 缓冲(*buffer*)。缓冲模式允许用于内部反馈，不允许作双向端口使用。

(4) 双向(inout)。对于双向信号，设计时必须定义为双向模式，以允许数据可以流入或流出该实体。双向模式也允许用于内部反馈。

4) 端口类型

(1) 布尔型(Boolean)。 布尔类型可取值“True”或“False”。

(2) 位(Bit)。 位可取值“0”或“1”。

(3) 位矢量(Bit_Vector)。位矢量由IEEE库中的标准包Numeric_Bit支持。该程序包中定义的基本元素类型为Bit类型， 而不是Std_Logic类型。

(4) 整数(Integer)。整数可用作循环的指针或常数，通常不用于I/O信号。

(5) 非标准逻辑(Std_ulogic)和标准逻辑(Std_logic)。非标准逻辑和标准逻辑由IEEE_ Std_logic_1164支持，访问该程序包中的项目需要有Library和use语句。

【例 2】一个作为设计实体的二选一电路的描述如下：

```
library    IEEE;
use  IEEE.std_logic_1164.all;

entity    twoto1 is
        generic(m:TIME:=1ns);
        port    (
                d0:    in    bit;
                d1:    in    bit;
                sel:    in    bit;
                q:    out    bit
            );
end    twoto1;
```

```
architecture    twoto1    of    twoto1    is
        signal    tmp:bit;
begin
   Cale: process (d0, d1, sel)
          variable    tmp1, tmp2, tmp3:bit;
            begin
                 tmp1：=d0    and    sel;
                 tmp2：=d1    and  (not sel);
                 tmp3：=tmp1    or    tmp2;
                 tmp<=tmp3;
              q<=tmp    after    m;
                end process;
end    twoto1;
```

【例 3】 利用半加器构建全加器。VHDL可以通过已有的基本模块来构造更大的模块或更高一层次的模块。例如，它可以利用现有的半加器模块来构造一个如图2-6所示的全加器:

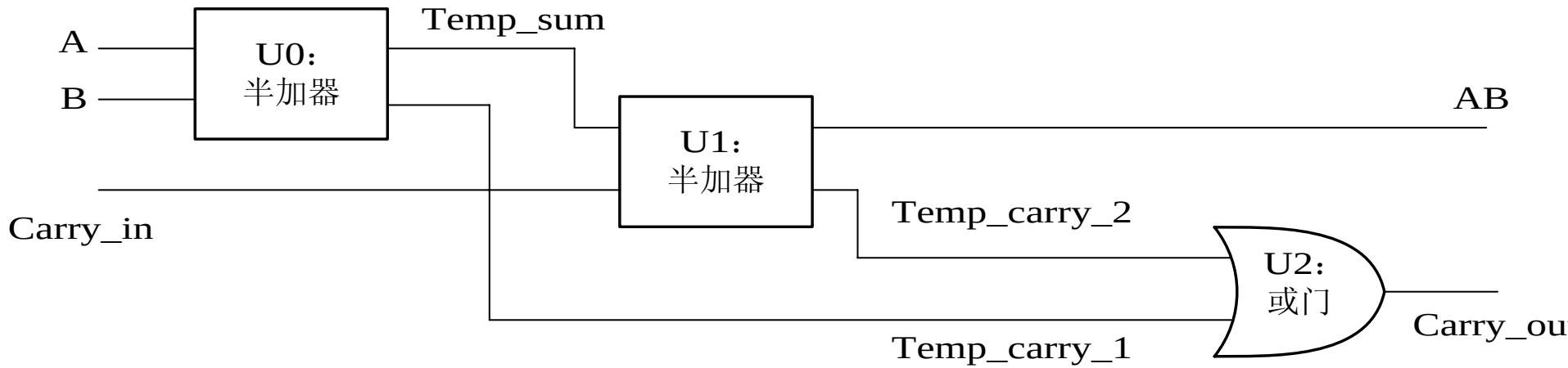



图2-6 由半加器构造的全加器

```
entity    Full_adder  is
          port     (
                    A:in Bit;
                    B: in Bit;
                    Carry_in: in Bit;
                    AB: out Bit;
                    Carry_out: out Bit );
          end    Full_adder;

architecture     Structure     of     Full_adder is
- - 连接元件所用的内部信号说明
          signal     Temp_sum: Bit;
```

```
        signal    Temp_carry_1: Bit;
        signal    Temp_carry_2: Bit;
-- 元件说明
component     Half_adder
        port (
        X: in  Bit;
        Y: in  Bit;
        Sum: out  Bit;
        Carry: out    Bit );
end component;
component    Or_gate
        port (
                In1: Bit:
                In2: Bit;
```

```
                Out1:    out     Bit );
end component;

-- 元件例化
begin
    U0: Half_adder port map (X => A, Y => B, Sum => Temp-sum , Carry =>
Temp_carry_1 );
    U1: Half_adder port map (X => Temp_sum , Y => Carry_in , Sum => AB ,
Carry => Temp_Carry_2 );
    U2: Or_gate port map(In1 => Temp_carry_1, In2 => Temp_carry_2 , Out1
=> Carry_out );
    end   structure;
```

注意:

(1) In1: *in* Bit; In2: *in* Bit; 可写成In1:Bit，In2: Bit;，因为*in*是缺省的I/O状态。

(2) “-- -”为注释行标志, 该标志后的所有字符均为注释内容。

(3) 由“*component ... end component*; ”注明的一段为元件说明语句, 给出了该元件的外端口情况, 或者说是给出了一个元件的模板。

(4) 由“- -component instantiation statements”说明的为元件例化语句部分。 该语句将元件说明中的端口映射到实际元件中的端口, 即将模板映射到现实电路。

## 1. 实体说明

实体类似原理图中的模块符号，它并不描述模块的具体功能， 实体的每一个I/O信号被称为端口，它与部件的输入 / 输出或器件的引脚相关连。 实体说明的一般形式是：

*entity* 实体名 *is*
［*generic*(类属表) ； ］
［*port*(端口表)； ］
［*declarations*说明语句； ］
［*begin*
实体语句部分］；
*end*［实体名］；

说明：

(1) 实体名和所有端口名都由字符串组成(称为标识符)。 该字符串中的任意字符可以是“a”到“z”、“A”到“Z”， 或数字“0”到“9”， 以及下划线“ -”。 字符串的第一个字符必须是字母， 中间不包括空格，且最后一个字符不可以为下划线， 两个下划线不允许相邻。

(2) ［ ］表示其中的部分是可选项。

(3) 对VHDL而言， 大小写一视同仁， 不加区分。

(4) 实体说明以 entity 实体名is开始， 至 end［实体名］结束， 最简单的实体说明是：

*entity* E *is*

*end；*

除此之外， 其余各项皆为可选项。

(5) 类属(*generic*)语句必须放在端口语句之前，用于指定由环境决定的参数。例如，在数据类型说明上用于传递位矢量长度、 数组的位长以及器件的延迟时间等参数。类属语句的一般形式为

*generic* (［常量］名字表：［*in*］类型标识［：=初始值］; ...);

例如，在二选一电路的描述中的generic(m： time：[KG-*3]=1ns)指定了结构体内延时m的值为 1ns。 又如：

```
entity    AndGate    is
generic(N:NatUral：=2);
 port(inputs:  in  Bit_vector(1  to  N);    --类属参数N规定了位矢量(Bit_Vector)inputs的长度
result: out Bit);
end   AndGate;
```

(6) 端口(*port*)说明是设计实体之外部接口的描述，规定了端口的名称、数据类型和输入输出方向, 其一般书写格式是：

*port*(端口名{，端口名}：［方向］子类型符［*bus*］［:=初始值］{；端口名{，　端口名}：［方向］子类型符［*bus*］［:=初始值］})

其中，方向用于定义外部引脚的信号方向是输入还是输出， 共有5种方向：*in*， *out*，*inout*，*buffer*，*linkage*。*in*表示信号自端口输入到结构体；*out*表示信号自结构体输出到端口；*inout*表示该端口是双向的；*buffer*说明端口可以输出信号，且结构体内部可以利用该输出信号；*linkage*用于说明该端口无指定方向，可以与任何方向的信号连接。

(7) 说明语句(*declaration*)可以包括*subprogram*说明、*subprogram*定义（或称*subprogram*体）、*type*说明、*subtype*说明、 *constant*说明、 *signal*说明、*file*说明、*alias*说明、*attribute*说明、 *attribute*定义、*use*语句、 *disconnection*定义。

用于对设计实体内所用的信号、常数、数据类型和函数进行定义， 这种定义对该设计实体是可见的。

## 2. 结构体

结构体是对实体功能的具体描述，必须跟在实体后面。通常，编译实体后才能对结构体进行编译，如果实体需要重新编译，那么相应的结构体也应重新编译。结构体的一般结构描述如下：

*Architecture* 结构体名 *of* 实体名 *is*

［说明语句；］

*begin*

［并行处理语句；］

*end*［结构体名］；

说明：

(1) 结构体的名称应是该结构体的惟一名称，*of*后紧跟的实体名表明了该结构体所对应的是哪一个实体。*is*用来结束结构体的命名。 结构体名称的命名规则与实体名的命名规则相同。

(2) 说明语句的内容除了实体说明中可有的说明项外，还可以包括元件说明(*component*)和组装说明(也称配置*configuration*)语句。说明语句用于对结构体内所用的信号、常数、数据类型和函数进行定义，且其定义仅对结构体内部可见。例如在对二选一电路的描述中：

```
architecture  connect  of  mux  is
signal  tmp：  BIT;
-- 对内部信号tmp进行定义
-- 信号定义和端口语句一样，  应有信号名和数据类型的说明
-- 因它是内部连接用的信号，  故没有也不需要方向说明
begin
...
end  connect;
```

(3) 处于*begin*和*end*之间的并行处理语句(即各语句是并发执行的)用于描述该设计实体(模块)的行为和结构。它包括block语句、 *process*语句、*procedure*调用语句、 *assert*语句、*assignment*语句、*generate*语句、*component instance*语句。

有关这部分语句的详情也会在后面几节中继续介绍。 二选一电路描述中的进程语句如下：

```
cale：process  (d0， d1， sel)
                variable   tmp1， tmp2， tmp3： BIT；
                begin
                  ...
end process；
```

(4) 一个实体说明可以对应于多个不同的结构体。即对于对外端口相同而内部行为或结构不同的模块，其对应的设计实体可以具有相同的实体说明和不同的结构体。所以，一个给定的实体说明可以被多个设计实体共享，而这些设计实体的结构体不同。从这个意义上说，一个实体说明代表了一组端口相同的设计实体(如两输入端的“与非门”和两输入端的“或非门”等)。

所以VHDL规定：对应于同一实体的结构体不允许同名，而对应于不同实体的结构体可以同名。

## 2.2.2 包(Package)、配置(Configuration)和库(Library)

### 1. 包

在实体说明和结构体中说明的数据类型、常量和子程序等只对相应的结构体可见，而不能被其它设计实体使用。为了提供一组可被多个设计实体共享的类型、 常量和子程序说明，VHDL提供了包(package)。

包用来罗列要用到的信号定义、常数定义、数据类型，元件语句、 函数定义和过程定义等。它是一个可编译的设计单元， 也是库结构中的一个层次。

包分为包说明( package declaration)和包体(package body)两部分。

包说明的一般形式是:

*package*包名 *is*

{ 说明语句; }

*end*［包名］;

包体的一般形式是:

*package body*包名 *is*

{ 说明语句; }

*end*［包名］;

说明：

(1) 包说明和相应的包体的名称必须一致。

(2) 包说明中的说明语句可包括

subprogram说明、type说明、subtype说明、constant说明、signal说明、file说明、 alias说明、 attribute说明、attribute定义、*use*语句、*disconnection*定义,即除了不包括子程序体外，与实体说明中的说明语句情况相同。包体中的说明语句可包括*subprogram*定义、*type*说明、*subtype*说明、*constant*说明、*file*说明、*alias*说明、*use*语句。

VHDL中的*subprogram*(子程序)概念与一般计算机高级语句中子程序的概念类似。子程序包括过程（*procedure*）和函数(*function*)， 分别由子程序说明和子程序体（子程序定义）两部分组成。可以出现在相应的实体说明、 结构体、 包说明和包体中，供其它语句调用。

包说明可定义数据类型，给出函数的调用说明， 而在包体中才具体地描述实现该函数功能的语句（即函数定义）和数据的赋值。这种分开描述的好处是，当函数的功能需要作某些调整时，只要改变包体的相关语句就行了，这样可以使重新编译的单元数目尽可能少。

(3) 可见性。包体中的子程序体和说明部分不能被其它VHDL元件引用，只对相应的包说明可见，而包说明中的内容才是通用的和可见的（当然还必须用*use*子句才能提供这种可见性）。下面即为一个包说明及其相应包体的例子：

```
package    Logic    is
        type    Three_level_logic is (′ 0′ , ′ 1′ , ′ z′ );
        function    invert (input： Three_level_logic)return Three_level_logic;
end    logic;

package body    Logic    is
        function    invert(input:Three_level_logic)return    Three_level_logic is
```

```
begin
    case  input is  when  '0' => return '1';
                          when '1' => return '0';
                          when 'z' =>return 'z';
    end case;
end invert;
end  Logic;
```

在上例中， 第一段是包说明，其中第三行是函数invert说明；第二段是包体，第二行开始函数的定义，给出了函数的行为。这部分内容只对包说明(即第一段)可见。所以，包说明包含的是通用的、可见的说明，而包体包含的是专用的、 不可见的说明。

(4) 在一个设计实体中加上*use*子句(在实体说明之前), 可以使包说明中的内容可见。如：

use IEEE.STD_LOGlC_1164．*all*;

*all*表示将IEEE库中的 STD_LOGIC_1164包中的所有说明项可见。又如，

*use* Logic.Three_level_logic;

表示将用户自定义的包Logic中的类型Three_level_logic对相应的设计实体可见。

(5) 包也可以只有一个包说明，因为如果包说明中既不创造子程序说明也没有待在包体中赋值的常数(deferred constant)，所以，包体就没必要存在了。

## 2. 配置

设计者可以利用配置语句为待设计的实体从资源文件(库或包)中选择描述不同行为和结构的结构体。在仿真某个实体时，可以利用配置语句选择不同的结构体，以便进行性能对比得到最佳性能的结构体。在标准VHDL中，配置并不是必需的，因为系统总是将最后编译到工作库中的结构体作为实体的默认结构体。 但是，使用配置来规定实体给了设计者一种自由——设计者可以自由选择结构体。

配置语句的一般形式为：

*configuration* 配置名 *of*实体名 *is*

［配置说明部分： *use*子句或 *attribute* 定义；］

［语句说明；］

*end*［配置名］；

配置语句根据不同情况，其语句说明有繁有简， 下面以一个微处理器的配置为例作一些简要说明。

```
--; an architecture of a microprocessor:
architecture  Structure_View of  Processor  is
--; component说明语句
    Component  ALU  port(...) end component;
    Component  MUX  port(...) end component;
begin
--; component例化语句:
    A1: ALU  port map  (...);
    M1: MUX  port map(...);
    M2: MUX  port map(…);
```

```
end  Structure_View;
-- a configuration of the microprocessor;
Library  TTL.Work;
configuration    V4_27_87  of  processor  is
        use Work．All;
        for  Structure_View
--组装说明
for    A1：  ALU
        use configuration  TTL．SN74LS181;
end     for;
for     M1, M2：  MUX
use entity Multiplex4(Behavior);
end for;
end for;
end V4_27_87;
```

其中：

*configuration* V4_27_87 *of* Processor *is*

...

*end* V4_27_87;

属于配置语句部分，为实体Processor选择了结构体Structure_View(用语句*for* Structure_View … )； 结构体 Structure-View仅给出了元件ALU、 MUX的模板， 而没有给出任何实质的行为或结构描述，所以配置语句中又采用元件配置(Component Configuration)，如：

*For* A1: ALU

*use configuration* TTL．SN74LS181;

*end* for;

为元件ALU选择标准库TTL中的配置SN74LS181；

*for* M1, M2： MUX

*use entity* Multiplex4(Behavior)；

*end for*;

将元件 MUX(M1、M2)组装到库WORK中的实体Multiplex4及相应的结构体Behavior上，使元件具有具体的行为或结构。类似元件配置的语句也可用于结构体中，称组装规则。例如，在原来的结构体的说明部分增加一句：

*for* M1, M2: MUX use entity Multiplex4(Behavior);

与在*configuration* V4_27_87中使用组装说明的目的和意义相同。组装规则的一般形式是：

*for* 元件例示标号：元件名 *use* 对应对象；

其中对应对象可以是某个配置*configuration*或实体*entity*。

组装规则就是将元件例示语句中的元件(如M1，M2：MUX)组装到实体Multiplex4 及其相应的结构体(Behavior)或已有的某个组装说明上。这样，配置语句(组装说明)为要设计的实体选择了结构体，元件配置或组装规则将元件与某个实体及其相应的结构体对应起来。

### 3. 库

库是经编译后的数据的集合，它存放已经编译的实体、结构体、包和配置。 库由库元组成，库元是可以独立编译的VHDL结构。 VHDL中有两类库元，即基本元和辅助元。基本元包括实体说明、 包说明和配置， 辅助元为包体和结构体。基本元对同一库中其它基本元都是不可见的，必须用use子句才能提供可见性。

(1) 库的种类。在VHDL语言中存在的库大致可以归纳为5 种：IEEE库、STD库、ASIC库、用户定义的库和WORK库。

IEEE库中汇集着一些 IEEE认可的标准包集合，如STD_LOGIC_1164； STD库是 VHDL的标准集。其中存放的STANDARD包是VHDL的标准配置，如定义了Boo1ean、Character等数据类型； ASIC库存放着与逻辑门一一对应的实体；用户为自身设计需要所开发的共用包集和实体等可以汇集在一起，定义为用户定义库；WORK库是现行工作库， 设计者所描述的 VHDL语句不加任何说明时，都将存放在WORK库中，例如，用户自定义的包在编译后都会自动加入到WORK库中。

(2) 库的使用。前面提到的 5 类库除了WORK库外，其它 4 类库在使用前都必须作说明，用库子句（library）对不同库中的库元提出可见性。library的说明总是放在设计单元的最前面， 其一般形式为：library库名； 接着用 use子句使库中的包和包中的项可见。 例如：

```
library    IEEE;

Use    IEEE．STD_LOGIC_1164．all   ;
```

也就是说，对于同一库中不同的库元，必须用use子句提供所需的可见性，而对于不同库中的库元，则必须用库子句加上use子句来提供相应的可见性。

(3) 库的作用范围。库语句的作用范围从一个实体说明开始到它所属的结构体和配置为止。 当一个源程序中出现两个以上的实体时，库语句应在每个实体说明语句前书写。 例如：

```
library    IEEE;    -- 库使用说明
    use    IEEE.STD_LOGIC_1164.all;
    entity and1    is
    ...
    end    and1;
    architecture    rt1    of    and1    is
    ...
    end    rt1;
```

```
configuration    s1    of    and1    is
   ...
end    s1;

library    IEEE;  -- 库使用说明
use    IEEE．STD_LOGIC_1164．all;
entity    or1    is
...
end    or1;
configuration    s2    of    or1    is
...
end    s2;
```

## 2.2.3 设计实例

以上从设计硬件电路的角度出发，介绍了完整的VHDL语言程序应具备的5个部分：实体+结构体，并配合以相应的资源(包、 库、 配置)。

采用 VHDL语言进行硬件设计时，采用自上而下的设计方法， 逐步将设计内容细化，最后完成系统硬件的整体设计。下面以设计一个小规模处理器mp为例，简要说明VHDL程序的基本结构。

尽管mp是小规模的处理器，但是仍考虑采用大规模电路自上而下的设计方法。 所谓自上而下的设计方法，即先将要设计的硬件系统(如微处理器mp)看成一个顶部模块，对应于VHDL程序中的一个设计实体(entity mp)，然后按一定的标准(如功能)将该系统分成多个子模块， 参见图 2-7。

