硬件描述语言VHDL设计

计算机科学与技术学院
张剑贤

内容纲要

- VHDL语法结构
- 组合逻辑电路设计
- 时序电路设计
- 有限状态机设计

一、VHDL语法结构

- 1.1 VHDL概述
- 1.2 VHDL程序结构
- 1.3 VHDL语言元素
- 1.4 VHDL基本逻辑语句
- 1.5 VHDL描述方式

1.1 VHDL概述

常用硬件描述语言：

1、ABEL-HDL

2、AHDL

3、VHDL

4、Verilog HDL

} IEEE标准

1.1 什么是VHDL（HDL）？

- VHDL：
- VHSIC (Very High Speed Integrated Circuit)
- Hardware
- Description
- Language

VHDL发展

- IEEE工业标准硬件描述语言
- 用于仿真及综合的高级描述语言

80 年代初　由美国国防部在实施超高速集成电路（VHSIC）项目时开发的。

1987年　IEEE 协会批准为 IEEE 工业标准称为 IEEE1076-1987。

1993年　更新为 93 标准，IEEE1076. 93。

1996年　IEEE1076.3成为综合标准

VHDL和Verilog HDL

- Verilog HDL

  由Verilog 公司开发，1995年成为IEEE标准。

  优点：与C语言风格类似，简单易学

  缺点：数据类型定义模糊化

- VHDL

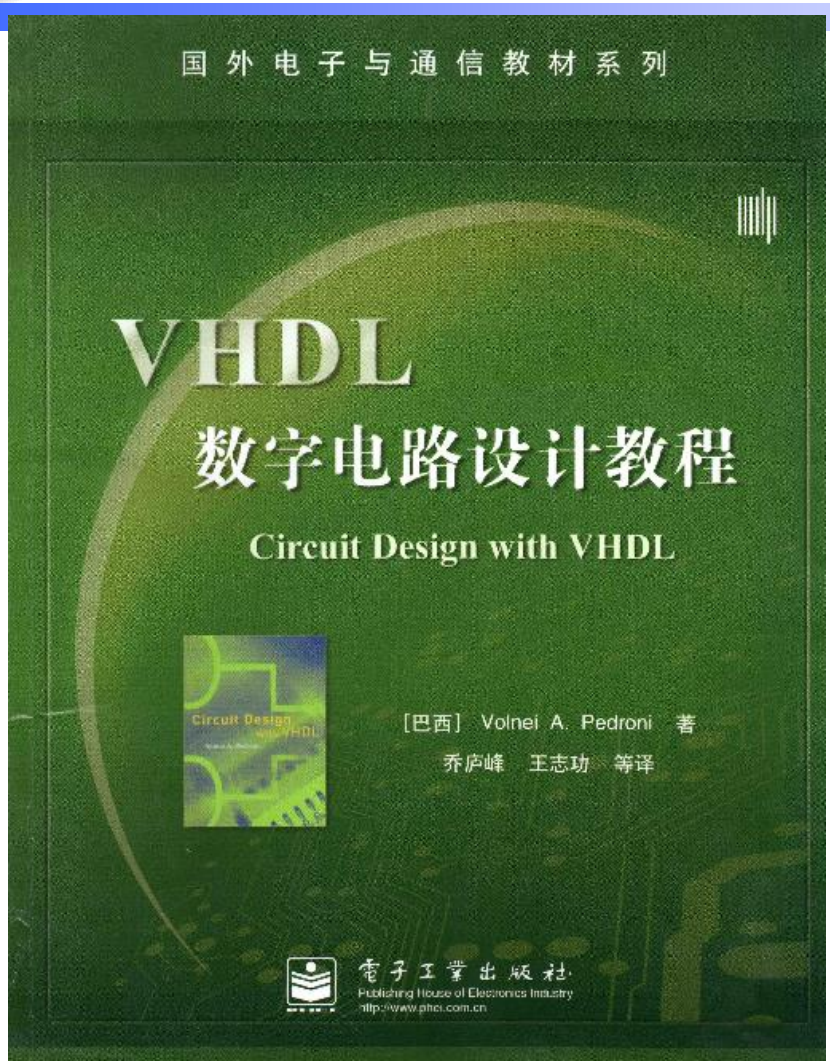
  优点：功能强大、通用性强，语法严谨

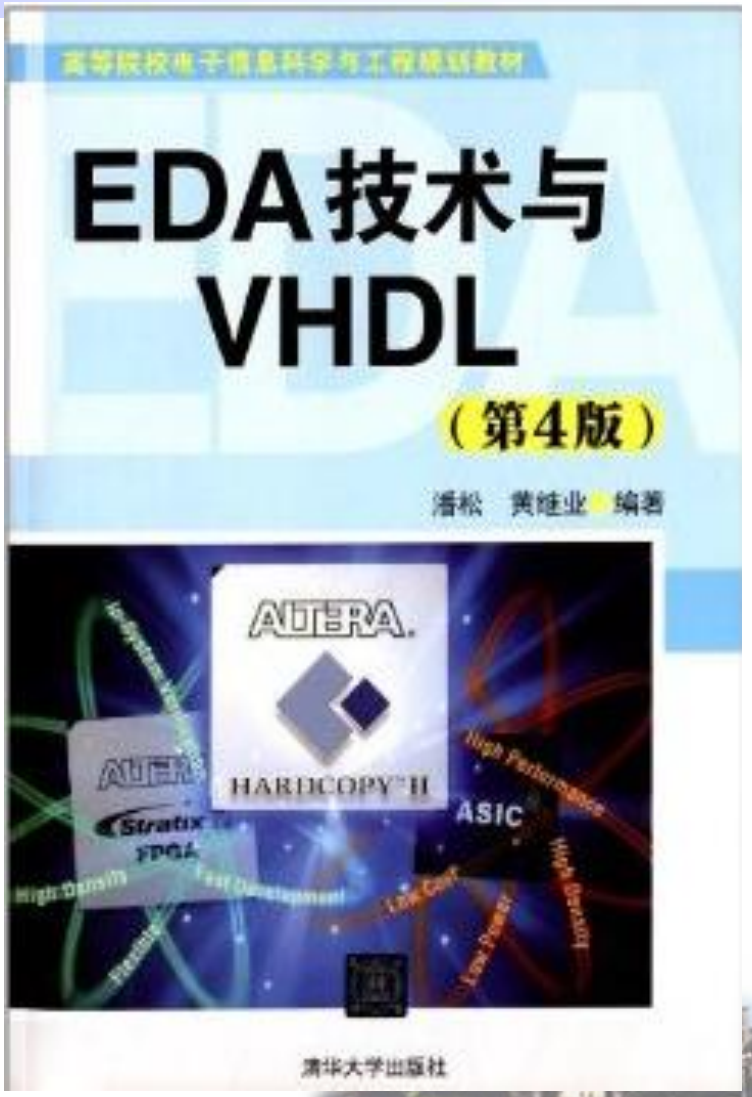
  缺点：语法结构复杂

参考文献

- www.vhdl.org
- www.verilog.org
- VHDL在线参考　www.acc-eda.com/vhdlref/index.html
- Verilog常见问答 http://parmita.com/verilogfaq
- http://china.xilinx.com/

