第二章

硬件描述语言VHDL

VHDL是什么？

- HDL → Hardware Description Language；
- VHSIC → Very High Speed Integrated Circuit；
- VHDL → VHSIC Hardware Description Language；

- VHDL是一个硬件描述语言，IEEE审定的工业标准；
- 80年代初期美国政府超高速集成电路（ VHSIC ）发展计划的衍生物；
- VHDL的IEEE国际标准：IEEE Std 1076
  - IEEE std 1076-1987, 1076-1993, 1076-2000 ， 1076-2002…..

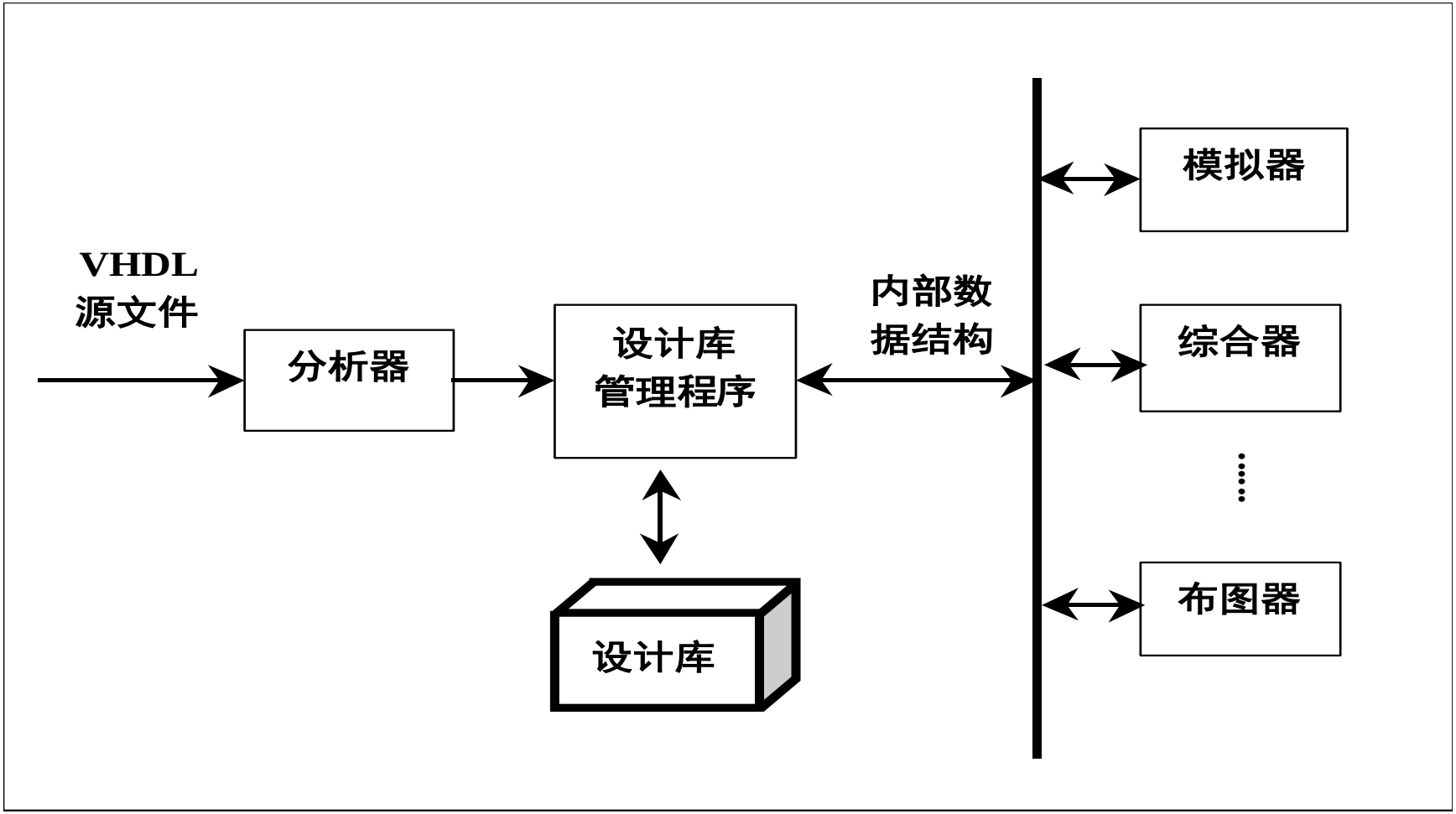
VHDL的发展历程

- 1981年6月，美国成立了VHDL工作小组；
- 1983年6月，由 Intermitrics, IBM和 Texas Instrument 组成开发小组，任务是：
  - 提出语言版本；
  - 开发其软件环境。
- 1987年12月，IEEE公布了 IEEE-1076 作为HDL的第一个标准；
- 1993年， IEEE公布了VHDL_93;
- 1999年： VHDL_AMS（Analog Mixed Signal）
- 1999年： VHDL1076.6(RTL可综合子集)；
- 2000年1月公布了VHDL1076-2000;
- 2000年5月公布了VHDL1076-2002;

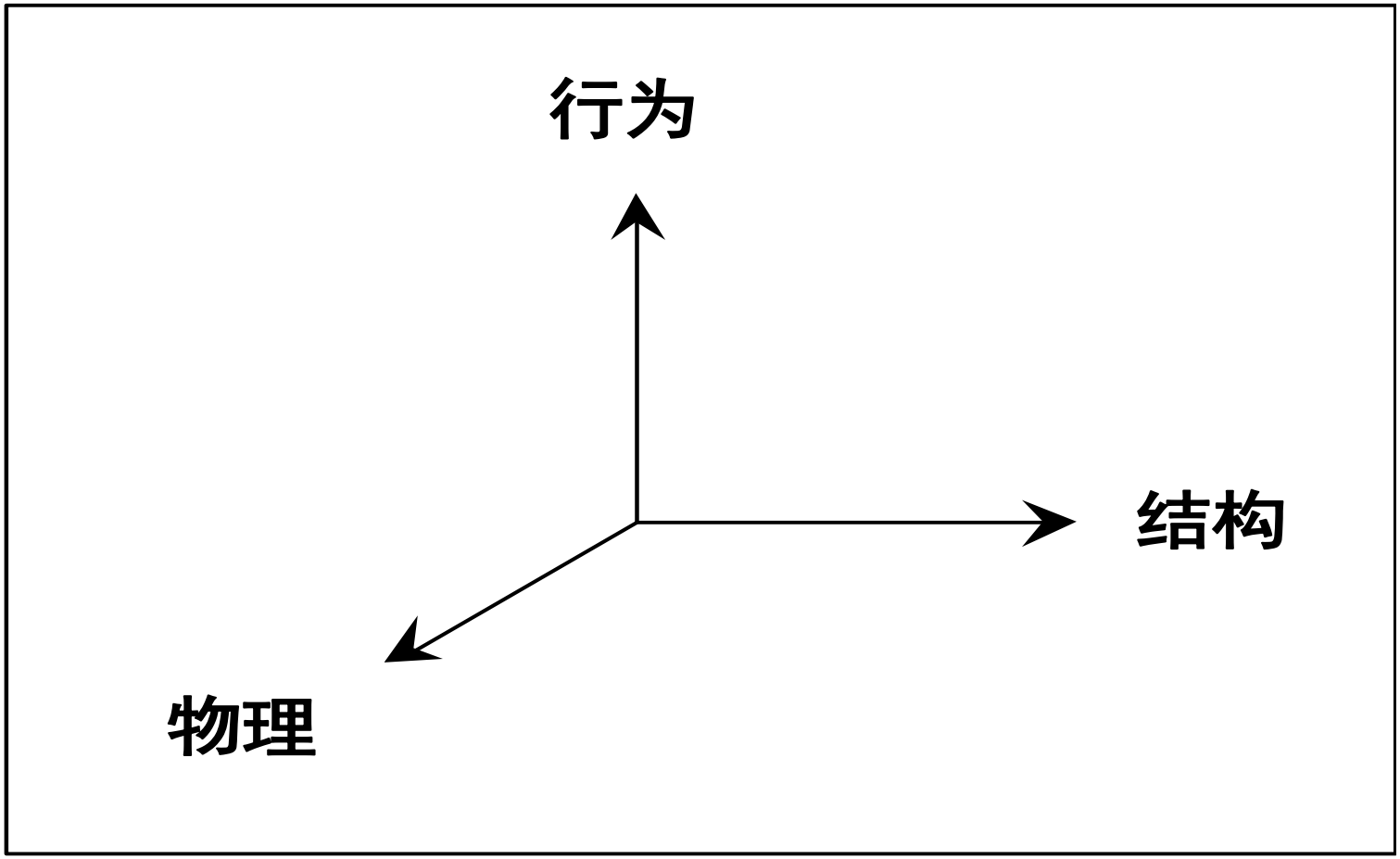
VHDL的主要目标

- 覆盖多个层次的广谱的HDL
  - 面向高层次的行为（算法）描述；
  - RTL级描述；
  - 门级描述；
  - 电路级描述（不太方便）；
  - 物理参数描述（延时，功耗，频率，尺寸.....）；
- 可读性好，既可被计算机接受，又可作文档；
- VHDL描述与工艺无关 ⇒ 有较长的生命期；
  （与工艺有关的参数，可通过VHDL的Attribute加进去）
- 因为是标准 ⇒ 有通用性。

VHDL的设计环境



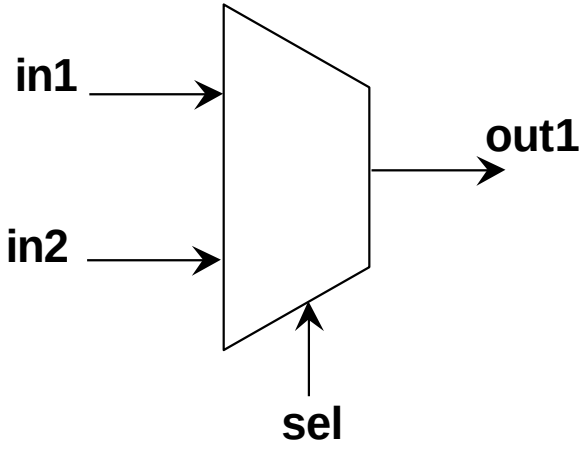
HDL的 3 方面特性



VHDL行为描述和结构描述

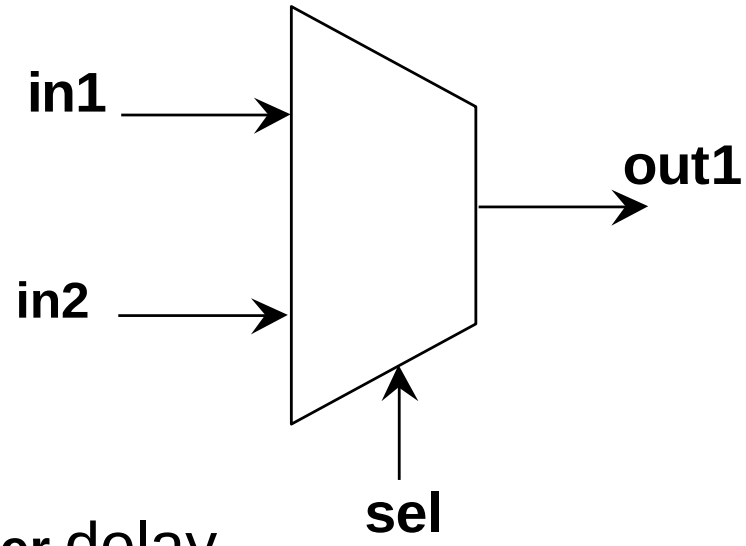
- **以2选1多路器为例**

  实体说明（下）对应于框图（左）



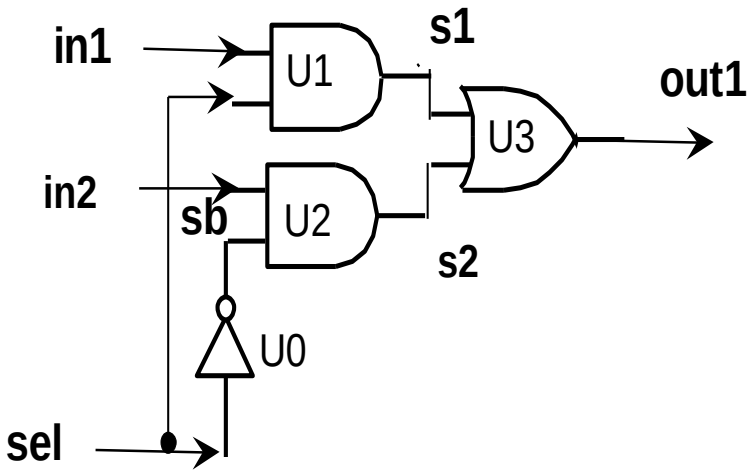

```
entity mux is
  begin
    port( in2, in2, sel :  in  bit ;
                 out1:           out  bit );
    generic( delay := 5ns );
  end;
```

VHDL行为描述



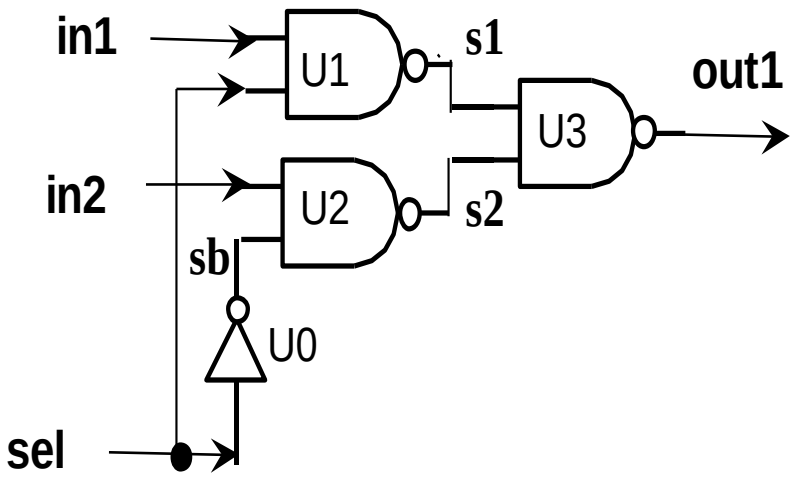

```
architecture muxbh of mux is
    begin
      if sel = '1' then out1 <= in1 after delay
      else out1 <= in2 after delay;
  end;
```

VHDL结构描述之一




```
architecture mux1 of  mux is
  begin
     NOT: sb == U0 ( sel ) --示意
     AND2: s1 == U1( sel, in1 );
     AND2: s2 == U2 ( sb, in2 );
     OR2:  out1 == U3 ( s1, s2 ）;
  end;
```

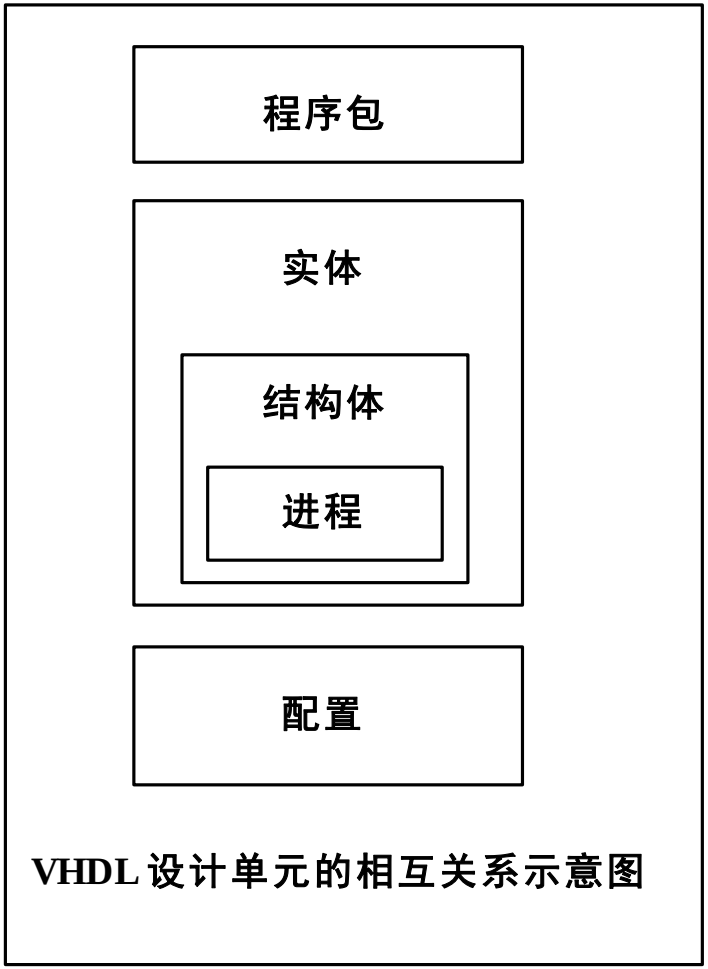
VHDL结构描述之二




```
architecture mux2 of  mux is
  begin
     NOT: sb == U0 ( sel )   --示意
     NAND2: s1 == U1( sel, in1 );
     NAND2: s2 == U2 ( sb, in2 );
     NAND2:  out1 == U3 ( s1, s2 );
  end;
```

VHDL的基本概念和主要特点

- VHDL基本单元：（4种说明）
- 程序包（**package** ），属任选项。
- 实体（**entity**），是必选项。
- 结构体（**architecture**），是必选项。
- **配置（configuration ），属任选项。**



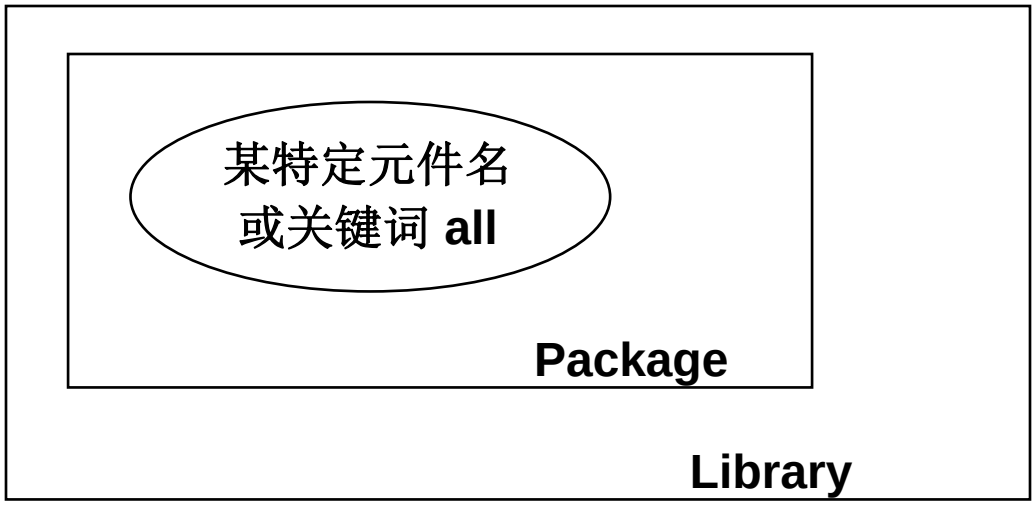
VHDL 设计单元的相互关系示意图

程序包

- 程序包分2部分：
  - package declaration
  - package body
- 用法：
  use < 程序包名>

程序包的内容：

- 数据类型定义
- 函数
- 过程



编译单元和设计库

- 编译单元共计5种：
  - 实体，结构体；
  - 包，包体；
  - 配置单元；
- 编译单元均可单独作为一个文件，也可放在一起。

- 设计库：存放各编译单元的地方。例：
- STD库：
  - STANDARD包；
  - TEXTIO包；
  - …..
- WORK库：当前的编译单元所在的库。
- 其它库

程序包（续）

- 将程序包的声明部分和包体部分分开，可以实现内部保密的目的；
- 程序包可用来定义多种数据类型以及对这些数据类型的运算。
- 数据抽象化：

  如果程序包提供了对某数据类型的所有运算，就可看作把类型的实际定义从设计中提取了出来，也就是说使用这种数据类型的用户不必了解这种类型的对象是如何构造的，而只需了解这种数据类型支持哪些运算即可。

程序包（续）

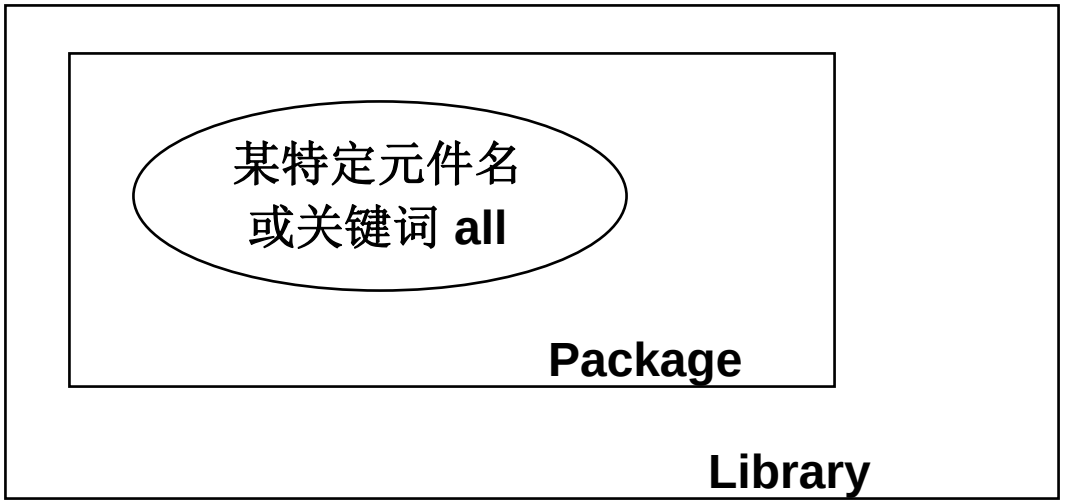
- 常用的程序包:
  - IEEE.std_logic_arith -- 算术运算函数
  - IEEE.std_logic_signed -- 带符号的算术运算函数
  - IEEE.std_logic_unsigned -- 不带符号的算术运算函数
  - IEEE.std_logic_1164 -- std_logic 数据类型及有关函数
  - ALTERA.maxplus2 -- Altera 宏单元的元件说明