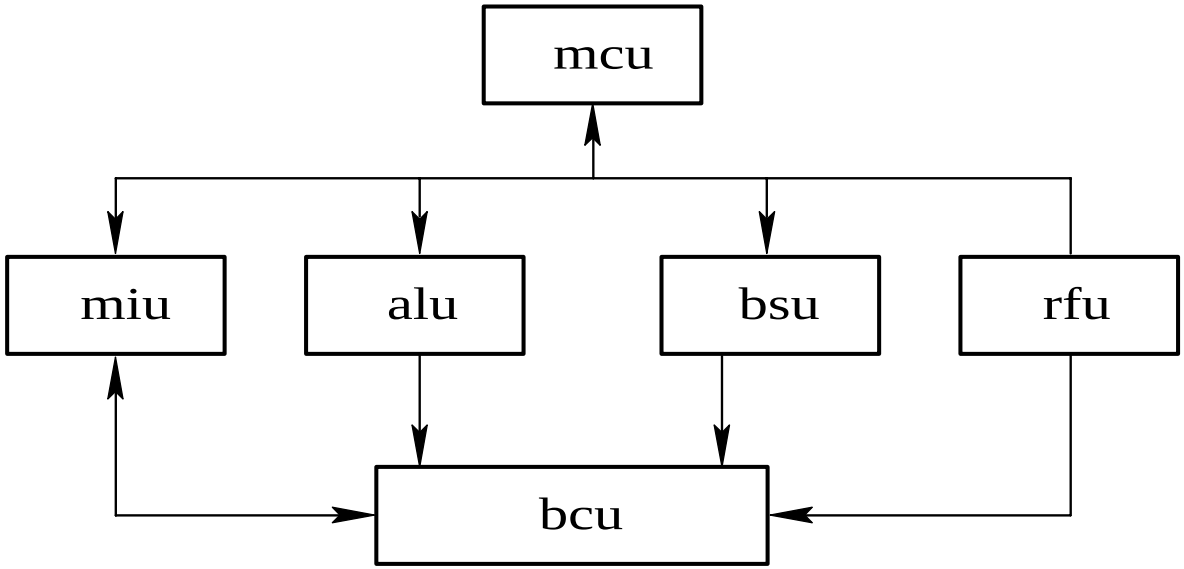



图 2 -7 处理器的 6 大部分

表 2-1 处理器各部分的功能说明

| | |
|---|---|
| alu | 算术逻辑单元 |
| bcu | 总线控制单元 |
| bsu | 循环移位单元 |
| mcu | 主控单元 |
| miu | 存储器接口单元 |
| rfu | 各存储器的控制 |

在图 2-7 中，处理器mp按功能被分为 6 个子模块： mcu、miu、 alu、bsu、 rfu、 bcu(具体功能见表 2-1)。这些子模块对应于设计实体 mp中的各个元件，用结构体中的component说明语句对元件的名字和接口进行说明：

```
-- TOP LEVEL;
-- Package declarations
library    IEEE;
use     IEEE.STD_LOGIC_1164.    all;
use     IEEE.STD_LOGIC_1164 EXTENSION. all;
library    WORK;
use     WORK.mp_package.all;
...
- --; entity declaration of mp
```

```
entity  mp is
 generic (...);
 port(...);
 begin
 ...
 end    mp;

 --; an architecture of mp;
      architecture    struct_view    of    mp    is
      component    mcu    port    (...);    end component;
      component    alu    port(...);    end component;
      component    bcu    port(...);    end component;
      component    bsu    port(...);    end component;
      component    miu    port(...);    end component;
      component    rfu    port(...);    end component;
```

```
begin
        l--1：  mcu   port map(...);
        1--2：  alu   port map(...);
        1--3：  bcu   port map(...);
        1--4：  bsu   port map(...);
        1--5：  miu   port map(...);
        1--6：  rfu   port map(...);
end   struct_view;
-- a configuration of map;
configuration  of   V_5_30 of  mp  is
        use WORK.all;
        for  struct_view
                for  1- -1：  mcu use entity   work.mcu;
                end for;
```

```
        for   1- - 2：  alu   use entity work.alu；
        end for;
        for   1- - 3：  bcu use entity   work.bcu；
        end for;
        for   1- - 4：  bsu use entity    work.bsu；
       end for;
        for  1- - 5: miu   use entity   work.miu；
       end for;
        for   1 - - 6：  rfu  use  entity  work.rfu；
      end   for;
      end for;
end    V_5_30;
```

在程序中，实体mp对处理器的外部引脚进行了说明，结构体则对处理器内部结构及相互关系进行了描述：

(1) 在结构体的说明部分，使用元件说明语句(如*component* mcu *port* (...)；*end component*；)描述了子模块的名称(mcu)和端口(形式端口)。

(2) 在结构体的语句部分， 用元件例化语句(1- -1：mcu *port map*(...)； )将元件标号、元件名称的对应关系进行描述，给出形式端口与实体中的端口、实际信号以及各子元件间的连接关系。

(3) 用*Configuration*语句(如*configuration* V_5_30 *of* mp *is* ...*end* V_5_30； )或一些组装规则将各个实际元件与器件库中的特定实体对应起来，从而使这个设计实体完成了该处理器的顶层设计。它描述了该处理器的外部端口和各个子模块间的相互关系，建立了一个 VHDL的外部框架。

(4) 所谓器件库中的特定实体，是指与各个子模块相对应的各个设计实体， 它们将各个子模块的功能和行为细化。这种对各个子模块的VHDL设计是该系统的次一层设计。

(5) 如果子模块又可以分成几个小模块，则将进行该系统的更次一层设计(方法相同), .....，如此细化下去，直到最底层设计。

这样由上至下进行系统硬件设计，其好处是：在程序设计的每一步都可进行仿真检查，有利于尽早发现系统设计中存在的问题。

VHDL是一种结构严密、语法严谨的语言。为了更灵活地掌握这种硬件设计方法，从总体上把握全局，而不至于被其中繁多的语法混淆思路，迷失设计方向，在前面充分讨论了VHDL程序的基本框架和设计思路的基础上，下面讨论VHDL语言的数据类型、操作符和对硬件系统的描述方式。

# 2.3 VHDL语言的基本数据类型和操作符

## 2.3.1 数的类型和数的字面值

VHDL中的数分整数、 浮点数、字符、字符串、位串及物理数等六种类型。 各类数的书写形式不同，而数的类型正是由其书写形式所决定。 通常把数的书写形式称为字面值。

## 1. 数字

VHDL中的数字可以用十进制数表示，也可以用二、八、十六进制数来表示。

(1) 十进制文字表示格式为：

十进制数文字::=整数［.整数］［指数］

整数::=数字｛［下划线］数字｝

指数::=E［＋］整数 | E－整数

注意： 在相邻的数字之间插入下划线，对十进制数的数值并没有影响，十进制数前面可以加若干个“0”，但不允许在数字之间存在空格， 数字与指数之间也不允许有空格。例如， 以下数字书写是合法的：

```
257                    -- 十进制整数
0                      -- 十进制整数
12_4_6A                -- -等效于1246a
0.09                   -- 十进制浮点数
4.1E-4                 -- 十进制浮点数
```

(2) 非十进制数， 其定义格式如下：

以基表示的数::=基＃基于基的整数［.基于基的整数］＃指数

基::=整数

基于基的整数::＝扩展数字｛［下划线］扩展数字｝

扩展数字::=数字｜字母

扩展数字指除了数字（“0”～“9”）外， 还包括“A”～“F”表示十六进制数的字母(大小写不分)。 同样， 在以基表示的数中插入下划线对其数值无影响。 基数的最小值为2，最大值为16。 基数和指数都必须用十进制数表示。 下面是一些例子：

- -整数文字， 值为254的十进制数可以用下列不同基的数表示：

2＃1111_1110＃　　8＃376＃　　16＃FE＃　　016＃0FE＃

- - 224(00E0H)的十六进制指数表示:

16＃E＃E1

- - 以基表示的浮点数

2＃1.1111　-1111＃E4　　(等于十进制数31.9375)

16＃0.F＃E0　　(等于十进制数0.9375)

## 2. 字符、字符串、位串和物理数

字符、字符串、位串均用ASCII字符表示。单个ASCII字符用单引号括起称为字符，如‘j’。一串ASCII字符用双引号括起来的字符序列(可以为空)称为字符串，如“CDMA”。位串是用字符表示多位数码，位串可用二进制、八进制或十六进制表示， 如数码2748用位串表示则为：

B″101010111100″ -- 二进制位串表示的数码2748

X″ABC″ -- 十六进制位串表示的数码2748

O″5274″ -- 八进制位串表示的数码2748

″A″ -- 字符串中有一个字符A，注意与字符′A′区别

物理数的字面值由一个整数或浮点数加上一个物理单位组成。 如：

1.7ps -- 1.7ps

15kohm -- 15 kΩ

此例中，ps和kohm均非关键字，它是专门定义的物理单位。

## 2.3.2 对象和分类

VHDL的数据对象包括信号(signal)、 变量(variable)、常数(constant)和文件(file) 四类。 其中， 文件包含一些专门类型的数值， 它不能通过赋值来更新文件的内容， 文件可以作为参数向子程序传递，通过子程序对文件进行读写操作。因此，文件参数没有模式。除了文件外， 其它三类数据对象的区别有：

(1) 在电子电路设计中，这三类对象都与一定的物理对象相对应。例如，信号对应硬件设计中的某一条硬件连接线，常数代表数字电路中的电源和地，变量与硬件的对应关系不太直接， 通常代表暂存某些值的载体。

(2) 变量和信号的区别在于变量的赋值被立即执行， 信号的赋值则有可能延时。

(3) 三种对象的含义和说明场合见表 2-2。

**表 2-2 三种对象的含义和说明场合**

| 对象类别 | 含　　义 | 说明语句的场合 |
| --- | --- | --- |
| 信号 | 说明全局量 | *architecture*、*package*、*entity* |
| 变量 | 说明局部量 | *process*、*function*、*procedure* |
| 常数 | 说明全局量 | 以上均可 |

## 1. 对象说明

每个对象都有类型，该类型决定可能取值的类型。constant、variable、 signal 三类对象说明的一般形式是：

*constant* 常数名表：数据类型 [：=表达式值]；

*variable* 变量名表：数据类型 [：=表达式值]；

*signal* 信号名表：数据类型 [信号类别] [：=表达式值]；

说明：常数名表、变量名表和信号名表，均由一个标识符或以“，”隔开的多个标识符组成。“：=表达式”为常数、变量、信号赋初值。通常常数赋值在常数说明时进行，且常数一旦被赋值就不能改变。

信号类别只有 *bus*或 *register*两种类型，是可选项。对象说明的示例如下：

*constant* Vcc： real：= 5.00;

*variable* x，y： integer *Range* 0 *to* 255：=10;

*signal* ground： Bit：=′0′；

## 2. 变量和信号的区别

(1) 物理意义不同。信号是电子电路内部硬件连接的抽象； 变量没有与硬件对应的器件。

(2) 赋值符号不同。 信号赋值用“<＝”符号(如S1＜＝S2)， 变量赋值用“： =”符号(如 temp3： =temp1 + temp2 ； )。

(3) 变量赋值不能加延时，且语句一旦被执行，其值立即被赋予变量。信号赋值可以加延时，使赋予信号的值在一段时间后代入。如：S1＜＝S2 *after* 10ns；S2的值经过10ns的延时后才被代入S1。 而有延时的变量赋值是不合法的，如 temp3： =temp1 + temp2 *after* 10ns， 是非法的。

(4) 信号是全局量，可用于进行进程间的通信，可用于architecture、 package、 entity的说明部分；变量是局部量，只能用于process、 function、 procedure 之中。

从上面几点不难看出，将变量和信号予以区别的根本出发点是它们对应的物理意义不同。

## 2.3.3 数据类型

VHDL提供了多种标准的数据类型，放在STD库的Standard包中。另外，为使用户设计方便，还可以由用户自定义数据类型。

VHDL的数据类型分四类，标量类型(Scalar Types)、复合类型(Composite Types)、存取类型(Access Types)、文件类型(File Types)。 限于篇幅，本节仅介绍最常用的标量类型和复合类型。 存取类型和复合类型在具体使用时，可以查阅工具软件的在线帮助。

## 1. 标量类型

标量类型是指其值能在一维数轴上从大到小排列的数据类型。标量分整(数类)型(Integer)、浮点(类)型(Float)、物理(量)类型(Physical)、(可)枚举类型(Enumeration)。(1) (可)枚举类型。(可)枚举类型是非常强的抽象建模工具， 设计者用(可)枚举类型严格地表示一个特定操作所需的值。一个(可)枚举类型的所有值都是由用户定义的，这些值为标识符或单个字母的字面值。字面值是用引号括起的单个字符，例如：′X′、′0′、′1′、′Z′等。标识符像一个名字，如reset。(可)枚举类型的定义格式为：

*type* 数据类型名 *is* (元素，元素，…)。

它定义的是一组由括号括起的标识符或字符表。 例如，用户可自定义枚举类型：

*type* Switch_level *is* (′0′，′1′，′x′);

又如，VHDL预定义的(可)枚举类型有： Character， Bit，Boolean，Severity_level(错误等级)，用于提示系统当前的工作状态：NOTE， WARNING，ERROR，FALLURE)。这些预定义放在Standard包内。

(2) 整(数类)型和浮点(类)型。VHDL定义的整(数类)型和浮点(类)型与我们一般理解的整数和实数相同。在VHDL中已预定义的整数范围是-($2^{31}$-1)～($2^{31}$-1)；预定义的实数范围是-(1.0×$10^{38}$ )～(1.0×$10^{38}$)。

VHDL中还可以自定义整(数类)型和浮点(类)型， 它们分别是以上两个类型的子集。自定义整(数类)型或浮点(类)型的一般形式是：

Type 数据类型 is 原数据类型名 约束范围；

其中，“约束范围”用“range边界1 to/downto边界2”表示。例如，定义一个用于数码显示的只能取 0~9 的整数：

*Type* digit *is* integer *range* 0 *to* 9 ;

定义一个只能取 $-10^4 \sim 10^4$的实数：

*type* current *is* real *range* -1E4 *to* 1E4;

(3) 物理类型。一个物理类型的数据应包含整数和单位两部分。物理(量)类型的定义包括一个域限制、一个基本单位和几个次级单位。每个次级单位是一个整数乘以基本单位。例如，定义一个名为 Distance的物理(量)类型：

```
type    Distance  is  range 0 to  1 E16
units
- - 基本单位
A；    angstrom埃
- - 次级单位
nm = 10A;
um = 1000nm;
mm = 1000um;
```

cm = 10mm;

m = 1000mm;

km ＝ 1000m;

*end units*;

Distance物理量的说明和运算， 如：

X： Distance;

X： ＝5A + 13um-50nm;

由上面的例子可以看出， 物理(量)类型定义的一般形式是：

*type* 数据类型名 is 范围

*units*

基本单位;

次级单位：

*end units*;

**说明**：物理量类型的范围最大为-($2^{31}$-1)~($2^{31}$-1)，且必须包含1，否则基本单位就无意义。次级单位是一个整数乘以基本单位。

VHDL预定义了物理量类型TIME， 放在Standard包中：

```
Type  Time   is   range  -(2 * * 31-1) to (2 * * 31-1)
units
 fs;                  -- 基本单位， 飞秒        10-15 s
ps = 1000 fs;         -- 皮秒                   10-12 s
ns = 1000 ps;         -- 纳秒                   10-9 s
μ s = 1000 ns;        -- 微秒                   10-6 s
ms = 1000 us;         -- 毫秒                   10-3 s
sec = 1000 ms;        -- 秒
min = 60 sec;         -- 分钟
hr = 60 min;          -- 小时
end units;
```

在系统仿真时， 时间数据用于描述信号延时。

## 2. 复合类型

复合类型即其值可分成更小对象的类型。复合类型有两种：数组和记录。数组类型(Array Types)是同一类型的分组，而记录类型(Record Types)把不同类型的元素分为一组。数组类型对线性结构(如RAM和ROM)的建模很有效，而记录类型对数据包、指令等的建模很有效。

(1) 数组类型。数组是类型相同的数据集合在一起所形成的新的数据类型，它可以是一维的、 二维的或多维的。 数组定义的一般形式是:

*type* 数组类型名 *is array* （下标范围） *of* 原数据类型名;

说明：下标范围的限定必须用整数或枚举类型来表示，如：

*type* My_ word is array (integer 0 *to* 31) *of* Bit;

用整数下标定义一个32位长的字。 又如， 先定义:

type instruction *is* (ADD，SUB， INC， SRL， SRF， CDA，LDB， XFR);

枚举类型，再定义数组下标取值范围是枚举量：

type insflag is array (instruction ADD to SRF) of Integer;

VHDL中预定义的数组类型有字符串 string和位矢量bit_vector。它们被放在STD库的 Standard包中。

(2) 记录类型。一个记录类型的数据对象可具有不同类型的多个元素。一个记录的各个字段可由元素名来访问。换句话说， 记录类型是将不同类型数据和数据名组织在一起而形成的新类型。 定义记录类型的一般形式为：

*type* 数据类型名 *is record*

元素名： 数据类型名;

元素名： 数据类型名;

...

*end record*;

例如:

```
type    bank    is record        - - 定义一个bank记录
r0：    integer;
inst：  instruction；
end  record;
```

记录的使用：

```
signal   r_bank：  bank;  - - 定义一个bank类型的信号r_bank
signal   result：  integer；
result＜= r_bank.r0；        - - 用“.”表示对记录的引用
```

## 2.3.4 运算操作符

VHDL为构成计算表达式提供了 23 个操作符。这些操作符预定义为 4 类： 算术运算符、逻辑运算符、关系运算符、连接运算符。 按优先级由低到高的顺序排列的VHDL操作符如表 2-3 所示。

# 表 2-3 VHDL的操作符

| 操作符类型 | 操　作　符 | 功　　能 |
|---|---|---|
| 逻辑运算符 | AND | 逻辑与 |
| | OR | 逻辑或 |
| | NAND | 逻辑与非 |
| | NOR | 逻辑或非 |
| | XOR | 逻辑异或 |
| 关系运算符 | = | 等号 |
| | /= | 不等号 |
| | < | 小于 |
| | > | 大于 |
| | <= | 小于等于 |
| | >= | 大于等于 |
| 算术运算符 | +、- | 加、减 |
| 连接运算符 | & | 连接 |
| 算术运算符 | +、- | 正、负 |
| | * | 乘 |
| | / | 除 |
| | MOD | 求模 |
| | REM | 取余 |
| | * * | 指数 |
| 逻辑运算符 | ABS | 取绝对值 |
| 算术运算符 | NOT | 求反 |

VHDL的操作符的意义、用法和高级语言基本相同。 值得注意的是，连接运算符用于位的连接，如：

```
signal    temp_b：   bit_vector(3 downto  0)；

signal    en：  bit：=1；

- - &将4个en相连为位矢量“1111”赋予temp   -b:

    temp_b：=en&en&en&en；
```

# 2.4 VHDL结构体的描述方式

研究微电子器件的两个基本问题是它的执行功能和逻辑功能。 相应地，VHDL程序对硬件系统的描述分为行为描述和结构描述。 行为描述和结构描述有 3 个不同。

(1) 与硬件的对应关系不同。行为描述是对系统书写模型的描述，结构描述是对系统的子元件和子元件之间相互关系的描述。 在与硬件的对应关系上，结构描述更明显、更具体。

(2) 语句不同。行为描述的基本语句是进程语句， 结构描述的基本语句是元件例化语句。

(3) 用途不同。行为描述方式用于系统数学模型或系统工作原理的仿真，而结构描述方式用于进行多层次的结构设计， 能做到与电原理图的一一对应，可以进行逻辑综合。

下面将对VHDL的行为描述语句作一介绍。至于结构描述语句(包括component语句和元件例化语句)，前面已有所表述， 这里就不再赘述。

## 2.4.1 顺序描述语句

顺序执行语句只能出现在进程 process或子程序 program中，用于定义进程或子程序的算法。 顺序描述语句(Sequential Statement)有以下几种： wait语句、 断言(assert)语句、 信号赋值语句、变量赋值语句、 过程调用、if语句、case语句、循环语句(loop)、 next语句、 exit语句、 return语句、null语句。 一个典型的例子是：

```
entity SRFF is
        port(s，r：in bit；q，qBar：out bit);
        end SRFF;

        architecture behavior of SRFF is
                begin
                process
                                variable Last_state：bit：='0';
                begin
                --下面是顺序执行语句
                assert not ( s ='' and r ='1' )
                        report" Both s and r equal to 1"
                        severity Error
                if s ='0' and r ='0' then
                        Last_State：=Last_State;
                elsif s ='0' and r ='1' then
                        Last-State：='0';
```

```
        else
                Last_State：='1';
        end if
                q <= Last_State    after    2ns;
                qBar <= not q;
        Wait on  r, s;
        end process;
end   SRFF;
```

## 1. wait语句

进程在仿真进行中的两个状态(激活、暂停)的变化受 wait语句控制。有四种wait语句以设置不同的条件：

*wait*； 无限等待；

*wait on* 信号名表；当信号名表中任一信号发生变化时， 进程结束暂停状态，被激活；

*wait until* 条件；只当条件成立时， 进程才被重新激活；

*wait for*时间表达式； 时间限定， 时间到则被激活；

在上面RS触发器的行为描述程序中，*wait on* r, s;表示信号r, s中任意一个发生变化都将使进程被激活。

## 2. 断言语句

断言语句主要用于程序仿真，以便调试时进行人机对话，监视系统当前工作状态和给出警告或错误信息。断言语句的一般形式是：

*assert* 条件

[*report*输出信息]

[*severity*等级]

说明：

(1) 当执行*assert*语句时，就会对条件进行判断。 如果条件为真， 则执行*assert*以后的另一个语句； 如果条件为假， 则表示系统出错， 输出错误信息和错误严重等级提示。 例如，在RS触发器的行为描述程序中， 条件not( s =′ 1′ and r = ′ 1′ )为真，即 s和r不同时为 1 时，系统不出错， 跳过*assert*语句；反之，当条件为假，即s和r同时为1时， 则执行*assert*语句。

(2) *report*后跟的是设计者写的字符串，用于说明错误的原因， 用“ ”括起来。例如，在RS触发器的行为描述中，*report*“Both s and r equal to 1”说明出现了RS触发器的r和s同时为1的错误。

(3) severity后跟的是错误严重程度的级别。VHDL中分为 4 个级别： FAILLURE、 ERROR、 WARNING、 NOTE。 在RS触发器的行为描述中， 错误级别为ERROR， 模拟会终止。

### 3. 信号赋值语句

信号赋值语句的一般形式为：

目的信号量<= 信号量表达式；

**说明：**

(1) 信号赋值语句用于将右边信号量表达式的值赋予左边的信号量, 而且“<=”两边信号量的类型和长度应该一致, 如

a<＝b;

(2) 信号量表达式中可以有延时，如RS触发器程序中，

q<= Last_state *after* 2ns;

## 4. 变量赋值语句

变量赋值语句的一般形式为：

目的变量：=表达式；

说明：

(1) 该语句表明将右边的值赋予左边的目的变量， 但左右两边的类型必须相同。

(2) 右边的表达式可以是变量、 信号或字符常量， 如Last_state：=′0′。

**5. if语句**

if语句的基本意义和用法同高级语言。 其一般形式为：

*if*条件 then

顺序处理语句

*elsif* 条件 then

顺序处理语句

［*else*顺序处理语句］

*endif*;

具体例子可以参考RS触发器程序。

### 6. case语句

*case*语句与 *if*语句功能类似，用于根据指定的条件执行某些语句。但*case*语句的可读性比 *if*语句强。 *case*语句的一般形式是：

*case* 表达式 is

*when* 条件表达式1 =＞ 顺序处理语句

*when* 条件表达式2 =＞ 顺序处理语句

...