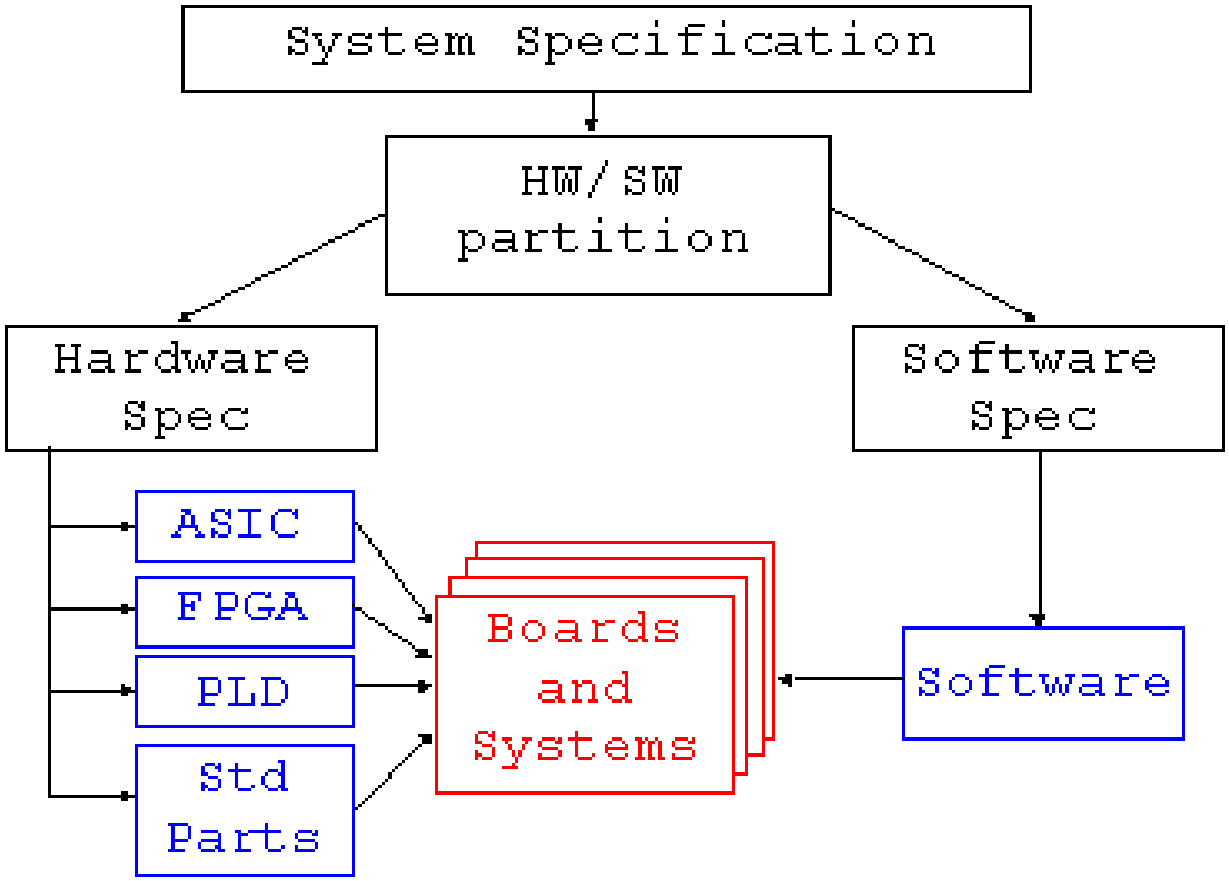
参考书籍


国外电子与通信教材系列
VHDL
数字电路设计教程
Circuit Design with VHDL
[巴西] Volnei A. Pedroni 著
乔庐峰 王志功 等译
電子工業出版社
Publishing House of Electronics Industry