如何使程序包成为可见

- **library** <library name>; **-- 打开一个库**
- **use** <library name>.<package name>**.all;** **-- 打开一个程序包**

  也可用某个特定的元件名代替 all
  - < library name >**可以是 IEEE 或 ALTERA;**
  - **在 MAX+PLUS II中,** <library name> **是**

    c:\maxplus2\max2vhdl **的一个子目录 。**



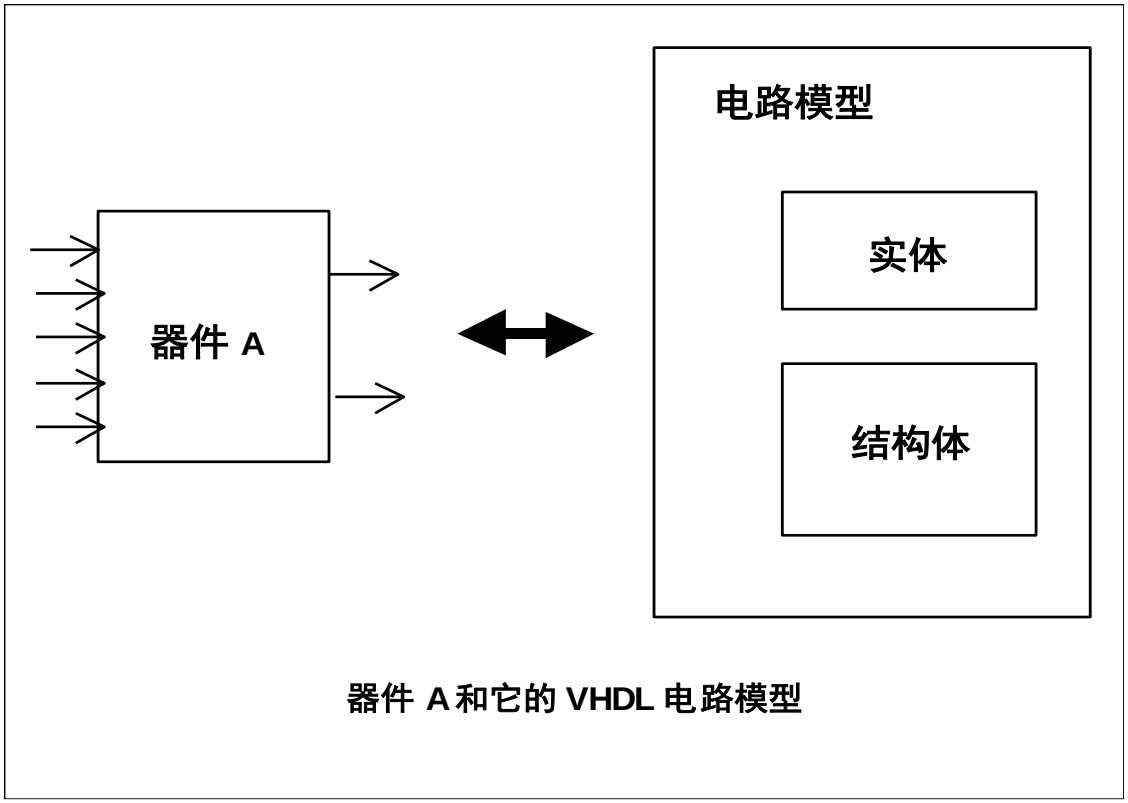
用户自定义程序包

- 用户自定义程序包必须和自己的设计在同一目录下。
- 使用自定义程序包的方法:

```
library work;          -- 此语句可以省略，因为WORK库
                          永远可见（VHDL的默认规则）。
use work.<package name>.all;
```

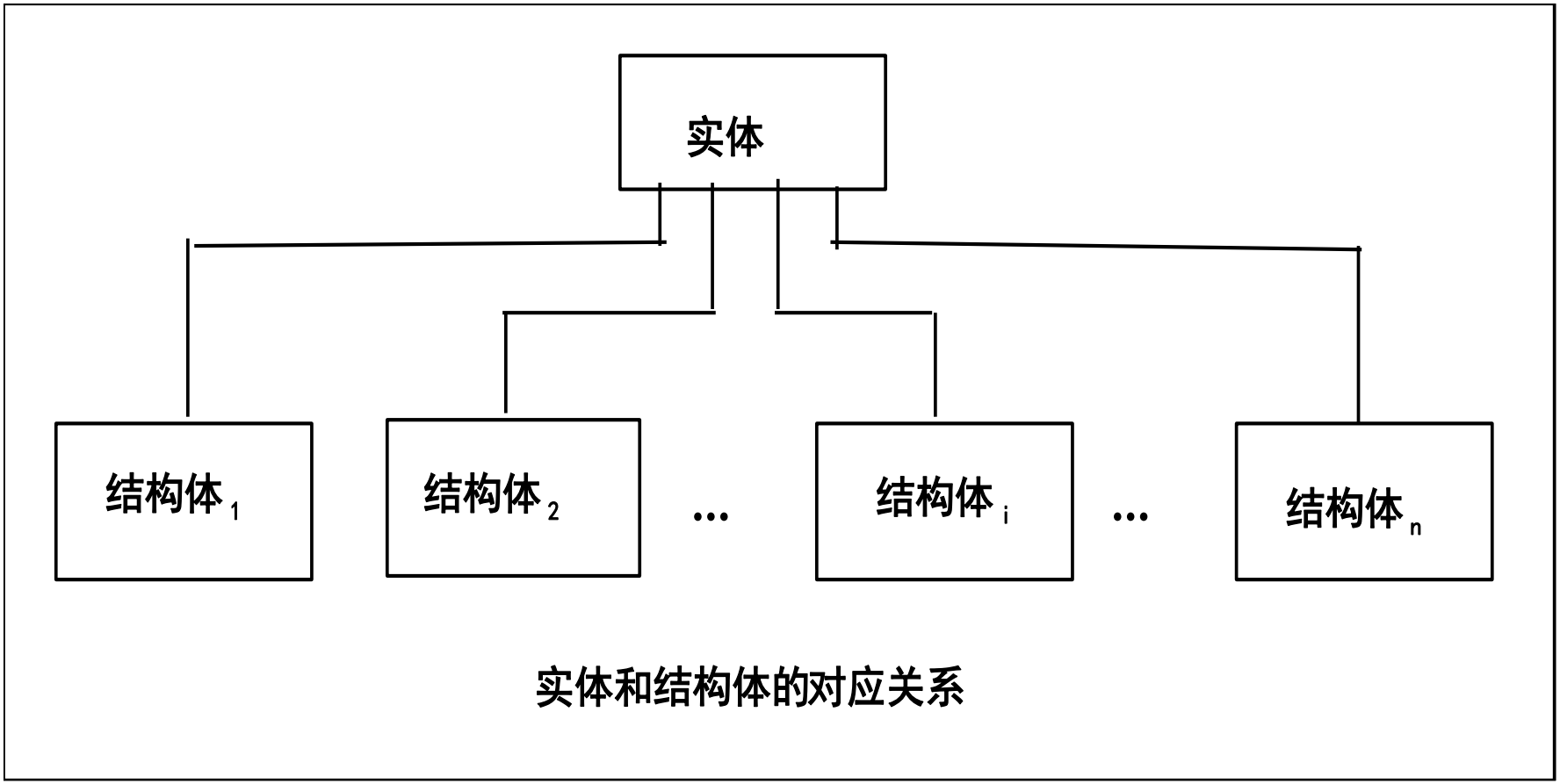
实体和结构体

- 实体说明：描述电路的接口信息，简称实体，保留字entity。
- 结构体：描述电路的行为或结构，保留字architecture。



器件 A 和它的 VHDL 电路模型

实体和结构体的对应关系



实体和结构体的对应关系

实体说明的语法形式

-- 提供公共信息

entity 〈实体名〉 **is**

　　[port(〈端口表〉);] ┐
　　[generic(〈类属表〉);] ┘ 〈实体说明部分〉外部可见

begin

　　[〈实体语句部分〉;] ┐ 外部不可见

end [〈实体名〉];

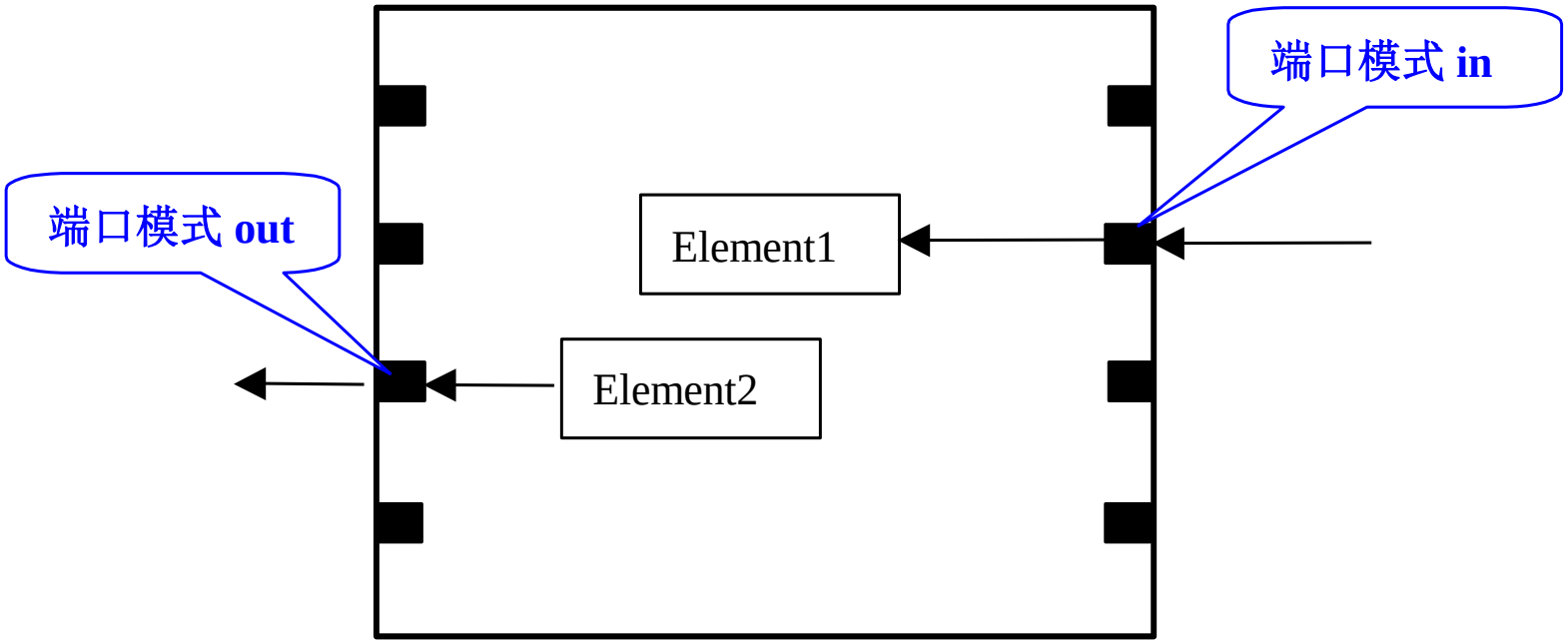
实体说明

- 公共信息中的可见部分：
  - 端口名称
  - 端口信息
    - 端口模式： in, out, inout, buffer
    - 信号取值类型： 二值逻辑、多值逻辑、整数、实数、记录、数组……
  - 类属信息
- 公共信息中外部不可见部分：

  -- 实体语句部分是可选项，包括：
  - 类型说明；
  - 断言语句（用于约束条件的判断，典型例子是触发器的建立时间和保持时间）等。

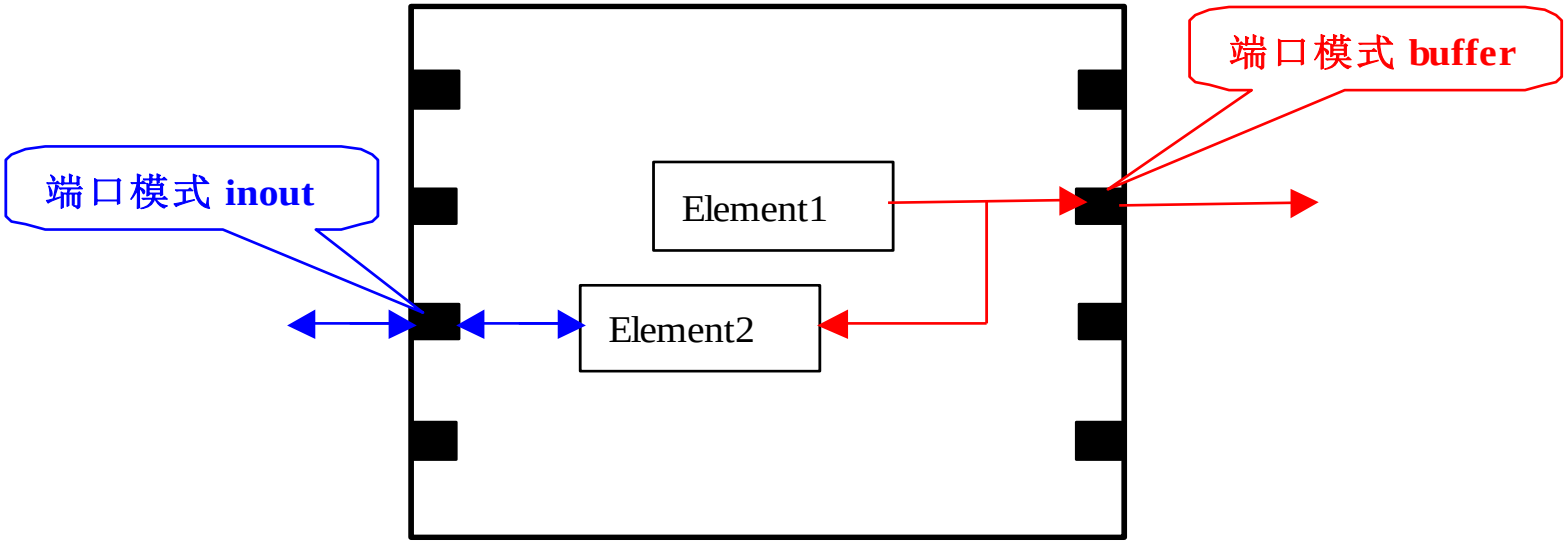
端口模式

- 端口模式： in, out, inout, buffer
  - 连接到in端口的信号，只能出现在赋值符号的右边；
  - 连接到out端口的信号，只能出现在赋值符号的左边；



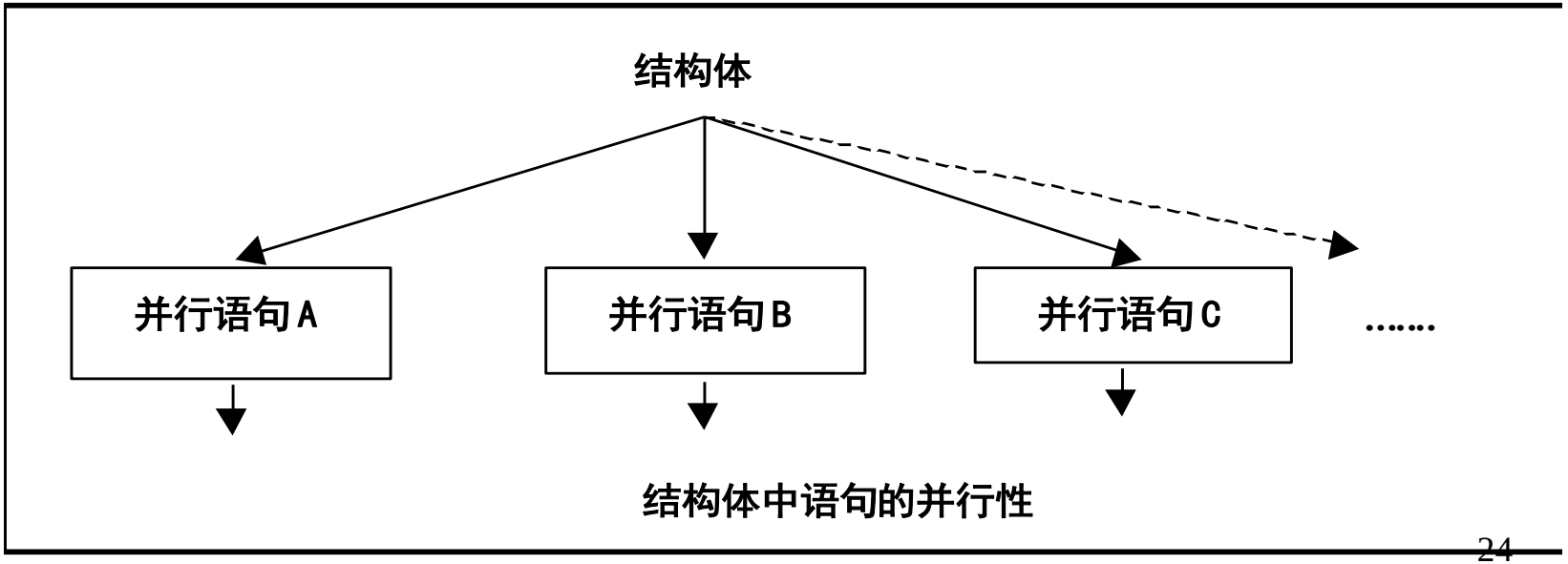
端口模式（续） -- inout 与 buffer 的区别

- 端口模式： in，out，inout，buffer
  - 连接到inout端口的信号，可以出现在赋值符号的左边或右边；（用途：和双向总线相连）
  - 连接到buffer端口的信号，可以出现在赋值符号的左边或右边。（和out类似，但可出现在赋值符号的右边）



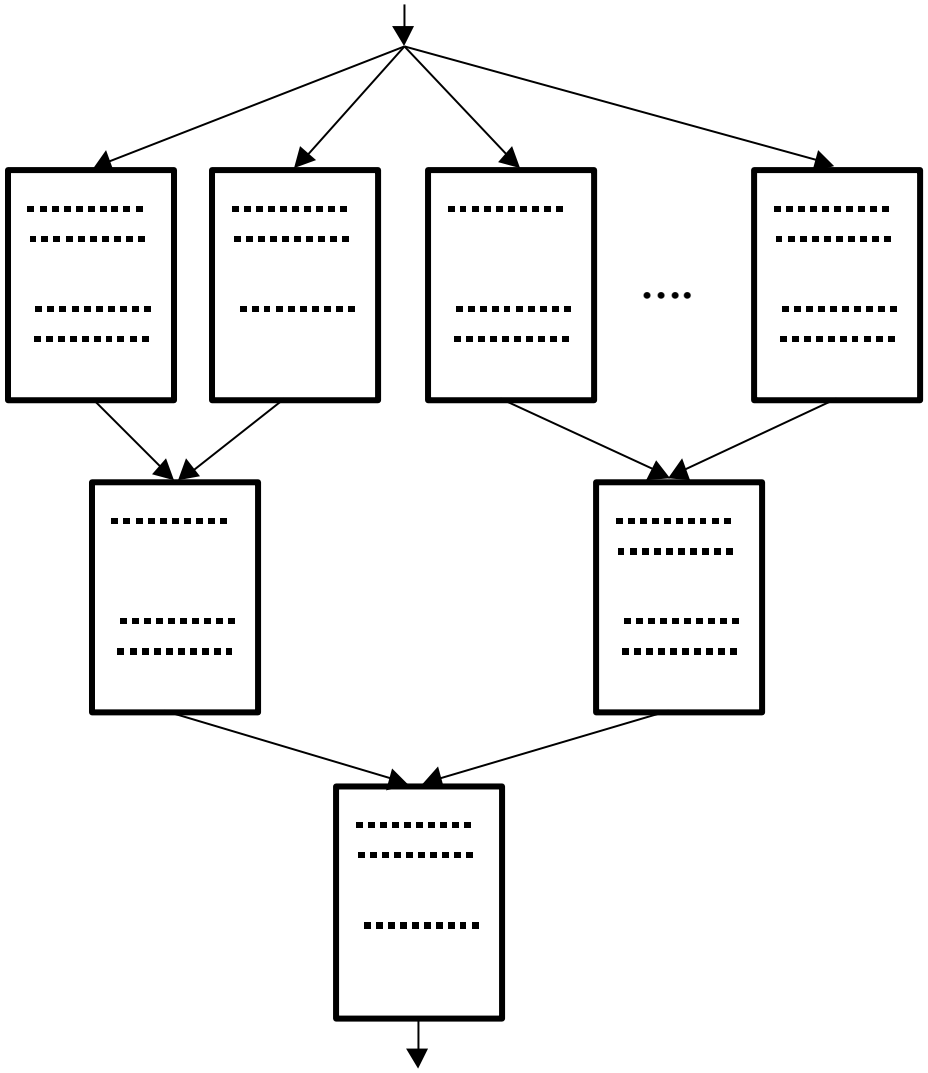
结构体（Architecture）的语法形式

```
architecture <结构体名> of <实体名> is
        <说明语句集>
begin
        <并行语句集>
end [ architecture ] [ <结构体名> ];
```