*end case*;

其中， when 表达式可以有4种形式：

*when*值 =＞顺序处理语句；

*when*值|值|值|...值|=＞顺序处理语句；

*when others* =＞ 顺序处理语句；

*when*值*to*值＝＞顺序处理语句； -- 表示在一定取值范围内执行顺序处理语句。

例如，RS触发器程序中的if语句又可以写为：

```
Case  s&r is
    when    00 ＝＞ Last_state： =Last   -state;
    when    01＝＞ Last_state： =0;
    when others =＞ Last_state： =1;
end case;
```

## 7. 循环语句

loop语句有两种表达形式：

(1) ［标号： ］ *for* 循环变量 in 离散范围 *loop*

顺序处理语句

*end loop* ［标号］；

例如，对数1～9进行累加运算：

sum： =0;

assume: *for i in* 1 *to* 9 *loop*

sum： =sum + 1;

*end loop* assume;

(2) ［标号： ］*while* 条件 *loop*

顺序处理语句

*end* ［标号］；

例如，上例又可写为：

```
sum： =0;
i： =1;
assume: while ( i < 10 )loop
sum： =sum + 1;
i： =i + 1;
end loop assume;
```

**8. null语句**

null语句表示一种只占位置的空处理操作。

**9. 其它**

其它顺序执行语句，如过程调用语句和*return*语句、*exit*和*next*语句都和软件编程的高级语言类似， 这里不再赘述。

## 2.4.2 并发描述语句

并发描述语句(Concurrent Statement)用于描述硬件系统并发工作的操作。 VHDL规定的并发描述语句有进程语句(process)、并发过程调用语句、并发信号赋值语句、 并发断言语句、 元件例化语句、生成语句 (generate)、 块语句(block) 。

## 1. 进程语句

进程语句是VHDL中描述硬件系统并发行为的最基本语句。进程语句前面已多次提到。

其一般形式为：

［标号：　］ *process*［(敏感信号表)］

［进程说明部分］

*begin*

{ 顺序处理语句 }

*end process* ［标号］；

说明：

(1) 敏感信号表中只要有一个信号发生变化， 进程就将启动。

(2) 进程说明部分用于对进程中用到的数据类型、子程序加以说明，包括：*subprogram*说明、*subprogram*体、*type*说明、*subtype*说明、*constant*说明、*variable*说明、*file*说明、*alias*说明、*attribute*说明、*attribute*定义、*use*语句。

进程语句的例子可以参见RS触发器程序，其中*Process* (set, reset)中的“set”和“reset”是敏感信号表中的两个激励信号，当敏感信号表中的某个信号变化时进程被激活。

## 2. 块语句

块语句可用于描述局部电路， 其格式为：

［标号:］ *block*

［块头］

［说明语句］

*begin*

{ 并发处理语句 }

*end block* ［标号］

说明：

(1) 块头用于信号的映射或类属参数的定义，通常用port语句、port map语句、generic语句和generic map语句来实现。

(2) 说明语句同结构体的说明语句，主要是对块内所要用到的对象加以说明。

(3) 一个块语句可以与一个局部电路对应。

例如一个cpu芯片，为简化起见，假设cpu只由alu和reg模块组成。 每个模块的行为分别用 block语句来描述:

```
architecture  cpu_blk  of  cpu  is
signal  ibus，dbus：Bit_vector(31 downto  0)；
begin
a1u：block
        signal  qbus：Bit_vector(31 downto  0)；
        begin
          …                          -- - 并发处理语句
```

```
        end block    alu;

    reg：block

            signal   qbus：Bit_vector(31 downto  0)；

            begin

                ...                          - - 并发处理语句

            end  block   reg；

    end   cpu_blk；
```

注意：块语句与进程语句的最大区别是，块的语句部分是并发处理语句， 进程的语句部分是顺序执行语句。

### 3. 并发信号赋值语句

并发信号赋值语句的一般形式为：

目的信号量＜＝敏感信号量表达式；

信号赋值语句在进程外(但仍在结构体中)使用时， 作为并发语句形式存在；在进程内使用时，作为顺序执行语句形式存在。 如：

```
architecture  behave  of  a_var is
begin
    out<=a(i); - -并发信号赋值语句
end  behave;
```

等价于

```
architecture  behave  of  a_var is
begin
    process(a，  i);
        begin
            out<=a(i);     - - 顺序信号赋值语句
    end  process;
end   behave;
```

由此可见，并发信号赋值语句与一个具有信号赋值的进程等价。并发信号赋值语句还可以分两种形式：条件信号赋值语句和选择信号赋值语句。

(1) 条件信号赋值语句的一般形式是：

目的信号量＜＝表达式1 *when*条件 1 *else*

表达式2 *when*条件 2 *else*

表达式3 *when*条件 3 *else*

表达式n；

表示若when后指定的条件满足，则将相应表达式的值代入目的信号量； 否则判断下一个表达式的条件。例如，利用条件信号赋值语句来描述一个四选一逻辑电路：

```
entity    mux4 is
    port( i0, i1, i2，  i3, a, b：  in STD_LOGIC；    q: out STD_LOGIC);
end  mux4;

Architecture  rt1 of    mux4  is
   signal    sel：   STD_LOGIC_VECTOR(1    downto    0 );
   begin
   sel＜= b＆a;
   q ＜＝ i0    when    sel＝“00”    else
          i1    when    sel＝“01”    else
          i2    when    sel＝“10”    else
          i3    when    sel =“11”;
   end    rt1;
```

由此可见，条件信号赋值语句相当于一个带有if语句的进程。

(2) 如果说条件信号赋值语句与带有if语句的进程等价，

那么选择信号赋值语句与带有case语句的进程等价。 其形式为：

*with* 表达式 select

目的信号量 <= 表达式1 *when* 条件 1；

表达式2 *when*条件 2；

表达式n *when*条件n；

例如，上例中的条件信号赋值语句可以用选择信号赋值语句代替：

```
with    sel    select
        q<=  i0    when“00″ ;
             i1    when“01″ ;
             i2    when“10″ ;
             i3    when“11″ ;
```

## 4. 生成语句

一个实际电路往往会由许多重复的基本结构组成，生成语句可用来简化这一类电路的描述。 见下面描述 8 个倒相器的例子:

```
entity    Invert_8 is
        port     (
                  Inputs: Bit_vector (1 to    8 );
                 Outputs: out  Bit_vector (1 to 8 ) );
end     Invert_8;
```

```
architecture    Invert_8    of    Invert_8  is
    component  Inverter
        port    (I1: Bit;
                 O1: out   Bit );
        end component;
        begin
                 G:    for  I  in  1 to 8 generate
                Inv: Inverter  port  map (Inputs(I), Outputs (I));
        end generate;
end Invert_8;
```

上述倒数第三行的端口映射方式称为“位置映射”，它将第一个实际端口Inputs(I) 与元件说明中的第一个局部端口I1建立联系，而将第二个实际端口Outputs(I) 与第二个局部端口O1建立联系。 如果改成下列映射形式，就称为“命名映射”：

```
Inv: Inverter  port map     (I1 => Inputs(I), O1 => Outputs(I));
```

# 2.5 Active-VHDL上机准备

## 2.5.1 Active-VHDL的安装与启动

### 1. 系统配置

Active-VHDL所需的硬件条件并不高，Pentium以上PC机，最少有 32 MB内存。 系统运行环境可在Windows 98/NT/2000 操作系统下，安装系统约需要 160 MB硬盘空间。 内存大小可能限制仿真周期长短和仿真时间，但我们只做教学、学习时，不考虑这些问题。

## 2. 安装步骤

(1) 上网下载Active　VHDL 3.3 教学程序。

(2) 展开该软件后，选中main.exe文件并输入密码*jackpot* 1999后开始安装。

(3) 安装完毕后， 将破解文件中的两个关键文件直接展开释放到：

After installation of Active VHDL 3.3,

copy license.dat into Dat subdirectory of Active VHDL;

e.g.C:\Program Files\Aldec\Active VHDL\Dat

copy RMCL.dll into BIN subdirectory of Active VHDL;

e.g. C:\Program Files\Aldec\Active VHDL\BIN

其中， 盘符可以变化， 例如

d:\ Program Files\Aldec\Active VHDL\Dat

(4) 电脑重新启动后， 即可运行Active-VHDL。

## 2.5.2 EDIT Plus安装使用

### 1. 为什么要使用EDIT Plus

EDIT Plus是功能强大的文本编辑器，可以无限制地撤消、重做，支持表达式查找替换，能够同时搜索、编辑多个文件，具有监视剪贴板功能， 可以同步自动地将剪贴板上的文本粘贴到EDIT Plus 的编辑窗口中。另外， 它也是一个好用的 HTML 编辑器，也支持 C/C++、Perl、Java 等语句的关键词突出显示，除了可以突出显示HTML标签外，还内嵌有完整的 HTML和 CSS 命令功能。它还会结合 IE 浏览器于 EDIT Plus 窗口中， 让您可以直接预览编辑好的网页。它非常适合编辑网页与程序的撰写，对于HTML、ASP、 JavaScript、 VBScript、 Perl、Java、C/C++等语言，可将程序代码以鲜明色彩显示， 也可以自定义需彩色显示的文字集。

## 2. 安装步骤

(1) 网上下载EDIT Plus。

(2) 下载并释放EDIT Plus。

(3) 安装并注册。

(4) 可以通过下载中文化程序简体版来汉化EDIT Plus。

## 2.5.3 熟悉Active-VHDL的集成环境

### 1. 认识Active-VHDL的文件类型

打开Active-VHDL的文件菜单，该菜单中“New”命令包括了Active-VHDL中的基本文件。下面将对Active-VHDL有关文件类型进行总结分类：

(1) VHDL Source。 该项为VHDL源文件，其扩展名为.vhd。

(2) Waveform。该项为波形文件，主要用于Active VHDL在进行仿真之后显示各信号的波形变化。

注意： 运行仿真之后才能生成波形文件， 其扩展名为.wfv。

(3) List。 该项为信号列表文件，用于显示各信息数值的变化。

(4)Basic Script。该项用于建立Basic Script文件，在Active-VHDL中支持用Active-VHDL 编写的宏，可以用于一系列重复的操作，主要用于仿真中。

(5) Text Document。该项为文本文档，主要用于编写一些和设计有关的说明文件。

(6) 位图文件。 其扩展名为.bmp。

(7) 设计文件。 其扩展名为.adf或.pdf， 建议使用.adf。

(8) 测试矢量文件。 其扩展名为.asc。

(9) State Disgram。该项为状态转换图文件，在Active-VHDL中，主要用于有限状态机(fsm)的设计。

## 2. 学会Comment Block和Uncomment Block的用法

Comment Block用于把当前所选定的文本块变为注释。该文本块被注释后，在编译时将不作为程序的一部分存入编译数据库中。在不知哪一部分出问题的情况下多用于VHDL调试中。被注释的文本块前面将加上“--”注释号。

Uncomment Block用于把当前所选定的注释文本块前的注释号去掉， 还原成程序描述的一部分。

### 3. ZOOM选项的用法

在波形调试时， 点击In选项可放大1倍，点击Out选项可缩小 1 倍，用于调整波形大小到合适位置。 点击Full选项可以将波形显示正好和波形显示窗口大小吻合， 适合总体观察波形形状。

## 4. Refresh命令的用法

点击Refresh命令可刷新， 将变形的波形恢复正常。

## 5. Design菜单的用法

(1) Add files to Design。该项用于在当前设计中加入一些新的文件。 单击该选项后，可以选择VHDL源文件， 测试基准文件， 以及将宏文件加入到当前设计中的文件之中。

(2) Compile all。 该项用于编译当前设计中的所有文件。

(3) 每次仿真中只可以使用一个测试基准(testbench)文件，而且必须在仿真之前将其置于顶层。设置方法：选择Design菜单的“Setting”命令中的“Top level Selection”选项卡， 将测试基准文件置于顶层。

## 6. 仿真菜单的用法

(1) Initialize simulation。该选项用于初始化，把一些仿真要用到的信息先存入数据库， 然后进行仿真。

(2) 运行命令。该项用于运行仿真。该命令分Run、Run until、Run for三种。 Run命令用于运行仿真，并直到仿真结束为止；Run until命令用于从头运行仿真到某一指定的时间点；Run for是从当前仿真所处的时间点上继续运行的时间。

## 2.5.4 Active-VHDL自带范例的调试流程

调试流程如下：

(1) 打开设计文件(例如可选计数器)。熟悉New Design、Open Design和Close Design命令。

(2) 调整并理解View菜单中各个窗口显示。

(3) 调出并阅读设计文件源程序。

(4) 编译和改错。

(5) 使用“Setting”命令置testbench文件于顶层。

(6) 添加信号。

(7) Run(仿真)。

(8) 阅读并分析波形。

## 2.5.5 测试基准中的VHDL激励信号

VHDL可以很有效地作为测试基准中激励信号的编程语言， 激励信号中的测试矢量可以直接用VHDL语言来编写，这使得测试矢量的编写与模拟器无关。

例如，对于半加器、全加器施加激励信号，测试矢量可以如表 2-4 所示(不是惟一的)。

# 表 2-4 测 试 矢 量

| 时间 t/ns | x | y | Cin |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 50 | 0 | 0 | 1 |
| 100 | 0 | 1 | 0 |
| 150 | 0 | 1 | 1 |
| 200 | 1 | 0 | 0 |
| 250 | 1 | 0 | 1 |
| 300 | 1 | 1 | 0 |

续表

| 时间 t/ns | x | y | Cin |
|---|---|---|---|
| 350 | 1 | 1 | 1 |
| 400 | 0 | 1 | 1 |
| 450 | 1 | 0 | 1 |
| 500 | 0 | 0 | 1 |
| 550 | 1 | 1 | 0 |
| 600 | 0 | 1 | 0 |
| 650 | 1 | 0 | 0 |
| 700 | 0 | 0 | 0 |

## 2.5.6 Active-VHDL中测试基准自动生成流程

(1) 设计半加器时， 设计浏览器(Design Browser) 如图 2-8 所示。

(2) 在“Design”菜单项下选择“Compile all”命令项，对所有源文件进行编译。 编译完成后，设计浏览器的每一个源文件前面都出现一个“＋”号， 表示其产生子项。

(3) 点击选中“半加器和全加器”前面的“＋”号， 将打开其下的子项。

(4) 用鼠标右击该子项， 将出现如图 2-9 所示的弹出式菜单。

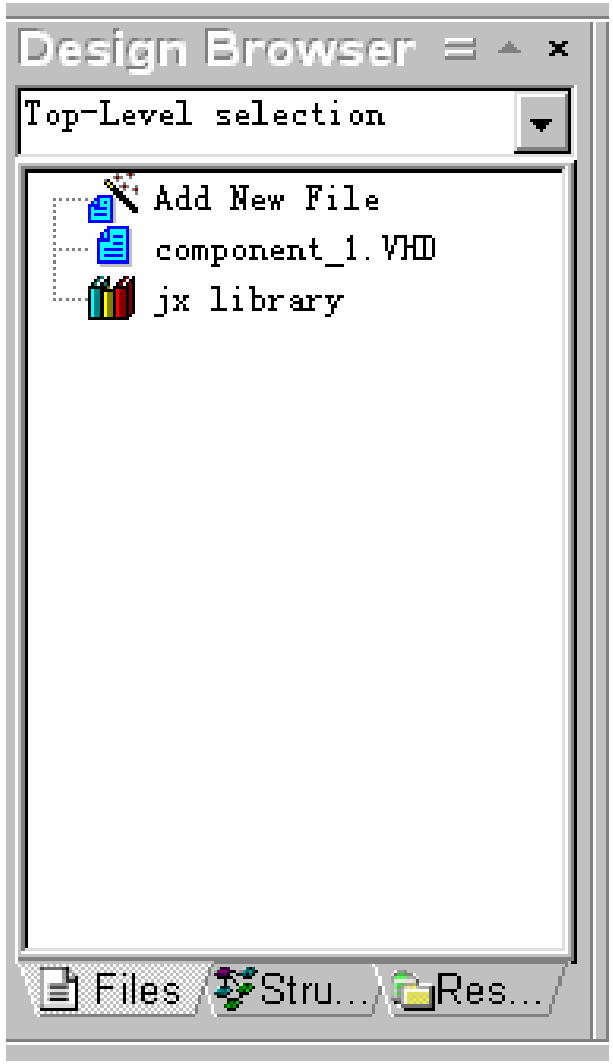

Design Browser
Top-Level selection
Add New File
component_1.VHD
jx library
Files
Stru...
Res...