EDA技术与
VHDL
（第4版）
潘松 黄继业 编著
ALTERA
HARDCOPY II
Stratix
FPGA
ASIC
清华大学出版社

VHDL在电子系统设计中的应用

- 电子系统的设计模块



VHDL在电子系统设计中的应用

- 电子系统设计的描述等级

1、行为级

2、RTL级（Register transfer level）

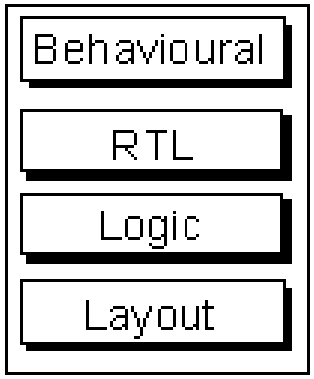
3、逻辑门级

4、版图级

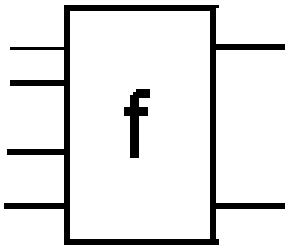
- 用VHDL可以描述以上四个等级

VHDL在电子系统设计中的应用

- 行为级



Behavioural

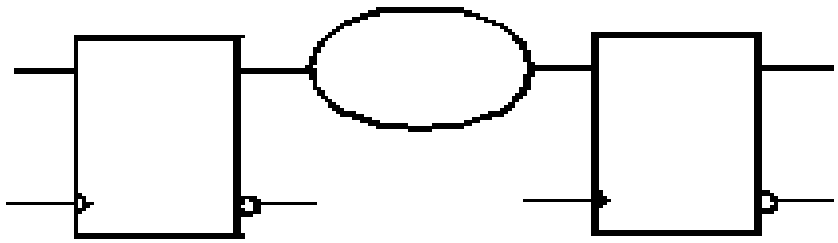


Function only, no architecture

Timing detail as required

VHDL在电子系统设计中的应用

- 寄存器传输级

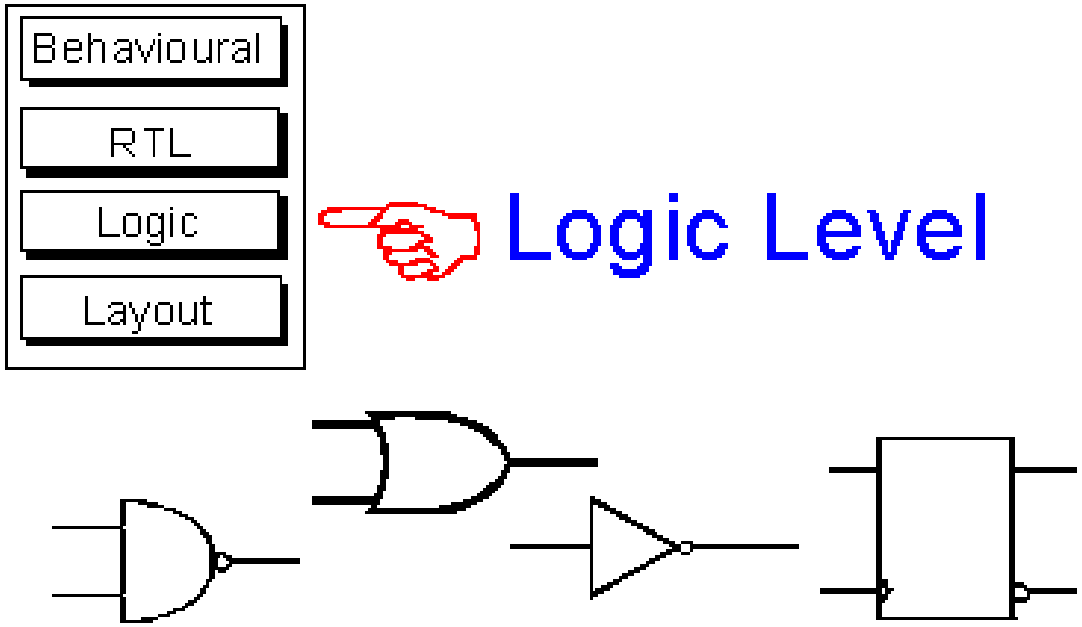


Cycle based timing

Function and register architecture

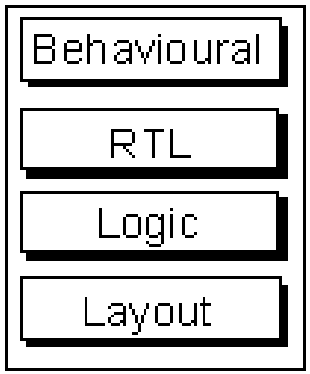
VHDL在电子系统设计中的应用

- 逻辑门级

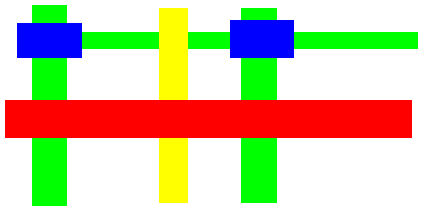


VHDL在电子系统设计中的应用

- 版图级



Layout Level



Layout on silicon

Timing, analog effects

1.2 VHDL程序结构

- 实体（Entity）
- 结构体（Architecture）
- 配置（Configuration）
- 库（Library）、程序包（Package）

VHDL程序基本结构

一个完整的VHDL程序

- **实体**（Entity）
- **结构体**（Architecture）
- **配置**（Configuration）
- **包集合**（Package）
- **库**（Library）