结构体中语句的并行性

并行与串行



并行语句

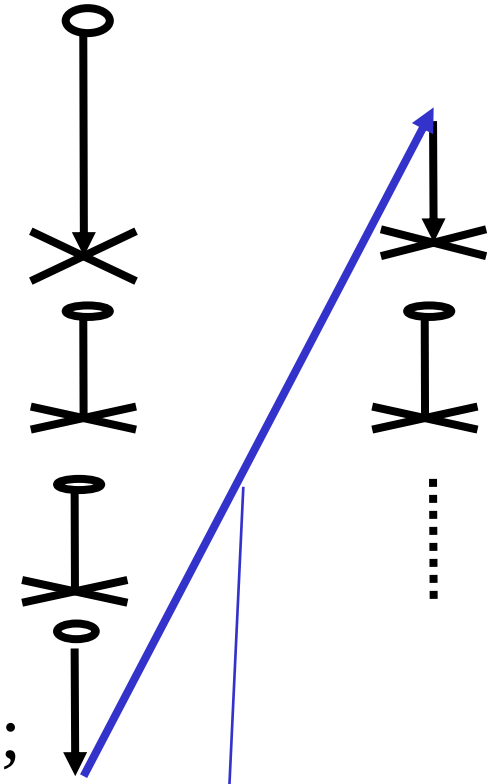
- 进程（process）语句；
  - 进程和进程之间并行执行。
  - 进程内部的语句顺序执行。
  - 简单的进程语句可以简化为并行信号赋值语句。

    例子见下页
- 并行信号赋值语句；
- 块（block）语句；
- 断言（assert）语句；
- 过程调用语句；
- 生成（generate）语句；
- 元件例化语句；

见后

进程内部执行过程

```
p1: process
begin
   …
  wait on s1;
   …
  wait on s2;
   …
  wait on s1;
   …
end process p1;
```



无限循环

- 进程是一个无限循环；
- 进程中的语句顺序执行；
- 进程中允许有多个wait语句；
- 遇到wait语句进程即被挂起，直到条件满足，进程被激活，接着向下执行；
- 进程间通过信号而相互激励/通信；
- 注意延迟时间的处理

结构体（Architecture）中的进程

（进程之间并行执行）

```
entity  subtracter  is
        port ( in1, in2, in3 :
                  in  integer;
               out1, out2 :
                  out  integer );
end  subtrcter;
```

```
architecture  simplest  of  subtracter  is
 begin
    process( in1, in2 )     -- 进程语句之一
    begin
       out1  <=  in2  -  in1  after  5 ns;
    end  process;
   process( in2, in3 )      -- 进程语句之二
    begin
      out2  <=  in2  +  in3  after  4 ns;
    end  process;
end  simplest;
```

进程之间并行执行

进程内部顺序执行

简单的进程语句可以简化为并行信号赋值语句。

简单进程化简为单个并行语句

- **进程语句若满足以下2个特点：**
  - **进程语句中只有一个信号赋值语句。**
  - **该赋值语句右边的所有信号都是敏感信号。**

则可以简化为并行信号赋值语句 ⇒

```
architecture simplest of subtracter is
begin
    out1 <= in2 - in1 after 5 ns;  --并行信号赋值语句之一
    out2 <= in2 + in3 after 4 ns;  --并行信号赋值语句之二
end simplest;
```

block 语句

- block语句可以出现在architecture中，相当于一个语法括号。例：

```
L1:   block
begin
signal_1   <=   '0'   after   10ns;
.....

end block;
```

这里的语句是并行的

- 加上这个语法括号与否不影响语义
- 加上这个语法括号有助于概念清晰，增加可读性。

block语句（续）

- block 中 的 说 明 部 分 （ 选 项 ） ：
  在保留字block和begin之间可以写入说明语句
  - 信号说明，
  - 变量说明，
  - 类属说明
  - 端口说明…．
- 其可见范围限于该block之内。

被保护的块

- 被保护的块含保护表达式；
- 保护表达式跟在关键字BLOCK之后；
- 保护表达式是布尔表达式：
  - 当布尔表达式值为True时，块中包括的驱动源起作用；
  - 当布尔表达式值为False时，所有驱动源失去作用。

顺序执行语句

- 顺序语句只出现在进程和子程序(过程和函数)中；包括：
  - wait语句；
  - 顺序赋值语句：
    - 信号赋值符号： “ <= ” ；
    - 变量赋值符号： “ := ”;
  - 顺序控制语句：
    - 条件控制：if, case;
    - 循环控制：loop, for … loop, while … loop, next, exit;
  - 断言语句： assert, report;
  - 过程调用： 过程名(实际参数);
  - 返回语句： return;
  - 空语句： null;

Wait语句

- Wait : -- 休眠，直到永远；
- Wait on <敏感信号列表>: -- 休眠，直到敏感信号有事件发生；
- Wait for <时间表达式>: -- 休眠一段时间；
- Wait until <条件表达式>: -- 休眠，直到条件为真；

Wait on语句的简化形式：

位置在end 之前

```
process
begin
  Output <= A or B;
  wait on A, B;
end process;
```

↔

```
Process（A, B）
begin
  Output <= A or B;
end process;
```

并行语句与顺序语句

详见statements.ppt

结构体（Architecture）的用途

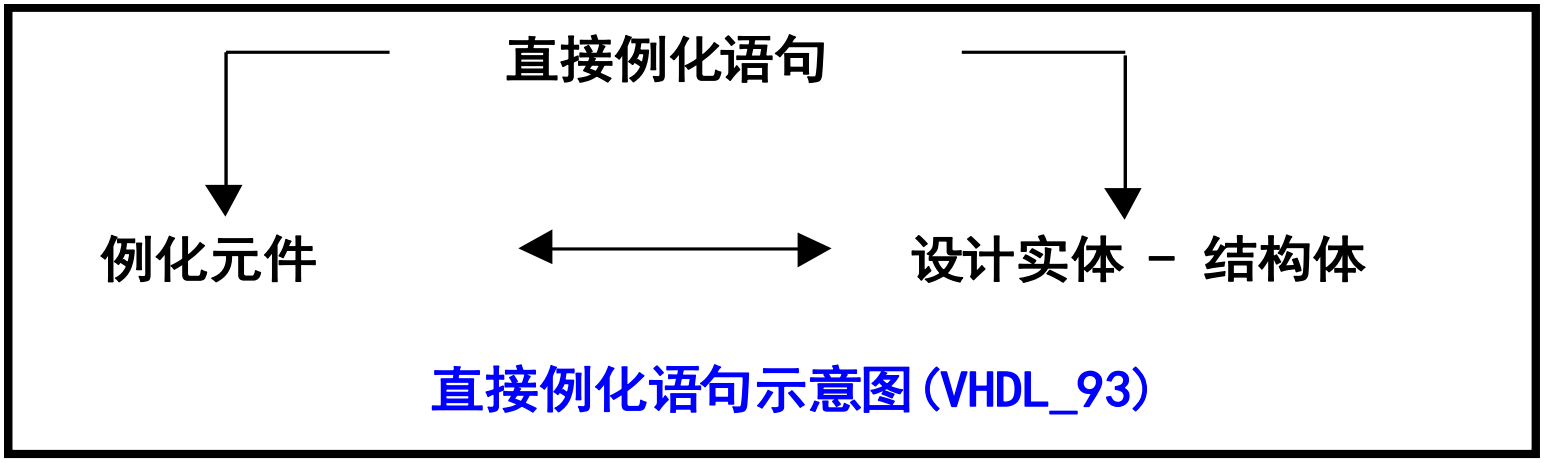
- 用于定义该设计单元的内部特性，有以下3种描述方式
  - 结构描述：描述该设计单元的硬件结构，即该硬件是如何构成的。主要使用元件例化语句及配置指定语句描述元件的类型及元件的互连关系；
  - 行为描述：描述该设计单元的功能，即该单元能做些什么。主要使用函数、过程和进程语句，以算法形式描述数据的变换和传送；
  - 数据流方式：以类似于寄存器传输级的方式描述数据的传输和变换。主要使用并行的信号赋值语句，既显式表示了该设计单元的行为，也隐式表示了该设计单元的结构；
- 结构体中的语句都是并行语句（包括进程）。

结构描述

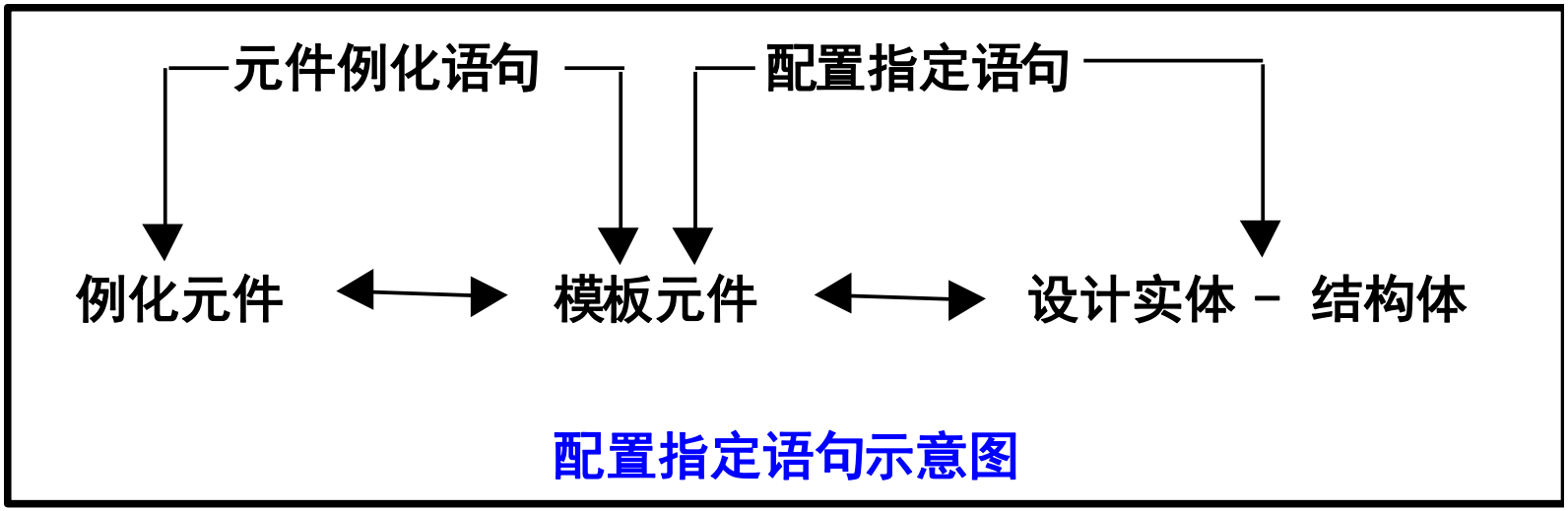
- 元件说明（ **component** declaration ）
- 元件例化（**component** instantiation）：

  高层次设计描述中把低层次描述当作子元件调用
- 配置指定（ **configuration** spefication ）
- 配置说明（**configuration** declaration）

元件例化与配置



直接例化语句示意图(VHDL_93)



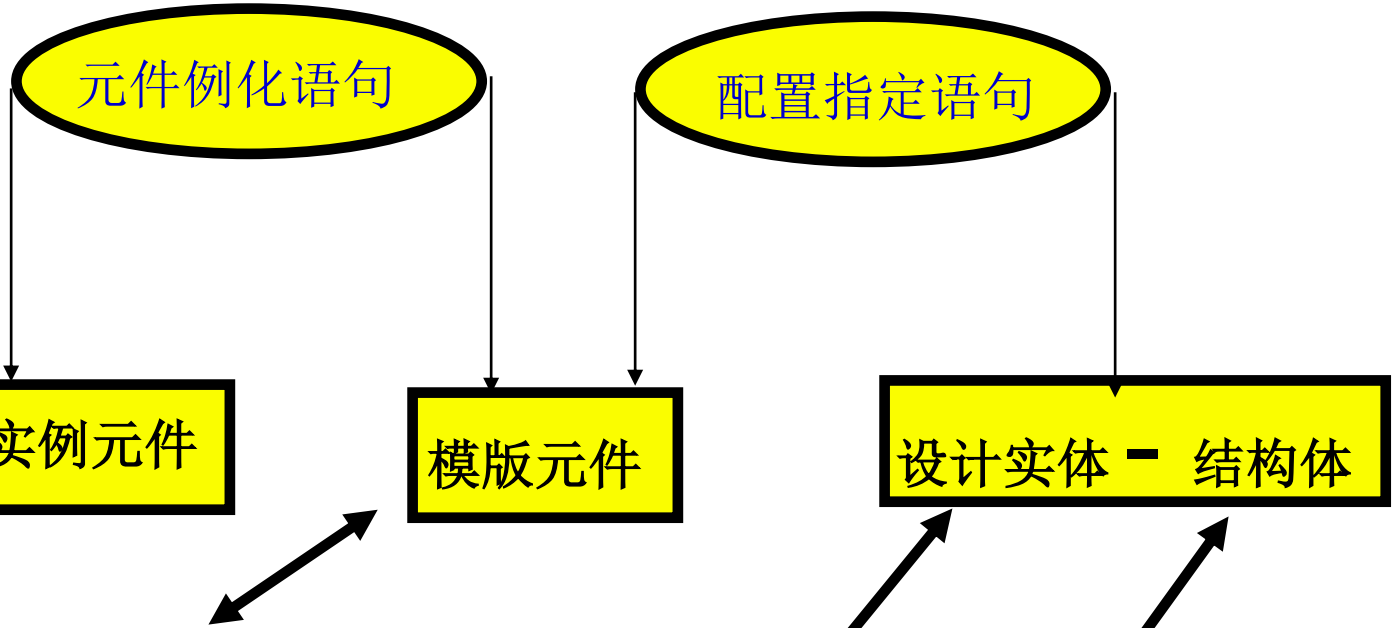
配置指定语句示意图

为什么要引入component？

- 元件（component）可以看作是1个插座，定义了1个虚拟的设计实体，通过元件例化语句把元件例化于结构体中。

- 通过配置说明或配置指定语句，把元件和实体（以及结构体）连接起来。可看作通过配置把电路插入插座。

元件例化语句举例：

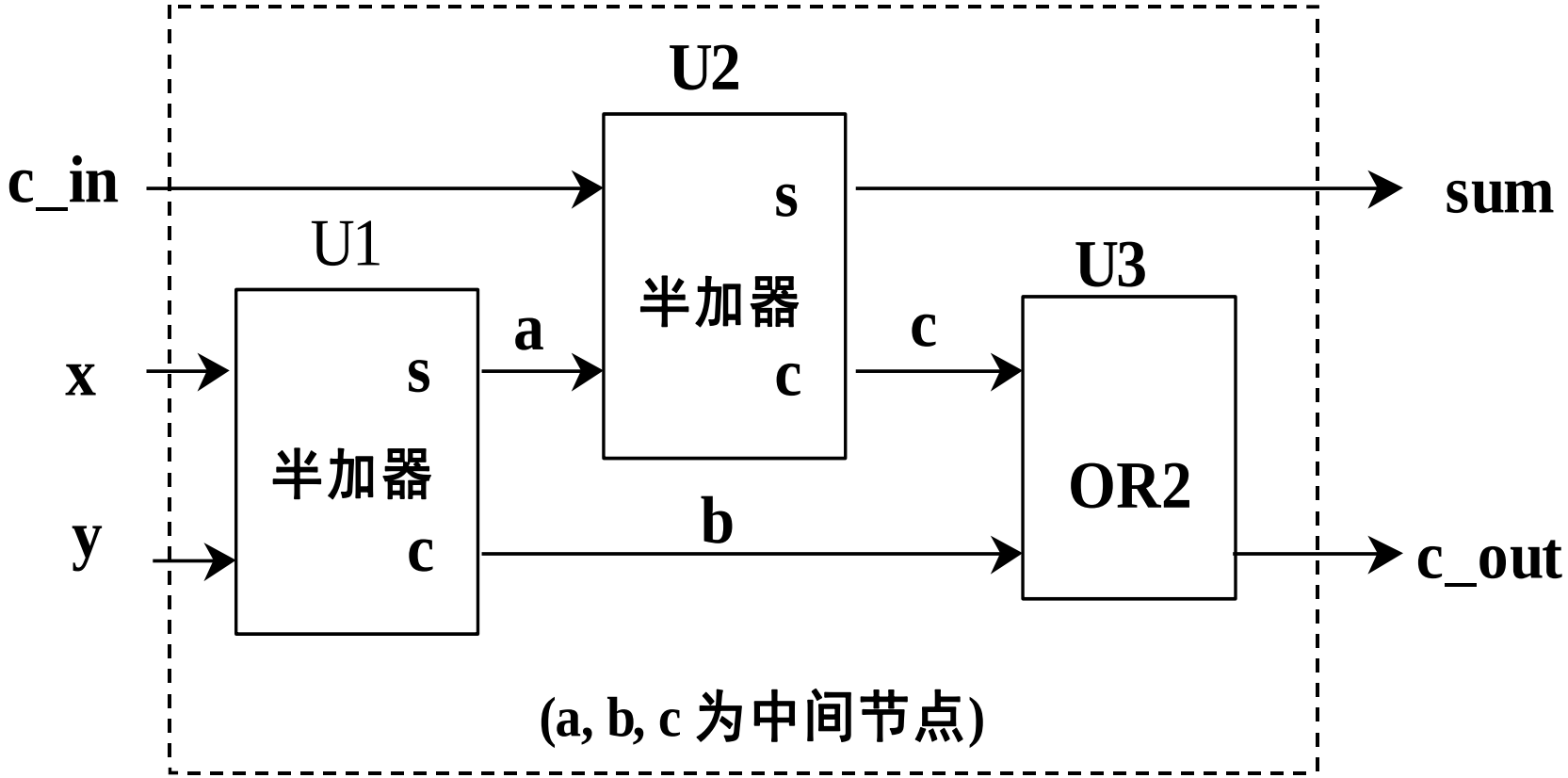



```
component  Inv
   port (In1  :  in  bit;   Out1  :  out  bit );
end  component;
for U1 : Inv  use  entity   Work.Inverter(Inverter_body);   --配置指定
     port map (I1 => In1,    O1 => Out1);
begin
  U1:  Inv  port map ( S,  S_bar );   --元件例化
end ;
```

实际信号

行为描述与结构描述 -- 举例

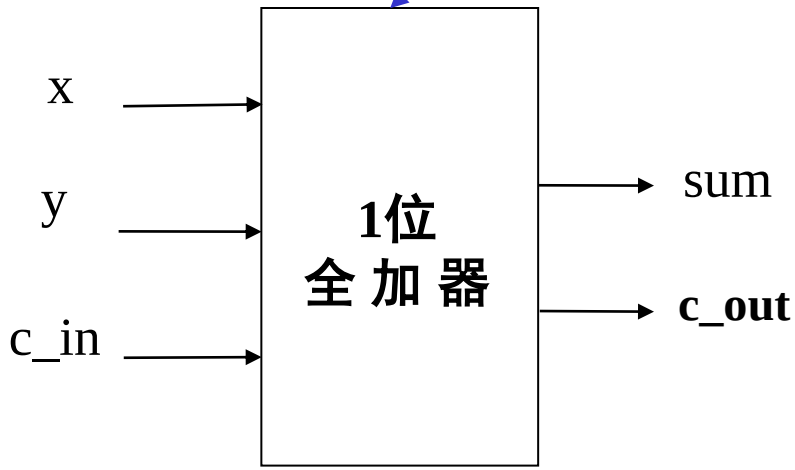
-- 1位全加器 --



行为描述与结构描述 -- 举例（续）

外观

- 1位全加器框图：



- ENTITY：

```
ENTITY full_adder IS
    GENERIC(tpd : Time := 10 ns);
    PORT(x, y, c_in : IN Std_Logic;
         sum, c_out : OUT Std_Logic);
END full_adder;
```

基本元件库和宏单元库 - 元件例化

- 芯片制造商一般都提供基本元件库和宏单元库。
- 这些库有利于提高芯片制造的质量和效率。
- Altera 公司也提供基本元件库和宏单元库。
  - Altera 公司的宏单元库和基本元件都在 ALTERA.maxplus2.all 中说明。
  - Altera 公司宏单元库的所有的端口（ ports）都只使用 STD_LOGIC 类型 或 STD_LOGIC_VECTOR类型。