图 2-8 设计浏览器

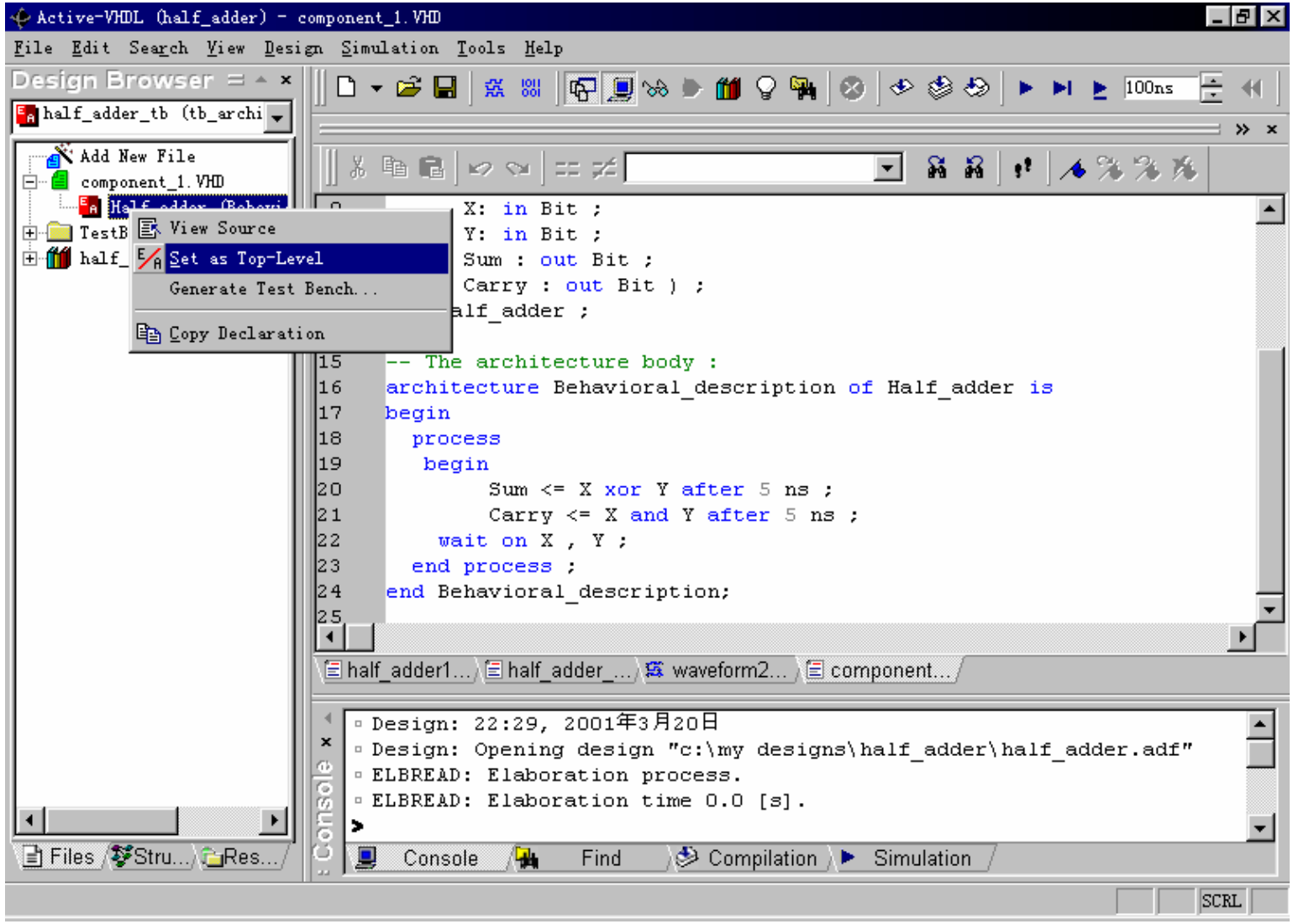

Active-VHDL (half_adder) - component_1.VHD
File Edit Search View Design Simulation Tools Help
Design Browser
half_adder_tb (tb_archi
Add New File
component_1.VHD
TestB
half_
View Source
Set as Top-Level
Generate Test Bench...
Copy Declaration
X: in Bit ;
Y: in Bit ;
Sum : out Bit ;
Carry : out Bit ) ;
alf_adder ;
-- The architecture body :
architecture Behavioral_description of Half_adder is
begin
process
begin
Sum <= X xor Y after 5 ns ;
Carry <= X and Y after 5 ns ;
wait on X , Y ;
end process ;
end Behavioral_description;
half_adder1...
half_adder_...
waveform2...
component...
Design: 22:29, 2001年3月20日
Design: Opening design "c:\my designs\half_adder\half_adder.adf"
ELBREAD: Elaboration process.
ELBREAD: Elaboration time 0.0 [s].
Files
Stru...
Res...
Console
Find
Compilation
Simulation
SCRL

图 2-9 右击快捷菜单

(5) 选择“Generate Test Bench...”项目， 将出现如图 2-10 所示的对话框。

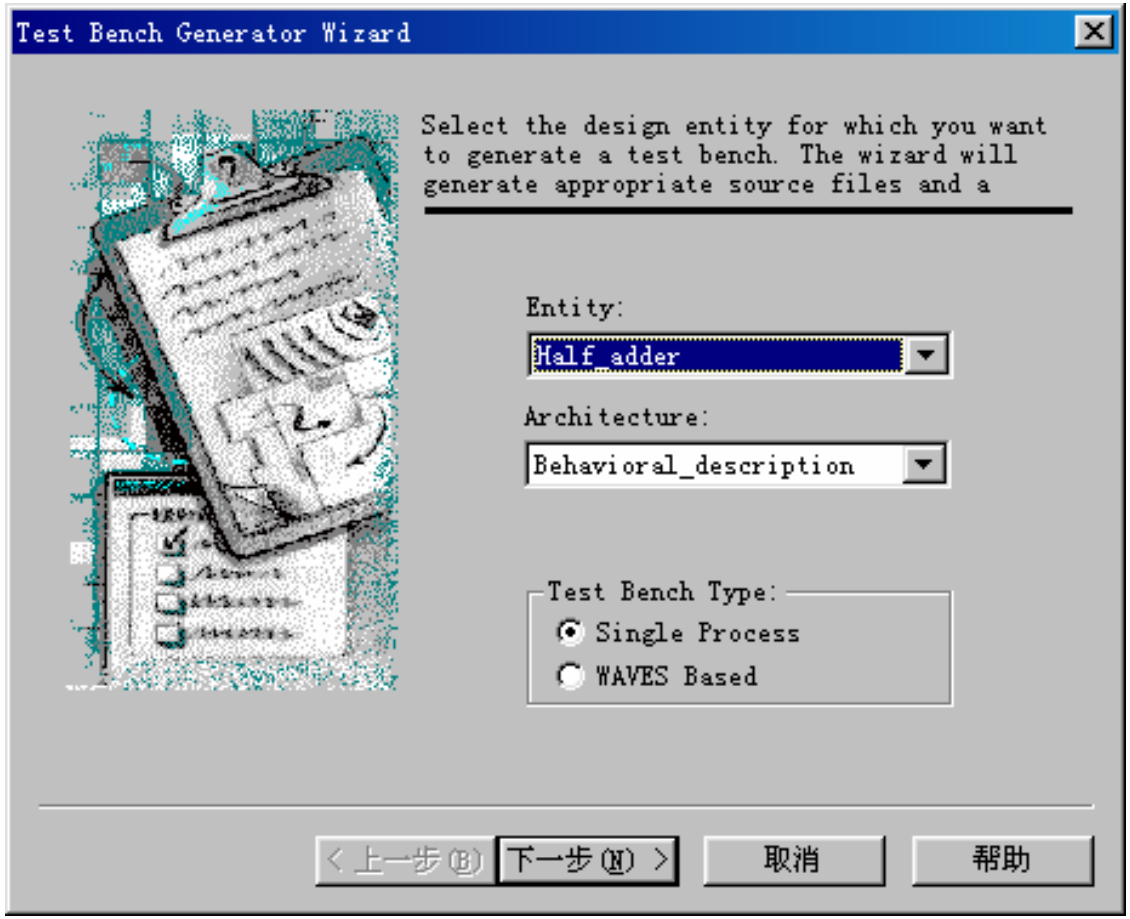



图 2-10 测试基准生成对话框

(6) 可选择要产生测试基准的实体和结构体， 以及测试基准的类型。

(7) 单击“下一步”按钮，按提示要求选择对话框有关选项。这时如果要从文件引入测试矢量， 可以选定“Test vectors from file”项。然后选择可以从中引入测试矢量的文件，否则将自动产生测试矢量。

(8) 单击“下一步”按钮，将出现选择对话框，按提示进行操作。

(9) 单击“完成”按钮， 将自动生成测试基准(testbench-for-half-adder)。

(10) 打开该文件， 在其中的“--add your stimulus here...”位置加上测试矢量描述即可， 参见图 2-11。 例如， 半加器可以是：

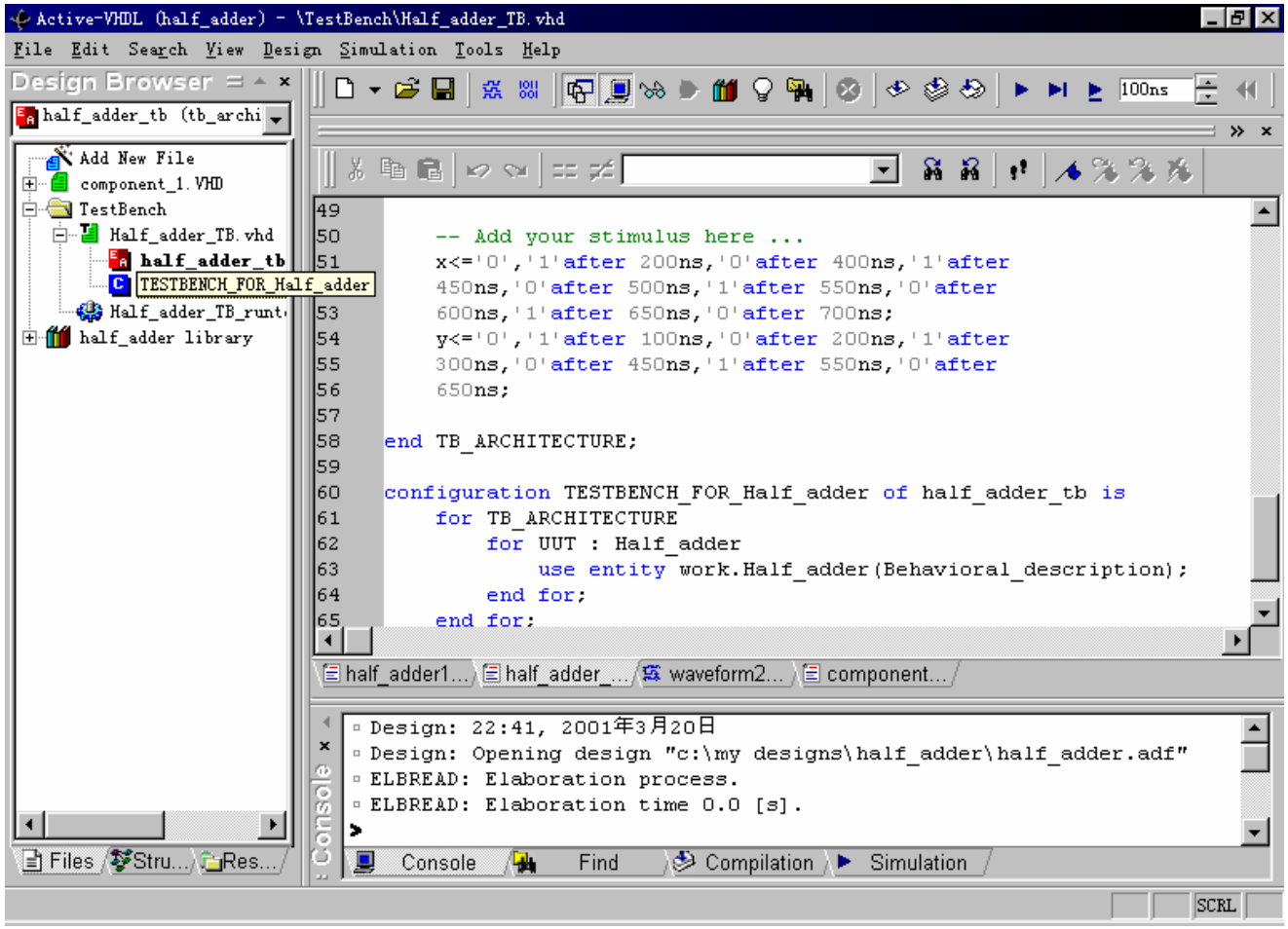



图 2-11 半加器测试基准示意图

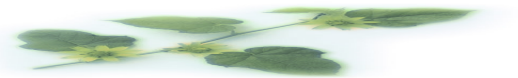
- - Add your stimulus here ...

x<='0','1' *after* 200ns,'0' *after* 400ns,'1' *after* 450ns,'0' *after* 500ns,'1' *after* 550ns,'0' *after* 600ns,'1' *after* 650ns,'0' *after* 700ns;

y<='0','1' *after* 100ns,'0' *after* 200ns,'1' *after* 300ns,'0' *after* 450ns,'1' *after* 550ns,'0' *after* 650ns;

如果是全加器，则再增加一句即可：

c_in<='0','1' *after* 50ns,'0' *after* 100ns,'1' *after* 150ns,'0' *after* 200ns,'1' *after* 250ns,'0' *after* 300ns,'1' *after* 350ns,'0' *after* 550ns;



(11) 注意：设定当前要仿真的实体可使用弹出式菜单中的“Set as Top　level”项。

在上面的例子中，我们写出了对于加法器adder所要施加的输入信号波形。注意在after子句后面的时间是绝对时间，不允许出现后面时间值小于前面时间值的情况。

## 2.5.7 半加器的波形分析

(1) 添加信号。使用“Compile all”命令后， 如报告编译无错误通过，则可以单击工具栏上“New Waveform”命令，点击如图 2-12 所示的“Add Signals”按钮，出现如图 2-13 所示的“Add Signals”对话框，按住“Shift”键，用鼠标点击起始信号项或按住“Ctrl”键逐一添加信号。