*实体部分*描述设计系统的外部接口信号（即输入/输出信号）

*结构体*用于描述系统的内部电路

*配置*用于从库中选取所需元件安装到设计单元的实体中

*包集合*存放各设计模块能共享的数据类型、常数、子程序等

*库*用于存放已编译的实体、结构体、包集合和配置

VHDL程序结构

例1　一个2输入的与门的逻辑描述

```
LIBRARY ieee;                                    --库说明语句
USE ieee.std_logic_1164.ALL;                     --程序包说明语句
ENTITY and2 IS
        PORT(a,b      : IN  STD_LOGIC;
             y        : OUT  STD_LOGIC);
END and2;
ARCHITECTURE and2x OF and2 IS
BEGIN
        y<=a AND b;
END and2x;
```

实体部分

结构体部分

结构关系

库、程序包

实体（**Entity**）

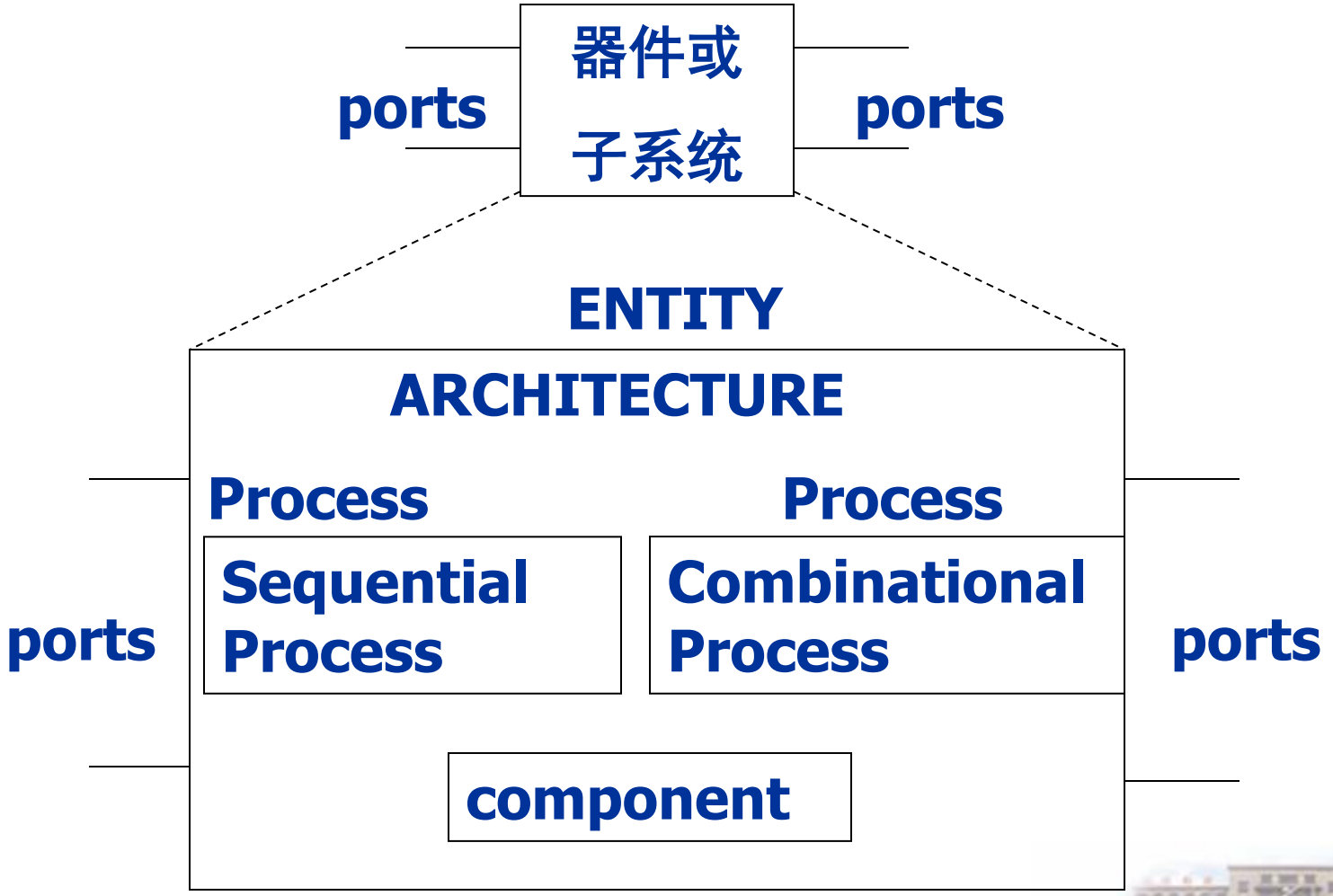
结构体

（**Architecture**）

进程
或其它并行结构

配置（**Configuration**）

实体结构


器件或
子系统
ports
ports
ENTITY
ARCHITECTURE
Process
Process
Sequential
Process
Combinational
Process
ports
ports
component

VHDL结构要点

1、ENTITY（实体）

格式：

ENTITY 实体名 IS

[类属参数说明]

[端口说明]

End ENTITY 实体名;

其中端口说明格式为：

PORT(端口名1，端口名N：方向：类型）