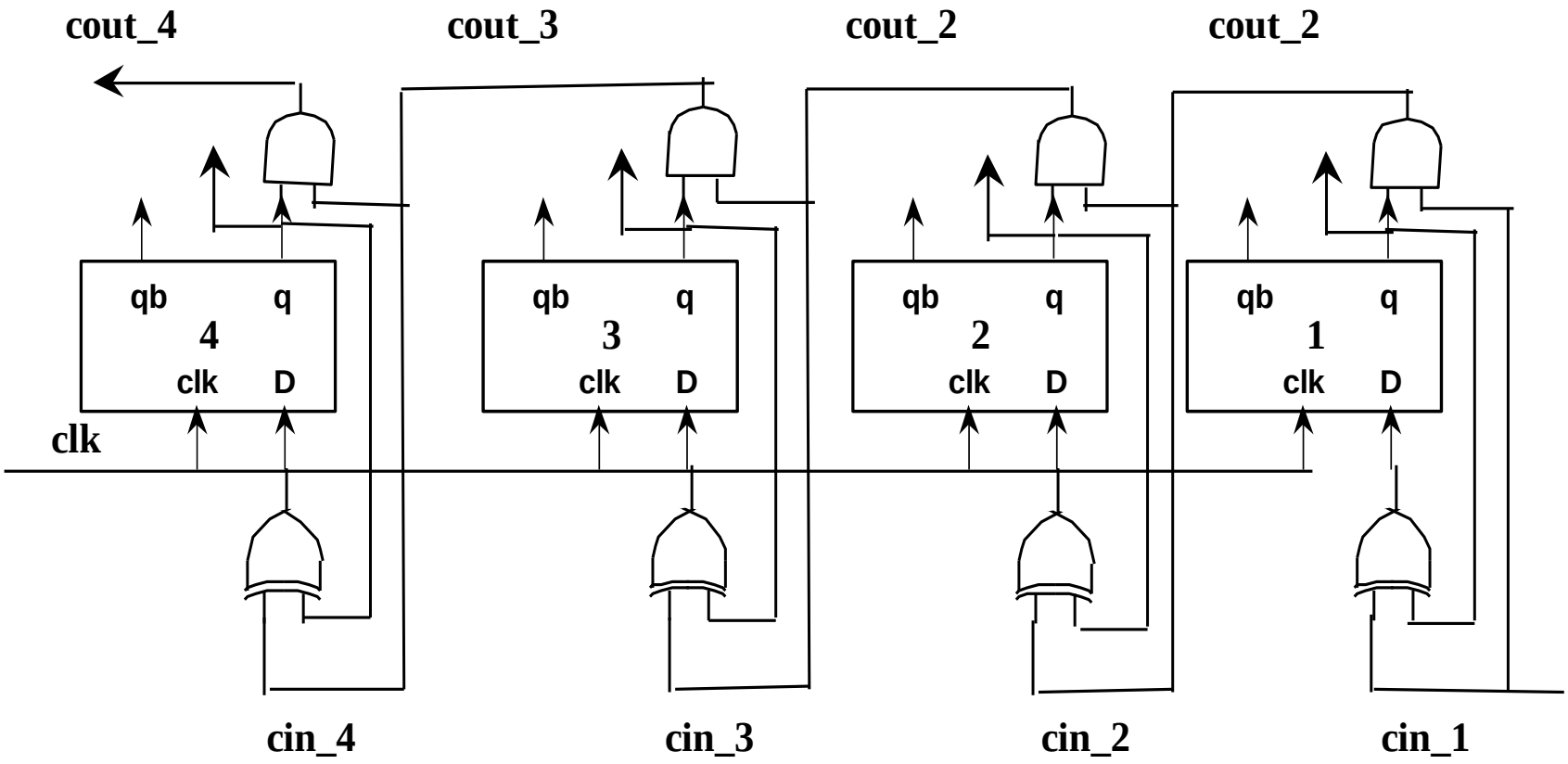
宏单元的例化

由于在此处打开了宏单元库，所以不再需要元件说明语句

```
LIBRARY  IEEE;
USE  IEEE.std_logic_1164.ALL;
LIBRARY  ALTERA;
USE  ALTERA.maxplus2.ALL;
ENTITY  macro  IS
          PORT(
                    clock, enable        : IN      std_logic;
                    Qa, Qb, Qc, Qd       : OUT     std_logic);
END  macro;
ARCHITECTURE  example  OF  macro  IS
BEGIN
   u1 :  gray4   PORT MAP (clk => clock,  ena => enable,  qa => Qa,
                    qb => Qb, qc => Qc, qd => Qd);
END  example;
```

设计分解举例

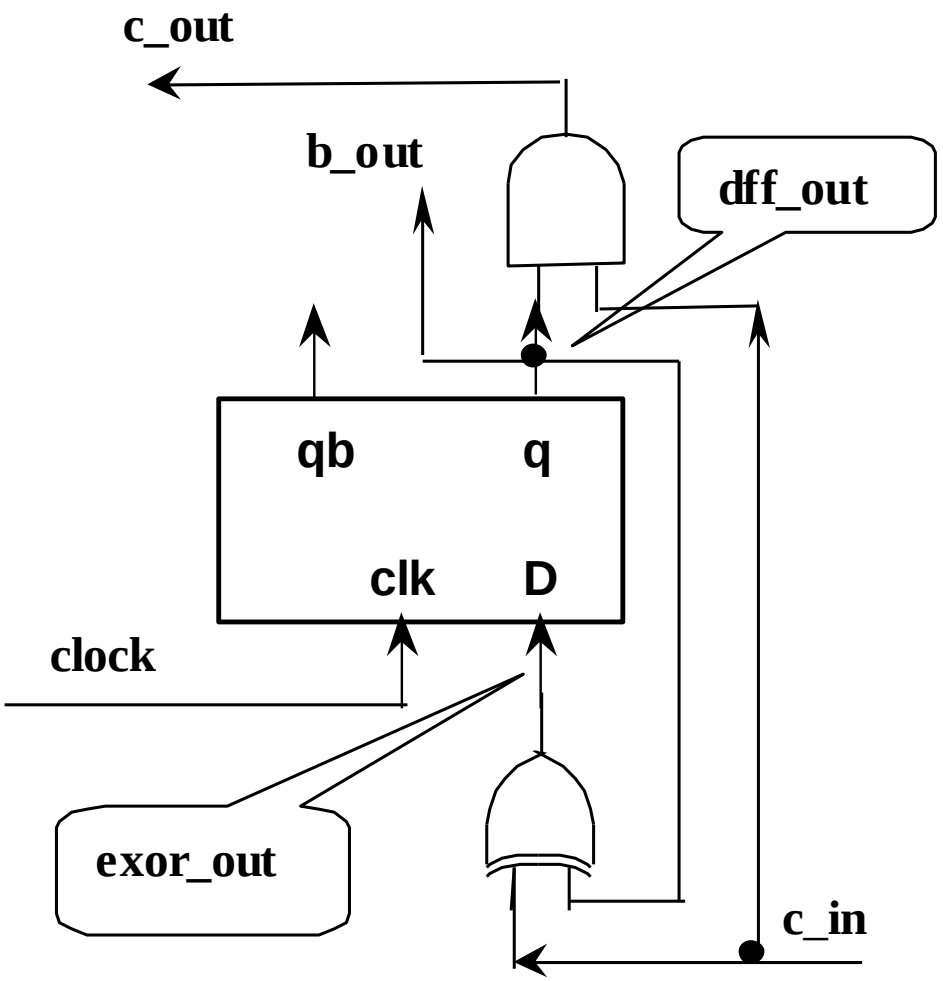
4位计数器：



设计分解举例（续）

◆ **1位计数器：**

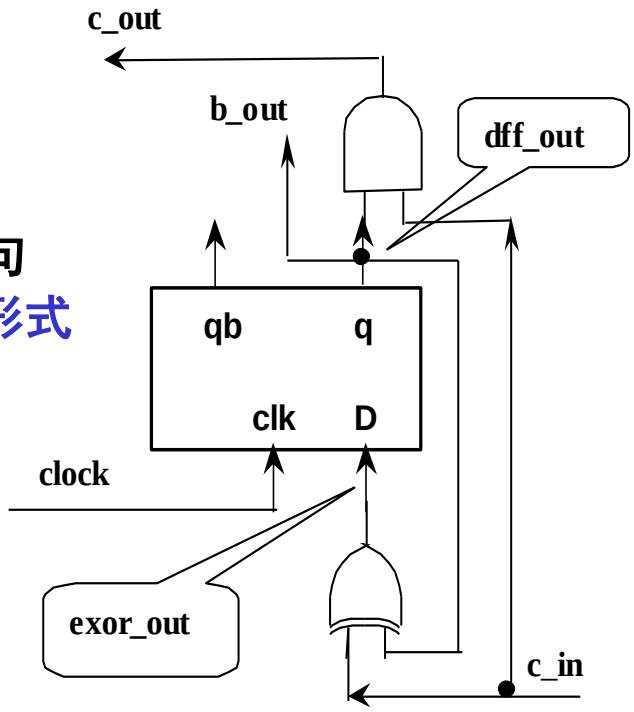
```
entity  counter_element  is
    port ( c_in, clock:  in  bit;
           c_out, b_out :  out  bit );
end  counter_element;
```



设计分解举例（续）

◆ **1位计数器的Architecture：**

```
architecture data_flow of counter_element is
    signal dff_out : bit := '0';
    signal exor_out : bit := '0';
begin
    L1: b_out <= dff_out;                    -- 并行赋值语句
    L2: exor_out <= dff_out xor cin;         -- 进程的简略形式
    L3: c_out <= dff_out and cin;
    L4: process ( clock)                     -- 进程
          begin
          if clock'event and clock = '1' then
                    dff_out <= exor_out;
          end if;
          end process;
end data_flow;
```

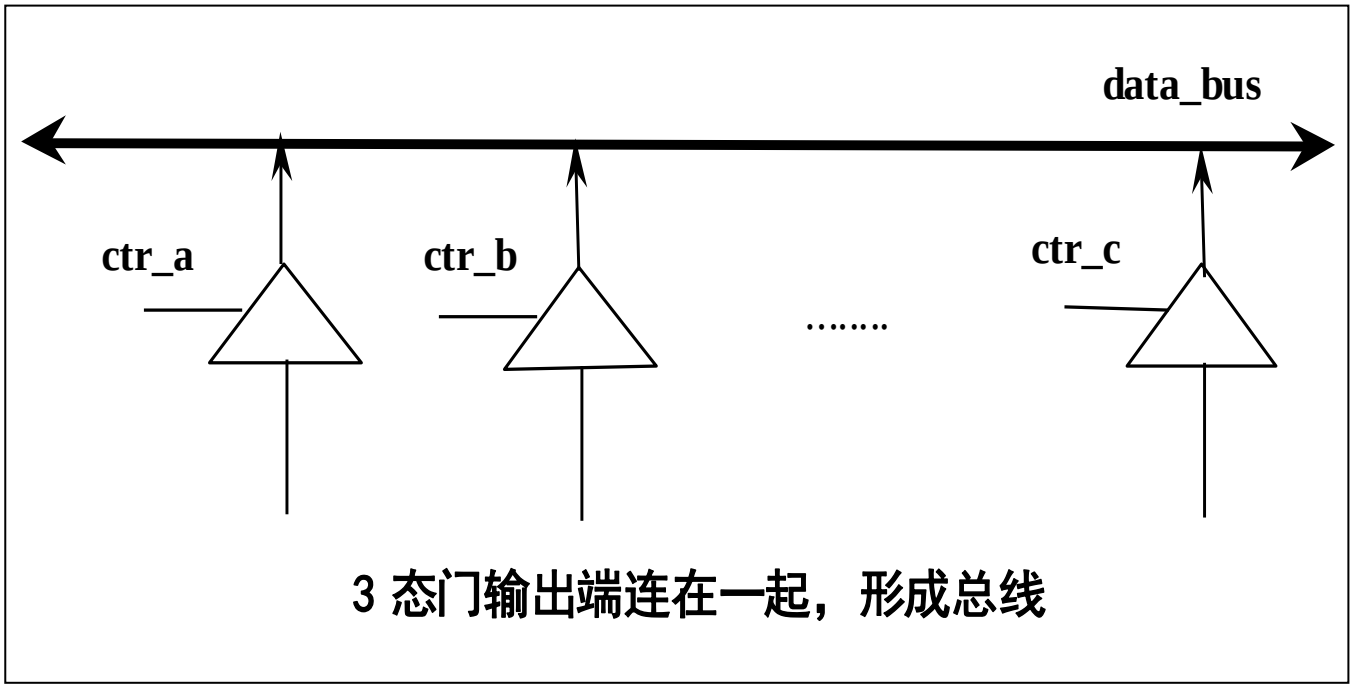


决断信号和决断函数

```
type   bit4   is   ( 'X', '1', '0', 'Z' )
type   bit4_vector   is   array ( integer range   < > )   of   bit4;
function   wired_or ( input :   bit4_vector )   return   bit4;
```

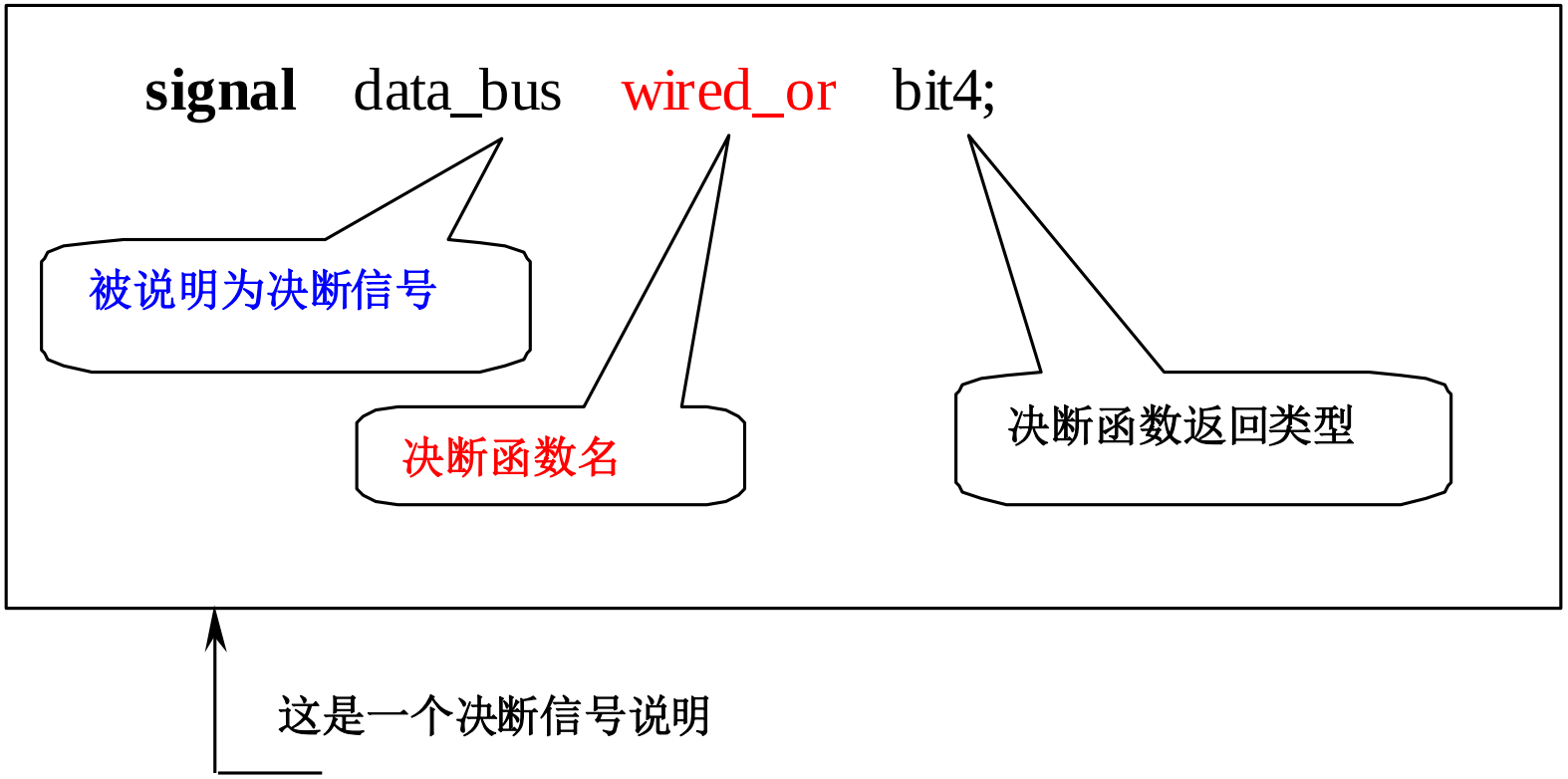
-- 这是一个**决断函数的声明**；

-- 该决断函数的内容放在对应的 package body 中；



3 态门输出端连在一起，形成总线

说明一个决断信号


signal data_bus wired_or bit4;
被说明为决断信号
决断函数名
决断函数返回类型
这是一个决断信号说明

总线应用举例

◆VHDL 提供了向量类型以表示总线;

◆最常见的向量类型是 : bit_vector, std_logic_vector，例：

```
signal  fred_bus :  bit_vector (7 downto 0);
signal  barney_bus : std_logic_vector (3 downto 0);
signal  betty_bus : std_logic_vector (0 to 3);
```

◆ 访问整个总线：

```
fred_bus <= "11111111";
```

◆ 访问整个总线中的某一位 (bit) :

```
bus (3) <= '1';
```

◆ 访问整个总线中的一段（slice）：

```
bus ( 3 downto 2 ) <= "11";
```

VHDL中的对象

- VHDL中的对象是存放值的容器，共有4类：
  - 信号:

    SIGNAL clock: bit;
  - 变量:

    VARIABLE sum: real;
  - 常量: 在被说明的时候被赋值（仅此一次）；

    CONSTANT sum: integer;
  - 文件（VHDL'93）

    FILE input: Text **IS IN** "STD_INPUT"

VHDL3种对象的比较

对象名	关键字	意义	值
信号	Signal	控制模块或进程间通信的机制，定义两个模块或进程间的数据通路。	时间序列 （波形），延迟赋值（延时）
变量	Variable	程序中临时使用的对象。	可变的单值，即时赋值（无延时）
常量	Constant	程序中不变的量。	初始化时确定，运行过程中不改变。

信号与变量的比较 --对综合的影响 --

	信　　号	变　　量
应用目的	表示电路的连接	保存值
应用范围	全局范围（任何地方）	局部范围（在进程内部）
何时取得新值?	在进程执行完毕之后（赋值语句执行之时不进行值的更新）	赋值语句执行之时立即进行值的更新

含中间变量的进程

```
ENTITY  var_ex  IS
   PORT ( x , a, b :  IN  bit;
          z ：  OUT  bit);
END var_ex;

ARCHITECTURE  example  OF var_ex  IS
BEGIN
 PROCESS ( x ,  a , b )
 VARIABLE   tmp :  bit ;

BEGIN
              IF ( x = '1' ) THEN
              tmp := a  AND  b;
              z <= tmp;
              ELSE
              z <= '1';
              END IF;
END PROCESS;
END example;
```

变量 ‘tmp’保存中间值

（前页VHDL描述）对应的逻辑图：

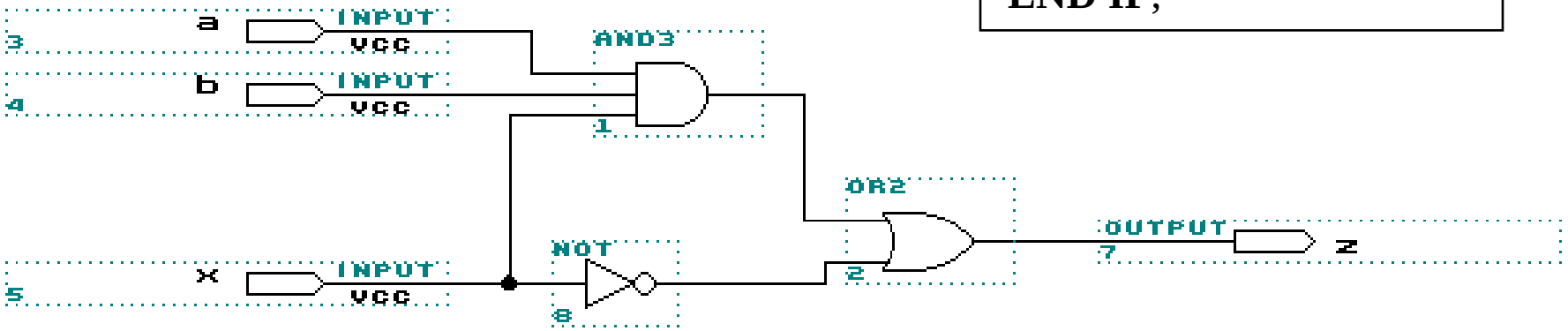
来自前页的描述

```
IF ( x = '1' ) THEN
    tmp := a  AND  b;
    z <= tmp;

ELSE
    z <= '1';

END IF;
```

变量 ‘tmp’保存中间值



基本元件之间通过 信号 实现连接

```
ENTITY  sig_ex  IS
   PORT (          a, b, c :  IN  bit;
                   y :  OUT  bit  );
END  sig_ex;


ARCHITECTURE  example   OF sig_ex
IS

SIGNAL   temp:  bit;

BEGIN

temp <= a  xor  b;
y <= temp  and  c;


END  example;
```