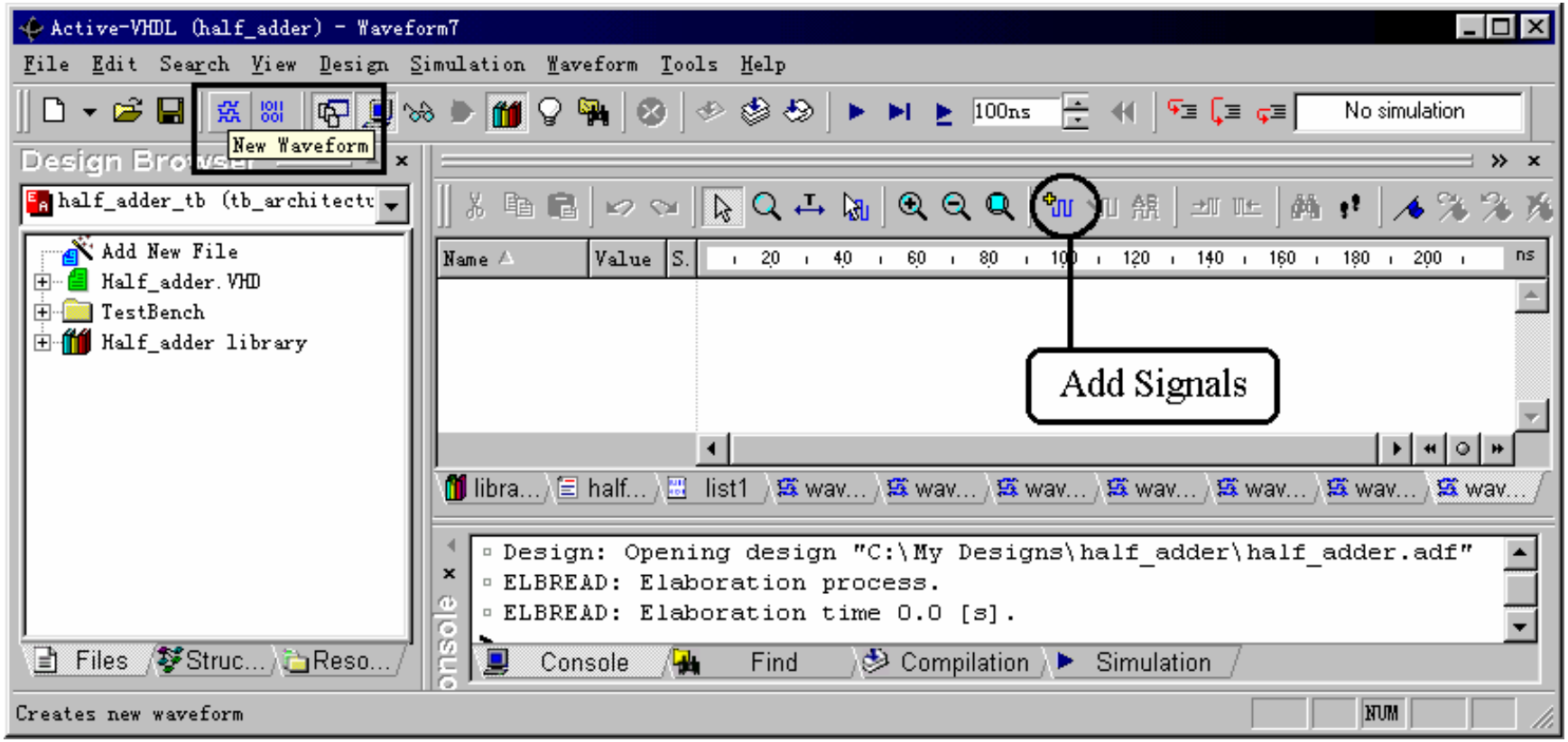



图 2-12 “Add Signals”按钮

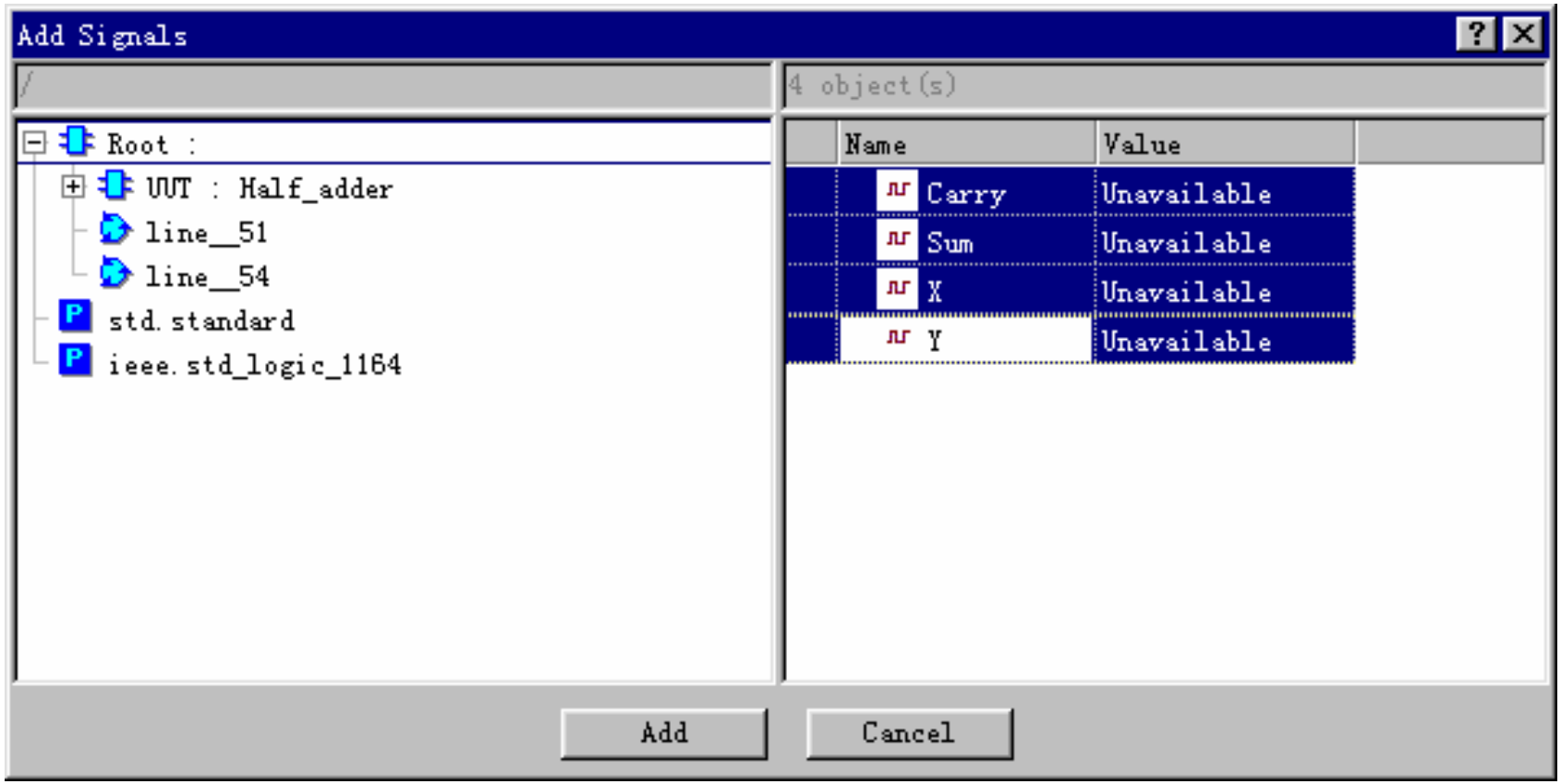



图 2-13 “Add Signals”对话框

(2) 单击“run”命令后出现波形如图 2-14 所示。

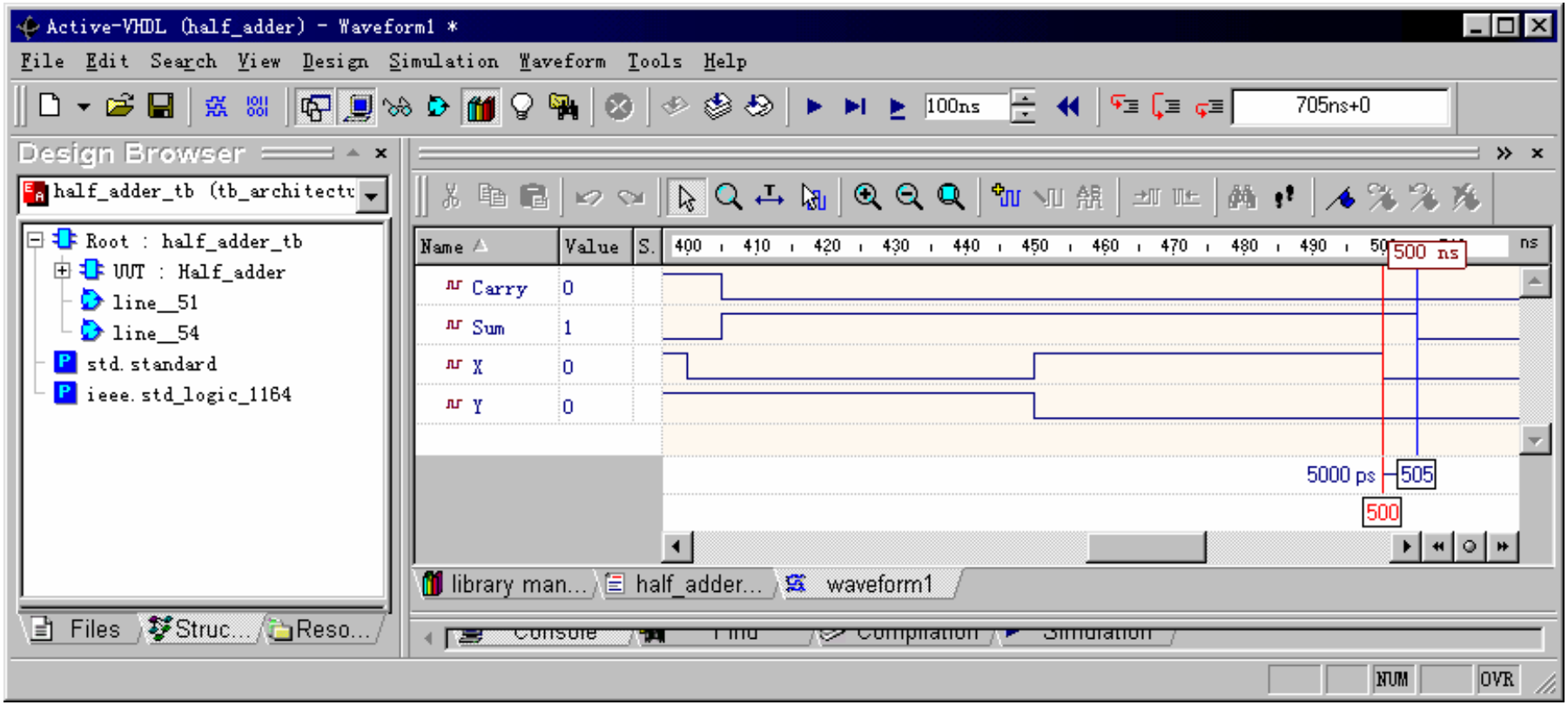



图 2-14 半加器的波形分析图

(3) 分析波形是否有问题。 注意： 当时间为 500 ns时，x和y=0, sum=1, carry=0; 时间为 505 ns时， x和y=0, sum=0 carry=0; 时间为 550 ns时， x和y全为1， sum=0, carry=0; 时间为 555 ns时， x和y全为1， sum=0, carry=1。

# 2.6 使用VHDL书写测试基准的方法

## 2.6.1 测试基准的作用

硬件系统通常是通过输入信号来驱动的，在不同的输入信号情况下会产生不同的输出结果。因此， 仿真输入信息（激励）的产生是对系统进行仿真的重要前提和必须进行的步骤。

验证工作就是将激励作用于在测模块（MUT， Module Under Test）的模型， 对在测模块的响应与期望响应相比较，并报告模拟期间发生的差异。这种验证方法常称为测试基准，其原理如图 2-15 所示。

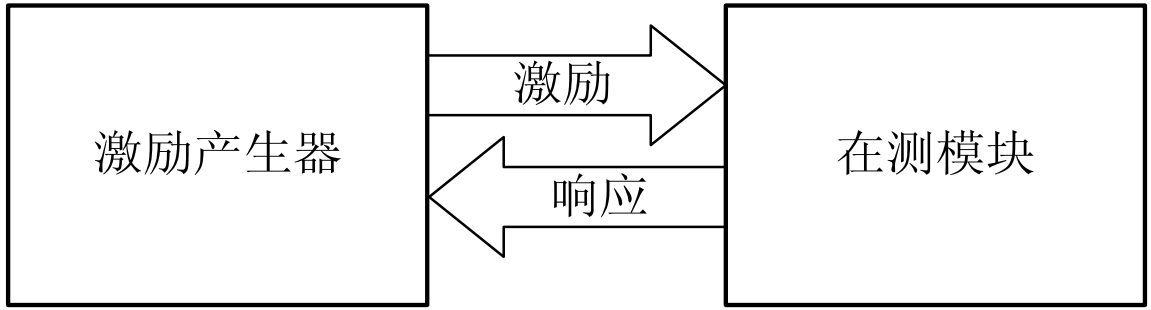



图 2-15 测试基准原理图

## 2.6.2 写激励所用的关键VHDL语句说明

### 1. 进程语句

进程语句之间是并行关系，而进程语句本身则定义了单独一组在整个模拟期间连续执行的顺序语句。进程语句的格式如下：

*PROCESS* ［（敏感信号表）］ ［IS ］

〈 说明区 〉

*BEGIN*

〈 顺序语句 〉

*END PROCESS*

编写激励时使用的进程语句通常不含有敏感信号表，而在顺序语句里含有大量WAIT的语句。

**注意：**含有敏感信号表的进程语句内不允许再显式出现*WAIT*语句，而不含敏感信号表的进程语句内必须至少含有一个*WAIT*语句。

不含有敏感信号表的进程语句可以看作是一个无限循环：在模拟期间，当进程的最后一个语句执行完毕之后，又从该进程的第一个语句开始执行。进程中的顺序执行期间，在未遇到*WAIT*语句的情况下， 模拟时钟不会停止前进。

2.WAIT语句

进程在仿真运行中总是处在下述两种状态之一：执行或挂起。进程状态的变化受等待语句的控制，当进程执行到，就将被挂起， 并设置好再次执行的条件。

1)*WAIT FOR*语句

*WAIT FOR*语句的完整书写格式为

*WAIT FOR*时间表达式;

*WAIT FOR*语句后面跟的是时间表达式，当进程执行到该语句时被挂起，直到指定的等待时间到时，进程再开始执行*WAIT FOR*语句的后继语句。

### 2) WAIT语句

*WAIT*语句是无限等待语句。对于希望只执行一次的不含有敏感信号表的进程语句，可以将*WAIT*语句置于该进程的最后，这样的话， 该进程在第一次执行到最后的*WAIT*语句时，进程进入无限制的等待状态。

## 2.6.3 激励的编写示例

有了前面的准备， 我们现在就以图 2-16 所示的非同步复位的D锁存器为例，用VHDL编写一个激励测试它的功能。用VHDL语言描述的非同步D锁存器如下：

```
library    VISUALLIB;
use    VISUALLIB.VER2VHDL.all   ;
use    VISUALLIB.VER2VHDL-MATH.all   ;

entity  DFF  is
    port (
```

```
        Q:    out wire;
        QN: out   wire;
        D:    in   wire;
        CP: in    wire;
        CLR: in   wire
        );
end    DFF;

architecture    DFF  of    DFF  is
  signal    O_int: wire;
begin
  process (CP, CLR)
```

```
begin
        if (CLR = ′ 0′ ) then
            O_int <= ′ 0′ ;
        elsif (CP′ event    AND    CP=′ 1′ ) then
            O_int <= D;
          end if   ;
    end process;
   Q <= O_int;
   QN <= not    (O_int);
end;
```

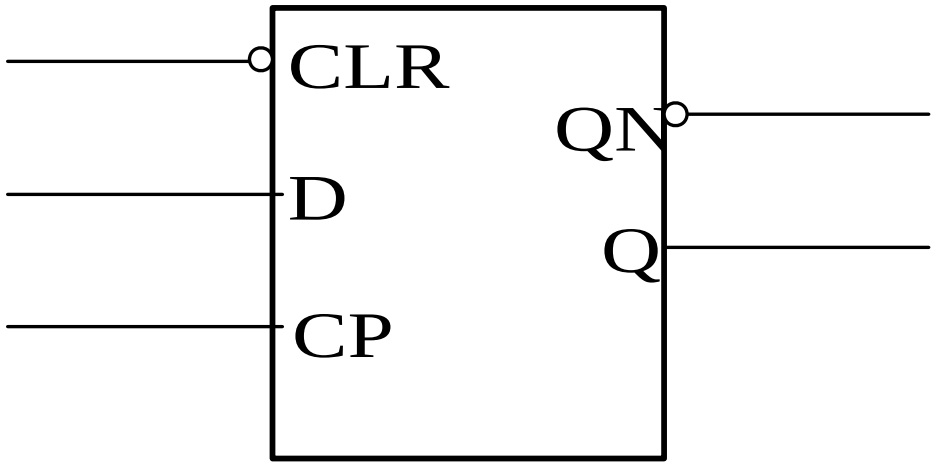



图 2-16 非同步复位的D锁存器

非同步复位的D锁存器是带有复位输入端CLR、上升沿触发的D触发器。当CLR=′0′ 时，其Q端输出被强迫置为“0”。CLR又称清零输入端；当CLR= ′1′ 时，只有在时钟上升沿脉冲过后， 输入端D的数据才传递到输出端。

根据以上对非同步复位的D锁存器的VHDL描述，我们设计如图 2-17 所示的激励波形(CP周期为10 ns)施加到DFF的输入端， 通过观察DFF的输出波形来确定其逻辑是否正确。

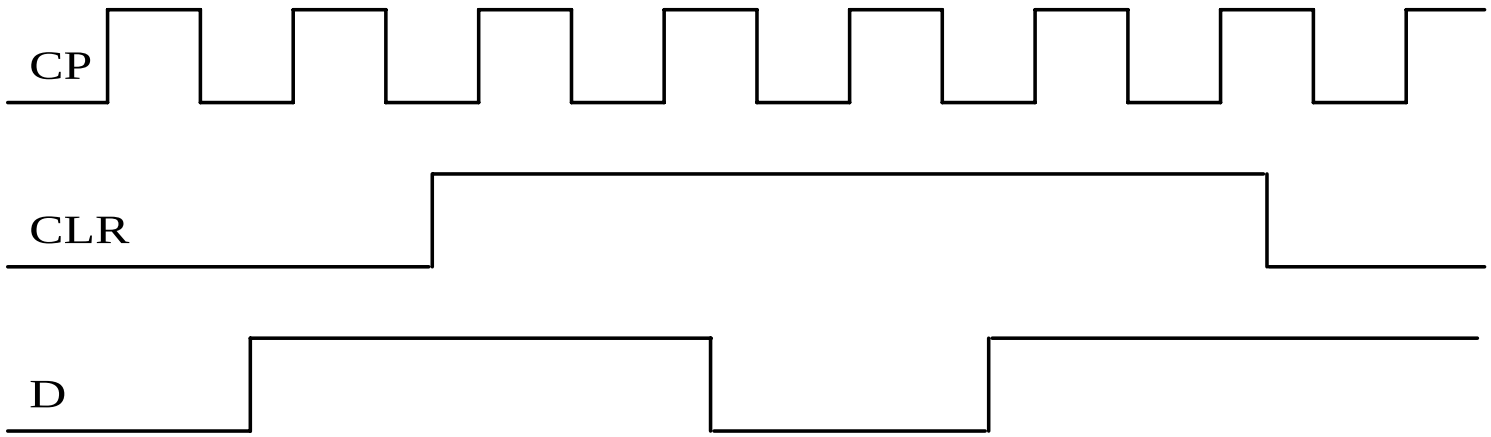



图 2-17 激励波形(CP信号的周期为10 ns)

下面给出使用VHDL语言对上述波形的描述：

```
process
begin
    CP <= ′ 0′ ;
    wait for 5000 ps;
    CP <= ′ 1′ ;
    wait for 5000 ps;
end process;
--这个process无限循环执行产生时钟信号CP
process
    begin
        CLR <= ′ 0′ ;        //TEST CLR
        D <= ′ 0′ ;
```

```
        wait   for     12500 ps;
        D <= ′ 1′ ;
        wait  for 10000 ps;
        CLR <= ′ 1′ ;
        wait  for     15000 ps;
        D <= ′ 0′ ;
        wait  for     15000 ps;
        D <= ′ 1′ ;
        wait  for  15000 ps;
        CLR <= ′ 0′ ;              - -TEST CLR

    wait   ;   - -无限等待， 表明此process只执行一次。
end process ;
```

下面给出在SUMMIT Visual HDL上对DFF的仿真程序以及最终的输入输出波形：仿真程序TEST.VHD:

```
library  VISUALLIB;
use  VISUALLIB.VER2VHDL.all  ;
use  VISUALLIB.VER2VHDL_MATH.all  ;

entity  test  is
end  test;
architecture  test  of  test  is
```

```
component    DFF
        port (
            Q: out   wire：='z';
            QN: out  wire：='z';
            D: in  wire：='z';
            CP: in  wire：='z';
            CLR: in  wire：='z'
           );
    end component;

        signal    Q: wire;
        signal    QN: wire;
        signal    D: wire;
        signal    CP: wire;
        signal    CLR: wire;
```

```
begin
        unit: DFF
        port  map(Q,
                  QN,
                  D,
                  CP,
                  CLR);
--以上实现对DFF的调用
  process
  begin
    CP <= ′ 0′ ;
    wait  for 5000 ps;
    CP <= ′ 1′ ;
    wait  for  5000 ps;
end process;
```

```
process
    begin
      CLR <= ′ 0′ ;
      D <= ′ 0′ ;
      wait for    12500 ps;
      D <= ′ 1′ ;
      wait for    10000 ps;
      CLR <= ′ 1′ ;
      wait for    15000 ps;
      D <= ′ 0′ ;
      wait for    15000 ps;
      D <= ′ 1′ ;
      wait for    15000 ps;
      CLR <= ′ 0′ ;

    wait;
 end process;
end   ;
```

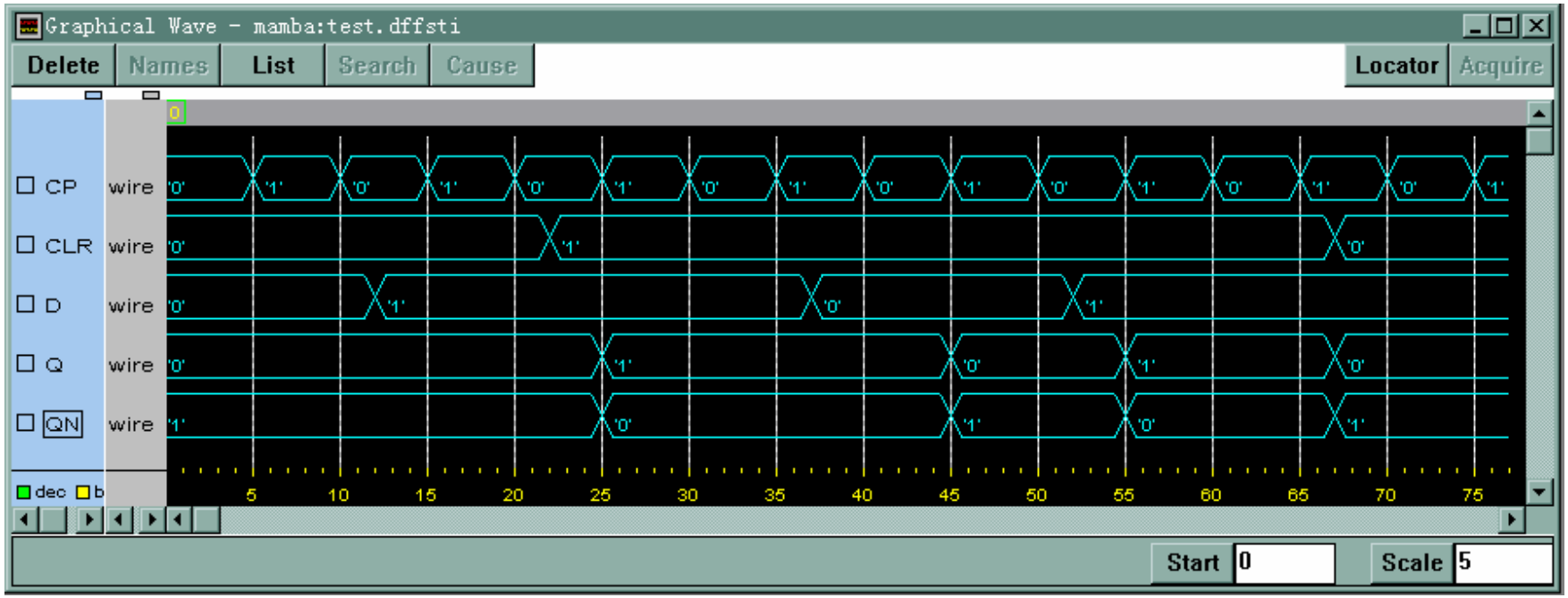



图 2-18 SUMMIT Visual HDL界面下的输入、 输出波形

# 2.7 基本逻辑电路的VHDL实现

## 2.7.1 组合逻辑电路设计

### 1. 二输入“异或门”电路

(1) 基本逻辑门有：与门(AND)、或门(OR)、与非门(NAND)、或非门(NOR)和异或门(XOR) 。这些门都是简单的组合电路，用布尔方程描述其逻辑功能很方便。例如，二输入“异或门”电路的逻辑表达式为：y=a⊕b。

(2) Active VHDL下编写的二输入异或门源代码：

```
library IEEE;
use IEEE.std_logic_1164.all ;
entity XOR2 is
  port (
      A: in STD_LOGIC;
      B: in STD_LOGIC;
      Z: out STD_LOGIC);
end XOR2;
architecture XOR2 of XOR2 is
begin
   process (a, b)
  begin
      Z<=A xor B;
    end process;
  end XOR2;
```