其中方向有: IN , OUT, INOUT, BUFFER, LINKAGE

1、ENTITY（实体）----类属说明

类属说明:

确定实体或组件中定义的局部常数。模块化设计时多用于不同层次模块之间信息的传递。可从外部改变内部电路结构和规模。

类属说明必须放在端口说明之前。

```
Generic (
          常数名称: 类型 [: = 缺省值]
         {常数名称: 类型 [: = 缺省值]}
              );
```

1、ENTITY（实体）----类属说明

类属常用于定义：

实体端口的大小、

设计实体的物理特性、

总线宽度、

元件例化的数量等。

```
entity mck is
        generic(width: integer:=16);
        port(add_bus:out std_logic_vector
            (width-1 downto 0));
```

1、ENTITY（实体）----类属说明

例：2输入与门的实体描述

```
entity and2 is
   generic(risewidth: time:= 1 ns;
           fallwidth: time:= 1 ns);
   port(a1: in std_logic;
        a0: in std_logic;
        z0: out std_loigc);
end entity and2;
```

注：数据类型 time 用于仿真模块的设计。

综合器仅支持数据类型为整数的类属值。

1、ENTITY（实体）----端口声明

端口声明：确定输入、输出端口的数目和类型。

```
Port （
      端口名称{，端口名称}：端口模式     数据类型；
      …
      端口名称{，端口名称}：端口模式     数据类型
            ）；
```