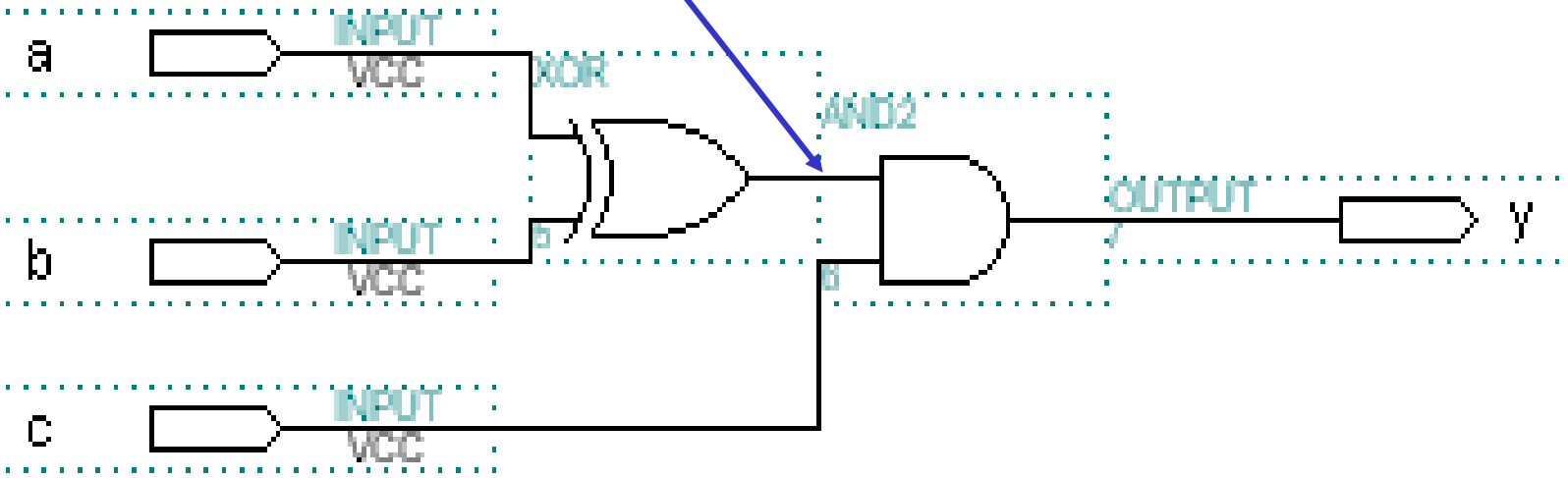
信号 ‘temp’用于实现基本元件的连接

（前页VHDL描述）对应的逻辑图：

来自前页的描述

```
ARCHITECTURE example OF sig_ex IS
SIGNAL temp: bit;
BEGIN
        temp <= a xor b;
        y <= temp and c;
END example;
```

信号 ‘temp’用于实现
基本元件的连接



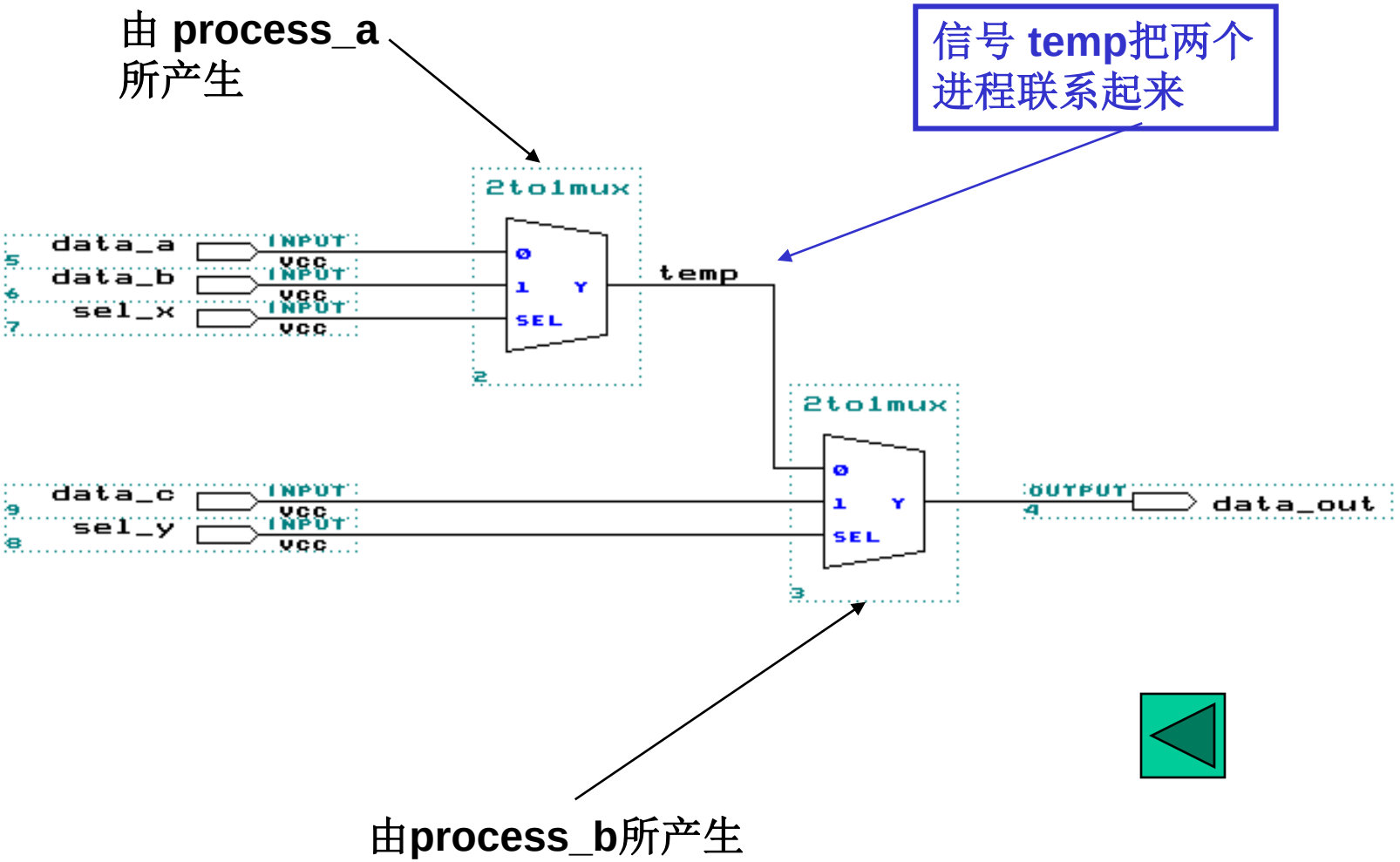
信号用于连接多个进程

```
ENTITY multiple IS
   PORT ( data_a, data_b, data_c, sel_x, sel_y  :  IN  bit ;
            data_out   :   OUT  bit );
END multiple;
ARCHITECTURE example OF multiple IS
SIGNAL  temp:  bit;
BEGIN
          process_a: PROCESS (data_a, data_b, sel_x)
             BEGIN
                        IF (sel_x = '0') THEN
                        temp <= data_a;
                        ELSE
                        temp <= data_b;
                        END IF;
             END PROCESS process_a;
          process_b: PROCESS (temp, data_c, sel_y)
             BEGIN
                        IF (sel_y = '0') THEN
                        data_out <= temp;
                        ELSE
                        data_out <= data_c;
                        END IF;
             END PROCESS process_b;
END example;
```

信号temp 在这里被赋值

信号temp在这里被使用

（前页VHDL描述）对应的逻辑图：



举例说明 变量 与 信号 的区别：

正确方案 (VARIABLE)

```
ENTITY mux_good   IS
    PORT ( i0, i1, i2, i3, a, b  :  IN  bit ;
                          q : OUT  bit ) ;
END mux_good;
ARCHITECTURE correct  OF  mux_good  IS
BEGIN
PROCESS ( i0, i1, i2, i3, a, b )
     VARIABLE muxval  :INTEGER  RANGE  0  TO  3;
BEGIN
                muxval := 0;
IF (a = '1') THEN
                muxval := muxval + 1;
END IF;
IF (b = '1') THEN
                muxval := muxval + 2;
END IF;
CASE muxval  IS
                WHEN 0 =>
                                q <= I0;
                WHEN 1 =>
                                q <= I1;
                WHEN 2 =>
                                q <= I2;
                WHEN 3 =>
                                q <= I3;
END CASE;
END PROCESS;
END correct;
```

变量muxval的新值已经得到

信号muxval 的新值还未得到

不正确方案(SIGNAL)

```
ENTITY mux_bad  IS
      PORT  (  i0, i1, i2, i3, a, b :  IN   bit;
                          q :    OUT  bit ) ;
END mux_bad;
ARCHITECTURE incorrect OF mux_bad IS
    SIGNAL muxval : INTEGER  RANGE  0 TO  3;
BEGIN
PROCESS ( i0, i1, i2, i3, a, b )
BEGIN
                muxval <=0;
IF (a = '1') THEN
                muxval <= muxval + 1;
END IF;
IF (b = '1') THEN
                muxval <= muxval + 2;
END IF;
CASE   muxval        IS
                WHEN 0 =>
                                q <= I0;
                WHEN 1 =>
                                q <= I1;
                WHEN 2 =>
                                q <= I2;
                WHEN 3 =>
                                q <= I3;
END CASE;
END PROCESS;
END incorrect;
```

变量用于非组合逻辑的描述

思考：若进程的某一次执行过程中，内部变量没有被赋值，它的取值应该如何？

```
ENTITY unsynth IS
        PORT ( sela, selb  : IN  bit;
                     dout : OUT  bit );
END unsynth;

ARCHITECTURE example OF unsynth IS
BEGIN
PROCESS (sela, selb)
        VARIABLE  temp :   bit ;
BEGIN


        IF (sela = '1') THEN
            temp := '1';
        ELSIF (selb = '1') THEN
            temp := '0';
        END IF;

        dout <= temp;
END PROCESS;
END example;
```

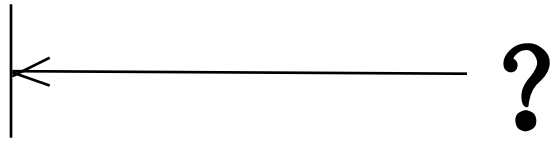
若**sela = '0'**且 **selb = '0'**
则变量 temp 保持原值不变

这个**VHDL**描述，定义了一个锁存器，而不是一个组合逻辑电路

例子中有几个寄存器?

```
ENTITY  reg1  IS
      PORT ( d, clk :  in  bit;
                  q      : out  bit );
END reg1;

ARCHITECTURE  reg1  OF  reg1  IS
SIGNAL a, b : bit;
BEGIN
      PROCESS (clk)
      BEGIN
            IF clk = '1'  THEN
                  a <= d;
                  b <= a;
                  q <= b;
            END IF;
      END PROCESS;
END reg1;
```



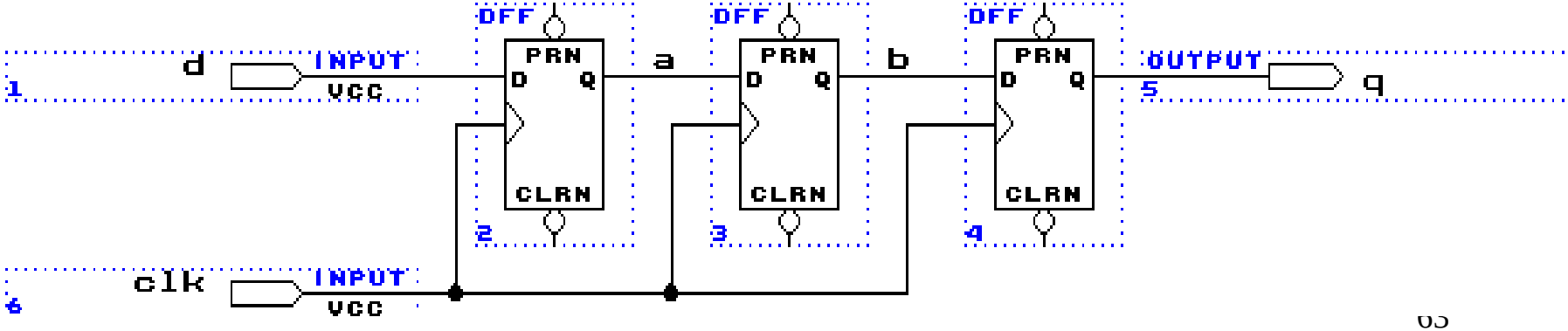
?

例子中有几个寄存器（续）？

来自前页：

```
entity  reg1  is
      port ( d, clk    :  in  bit;
                 q     : out  bit );
end reg1;
```

```
architecture  reg1  of  reg1  is
signal a, b : bit;
begin
      process (clk)
      begin
            if clk = '1'  then
                  a <= d;
                  b <= a;
                  q <= b;
            end if;
      end process;
end reg1;
```



例子中有几个寄存器？

```
ENTITY  reg1  IS
      PORT ( d ,  clk :  in  bit;
                  q : out  bit );
END reg1;

ARCHITECTURE  reg1  OF  reg1  IS
SIGNAL  a ,  b : bit;
BEGIN
      PROCESS (clk)
      BEGIN
            IF  clk = ' 1'   THEN
                  a <= d;
                  b <= a;
            END IF;
      END  PROCESS;
      q <= b;
END reg1;
```

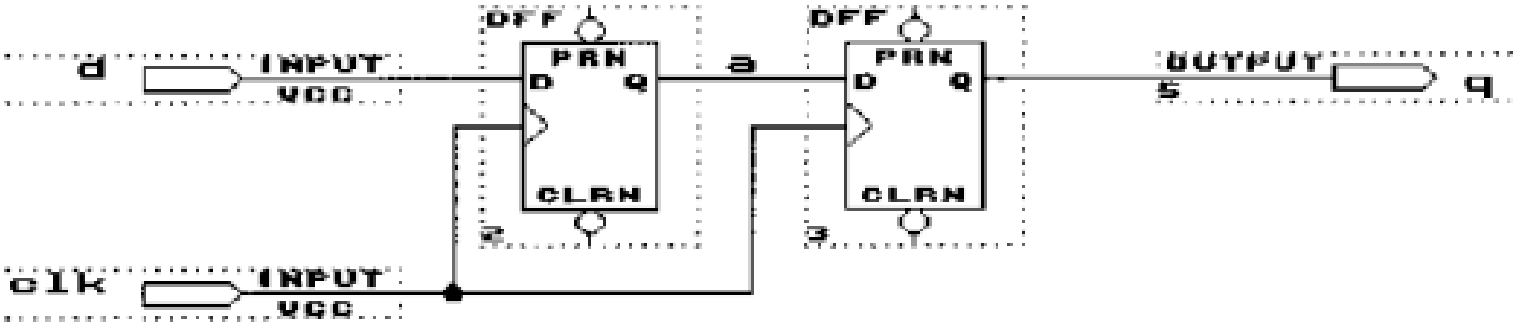
信号赋值语句移动了位置，它已经不再 对clk的边沿敏感。

例子中有几个寄存器？

```
ARCHITECTURE reg1 OF reg1 IS
SIGNAL a , b : bit;
BEGIN
    PROCESS (clk)
    BEGIN
        IF clk = ' 1'  THEN
            a <= d;
            b <= a;
        END IF;
    END PROCESS;
    q <= b;
END reg1;
```

来自前页的描述：

信号赋值语句移动了位置，它已经不再 对clk的边沿敏感。



例子中有几个寄存器？

```
ENTITY  reg1  IS
      PORT ( d , clk:   in  bit;
                  q :  out  bit);
END reg1;

ARCHITECTURE  reg1  OF  reg1  IS
SIGNAL a , b : bit;
BEGIN
      PROCESS (clk)
      BEGIN
            IF  clk = ' 1'   THEN
                  b <= a;
                  a <= d;
            END IF;
      END  PROCESS;
      q <= b;
END  reg1;
```

赋值语句的顺序改变了！

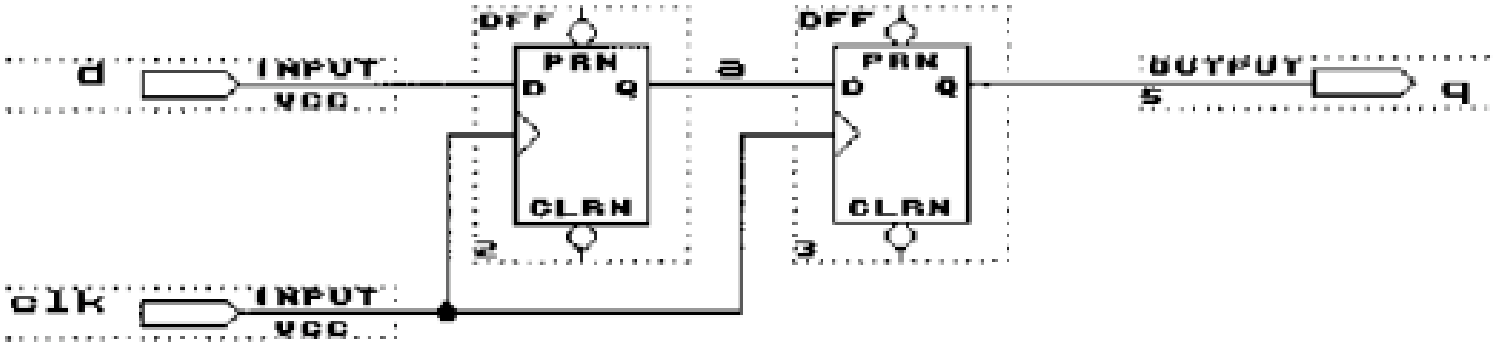
例子中有几个寄存器（续前）？

来自前页：

```
entity reg1 is
    port ( d , clk:  in bit;
                q :  out bit);
end reg1;
```

```
architecture reg1 of reg1 is
signal a , b : bit;
begin
    process (clk)
    begin
        if clk = ' 1'  then
            b <= a;
            a <= d;
        end if;
    end process;
    q <= b;
end reg1;
```

赋值语句的顺序改变不影响综合结果



例子中有几个寄存器 ？

```
ENTITY  reg1  IS
      PORT ( d , clk : in  bit;
                   q :  out  bit );
END  reg1;

ARCHITECTURE   reg1 OF  reg1  IS
BEGIN
      PROCESS (clk)
      VARIABLE a , b : bit;
      BEGIN
            IF  clk = ' 1'   THEN
                  a := d;
                  b := a;
                  q <= b;
            END  IF;
      END  PROCESS;
END  reg1;
```

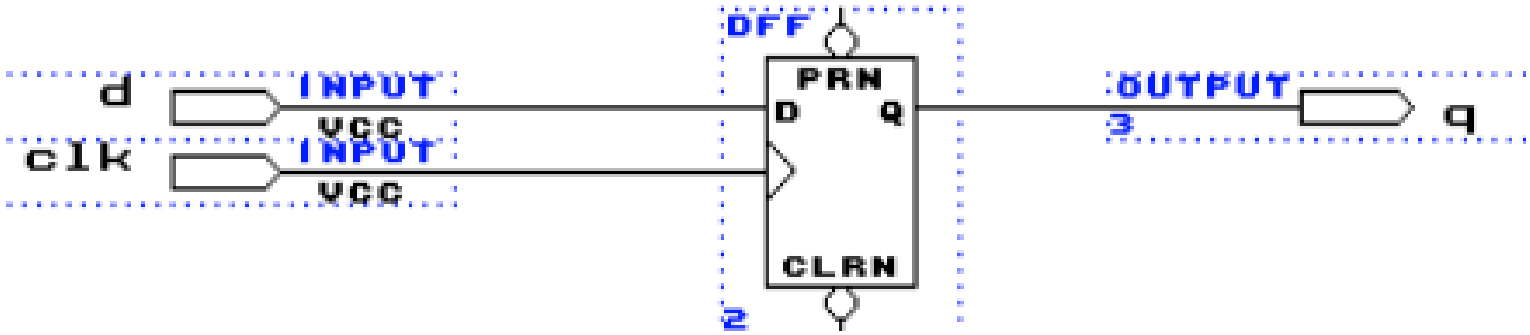
把信号改变为变量

例子中有几个寄存器（续前）？

来自前页的描述

```
PROCESS (clk)
VARIABLE a , b : bit;
BEGIN
      IF  clk = ' 1'   THEN
            a := d;
            b := a;
            q <= b;
      END  IF;
END  PROCESS;
```

- 变量赋值语句执行之后立即更新（值）
- 信号赋值语句要在全部活跃进程执行完毕之后才更新（值），
- 本例中的信号赋值语句对信号边沿敏感



和对象有关的其它问题

- 对象的取值 —— 值类型问题；

见下页

- 对象之间的运算：
  - 预定义运算符；
  - 子程序：实际是运算符的扩展
    - 函数；
    - 过程；
- 信号赋值的延时问题：
  - 传输延时；
  - 惯性延时；

VHDL的数据类型

- VHDL的每一个对象只能有1种类型，并且只能取该类型的值；
  整型； 实型； 枚举类型；
- 标准程序包STANDARD中有一些预定义的数据类型，例如：
  - 标量类型：
    - integer
    - real
    - boolean （属于枚举类型）
    - bit （属于枚举类型）
    - std_ulogic （属于枚举类型）
    - std_logic （属于枚举类型）
    - severity_level （属于枚举类型）
    - character （属于枚举类型）
    - time （带单位，属于物理类型）
    - ….
  - 复合类型（array, record）

VHDL的数据类型（续）

- **复合类型（array, record）**
  - **数组：**
    - **限定性数组：**

```
type opcode is (add, sub, complement);
type address is range 16#0000# to #FFFF# ;
```

    - **非限定性数组（ 在使用时指定具体范围 ）：**

```
type string is array ( positive range < > ) of character;
type bit_vector is array ( natural range < > ) of bit;
```