(3) 输入“异或门”仿真波形如图 2-19 所示。

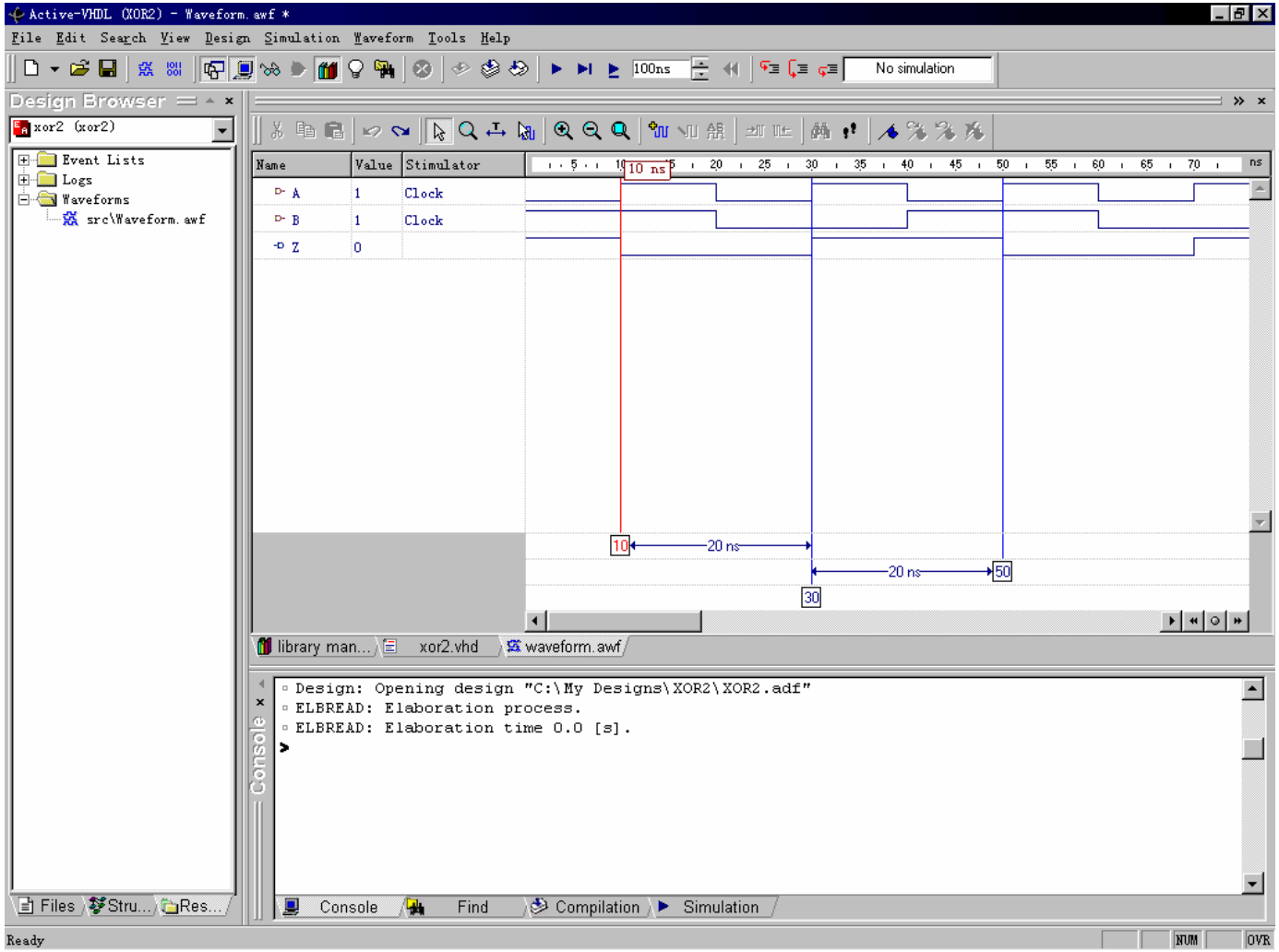



图 2-19 二输入“异或门”仿真波形

从图 2-18 中可看出二输入“异或门”的波形情况， 该仿真没考虑延时，所以在10 ns时， A、 B输入同为“1”， 则Z输出为“0”； 而在 30 ns时， A输入为“1”， B输入为“0”，则Z输出为“1”。

## 2. 三-八译码器

(1) 查看有关集成电路手册，三-八译码器(74LS138)由一个三输入“与非”门、4个反相器和一个三输入“或非门”构成。如果采用VHDL语言来描述和编写测试基准，其逻辑设计变得非常容易， 阅读起来也非常方便。

(2) Active VHDL下编写的三-八译码器源代码：

```
library IEEE;
use IEEE.std_logic_1164.all ;

entity e_decode is
        port (
              inputs: in STD_LOGIC_VECTOR (0 to 2);
              enables: in STD_LOGIC_VECTOR (0 to 2);
              outputs: out STD_LOGIC_VECTOR (7 downto 0)
         );
end e_decode;
```

```
Architecture  e_decode of     e_decode    is
    constant enabled:std_logic_vector(0 to 2)：=″ 100″ ;
    constant y0:std_logic_vector(7 downto 0)：=″ 00000001″ ;
    constant y1:std_logic_vector(7  downto 0)：=″ 00000010″ ;
    constant y2:std_logic_vector(7 downto 0)：=″ 00000100″ ;
    constant    y3:std_logic_vector(7 downto 0)：=″ 00001000″ ;
    constant    y4:std_logic_vector(7 downto 0)：=″ 00010000″ ;
    constant    y5:std_logic_vector(7 downto 0)：=″ 00100000″ ;
    constant    y6:std_logic_vector(7 downto 0)：=″ 01000000″ ;
```

```
    constant  y7:std_logic_vector(7 downto 0)：=〃10000000〃;
    constant    zero:std_logic_vector(0    to    2)：=〃000〃;
    constant    one:std_logic_vector(0    to    2)：=〃001〃;
    constant    two:std_logic_vector(0    to    2)：=〃010〃;
    constant    three:std_logic_vector(0    to    2)：=〃011〃;
    constant    four:std_logic_vector(0    to    2)：=〃100〃;
    constant    five:std_logic_vector(0    to    2)：=〃101〃;
    constant    six:std_logic_vector(0    to    2)：=〃110〃;
    constant    seven:std_logic_vector(0    to    2)：=〃111〃;

begin
    process(inputs, enables)
        begin
```

```
        if   enables=enabled  then
    case     inputs is
        when    zero =>outputs<=y0;
        when    one =>outputs<=y1;
        when    two =>outputs<=y2;
        when    three=>outputs<=y3;
        when    four =>outputs<=y4;
        when    five =>outputs<=y5;
        when    six  =>outputs<=y6;
        when    seven=>outputs<=y7;
        when    others=>null;
      end case;
  end if;
end  process;
end  e_decode;
```

(3) Active VHDL下编写的三_八译码器测试基准源代码：

```
library IEEE;
use IEEE.std_logic_1164.all ;

_ _ Add your library and packages declaration here ...

entity e_decode_tb is
end e_decode_tb;

architecture TB_ARCHITECTURE of e_decode_tb is
```

```
_ _ Component  declaration  of the tested unit
component    e_decode
     port(
       inputs: in  std_logic_vector(0 to 2);
        enables: in std_logic_vector(0 to 2);
        outputs: out std_logic_vector(7 downto  0));
end component;

_ _ Stimulus signals _ signals mapped to the input
and inout ports of tested entity
signal     inputs: std    _logic    _vector(0 to  2);
```

```
    signal    enables: std_logic_vector(0 to 2);
     _ _ Observed signals _ signals mapped to the output ports of tested entity
    signal    outputs: std_logic_vector(7 downto 0);

    _ _ Add your code here ...

begin

     _ _ Unit Under Test port map
     UUT: e    _decode
          port map
               (inputs => inputs,
               enables => enables,
               outputs => outputs);
```

enables<=″ 000″ , ″ 100″ *after* 10ns;

inputs<=″ 000″ , ″ 001″ *after* 50 ns, ″ 100″ *after* 100ns, ″ 010″ *after* 150ns,

″ 101″ *after* 200ns, ″ 111″ *after* 250ns, ″ 011″ *after* 300ns, ″ 110″ *after* 350ns;

*end* TB_ARCHITECTURE;

*configuration* TESTBENCH_FOR_e_decode *of* e_decode_tb *is*

*for* TB_ARCHITECTURE

*for* UUT: e_decode

```
        use  entity  work.e_decode(e_decode);
    end for;
  end for;
end  TESTBENCH_FOR_e_decode;
```

(4) 三—八译码器仿真波形如图2_20所示。

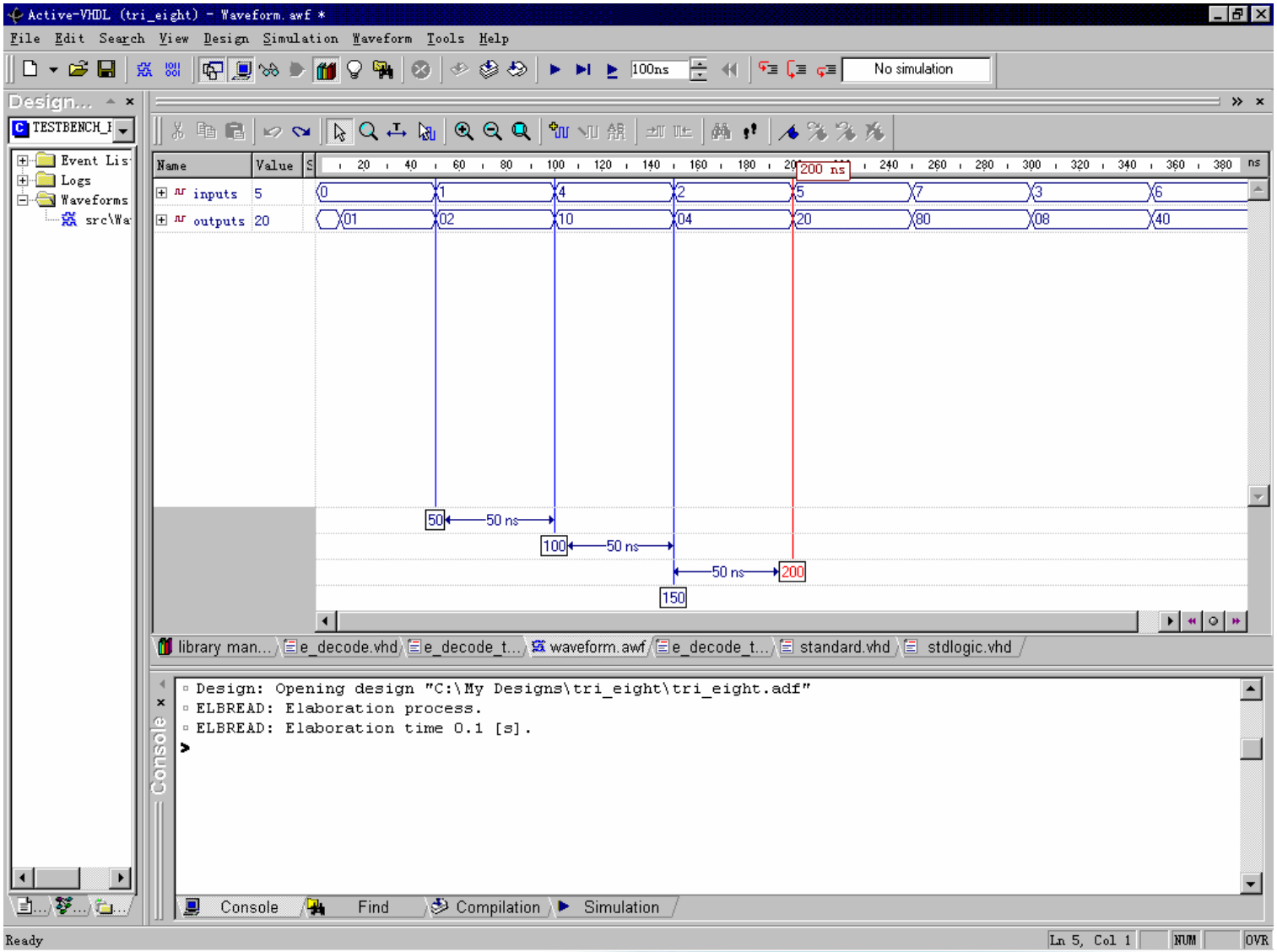



图 2 -20 三-八译码器仿真波形

结合本例中测试基准语句：

enables<=〞000〞,〞100〞*after* 10 ns; inputs<=〞000〞,〞001〞*after* 50 ns,〞100〞*after* 100 ns,〞010〞*after* 150 ns,〞101〞*after* 200 ns,〞111〞*after* 250 ns,〞011〞*after* 300 ns,〞110〞*after* 350 ns;

在本例所写的激励中，inputs输入端依次是“0”(对应000，下同)、“1”、“4”、“2”、“5”、“7”、“3”和“6”。 其输出outputs按三_八译码器的要求应为：“00000001”、“00000010”、“000100000”、“00000100”、 “00100000”、 “10000000”、“00001000”、“010000000”。其在图2-18中对应的值为：“01”(y0)、 “02”(y1)、“10”(y4)、 “04”(y2)、 “20”(y7)、“80”(y3)、“08”(y3)、“40”(y6)。

注意： outputs 输出“01”(y0)时， 有10 ns的延迟。

## 2.7.2 时序逻辑电路设计

时序电路可分为同步电路、 异步电路。 在同步电路中，所有触发器(或寄存器)的时钟都接在一根时钟线上； 而异步电路的各触发器(或寄存器)的时钟不接在一起。 在本书第四章将要介绍的FPGA、 CPLD器件中，常用的触发器为D触发器， 其它类型的触发器(如 T 、JK)都可由D触发器构成。

**1. D触发器**

(1) D触发器源代码：

```
library  IEEE;
use   IEEE.std_logic_1164.all  ;

entity  e_dff   is
    port (
          d:   in   BIT;
          clk:   in  BIT;
          q:   out   BIT);
end   e_dff;
architecture  e_dff  of   e_dff is
     begin
```

```
        process    (clk , d)
        begin
            if    clk′ event    and    clk=′ 1′      then
                q<=d;
            end if;
        end process;
end  e_dff;
```

(2) D触发器测试基准源代码：

```
entity    e_dff_tb  is
end    e_dff_tb;
architecture    TB_ARCHITECTURE    of    e_dff_tb is
        _ _ Component declaration of the tested unit
       component    e_dff
       port(
            d:    in BIT;
            clk: in BIT;
            q: out  BIT);
  end component;
  _ _ Stimulus signals _ signals mapped to the input and inout ports of tested
entity
```

```
signal    d: BIT;
signal    clk: BIT;
_ _ Observed signals _ signals mapped to the output  ports of tested entity
signal    q: BIT;
_ _ Add your code here ...
    begin
 _ _ Unit Under Test port map
      UUT: e    _dff
      port  map
              (d => d,
              clk => clk,
              q => q);
clk<='  0'  , '  1'  after 50 ns;
```

d<='0', '1' *after* 50 ns, '0' *after* 100 ns, '1' *after* 150 ns, '0' *after* 200 ns;

_ _ Add your stimulus here ...

*end* TB_ARCHITECTURE;

*configuration* TESTBENCH_FOR_e_dff of e_dff_tb *is*

*for* TB_ARCHITECTURE

*for* UUT: e_dff

*use entity* work.e_dff(e_dff);

*end for*;

*end for;*

*end* TESTBENCH_FOR_e_dff;

(3) D触发器仿真波形如图 2-21 所示。请对照D触发器的真值表，重新编写测试基准中有关语句， 得出新的仿真波形。

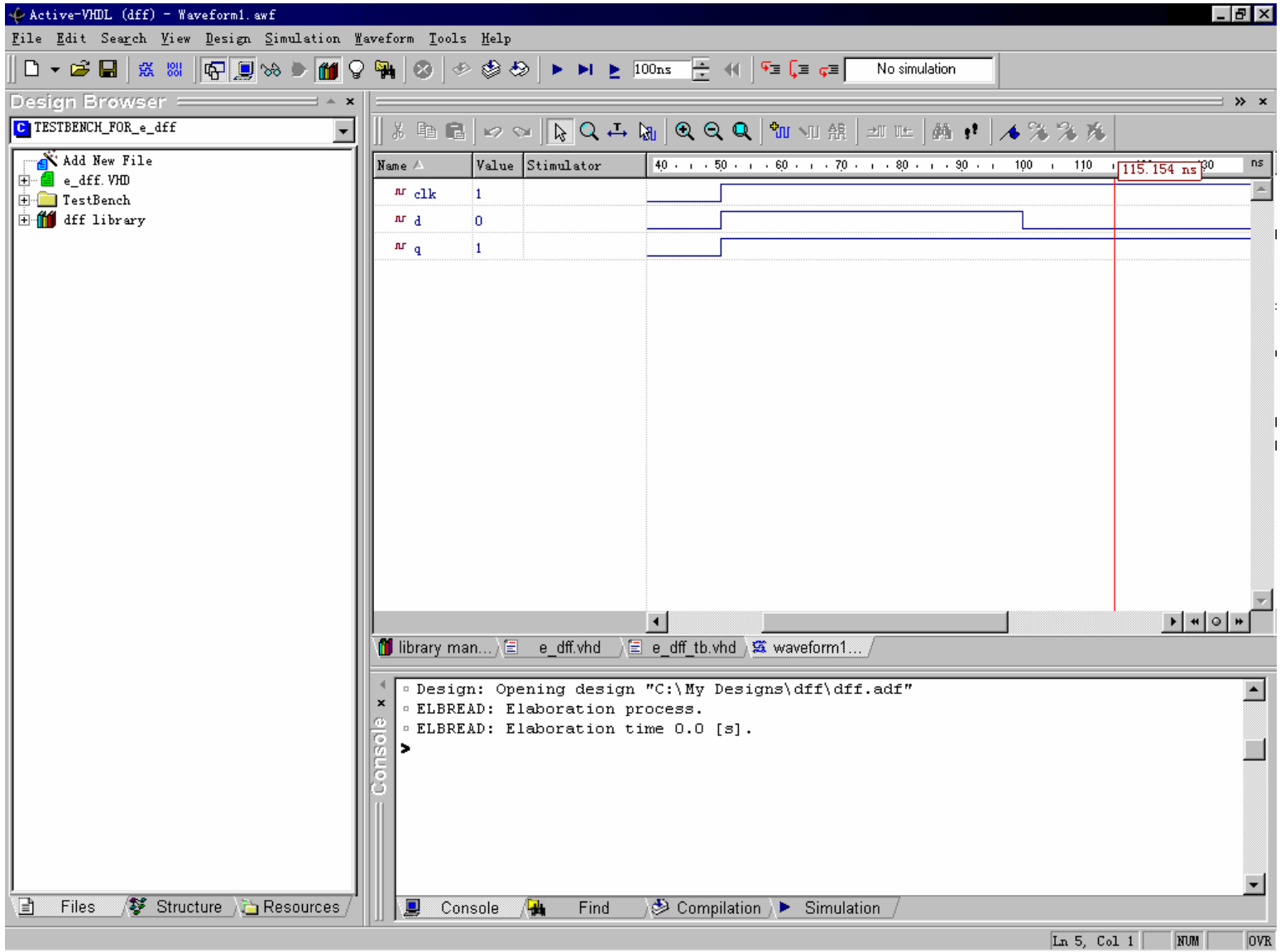



图 2-21 D触发器仿真波形

## 2. 可逆计数器

在时序应用电路中，计数器的应用十分普遍， 如加法计数器、 减法计数器、 可逆计数器等。 Active-VHDL提供了一个 8 位计数器范例，很值得学习研究。

(1) 计数器源程序：

```
library  IEEE;
use     IEEE.std_logic_1164.all    ;
```

```
entity    counter8    is
        port    (
         CLK: in  STD_LOGIC;
         RESET: in  STD_LOGIC;
         CE, LOAD, DIR: in  STD_LOGIC;
         DIN:  in  INTEGER  range  0  to  255;
         COUNT: out  INTEGER  range  0  to  255);
end    counter8;

architecture    counter8    _arch    of    counter8    is
begin
process    (CLK, RESET)
_ _auxiliary variable COUNTER declaration
_ _the output port ″ COUNT″ cannot appear on the right side of assignment
_ _ statements
```

```
variable  COUNTER: INTEGER range 0 to 255;
begin
        if    RESET=′ 1′   then
           COUNTER： =0;
        elsif    CLK=′ 1′      and    CLK′ event    then
            if    LOAD=′ 1′      then
                  COUNTER： =DIN;
           else
              if    CE=′ 1′      then
                 if    DIR=′ 1′      then
                    if    COUNTER =255    then
                             COUNTER： =0;
                    else
                             COUNTER： =COUNTER + 1;
```

```
            end if;
        else
            if    COUNTER =0    then
                    COUNTER： =255;
            else
                    COUNTER： =COUNTER_1;
            end if;
          end if;
      end if;
    end if;
  end if;
  COUNT <= COUNTER;
end process;
end    counter8_arch;
```