端口模式：

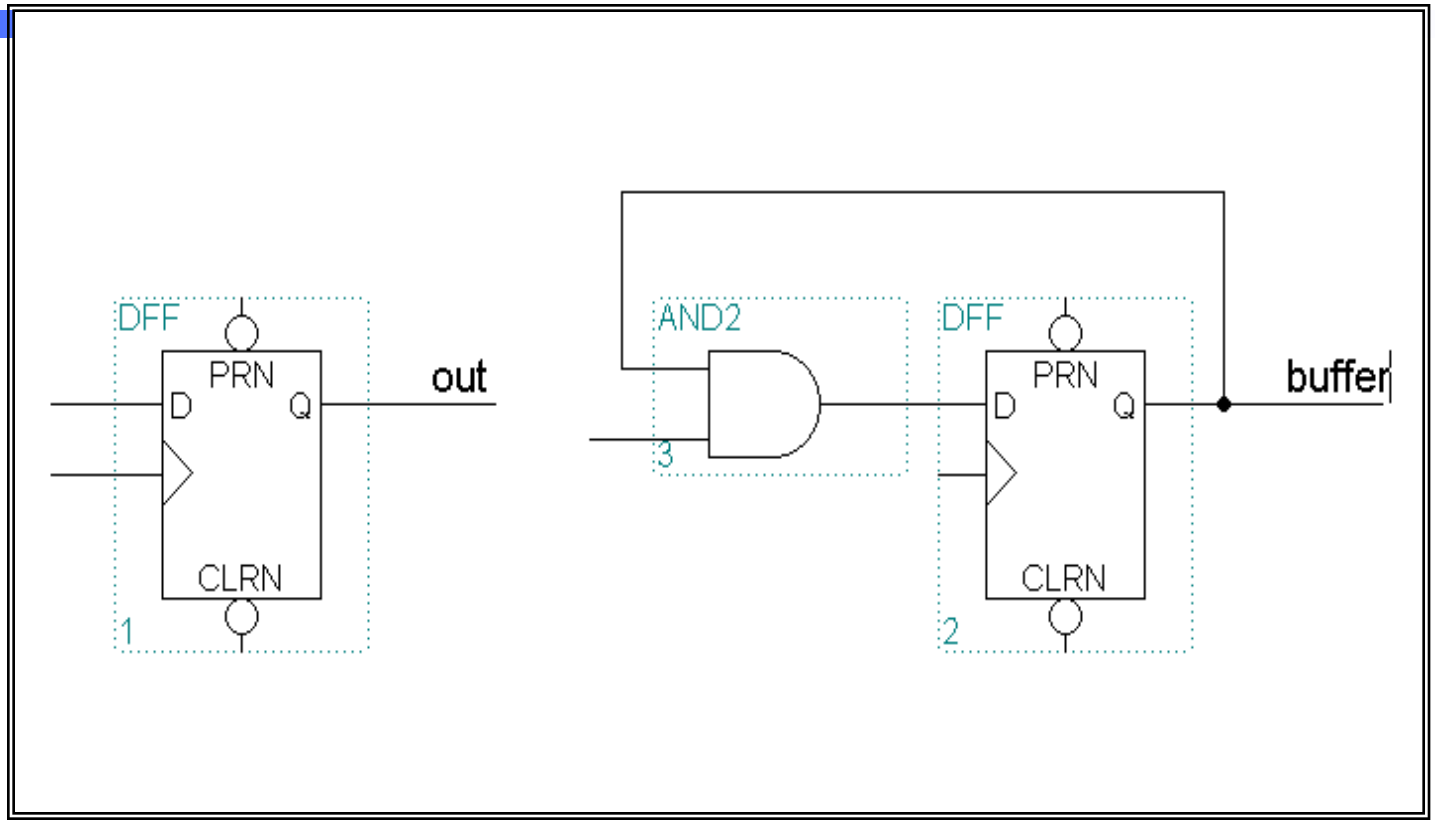
in: 输入型，此端口为只读型。

out: 输出型，只能在实体内部对其赋值。

inout:输入输出型，既可读也可赋值。

buffer: 缓冲型，与 out 相似，但可读。

out 和 buffer 的区别



1、ENTITY（实体）----数据类型

指端口上流动的数据的表达格式。为预先定义好的数据类型。

如：bit、bit_vector、integer、std_logic、std_logic_vector 等。

例：

```
entity nand2 is
 port (
   a,b:in bit;
   z: out bit
      ) ;
end entity nand2;
```

```
entity m81 is
 port(
   a:in bit_vector(7 downto 0);
   sel:in bit_vector(2 downto 0);
      b:out bit);
end entity m81;
```

1、ENTITY（实体）----例程

- 例子 (HalfAdd)

```
entity HALFADD is
port (A,B : in bit;
      SUM, CARRY : out bit);
end HALFADD;
```