  - **记录：**

```
type instruction is record
    opcode_field: opcode;
    operand_1: address;
    operand_1: address;
end record;
```

VHDL的数据类型（续）

- 复合类型的数值可以用聚集（aggregate）表示法给出：

  一个聚集是一个用圆括号括起来的元素关联表，每个元素关联指定了数组或记录的一个元素的数值。元素关联以逗号分隔。例如：

  - 下面是类型Conversion_array的声明，其中用聚集表示离散范围：

    type Clock_level **is** (Low，Rising，High，Falling);

    type Conversion_array **is array** (Clock_level) **of** Bit;

  - 下面是Conversion_array类型的变量C的聚集举例：

    C := (′0′, ′1′, ′1′, ′0′);

以上是位置关联表示法以下是名字关联表示法：

C := (Low => ′0′，Rising => ′1′，High => ′1′，Failing => ′0′);

C := (Low | Falling => ′0′，Rising | High => ′1′)

聚集表示中使用保留字others

- 聚集中可用保留字**others**表示所有剩余的选项(即到目前为止还未指定的选项)，例如：
  - **constant** One: Rational := (**others** => 1);
  - **constant** Is_decimal_digit: Hex_index := (′0′ **to** ′9′ =>True, **others** => False);
  - **variable** Hex_value: Eight_digit := (**others** => ′0′);

枚举类型举例（九值逻辑）

```
TYPE  Std_ULogic  IS
      ( 'U',      -- Uninitialized
        'X',      -- Forcing  Unknown
        '0',      -- Forcing  0
        '1',      -- Forcing  1
        'Z',      -- High Impedance
        'W',      -- Weak Unknown
        'L',      -- Weak 0
        'H',      -- Weak 1
        '-'       -- Don't care
      );
```

程序包 std_logic_1164 中有此定义

这里不仅出现了“值”而且出现了强度。

整数类型（integer）及其子域

- 其作用同代数中的整数（在standard程序包中定义）；
- 整数的范围可以由编译器设定缺省值 指定；也可以由用户指定。
  - 用户可以随意给整数指定一个子域。例如：

    fred : INTEGER RANGE 0 to 255;

    思考：为什么要指定子域？

  - 若用户没有给对象指定一个子域（见下例），则INTEGER的范围被编译器设定的的缺省值所决定。

    例：fred： INTEGER;

例如是2^{32}

子类型举例（Std_Ulogic的子类型）

- 以下定义取自程序包 std_logic_1164：

```
SUBTYPE X01 IS Resolved std_ulogic RANGE 'X' TO '1';        -- ('X','0','1')
SUBTYPE X01Z IS Resolved std_ulogic RANGE 'X' TO 'Z';       -- ('X','0','1','Z')
SUBTYPE UX01 IS Resolved std_ulogic RANGE 'U' TO '1';       -- ('U','X','0','1')
SUBTYPE UX01Z IS Resolved std_ulogic RANGE 'U' TO 'Z';
                                                            -- ('U','X','0','1','Z')
--------------------------------------------------------------------------------------
```

Resolved是预定义的决断函数

```
--------------------------------------------------------------------------------------
FUNCTION Resolved ( s : std_ulogic_Vector ) RETURN std_ulogic;
--------------------------------------------------------------------------------------
```

预定义类型举例

- 预定义整数类型：INTEGER
  - 10进制整数：2, 0, 77459102, 77_459_102，10E9
  - 16进制整数：16#FFD1#
  - 8进制整数： 8#720#
  - 2进制整数： 2#10110010#， 2#1011_0010#
  - 指定范围的整数对象：

    signal a, b: Integer range –128 to 127;
  - 预定义子类型：

    Natural —— 0 to ∞； Positive —— 1 to ∞
- 预定义实数类型：REAL

  1.0， 0.0， 3.1415926， - 43.6E - 4

  signal f: Real range –1.0 to 1.0;
- type BOOLEAN is (TRUE, FALSE);
- type BIT is (′0′，′1′);

自定义类型举例

- **type** Byte **is range** -128 **to** 127;
- **type** Bit_position is **range** 7 **downto** 0;
- **type** Decimal_int **is range** -1E9 to 1E9;

- **Subtype** digital **is** integer **range** 0 **to** 9;
- **subtype** Register **is** Bit_Vector (7 **downto** 0);

- 注意**type**与 **subtype** 的区别
  - **type** month **is range** 1 **to** 12;
  - **subtype** summer **is** month **range** 6 **to** 8;

类型转换

- 赋值时，若值类型和对象类型不一致，要使用显式类型转换。
- 同一父类型的诸子类型相互兼容。

用类型标记实现类型转换

- 类型标记实际上是类型的名字；
- 类型标记转换仅适用于关系密切的标量类型，即整数和浮点数；
- 可以用类型转换函数实现类型转换。
- 枚举类型不能使用类型标记的方法进行类型转换；

举例： 假定有：

VARIABLE i : Integer;
VARIABLE r : Real;

则下列赋值语句可以正常工作：

i := Integer (r);
r := Real (i);

- - 把浮点数转换为整数时会发生舍入现象

预定义的类型标记

- 某些程序包中预定义了一些类型转换函数，即类型标记。
- 程序包Numeric_Bit中定义了有符号数Signed和无符号数Unsigned类型，可以用类型标记方法实现它们和Bit_Vector之间的类型转换。
- 程序包Numeric_Std中定义了有符号数和无符号数类型，可以用类型标记方法实现它们和Std_Logic_Vector之间的类型转换。
- 程序包std_logic_unsigned中定义了一些函数，使得可以对Std_Logic_Vector类型的对象方便地进行算术运算（如同无符号整数类型一样地进行）。
- 程序包std_logic_signed中定义了一些函数，使得可以对Std_Logic_Vector类型的对象方便地进行算术运算（如同有符号整数类型一样地进行）。

用类型转换函数实现类型转换

```
FUNCTION    To_Bit  (  s:  Std_ULogic;
    xmap : Bit := '0' )   RETURN  Bit  IS
BEGIN
 CASE  s  IS
 WHEN  '0' | 'L'  =>  RETURN ('0');
   WHEN  '1' | 'H'  =>  RETURN ('1');
   WHEN  OTHERS =>  RETURN  xmap;
 END CASE;
END;
```

- 左面的类型转换函数取自程序包 Std_Logic_1164 ；
- 调用转换函数To_Bit时，第二个参数可以被略去不写。此时，该参数取默认值。
- 用户也可以写出自己的类型转换函数，放入自定义程序包。

用常数实现类型转换

◆ 下面的例子使用常数把类型为Std_ULogic的值转换Bit 类型的值：

```
LIBRARY IEEE;
USE IEEE.Std_Logic_1164.ALL;
ENTITY typeconv IS
END;

ARCHITECTURE arch OF typeconv IS
 TYPE typeconv_typ IS ARRAY ( Std_ULogic ) OF Bit;
 CONSTANT typeconv_con : typeconv_typ := ( '0' | 'L' => '0',
                                           '1' | 'H' => '1', OTHERS => '0' );
 SIGNAL b: Bit;
 SIGNAL s: Std_ULogic;
BEGIN
  b <= typeconv_con (s);
END;
```

定义了一个常数typeconv_con，用来实现类型转换

尽量利用程序包中预定义的类型转换函数

- 程序包std_logic_arith中：定义了以下类型转换函数，用于把类型std_logic_vector转换为整数、有符号数、无符号数：
  - conv_integer;
  - conv_signed;
  - conv_unsigned;
- 程序包std_logic_signed中：定义了一些函数，使得可以对std_logic_vector类型的对象当作有符号数运算；
- 程序包std_logic_unsigned中：定义了一些函数，使得可以对std_logic_vector类型的对象当作无符号数运算；

VHDL其他数据类型

-- 访问类型和文件类型 --

- VHDL中的访问类型（保留字access ）相当于程序设计语言中的指针类型。访问类型的值是指针，它指向（或链接到）动态分配的、无名的、其他类型的对象。例如：
  - **type** Int **is range** 0 **to** 500; - - Int 是整数类型
  - **type** Int_Ptr **is access** Int; - - Int_Ptr是Int的访问类型
- 在声明一个访问类型的对象时，可以用保留字new来为其分配一个空间，并可指定一个初始值。例：
  - variable V_Int_Ptr_1: Int_Ptr := new Int′(362);
  - --访问值为Int类型, 初始值为362。
  - variable V_Int_Ptr_2: Int_Ptr := new Int;
  - --访问值为Int类型, 初始值为0 (Int′LEFT)。
  - variable V_Int_Ptr_3: Int_Ptr; - -访问值为空(null)。

VHDL其他数据类型（续）

◆ 文件类型（保留字file）：

文件类型和文件对象为VHDL设计与外部设计环境之间的通信提供了一个途径。例如（取自程序包textio）：

- **type** Text **is file of** STRING;
  - ➢ **Text为由可变长度ASCII记录组成的文件类型；**

- **file** INPUT: Text **is in** "STD_INPUT";
  - ➢ **INPUT为文件标识符；**
  - ➢ **保留字in表示输入文件；**
  - ➢ **双撇号中的字符串为实际文件名。**

预定义运算符

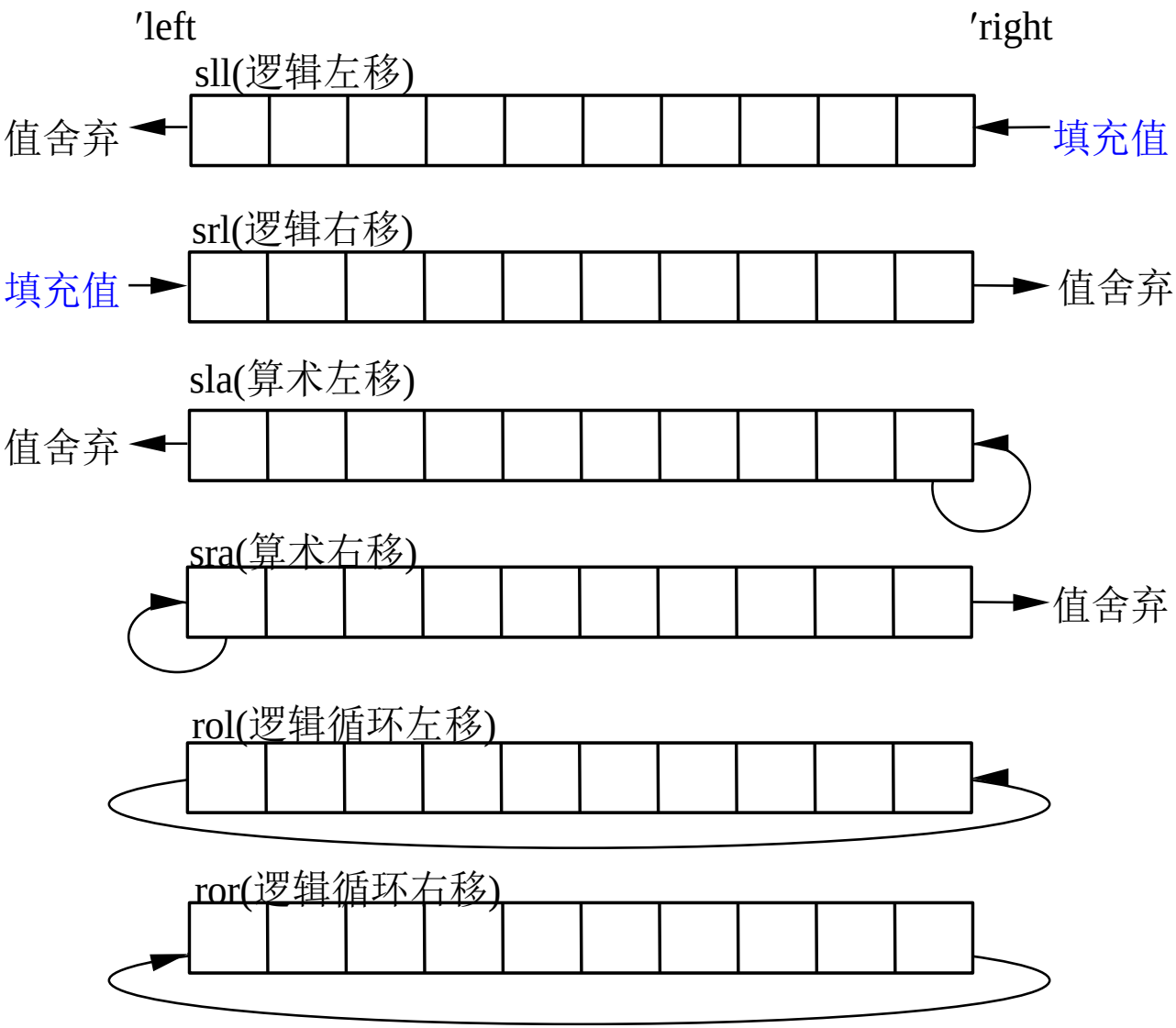
- **算术运算符:**
  - **一元算术运算符:**
    - **+ (正号)， –(负号),**
    - **abs (绝对值);**
  - **二元算术运算符:**
    - **+（加）， –（减）, *（乘）, /（除）,**
    - **mod（取模）, rem（取余）,**
    - ****（乘方）;**
- **逻辑运算符:**
  - **一元算术运算符： not（非）;**
  - **二元算术运算符:**
    - **and（与）, or（或）, nand（与非）,**
    - **nor（或非）, xor（异或）, xnor（异或非）;**

预定义运算符

- **关系运算符：**
  - =（等于）， /=（不等于）， <（小于）， >（大于），
  - <=（小于等于）， >=（大于等于）；
- **连接运算符：** **&（连接）**

  举例： **constant** S1: string **:=** ″ABC″;
  constant S2: string := ″DEF″;
  constant S3: string := ″S1 & S2″; --″ABCDEF″
- **移位运算符：**
  - **sll**（逻辑左移）, **srl**（逻辑右移），
  - **sla**（算术左移）, **sra**（算术右移），
  - **rol**（逻辑循环左移）, **ror**（逻辑循环右移）；

移位运算示意图



填充值定义：

数组元素类型的属性 LEFT 的值。

- bit_vector:

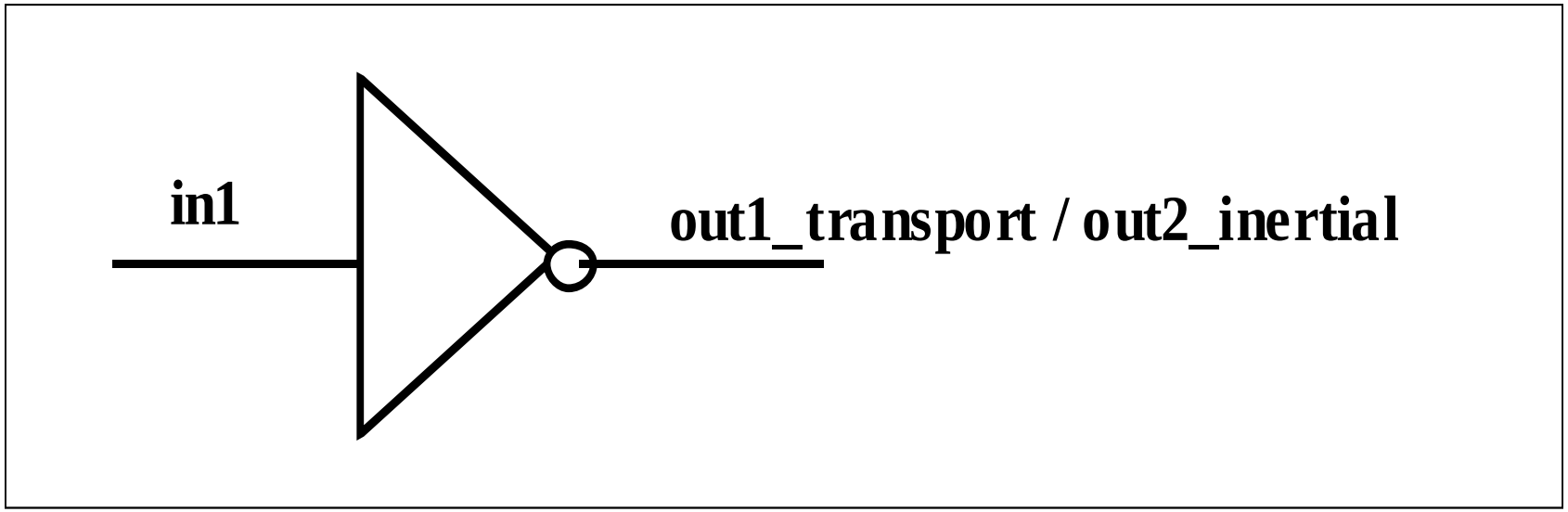
  '0'

- boolean向量:

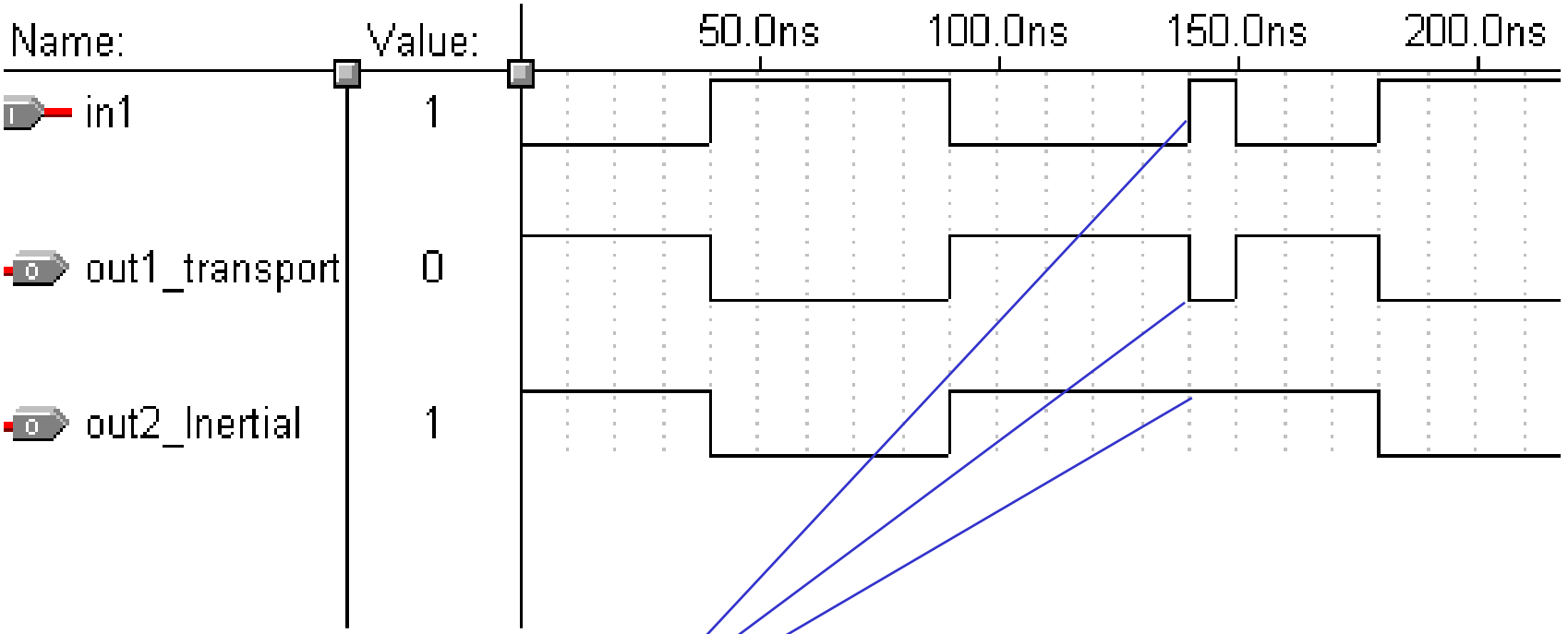
  False

信号赋值中的延时

- 延迟的分类：
  - 传输延迟 ：　关键字 TRANSPORT;
  - 惯性延迟：　关键字 INERTIAL， REJECT；
  - 波形见下页：



传输延时和惯性延时的比较



对窄脉冲的反应不同！

输入脉冲宽度小于惯性阈值 时，
惯性延迟元件的输出端将没有反应。

延时子句（AFTER）举例

- dout1 <= INERTIAL a AND b AFTER 5 ns;
  -- 惯性阈值 = 5 ns， 由AFTER指定；等价于： dout1 <= a AND b AFTER 5 ns;
- dout2 <= REJECT 3 ns INERTIAL a AND b AFTER 5 ns;
  --惯性阈值 3 ns， 由REJECT指定；
- dout1 <= TRANSPORT a AND b AFTER 5 ns;
  -- 传输延迟
- out_signal <= INERTIAL '1' AFTER 5 ns,
  '0' AFTER 10 ns， '1' AFTER 12ns;
  - 仅仅第1个AFTER子句是惯性延迟；其后的AFTER子句均为传输延时
  - 延时值必须是递增顺序，否则是语法错误。