(2) 测试基准源程序：

```
library IEEE;
use IEEE.std_logic_1164. all;
use IEEE.STD_LOGIC_TEXTIO. all;
use STD.TEXTIO. all ;
entity testbench is
end testbench;
architecture testbench_arch of testbench is
file RESULTS: TEXT open WRITE_MODE is "results.txt";
component counter8
    port (
        CLK: in STD_LOGIC;
        RESET: in STD_LOGIC;
```

```
        CE, LOAD, DIR: in  STD_LOGIC;
        DIN: in  INTEGER    range 0 to 255;
        COUNT: out  INTEGER range  0  to    255);
end component;
shared variable     end_sim: BOOLEAN：=    false;
signal     CLK, RESET, CE, LOAD, DIR: STD_LOGIC;
signal     DIN: INTEGER     range 0 to 255;
signal     COUNT: INTEGER     range 0 to 255;

procedure     WRITE_RESULTS (
      CLK           : STD_LOGIC;
      RESET         : STD_LOGIC;
```

```
        CE          : STD_LOGIC;
        LOAD        : STD_LOGIC;
        DIR         : STD_LOGIC;
        DIN         : INTEGER;
        COUNT       : INTEGER
                          ) is
variable    V_OUT: LINE;
begin
        _ _ write time
        write(V    _OUT, now, right, 16, ps);
        _ _ write inputs
        write(V_OUT, CLK, right, 2);
        write(V_OUT, RESET, right, 2);
        write(V_OUT, CE, right, 2);
```

```
        write(V_OUT, LOAD, right, 2);
        write(V_OUT, DIR, right, 2);
        write(V_OUT, DIN, right, 257);
        _ _write outputs
        write(V_OUT, COUNT, right, 257);
        writeline(RESULTS, V_OUT);
         _ _
end    WRITE_RESULTS;
begin
        UUT: COUNTER8
         port map     (
                  CLK => CLK,
                  RESET => RESET,
```

```
            CE => CE,
            LOAD => LOAD,
            DIR => DIR,
            DIN => DIN,
            COUNT => COUNT);
CLK_IN:     process
begin
        if    end_sim = false then
            CLK <= ′ 0′ ;
            wait  for 15 ns;
            CLK <=′ 1′ ;
            wait  for 15 ns;
```

```
        else
            wait;
        end if;
    end process;

STIMULUS: process
    begin
        RESET  <= '1';
        CE     <= '1';      -- count enable

        DIR    <= '1';      -- count up
        DIN    <= 250;      -- input value
        LOAD   <= '0';      -- doesn't load input value
    wait for   15 ns;
```

```
RESET  <= ′ 0′ ;
wait for    1 us;

CE  <= ′ 0′ ;                          - - don′ t count

wait for    200 ns;
CE        <= ′ 1′ ;
wait for    200 ns;
DIR       <= ′ 0′ ;
wait for    500 ns;
LOAD <= ′ 1′ ;
wait for    60 ns;
LOAD  <= ′ 0′ ;
wait for    500 ns;
```

```
        DIN  <= 60;
        DIR  <= ′ 1′ ;
        LOAD <= ′ 1′ ;
        wait for 60 ns;
        LOAD <= ′ 0′ ;
        wait for 1 us;
        CE   <= ′ 0′ ;
        wait for 500 ns;
        CE   <= ′ 1′ ;
        wait for 500 ns;
        end_sim： =true;
        wait;
end process;
WRITE_TO_FILE: WRITE    _RESULTS(CLK, RESET, CE, LOAD,
DIR, DIN, COUNT);
End  testbench_arch;
```

(3) 计数器仿真波形如图 2-22 所示。

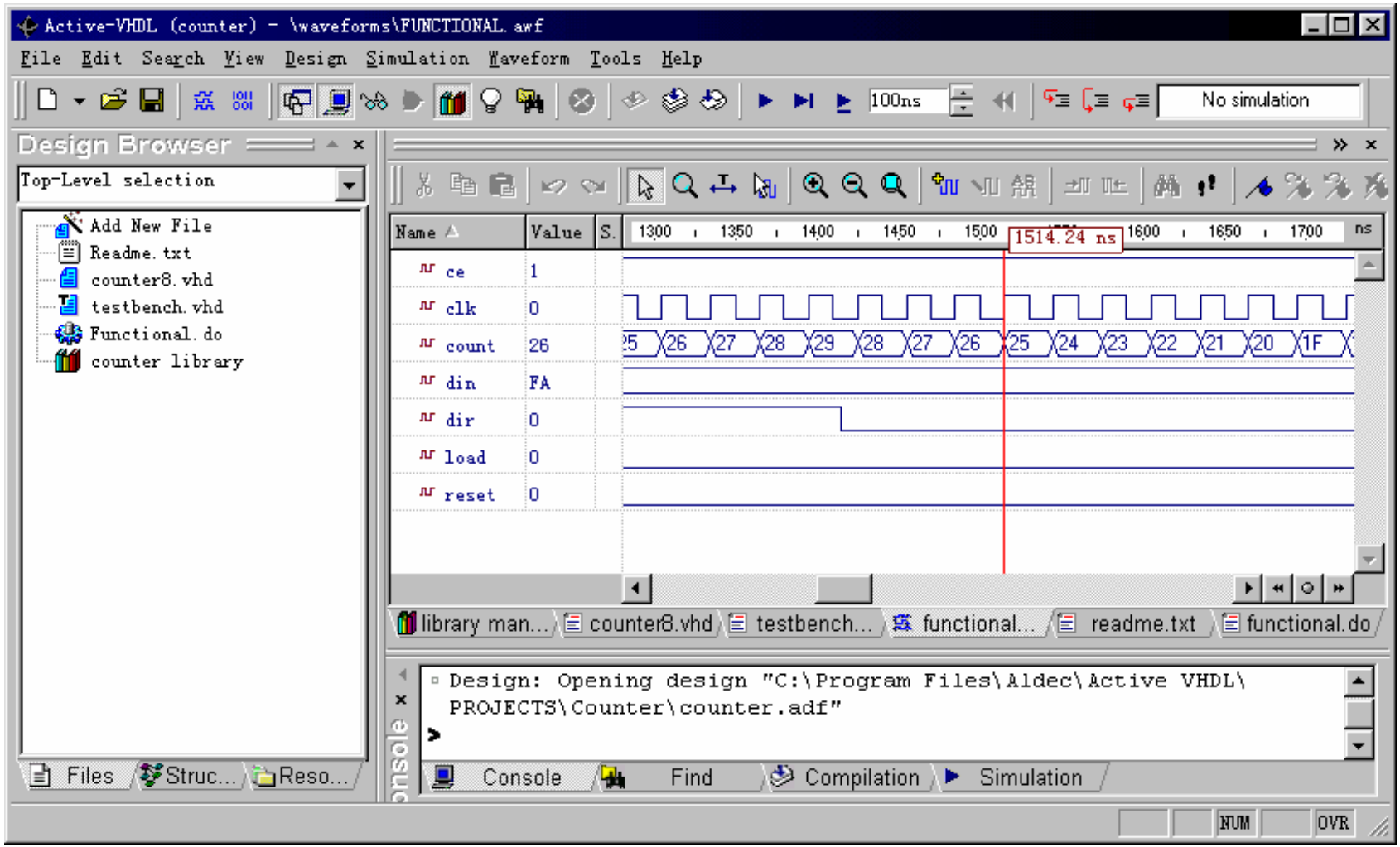



图 2-22 计数器仿真波形

(4) 提示： 结合计数器VHDL的源程序和仿真波形， 需要特别理解和注意以下几个方面：

① 一个 8 位计数器，它能计数的范围是 0～255。 同理， n 位的计数器所能计数的范围是 0～$2^n$-1。

② 减法计数器，计数方式除了和加法计数器的方向不同外， 其余是完全一样的。 在本例中， 由COUNTER：=COUNTER + 1;变为COUNTER： =COUNTER-1。 可逆计数器的计数方式是可加也可减。

③ 编写可逆计数器VHDL程序时，在语法上，就是把加法和减法计数器合并，使用一个控制信号决定计数器作加法或减法的动作。在本例中，利用“控制信号DIR”可以让计数器的计数动作加 1 或减 1 。

④ 在本例中，“控制信号DIR”在时间轴为 1415 ns时，由状态“1”切换到状态“0”，但由于此时并非时钟脉冲信号的上升沿，所以必须等到为 1425 ns时，计数器才开始改变成减法动作。这意味着由于芯片的传输延迟效应，计数器在延迟了 10 ns后，才输出减法计算的结果。

⑤ 为了防止信号超过一个时钟周期而造成计数器误操作、多计数，可利用微分电路加上计数器的结果构成同步计数器，从而避免系统不稳定的状况出现。

⑥ 本例中“CE”信号对波形的影响，如在 1015～1215 ns和3535～4035 ns期间，“CE”为0， 导致计数暂停。

⑦ 在仿真时，计数值到“FF”(即 255)后， 有看似噪声的现象出现，这就是出现计数值等于“100”(即FF+1的值)瞬间，但由于该时间十分短暂， 不致引起系统不稳定的状况。

### 3. 移位寄存器

在微处理器的算术或逻辑运算中，移位寄存器是一个基本的组件。移位寄存器的移位方式可以向左或向右移位，其主要由D触发器构成。下面是 8 位移位寄存器的VHDL 源程序：

```
library  IEEE;
use  IEEE.STD_LOGIC_1164.all  ;
use  IEEE.STD_LOGIC_ARITH.all  ;
use  IEEE.STD_LOGIC_UNSIGNED.all;
entity  Shift8  is
```

```
    port   (
            CP       : in  STD_LOGIC;          -- Clock
            DIN      : in   STD_LOGIC;         -- I/P Signal
            DIR      : in   STD_LOGIC;         -- Shift Control
            OP       : out   STD_LOGIC         -- Shift Result
           );
end    Shift8;
architecture   shifter   of   Shift8   is
   signal  Q:STD_LOGIC_VECTOR(7 DOWNTO 0); --Shift Register
  begin
      process (CP)
              begin
  if    CP′ event and CP=′ 1′ then
```

```
            IF DIR = ′ 0′ then     - - 8位左移
                Q(0) <= DIN;
                for I in 1 to 7 loop
                    Q(I) <= Q(I-1);
                end loop    ;
            else                          - - 8位右移
                Q(7) <= DIN;
                for I in 7 downto 1 loop
                  Q(I_1) <= Q(I);
                end loop    ;
            end if;
        end if;
    end process;
    OP <= Q(7) when DIR = ′ 0′ else                --移位输出
          Q(0);
end   shifter;
```

# 2.8 VHDL上机实践

## 2.8.1 VHDL数字电路的文本描述、 编译与仿真上机实验

### 1. 实验内容要求

熟悉VHDL语言的基本语法，熟悉Active　VHDL工具的使用。用VHDL文本描述语句输入一个 8 位全加器，编译仿真通过后将其加入库中。本实验结果要求以文档的方式传递。

## 2. 实验原理指导

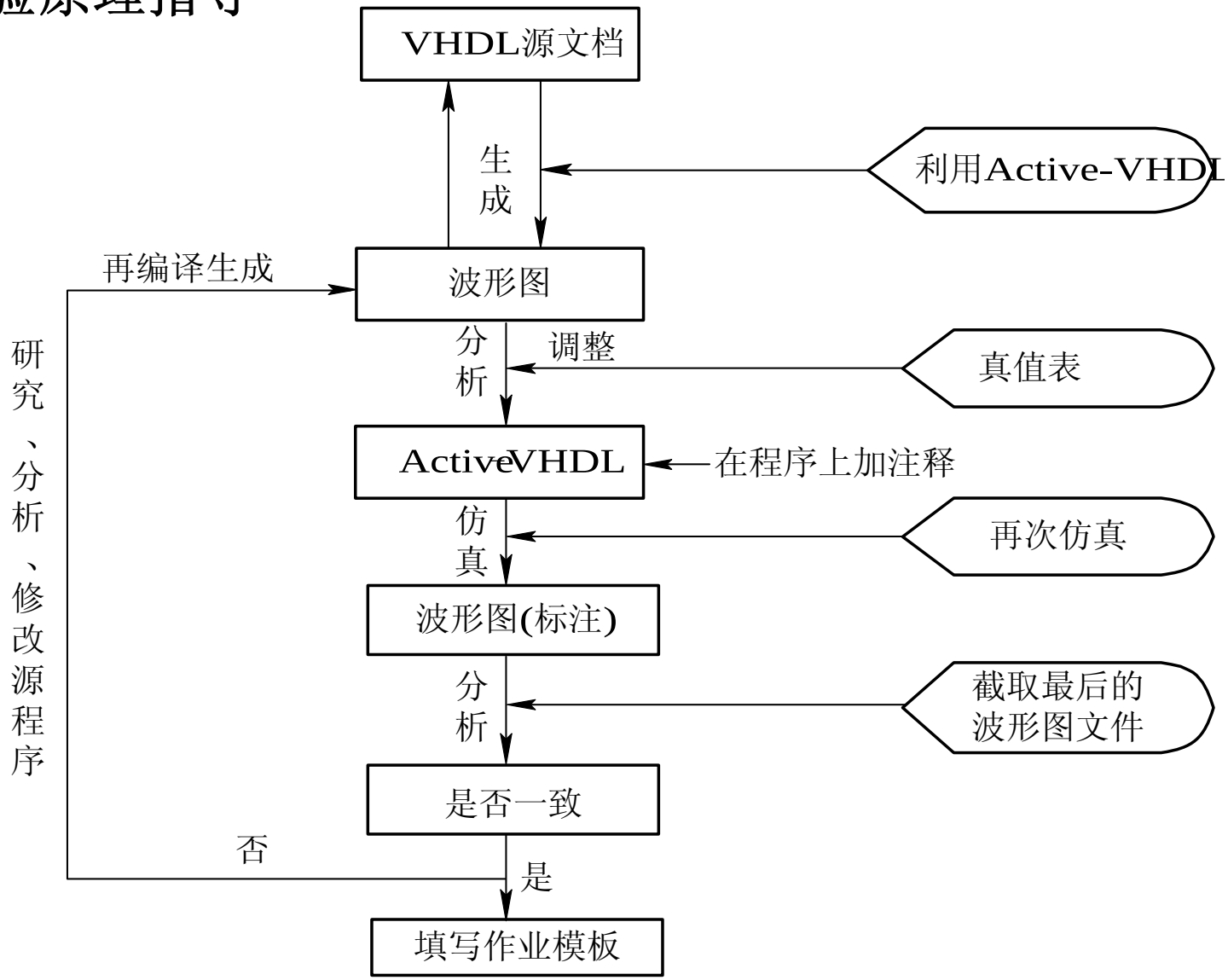



图 2-23 实验原理指导框图

### 3. 实验内容

8 位全加器、8 位全减器、 译码器、 计数器、 触发器（D、J-k、T）、移位寄存器、 八选一等基本器件至少任选一个。

## 4. 实验步骤指导

(1) 输入。例如输入一位全减器的电路描述，并保存该文件。

(2) 仿真电路并修正错误，分析波形，修改测试基准中部分内容， 分析波形变化情况。

## 2.8.2 交通灯控制器

交通灯控制器是典型的有限状态机(Finite State Machine，FSM)问题，设计的意图是在高速公路和乡村小路十字路口处实现交通灯无人自动管理。 此例最早见诸于由C.Mead和L.Conway合著的《超大规模集成电路系统导论》中的PLA设计举例。 数字系统的控制单元通常使用FSM或时钟模式时序电路来建模。每个控制步可以看作是一种状态，与每一控制步相关的转移条件指定了下一种状态和输出。有限状态机有两种类型： Moore型和Mealy型。对Moore型有限状态机， 输出只为有限状态机当前状态的函数，而对于Mealy型有限状态机，输出为有限状态机当前值和有限状态机输入值的函数。在实践应用中，有许多以控制为主的结构都可以模型化为行为综合期间有限状态机的通信网络。

## 1. 交通灯控制器分析和VHDL编程

设计一个如图2_24所示的控制交叉路口的交通灯控制器。在乡间小路的每一面上都有探测器来监测汽车出现的情况。只有在小路上发现有车时高速公路上的交通灯才有可能为红灯。一般情况下， 高速公路上的交通灯为绿色。

在图2_24中，F（Famroad）为乡间小路， H（Highway）为高速公路，C为探测器， HL为高速公路上的交通灯，FL为乡间小路上的交通灯。 本节以交通灯控制器的行为级设计为例，说明VHDL程序的设计方法。设计交通灯控制器按以下步骤进行：

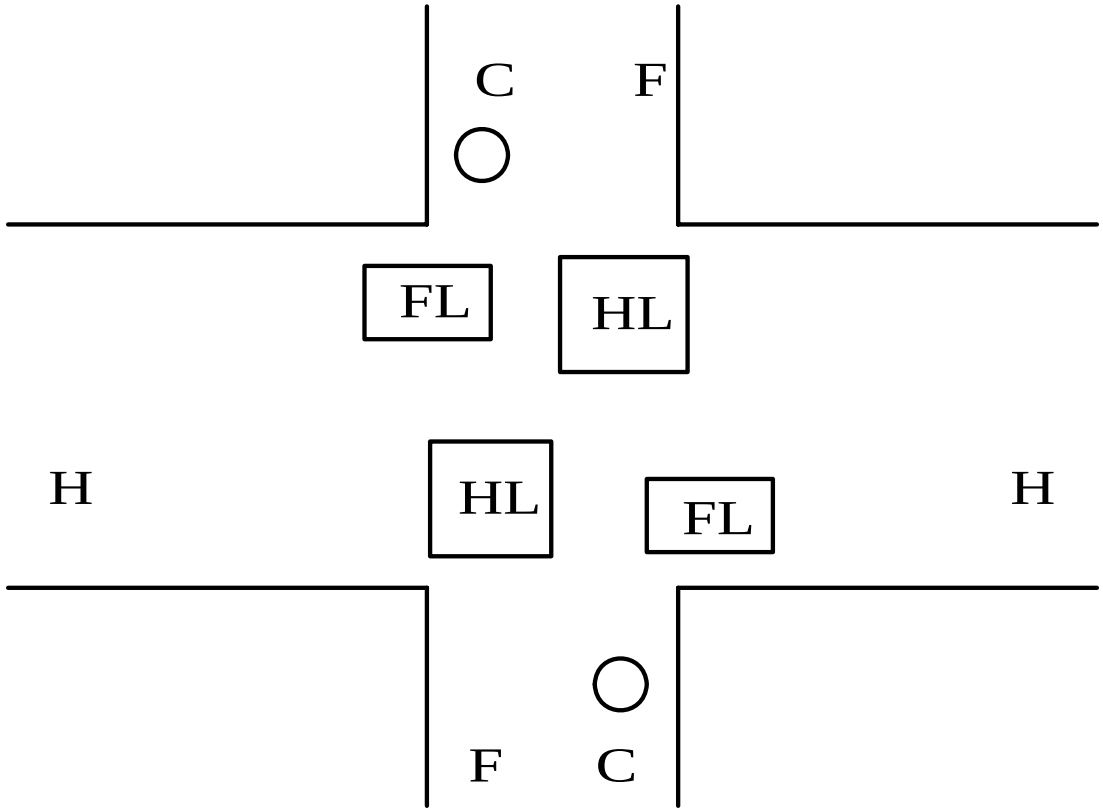



图 2-24 公路和小路的十字路口交通灯

(1) 规格设计，产生详细说明。首先要确定设计问题的特性。本例中， 交通灯控制器需要经历四个状态：

① (稳态)HL=绿，FL＝红：当小路上有汽车，且公路上的交通灯为绿的时间达到了限定时间(long _time)时，则HL=黄， FL不变，仍为红色；

② HL＝黄， FL=红：当公路灯为黄色的时间达到了限定时间(short _time)时，则系统转到 FL＝绿， HL＝红的状态；

③ FL＝绿，HL=红：当小路上没有汽车或小路上的交通灯为绿的时间达到了限定时间(long_time)时，则转到FL=黄，HL＝红的状态；

④ FL=黄，HL＝红：当小路上的交通灯为黄的时间达到了限定时间时，则转到第一个状态；如此循环下去。

(2) 定义系统类型。经以上分析，可知系统有四种状态。为了将这些信息编码化，应该定义一个拥有灯的颜色类型和系统状态类型的包。

```
package traffic_package is
    type color is (green, yellow, red);  -- 枚举类型定义颜色
    type statc is (highway_light_green, highway_light_yellow,
                   farmroad_light_green, farmroad_light_yellow);
end Traffic_Package;
```

注意： State类型定义了系统的每个状态。 可以注意到公路灯为绿时， FL＝红； 反之， FL=绿时， HL=红， 所以不需要单独设置交通灯为红色的状态。

(3) 产生接口，完成实体说明。这一步用于在设计实体的实体说明部分中定义系统的输入、输出。

因为系统要控制四条路上的灯，所以需要两个输出信号分别控制公路灯和小路灯，称作highway_light和farmroad_light，其值由Color类型决定。

因为当发现小路上有车时，系统就作出反应，所以系统需要一个输入，称Car_ on_farmroad, 它是布尔量。

系统中还有两个外部环境决定的常数：交通灯保持绿色的允许持续时间 long_time和交通灯保持黄色所应持续的时间short_time。VHDL中允许通过类属从外界输入这些值。这样便可以定义实体：

```
use work.traffic_package.all
    entity    traffic_light_controller    is
              generic   (long_time ：  time；  short_time：  Time)；
              port   (car_on_farmroad：  in boolean；
                        highway_light:    out    Color;
                        farmroad_light:    out    Color;
    end    Traffic_light_controller
```

(4) 结构体描述。如前所述，可得到输入输出名称和状态转换表，见表 2-5， 由此可以给出交通灯控制器的行为描述。

**表 2-5 交通灯控制器状态转换表**

| 状　　态 | 输　出<br>highway_light | 输　出<br>farmroad_light | 输　入<br>(time_out 表示超过限定的延迟时间) | 后继状态 |
|---|---|---|---|---|
| highway_light_green | green | red | car_on_farmroad =1<br>and time_out_long =1 | highway_light_yellow |
| highway_light_yellow | yellow | red | time_out_short =1 | farmroad_light_green |
| farmroad_light_green | red | green | car_on_farmroad =0<br>or time_out_long =1 | farmroad_light_yellow |
| farmroad_light_yellow | red | yellow | time_out_short =1 | highway_light_green |

为了确定一个新状态应保持多长时间，系统必须包含一个时间指示器(见结构体中最后一个进程)。每进入一个新状态， 相应的定时器就开始工作， 当超过 long_time和 short_time时， 将修改状态。 所以需要 3 个信号为计数器提供输入输出， 它们在结构体中命名为 start_timer， time_out_long, time_out_short。

由表 2-5 和以上分析， 可以得到交通灯控制器的行为描述：

```
architecture behavior of Traffic_light_controller is
    signal present _state:state： =highway_light_green;
    -- present_state用于保存系统当前所处的状态， 初始化为:
    -- highway_light_green;
    signal time_out_long： boolean： =false;
    signal time_out_short： boolean： =false;
    signal start_timer： boolean： =false;
begin
    control_process： -- 状态转换进程
        process (car_on_farmroad, time_out_long, time_out_short)
```

```
begin
  case   present_state   is
   when   highway_light_green =>
      if   car_on   _farmroad   and   time_out_long   then
          start_timer：= not start_timer;
           present_state <= highway_light_yellow;
      end if;
   when   highway_light_yellow =>
      if   time_out_short   then
           start_timer：= not start_timer;
           present_state <= farmroad_light_green;
    end if;
  when  farmroad_light_green =>
```

```
    if  not car_on_farmroad or time_out_long then
          start_timer：=not   start_timer;
       present_state <= farmroad_light_yellow;
      end if;
      when    farmroad   _light   _yellow =>
    if    time_out_short then
      start_timer：= not  start_timer;
    present_state <= highway   _light   _green;
    end if;
  end case;
end process;
```

```
-- 选择信号赋值语句完成所有状态对输出信号的控制;
highway_light_set:
        with  present_state  select
              highway_light <= green  when  highway_ligh_green;
                               yellow when  highway_light_yellow;
                               red    when  farmroad_light_green or
                              farmroad_light_yellow;
farmroad_light_set:
        with  present_state select
             farmroad_light <= green when farmroad_ligh_green;
                               yellow when  farmload_light_yellow;
                              red     when     highway_light_green or
                              highway_light_yellow;
```

```
-- 时间指示器
timer_process:
        process (start_timer)
                begin
                        time_out_long <= false, true  after  long_time;

                        -- 先关闭超时信号， 并在类属延时后激活它们

                        time_out_short <= false, true after short_time;
        end process;
end behavior;
```

注意： 用不同方法可以得出不同的程序， 根据以上分析方法来设计的交通灯控制器的源代码如下：

（1） 交通灯源文件名： e_traffic_con.vhd

```
package traffic_package is
    type  color is(green, yellow, red); --nemurical type defines colors
    type   state is(highway_light_green, highway_light_yellow,
farmroad_light_green, farmroad_light_yellow);
    end    traffic_package;
    use    work.traffic_package.all    ;
    entity     e_traffic_con is
       generic (long_time:time ： =80ns;
                    short_time:time ： =40ns);
```

```
        port (
                car_on_farmroad: in BOOLEAN;
                highway_light: out color;
                farmroad_light: out color
        );
end    e_traffic_con;

architecture    e_traffic_con of    e_traffic_con     is
    signal     present_state:state： =highway_light_green;
    signal     time_out_long:boolean： =false;
    signal     time_out_short:boolean： =false;
    signal     start_timer:boolean： =false;
```

```
begin
    control_process:
        process(car_on_farmroad, time_out_long, time_out_short)
          begin
            case   present_state    is
              when  highway_light_green=>
                if  car_on_farmroad  and  time_out_long then
                  start_timer<=not  start_timer;
                  present_state<=highway_light_yellow;
                end if   ;
              when   highway_light_yellow =>
                if   time_out_short  then
                   start_timer<=not   start_timer;
                   present_state<=farmroad_light_green;
                end if   ;
```

```
when  farmroad_light_green=>
    if  not  car_on_farmroad  or  time_out_long  then
       start_timer<=not  start_timer;
       present_state<=farmroad_light_yellow;
    end  if;
 when  farmroad_light_yellow=>
    if    time_out_short then
          start_timer<=not  start_timer;
          present_state<=highway_light_green;
    end if;
  end case;
end process;
--select signal tocontrol all state
 highway_light_set:
```

```
    with  present_state  select
     highway_light<=green when  highway_light_green,
                          yellow  when    highway_light_yellow,
                          red  when  farmroad_light_green | farmroad_light_yellow;
     farmroad_light_set:
with    present_state select
     farmroad_light<=green when  farmroad_light_green,
                        yellow when    farmroad_light_yellow,
                        red  when    highway_light_green | highway_light_yellow;


 --timer:
     timer_process:
          process(start_timer)
```

```
        begin
            time_out_long<=false, true after long_time;
            time_out_short<=false, true after short_time;
    end process;
-- <<enter your statements here>>

end  e_traffic_con;
```

(2) 交通灯测试基准文件名： e_traffic_con_tb.vhd

```
use  work.traffic_package.all;

        --Add your library and packages declaration here ...

entity     e_traffic_con_tb     is
    -- Generic declarations of the tested unit
   generic(
          long_time: time：  =80.0 ns;
          short_time: time：  =40.0 ns);
end    e_traffic_con_tb;
```

```
architecture    tb_architecture    of    e_traffic_con_tb    is
    -- Component declaration of the tested unit
    component    e_traffic_con
    generic(
        long_time: time：=80.0 ns;
        short_time: time：=40.0 ns);
    port(
        car_on_farmroad: in     boolean;
        highway_light:    out     color;
        farmroad_light: out     color);
end component;
```

```
    -- Stimulus signals _ signals mapped to the input and inout ports of
tested entity
    signal     car_on_farmroad: boolean;
    -- Observed signals- signals mapped to the output ports of tested entity
    signal     highway_light: color;
    signal     farmroad_light: color;
    -- Add your code here ...
begin
    -- Unit Under Test port map
    UUT: e_traffic_con
        port map
            (car_on_farmroad => car_on_farmroad,
```

```
                highway_light => highway_light,
                farmroad_light => farmroad_light);
            -- Add your stimulus here ...
            car_on_farmroad<=false, true after    50ns, false after  300ns,
true after  400ns,
            false  after    600ns;

end    tb_architecture;

configuration    TESTBENCH_FOR_e_traffic_con of e_traffic_con_tb is
        for     tb_architecture
```

```
        for     UUT: e_traffic_con
                use entity     work.e_traffic_con(e_traffic_con);
        end  for;
    end  for;
end     TESTBENCH    _FOR    _e    _traffic    _con;
```

交通灯的测试基准文件调试如图 2_25 所示。 交通灯控制器调试通过后的波形图如图 2_26 所示。

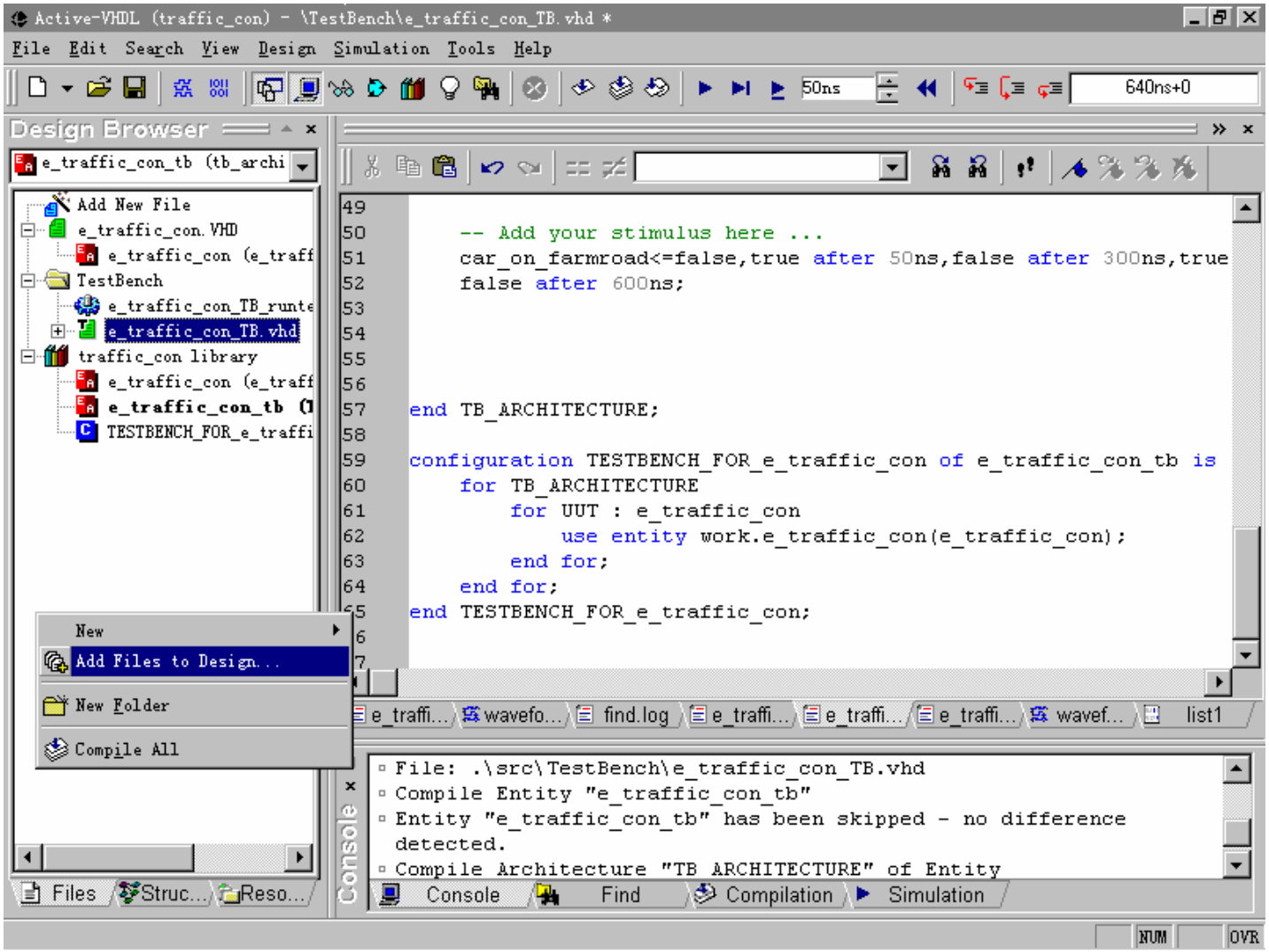



图 2-25 Active-VHDL下的测试基准文件调试

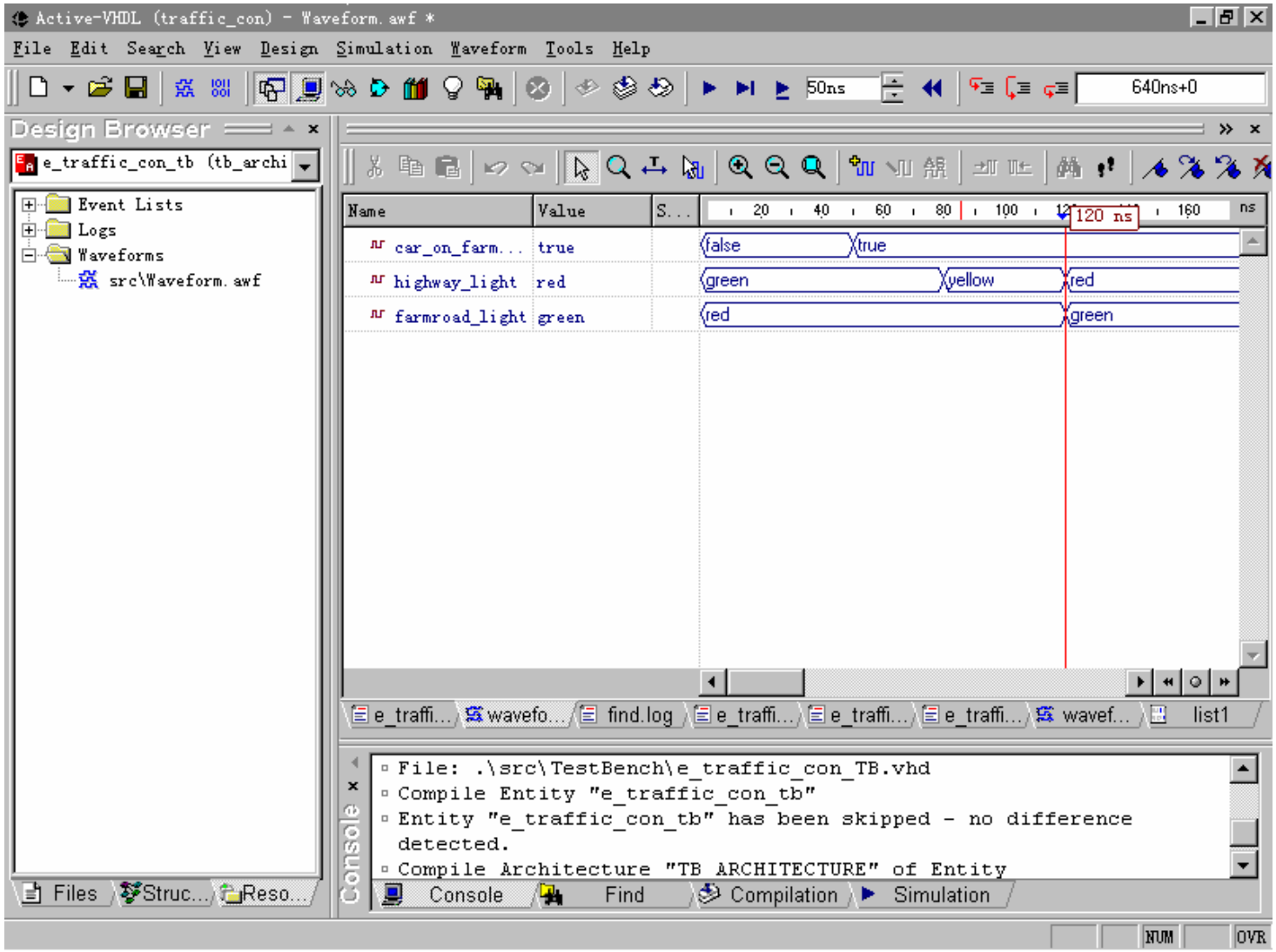



图 2-26 交通灯控制器波形图

## 2. 交通灯控制器状态表和状态图

设计一个如图 2-24 所示的控制交叉路口交通灯控制器。在乡间小路的每一面上都有探测器来监测汽车出现的情况。只有在小路上发现有车时高速公路上的交通灯才有可能为红灯。一般情况下， 高速公路上的交通灯为绿色。

交通灯控制器状态表如表 2-6 所示。其中，状态 4 为初始状态(实际上该状态为冗余状态)， 剩余 4 个状态为稳定状态。其状态转换图如图 2-27 所示。

# 表 2-6 交通灯控制器状态表

| | HL | FL | 值 |
|---|---|---|---|
| 状态0 | 绿 | 红 | 000 |
| 状态1 | 黄 | 红 | 001 |
| 状态2 | 红 | 绿 | 010 |
| 状态3 | 红 | 黄 | 011 |
| 状态4 | 红 | 红 | 111 |

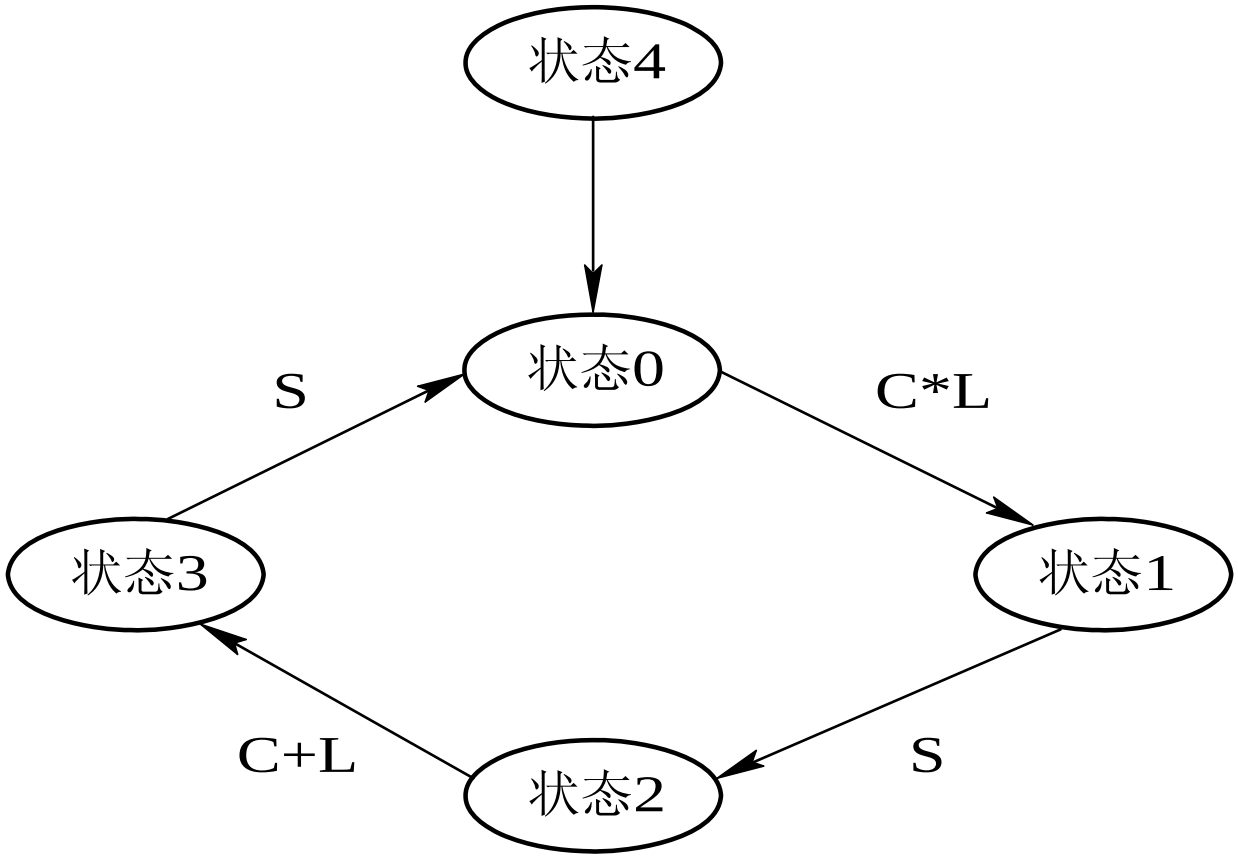



图 2-27 交通灯控制器状态转换图

在图 2_27 中，C表示小路上有车; L表示过了一段长的时间; S表示已过了一段短的时间; 操作符“*”表示逻辑与的关系; 操作符“+”表示逻辑或的关系。 描述中定义变量current_state表示当前状态，其类型为BIT_VECTOR(2 *downto* 0)，5 种状态分别由它的 5 个值来代表，如 111 代表状态 4。

### 3. 上机实验内容和步骤

(1) 调通并分析源程序和激励程序。程序中进程说明部分的几个变量的意义如下：

newstate：　　下一个状态值；

current_state：　当前状态值；

newHL：　　高速公路灯的状态，3位位长的每一位表示绿、 黄、红灯的亮、 灭状态；

newFL：　　乡间公路灯；

newST：　　用于启动外部计时器的输出位。

(2) 实验步骤指导如下：

① 将源程序输入到Active　VHDL；

② 将激励程序输入到Active　VHDL；

③ 编译、 仿真该电路；

④ 对波形结果进行分析；

⑤ 填写作业模板。 参见附录一给出的上机作业模板范例。

# 复习思考题

1. 什么是VHDL? 采用VHDL进行数字系统设计有哪些优点?

2. 信号和变量在描述和使用时有哪些主要区别?

3. 配置的主要功能有哪些? 试举例说明。

4. 试写出RS触发器测试基准， 并在Active　VHDL中验证。

5. 在一个设计文件中， 写出二输入与门、 与非门、 或门、或非门、 异或门的VHDL模型。

6. 编写八位全减器， 并比较它和一位全加器的异同。

7. 上机调试本章给出的八位移动寄存器VHDL语言描述，并编写测试基准以及分析波形。

8. 试编写一个八位可控制计数器，要求考虑计数值预置输入端、 计数输出端、 控制代码、 选通信号及计数时钟。