信号值的表示形式

波形元素 ≡ （信号名，信号值，信号值发生变化的时刻）

- 当前值；

未来事项序列

- 未来值链： 被称为信号的驱动源
  （名，值，时刻）--（名，值，时刻）--（名，值，时刻）

- 历史值链：当前值不断地变为历史值，供输出波形之用。

模拟过程中的处理方法

未来值链：

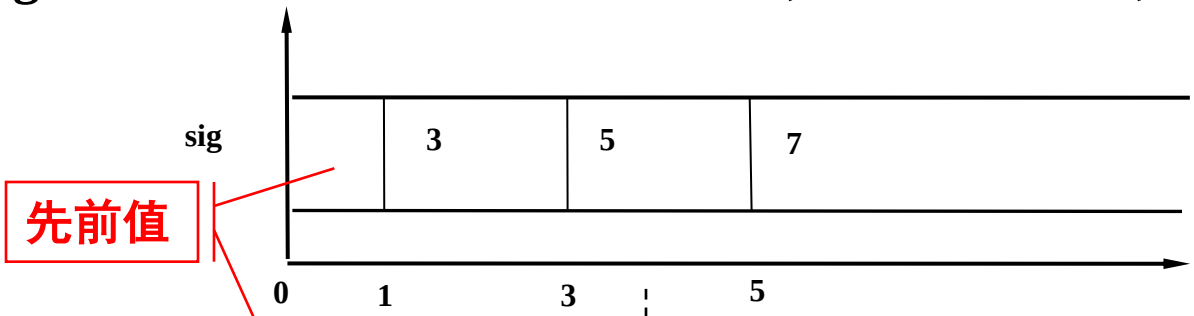
（名，值，时刻）--（名，值，时刻）--（名，值，时刻）

- 随着模拟时钟的推进，不断从未来值链中取出合适的值作为当前值。
- 原来的当前值则变为历史值。
- 未来值链中的波形元素动态地被增加 / 删除。

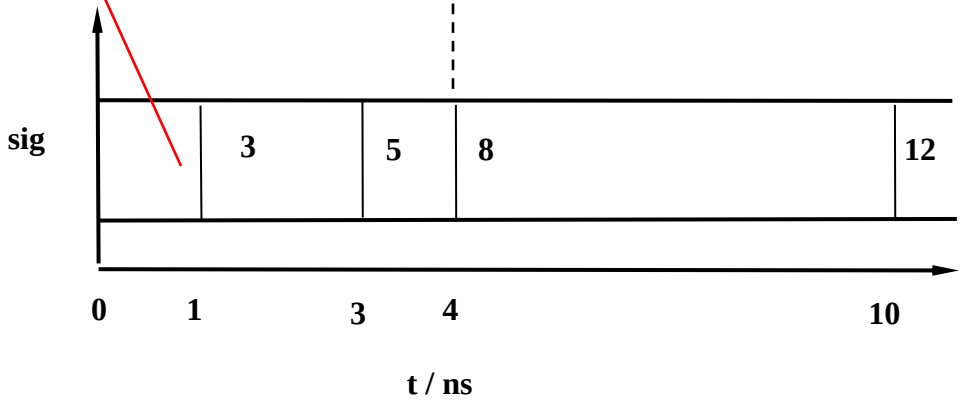
VHDL的时延模型（传输延迟）

- 第一个新事项之后的所有老事项均被删除。
- 同一信号赋值语句中第一个新事项之后的其他新事项均被添加。

sig <= TRANSPORT 3 AFTER 1 ns, 5 AFTER 3 ns, 7 AFTER 5 ns;



sig <= TRANSPORT 8 AFTER 4 ns, 12 AFTER 10 ns;



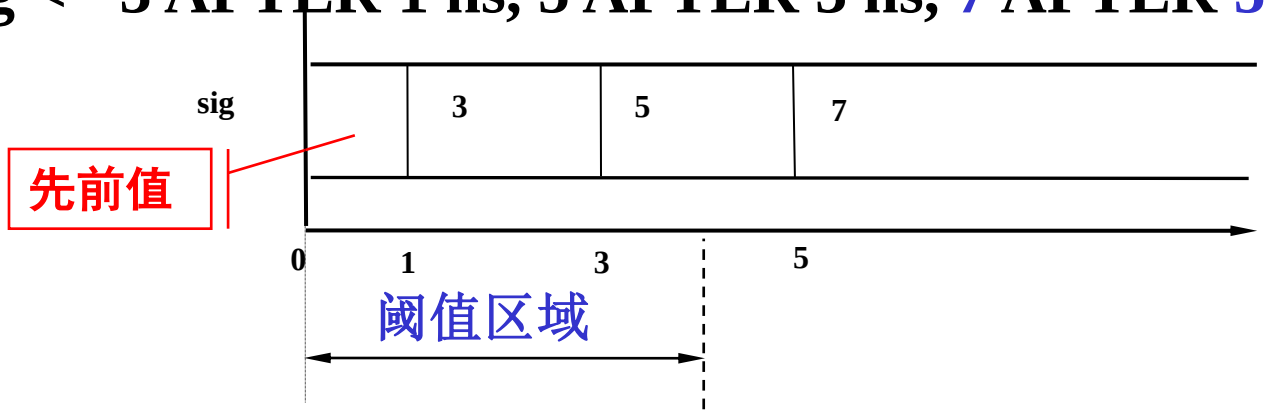
VHDL的时延模型（惯性延迟）

- 第一个新事项之后的所有老事项均被删除。
- 同一信号赋值语句中第一个新事项之后的其他新事项均被添加。
- 对于第一个新事项之前的老事项：
  - 若老事项值与第一个新事项之值相同，则保留之。反复执行此步骤；
  - 若老事项值与第一个新事项之值不同，删除该老事项以及该老事项之前的所有老事项。

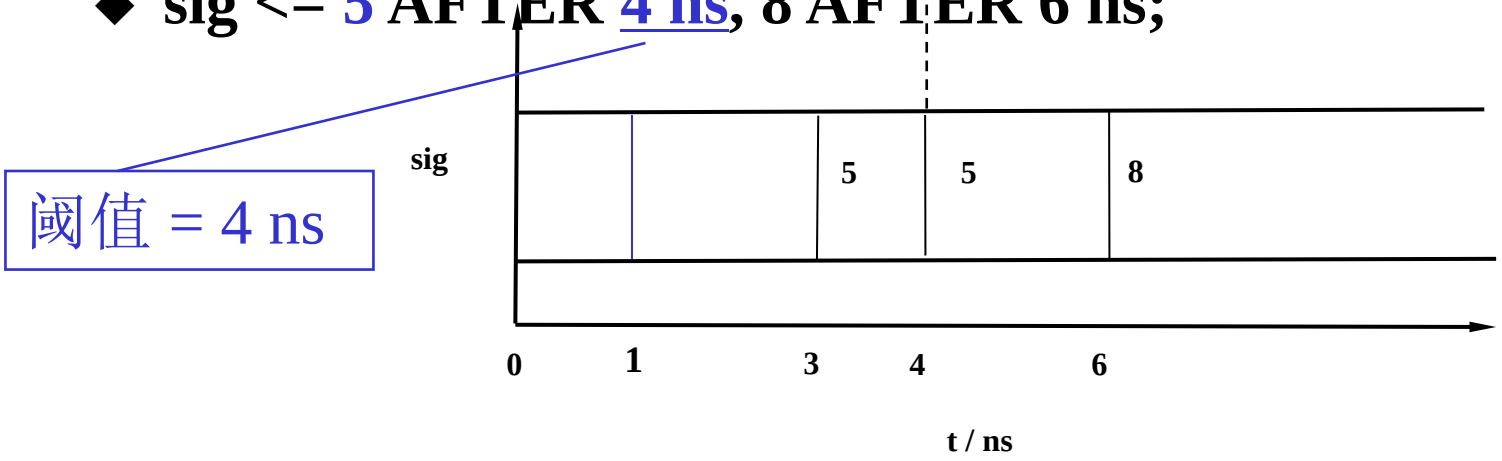
VHDL的时延模型（惯性延迟）

语句中有reject的情况：阈值由 reject 指定

- **sig <= 3 AFTER 1 ns, 5 AFTER 3 ns, 7 AFTER 5 ns;**



- **sig <= 5 AFTER 4 ns, 8 AFTER 6 ns;**



VHDL的时延模型（阈值惯性延迟）

语句中有reject的情况：阈值由 reject 指定

- 第一个新事项之后的所有老事项均被删除。
- 同一信号赋值语句中第一个新事项之后的其他新事项均被添加。
- 阈值定义：第1个新事项之前的一段时间 δT，
  δT 由关键字**REJECT指定。**
- 对于第一个新事项之前的老事项，
  - 若老事项值与第一个新事项之值相同，则保留之。反复执行此步骤；
  - 在阈值δT范围内，若老事项值与第一个新事项之值不同，删除该老事项以及该老事项之前的所有老事项。

sig <= **TRANSPORT** 3 **AFTER** 1 ns, 5 **AFTER** 3 ns, 6 **AFTER** 5 ns,
8 **AFTER** 6 ns, 6 **AFTER** 8 ns, 7 **AFTER** 15 ns;

sig <= REJECT 5 ns **INERTIAL** 6 **AFTER** 9 ns, 10 **AFTER** 18 ns;

VHDL 词法单元

- 注释： -- 表示从此到行尾为注释；
- 数字：
  - 下划线：在相邻数字之间插入下划线对十进制数的数值不产生影响，例如：123_456 = 123456;
  - 基：以基表示的数需要用#号括起来, #号之前的数字（2 ≤ 基 ≤ 16）代表基，例如：
    2#1111_1111# = 8#377# = 16# FF# = 255
  - 浮点数：
    - 十进制文字：E之前的数代表尾数；E之后的整数代表指数：
      例： 1.2E-6, 2.4E+3
    - 以基表示的浮点数实例：
      2#1.1111_1111#E4 （等于十进制数32.9375）
      16#FF.FF#E+1 （等于十进制数4095.9375）

VHDL 词法单元（续）

- 字符：被单引号括起来的ASCII字符（可以为空），例如：
  'A'，'*'，'"'，''（空），
- 字符串：被双引号括起来的ASCII字符串（可以为空），例如：
  "can be used for",
  "A",
  ""（空），
- 位串：被双引号括起来的数字序列，例如：
  B"1111_0101", --长度为8的二进制数
  X"F0F", -- 长度为3的16进制数
  O"070", -- 长度为3的8进制数
  - B, X, O 代表数字的基。

重载

- 类型重载；
- 子程序（ 函数和过程 ）名的重载；

类型重载

- 怎样消除二义性？

```
type   Four_level   is   ( Rising, High, Falling, Low );
type   Two_level    is   ( High,   Low );
        -- High 和 Low 被重载，会不会引起二义性？

signal   s1:   Four_level   := Low;
signal   s2:   Two_level    := Low;
```

由于类型说明的存在，
消除了二义性！

子程序（函数和过程）名的重载

- 怎样消除二义性？

function _invert (x: three_level_logic) **return** three_level_logic;
function _invert (x: matrix) **return** matrix;

由于提供了参量及返回值的类型信息，消除了歧义！

重载运算符的应用实例

-- 4位加法器 --

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity 4_full_adder is
   port (
      a, b: in std_logic_vector( 0 to 3 );
      cin: in std_logic;
      sum: out std_logic_vector ( 0 to 3 );
      cout: out std_logic
    );
end 4_full_adder;
```

该程序包中对运算符“+”定义了重载

```
architecture behav of 4_full_adder is
begin
   process (a, b, cin )
    variable x: std_logic_vector( 0 to 4 );
   begin
      x := ( '0' & a ) + ( '0' & b ) +
            ( "0000" & cin );
      sum <= x( 1 to 4 );
      cout <= x( 0 );
   end process ;
end behav;
```

运算符“+”被重载，要求各操作数的长度必须相等。

信号类

- 信号说明的一般形式为：

  SIGNAL 标识符表：子类型标识 [信号类] [:= 表达式];

  - 被保护信号必须为决断信号。
  - 如果在信号说明中出现信号类，则这样的信号为被保护信号。

- 信号类关键字：
  - REGISTER
  - BUS

属性

- VHDL中的某些项目类可以具有属性。
- 属性名的一般格式是：
  项目名'属性标识符
- 属性是一个表达式，可用在任何可使用表达式的位置；
- 有关范围的属性可用在表示范围的位置：
  - **type** Row_info **is array** (Window ' Range(1)) **of** Boolean;
  - **type** Column_info **is array**(Window ' Range(2)) **of** Boolean;
  - **subtype** Column_number **is** Integer **range**
    Window ' Range(2);

- 若想了解完整的预定义属性类，请参考VHDL语言参考手册[LRM93]。

属性 -- 事件和事项处理

下面是函数类属性：

- 活跃（active）：信号值从'1'变为'0'是一个活跃实例，而从'1'变为'1'也是一个活跃，唯一的准则是发生了某件事。
- 事件（event）：某个信号发生了事件，是指该信号值发生了变化；信号值从'1'变为'0'是一个事件，但从'1'变为'1'则不是一个事件。
- 事项处理（transaction）：某个信号发生了事项处理，是指该信号引起了一次计算，计算的结果可能使信号值发生变化，也可能不发生变化。某个信号发生了事项处理，意味着该信号一次活跃。
- 所有的事件都是活跃，但并非所有的活跃都是事件。

预定义属性实例

- 信号名'Event：如果在当前模拟周期内该信号发生了某个事件(信号值发生变化)，则返回True；否则，返回False。
- 信号名'Active：如果在当前模拟周期内该信号发生了事项处理，则返回True；否则，返回False。
- 信号名'Last_Event：返回从该信号前一个事件发生到现在所花费的时间。
- 信号名'Last_Value：返回该信号在最近一个事件发生以前的信号值。
- 信号名'Last_Active：返回该信号从前一个事项处理发生到现在所花费的时间。

预定义属性实例（续）

下面是信号类属性：

- 信号名'Stable[(*t*)]：建立一个Boolean类型信号，当该引用信号在*t*个时间单位内没有事件发生时，其值为True。
- 信号名'Delayed[(t)]：建立与该引用信号同样类型的信号，其值为该引用信号延迟*t*个时间单位后的值。
- 信号名'Quiet[(t)]：建立Boolean类型信号，若该信号已静待*t*个时间单位不活跃，则返回True；否则，返回False。
- 信号名'Transaction：建立一个Bit类型信号，当信号活跃时，信号值在每个模拟周期对其前值进行翻转。

利用属性的实例

- 检查时钟上升沿(对于Bit类型clk信号)：
  - clk'**Event AND** clk = '1'; -- 一种可能途径
  - **NOT** clk'**Stable AND** clk = '1'; -- 另一种可能途径
- 检查时钟下降沿(对于Bit类型clk信号)：
  - clk'**Event AND** clk = '0'; -- 一种可能途径
  - **NOT** clk'**Stable AND** clk = '0'; -- 另一种可能途径

利用属性的实例（续）

◆ **稳定性：**

- signal_name′**Last_Event** >= 10 ns;

  -- 信号上次事件至少发生在**10 ns**前

- signal_name′**Stable**(10 ns);

  -- 最少已稳定**10 ns**

利用属性的实例（续）

- **Std_Logic_1164中，有下面2个预定义函数用来检查时钟沿：**

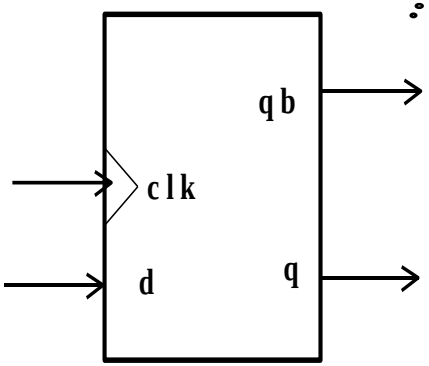
  FUNCTION Rising_Edge**(SIGNAL** s : Std_ULogic**) RETURN** Boolean**;**

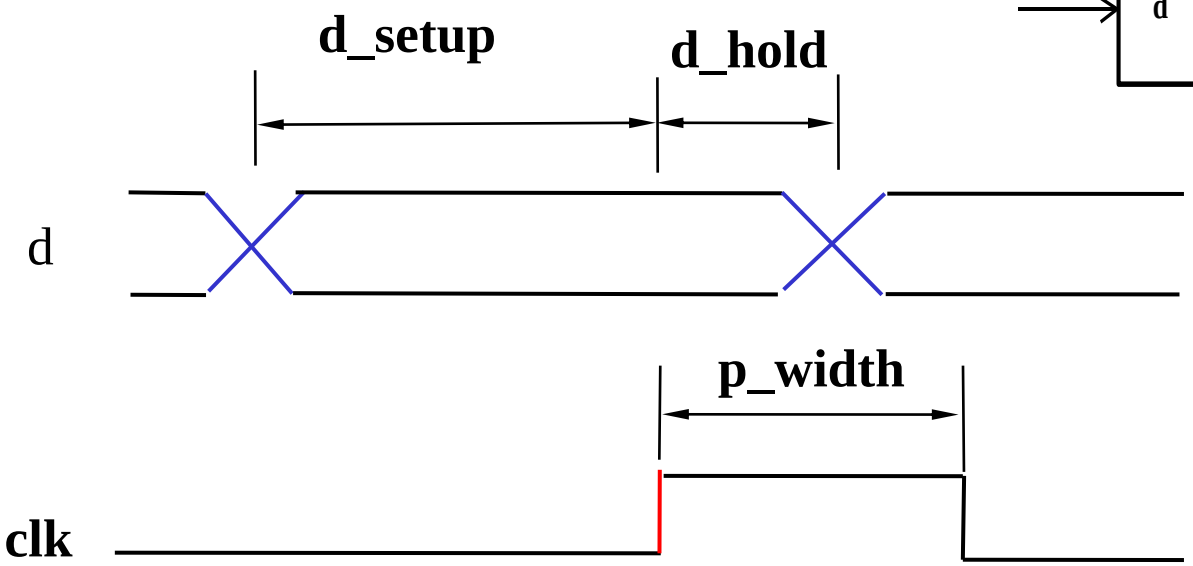
  FUNCTION Falling_Edge**(SIGNAL** s : Std_ULogic**) RETURN** Boolean**;**

- **可用于：**
  - **检查“建立时间”和“保持时间”：**
  - **脉冲宽度检查：**

触发器的“建立时间”和“保持时间”

- 使用预定义属性检查数据建立/保持时间和最小脉冲宽度：





实例 见下页

```
PROCESS(clk, d)
  BEGIN
    -- 在时钟上升沿，检查数据建立时间
    IF Rising_Edge(clk) THEN
      ASSERT (d′Last_Event >= d_setup)    -- d_setup 是预先定义的类属参数
        REPORT ″setup time error″ SEVERITY Warning;
    -- 在时钟下降沿，检查最小时钟脉冲宽度
    ELSIF Falling_Edge(clk) THEN
      ASSERT(clk′Last_Event >= p_width)
        REPORT ″clk minimum pulse width error″ SEVERITY Warning;
    -- 当数据在时钟上升沿之后变化时，检查数据保持时间
    ELSIF (d′Event) AND (clk′Last_Value = ′0′) THEN
      ASSERT(clk′Last_Event >= d_hold)    -- d_ hold 是预先定义的类属参数
        REPORT ″Data hold time error″ SEVERITY Warning;
    END IF;
     -- 边沿触发 D 触发器的操作
    IF Rising_Edge(clk) THEN
      q <= d;
      not_q <= NOT d;
    END IF;
END PROCESS;
```

数组对象的属性（例）

◆ 例： **VARIABLE** byte : Bit_Vector (7 **DOWNTO** 0);

数组对象的属性值

属　　性	值
byte′Left	7
byte′Right	0
byte′Low	0
byte′High	7
byte′Length	8
byte′Range	7 **DOWNTO** 0
byte′Reverse_Range	0 **TO** 7

枚举对象的属性（例）

◆ 例：**TYPE** operations **IS** (add, sub, mult, div);

属性值

属　　性	值
operations′Leftof(sub)	add
operations′Rightof(sub)	mult
operations′Pos(add)	0
operations′Val(3)	div

自定义属性

- **目的：** 通过属性插入附加信息。
- **实例：**

```
architecture  NAND_Impl  of  SR_FF  is
    component  NAND_Gate
        port ( A, B:  in  bit;  Y:  out  bit );
    end  component;
    attribute  Placement  of  NAND1:  lable  is  (200, 100);
    attribute  Placement  of  NAND2:  lable  is  (200, 90);
begin
    NAND1:  NAND_Gate  port  map ( s, q_bar, q );
    NAND2:  NAND_Gate  port  map ( s, q, q_bar );
end  NAND_Impl;
```

关键字

属性值

- 在此定义之后，当使用

NAND1′Placement

时，就可以得到预先定义的属性值（ **200， 100** ）

子程序

- 子程序分为过程和函数：
  - 过程调用是1条语句；
  - 函数调用是1个返回值的表达式。

函数和过程的区别

	函　　数	过　　程
形式参数允许的方式	IN	IN，OUT，INOUT
形式参数允许的对象类	常量，信号	常量，信号，变量
形式参数使用的默认对象类	常　量	常量(对 IN 方式) 变量(对 INOUT 和 OUT 方式)
等待语句，顺序信号赋值	不允许	允　许

子程序重载

-- 子程序的重载指两个或多个子程序使用相同的名字，它允许这些子程序使用不同类型的参数表。例如：

```
FUNCTION min (a, b : Integer)
    RETURN Integer IS
BEGIN
  IF a < b THEN
    RETURN a;
  ELSE
    RETURN b;
  END IF;
END min;
```

```
FUNCTION min(a, b : Real)
    RETURN Real IS
BEGIN
  IF a < b THEN
    RETURN a;
  ELSE
    RETURN b;
  END IF;
END min;
```

重载函数举例

- 重载函数**Std_Match**取自程序包**Numeric_Std**：

FUNCTION Std_Match (L, R : Std_ULogic) **RETURN** Boolean;

FUNCTION Std_Match (L, R : Unsigned) **RETURN** Boolean;

FUNCTION Std_Match (L, R : Signed) **RETURN** Boolean;

FUNCTION Std_Match (L, R : Std_Logic_Vector) **RETURN** Boolean;

FUNCTION Std_Match (L, R : Std_ULogic_Vector) **RETURN** Boolean;

- 当几个子程序重名时，下列因素决定使用哪一个子程序。其层次顺序如下：
  - 子程序调用中出现的参数数目；
  - 调用中出现的参数类型；
  - 调用使用带名字的参数关联时参数的名字；
  - 子程序为函数时返回值的类型。

子程序调用时形参数与实参的关联

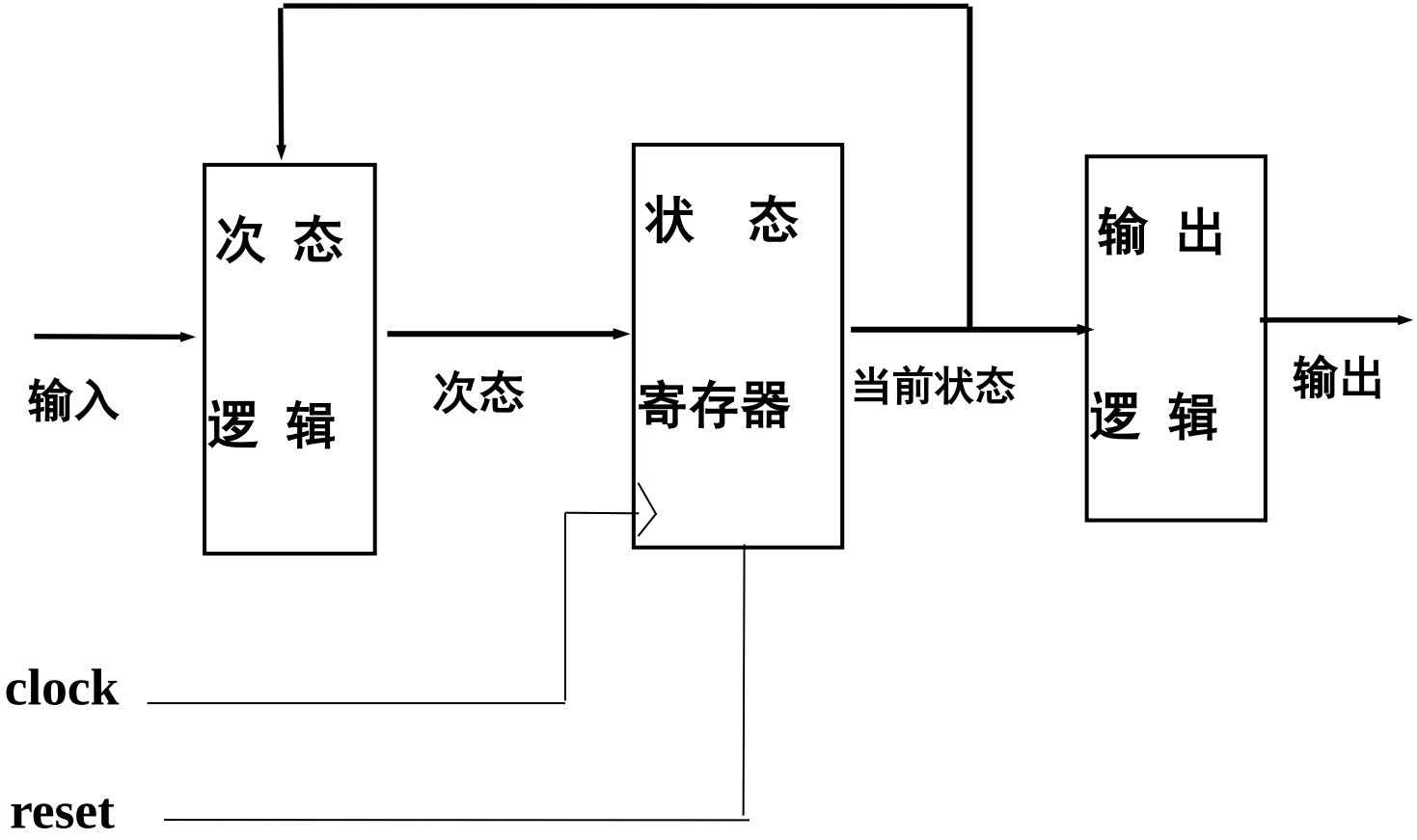
- 位置关联；
- 名字关联；
- 实例：
  - 假定已经定义了函数 ：

    FUNCTION min (a, b : Integer) **RETURN** Integer
  - 现在用实参（ x, y)调用函数 min ， 以下3种调用形式等效：

```
min( x,  y ) ;                  -- 位置关联
min( a => x,  b => y );         -- 名字关联（顺序无关）
min( b => y,  a => x );         -- 名字关联（顺序无关）
```

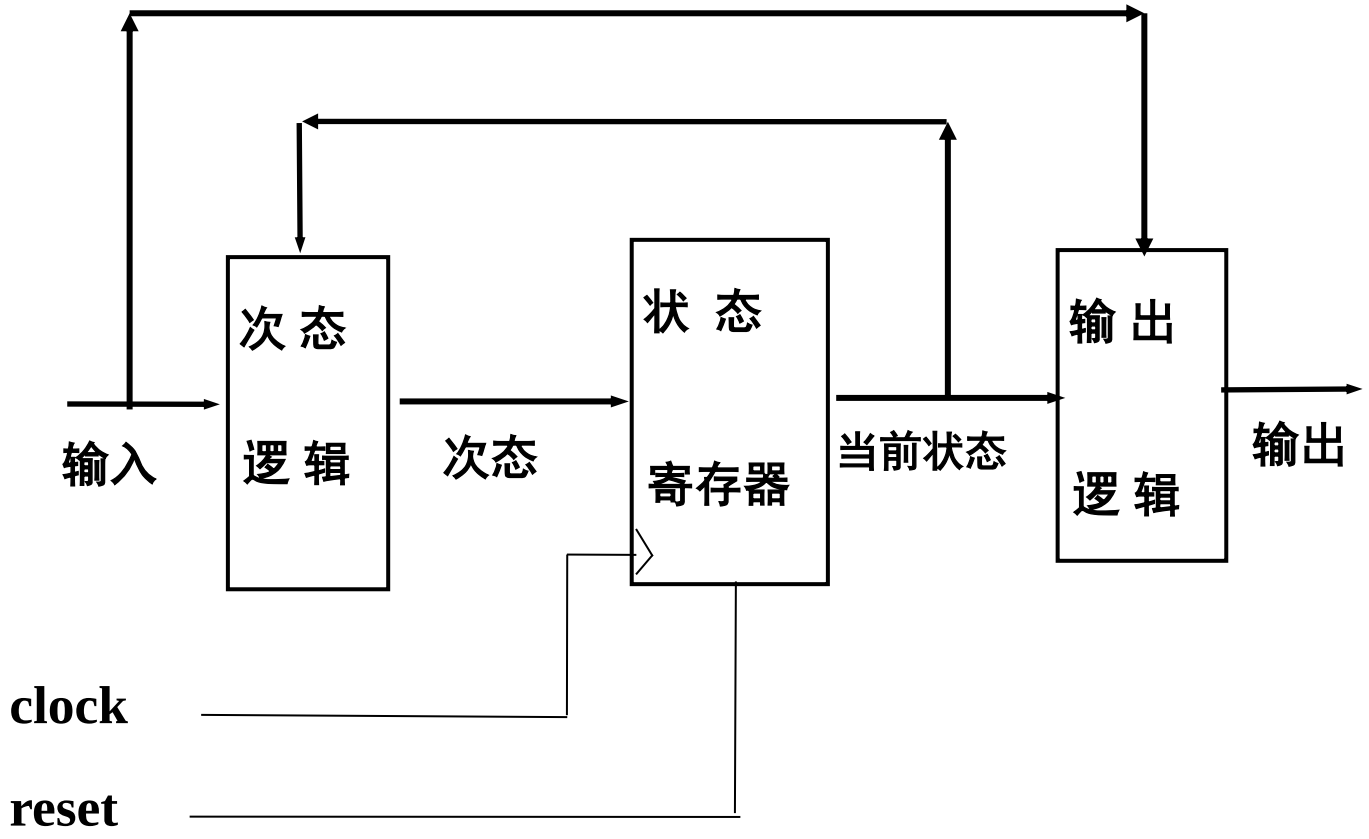
有限状态机（FSM）模型

- Moore型有限状态机：输出是且仅是FSM当前状态的函数；

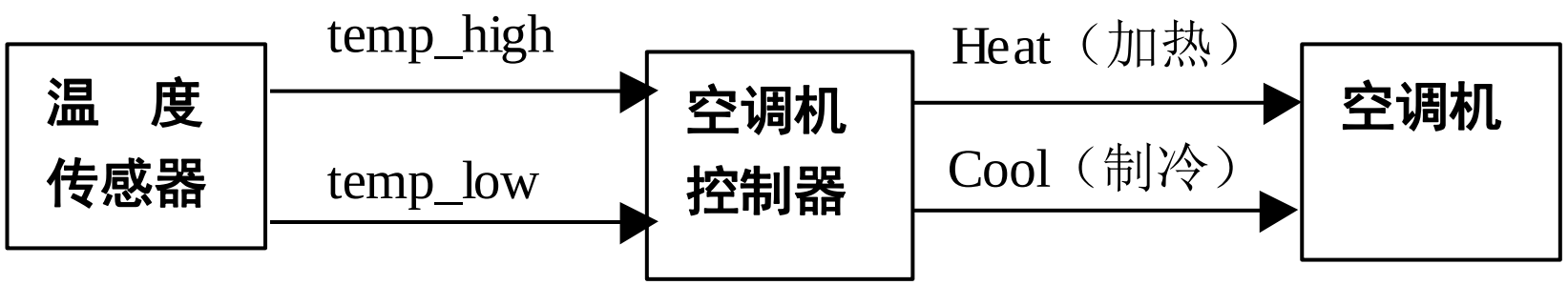


有限状态机（FSM）模型（续）

- **Mealy型有限状态机：输出是FSM当前状态和输入变量当前值的函数。**

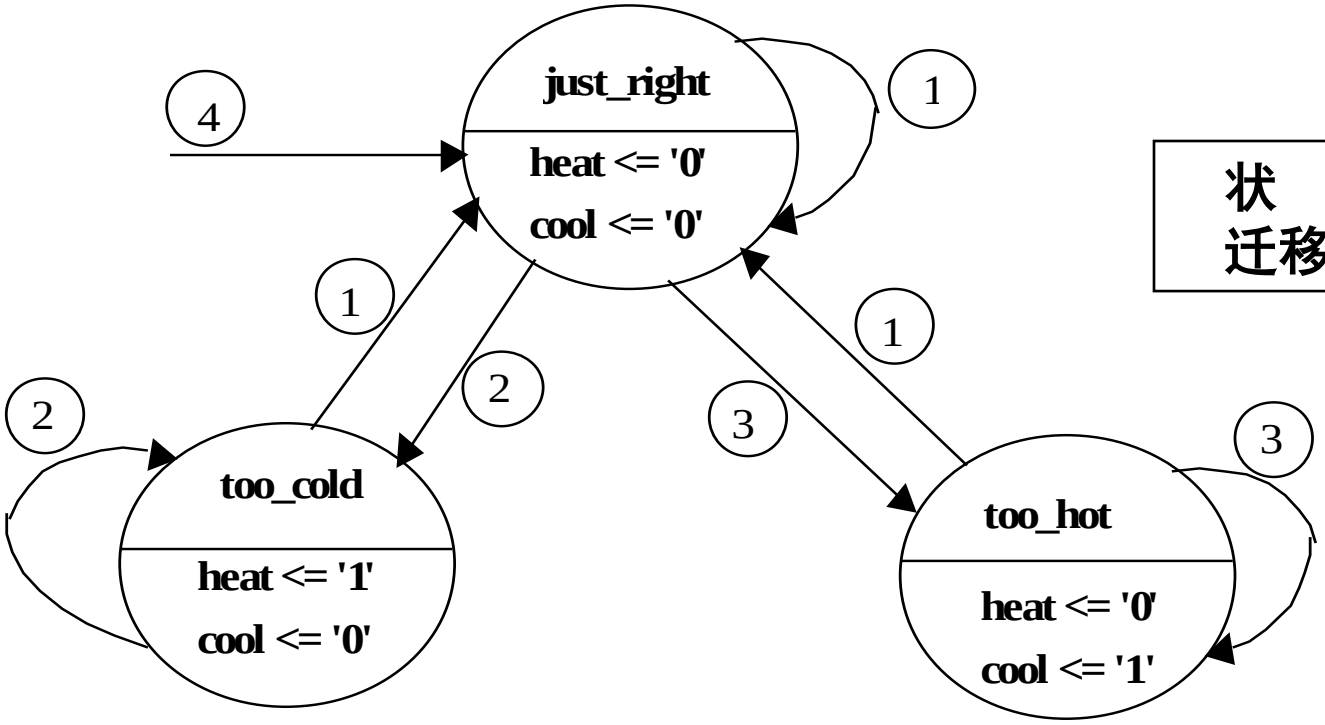


Moore型状态机 —— 空调机控制器



空调机控制器作用示意图

Moore型状态机 —— 空调机控制器



状　态
迁移图

输出是且仅是FSM当前状态的函数

(1) temp_high = '0' AND temp_low = '0'
(2)　　temp_low = '1'
(3)　　temp_high = '1'
(4)　　reset = '0'

Moore型状态机 —— 空调机控制器

◆ 状态迁移表：

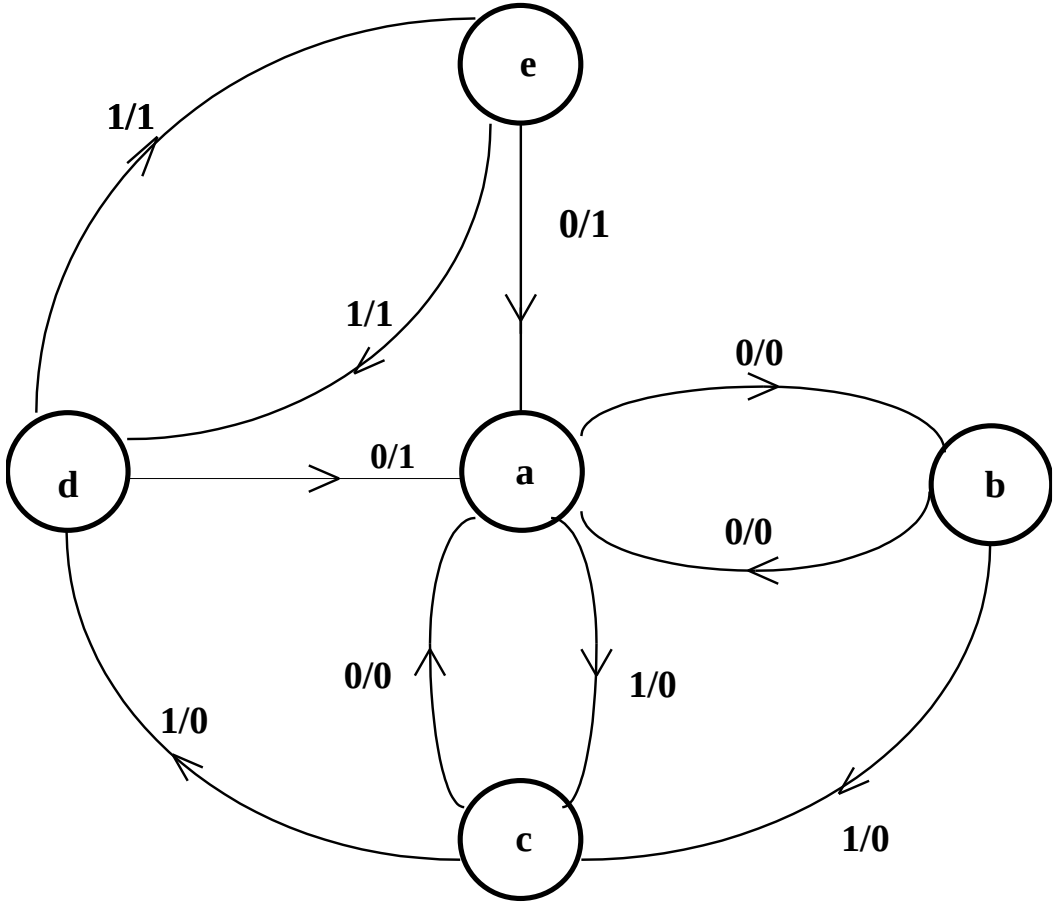
输出是且仅是FSM当前状态的函数

输入			当前状态	次态	输出	
reset	temp_high	temp_low			Heat	Cool
0	X	X	X	just_right	0	0
1	0	0	just_right	just_right	0	0
1	0	1	just_right	too_cold	0	0
1	1	0	just_right	too_hot	0	0
1	1	0	too_hot	too_hot	0	1
1	0	0	too_hot	just_right	0	1
1	0	1	too_cold	too_cold	1	0
1	0	0	too_cold	just_right	1	0

Mealy型状态机

- 状态迁移图：

输出是FSM当前状态和输入变量当前值的函数



Mealy型状态机

◆ 状态迁移表：

当前状态 State	输入信号 In	后继状态 Next	输出信号 Out
a	0	b	0
a	1	c	0
b	0	a	0
b	1	c	0
c	0	a	0
c	1	d	0
d	0	a	1
d	1	e	1
e	0	a	1
e	1	d	1

输出是FSM当前状态和输入变量当前值的函数

有限状态机的描述风格

◆ 描述方法的若干建议：

- 有1个状态变量，用它指定有限状态机的状态；
- 有1个时钟(假设为单相、全局时钟)；
- 状态转移指定；
- 输出指定；
- 同步或异步(可选，异步更好)复位信号。

◆ 状态编码的策略：

- 1热态位码；
- 顺序码；
- 随机码；

◆ 状态编码的确定：

- EDA工具指定（含优化）；
- 设计者指定。

设计者指定状态编码方法之一

◆ 用自定义属性人工指定状态编码实例（Max+Plus2）：

```
ARCHITECTURE Behavior OF simple IS
   TYPE  State_TYPE  IS  (A, B, C) ;
   ATTRIBUTE  ENUM_ENCODING  :  STRING ;
   ATTRIBUTE  ENUM_ENCODING  OF  State_type     :
                        TYPE  IS  "00 01 11" ;
   SIGNAL  y_present,  y_next : State_type ;
BEGIN
    .
    .
    .
END
.
```

自左向右按位置匹配：

A ➔ "00 "

B ➔ " 01"

C ➔ " 11"

设计者指定状态编码方法之二

```
LIBRARY ieee ;
USE ieee.std_logic_1164.ALL ;
ENTITY simple IS
        PORT ( Clock, Resetn, w      : IN  STD_LOGIC ;
                    z                : OUT STD_LOGIC ) ;
END  simple ;

ARCHITECTURE Behavior OF simple IS
        SIGNAL y_present, y_next : STD_LOGIC_VECTOR(1 DOWNTO 0);
        CONSTANT A    : STD_LOGIC_VECTOR(1 DOWNTO 0) := "00" ;
        CONSTANT B    : STD_LOGIC_VECTOR(1 DOWNTO 0) := "01" ;
        CONSTANT C    : STD_LOGIC_VECTOR(1 DOWNTO 0) := "11" ;
BEGIN
        PROCESS ( w, y_present )
        BEGIN
            CASE y_present IS
                WHEN A =>
                    IF w = '0' THEN y_next <= A ;
                    ELSE y_next <= B ;
                    END IF ;
```

设计者指定状态编码方法之二（续）

```
                WHEN  B =>
                      IF  w = '0'  THEN  y_next <= A ;
                      ELSE  y_next <= C ;
                      END IF ;
                WHEN  C =>
                      IF  w = '0'  THEN  y_next <= A ;
                      ELSE  y_next <= C ;
                      END IF ;
                WHEN OTHERS =>
                      y_next <= A ;
          END CASE ;
     END PROCESS ;
     PROCESS ( Clock, Resetn )
     BEGIN
          IF  Resetn = '0'  THEN
               y_present <= A ;
          ELSIF  (Clock'EVENT AND Clock = '1')  THEN
               y_present <= y_next ;
          END IF ;
     END PROCESS ;
     z <= '1'  WHEN  y_present = C  ELSE  '0' ;
END Behavior ;
```

有限状态机的描述风格

描述风格	功 能 划 分	所用进程数目
风格 A	1. 次态逻辑 2. 状态寄存器 3. 输出逻辑	3
风格 B	1. 次态逻辑、状态寄存器、输出逻辑	1
风格 C	1. 次态逻辑、状态寄存器 2. 输出逻辑	2
风格 D	1. 次态逻辑 2. 状态寄存器、输出逻辑	2
风格 E	1. 次态逻辑、输出逻辑 2. 状态寄存器	2

本章小结

- VHDL语法结构和执行过程
- VHDL组成：

  entity (architecture),

  configuration

  package (package body)
- 进程概念：
  - 进程内顺序执行，进程之间并行执行；
  - 进程只有激活或挂起，永不停止（无限循环）。
- 信号与变量：
  - 信号：进程间的联系；
  - 变量：进程内部的临时变量。
- 元件（层次化）例化，配置的作用。