其内部结构将由Architecture来描述

2、ARCTHITECTURE(构造体）

格式：

Arcthitecture 构造体名 **of** 实体名 **is**

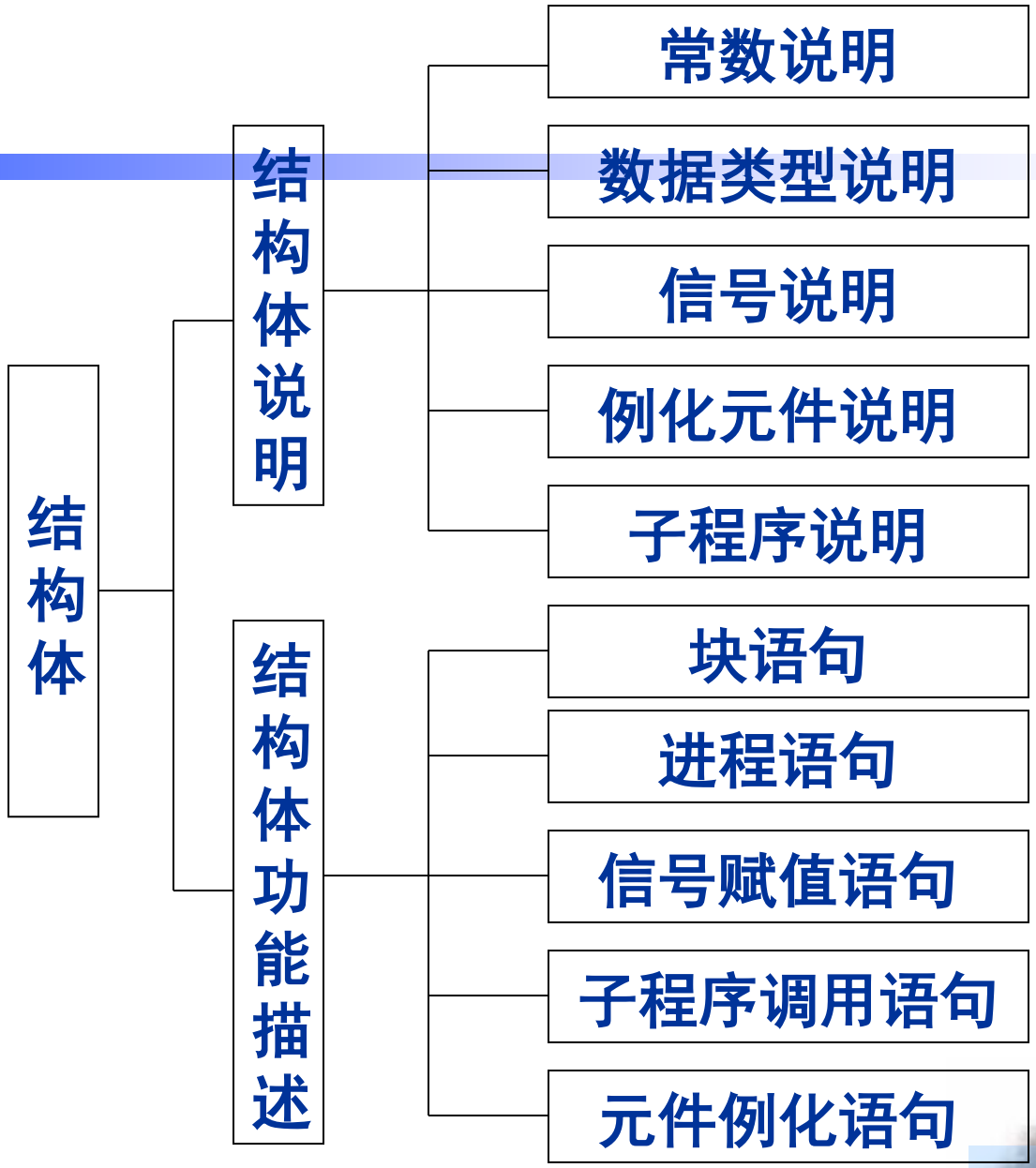
[定义语句] 内部信号、常数、元件、数据类型、函数等的定义

begin

[并行处理语句:block / process /function / procedure]

end 构造体名；

注：同一实体的结构体不能同名。定义语句中的常数、信号不能与实体中的端口同名。


结构体
结构体说明
常数说明
数据类型说明
信号说明
例化元件说明
子程序说明
结构体功能描述
块语句
进程语句
信号赋值语句
子程序调用语句
元件例化语句

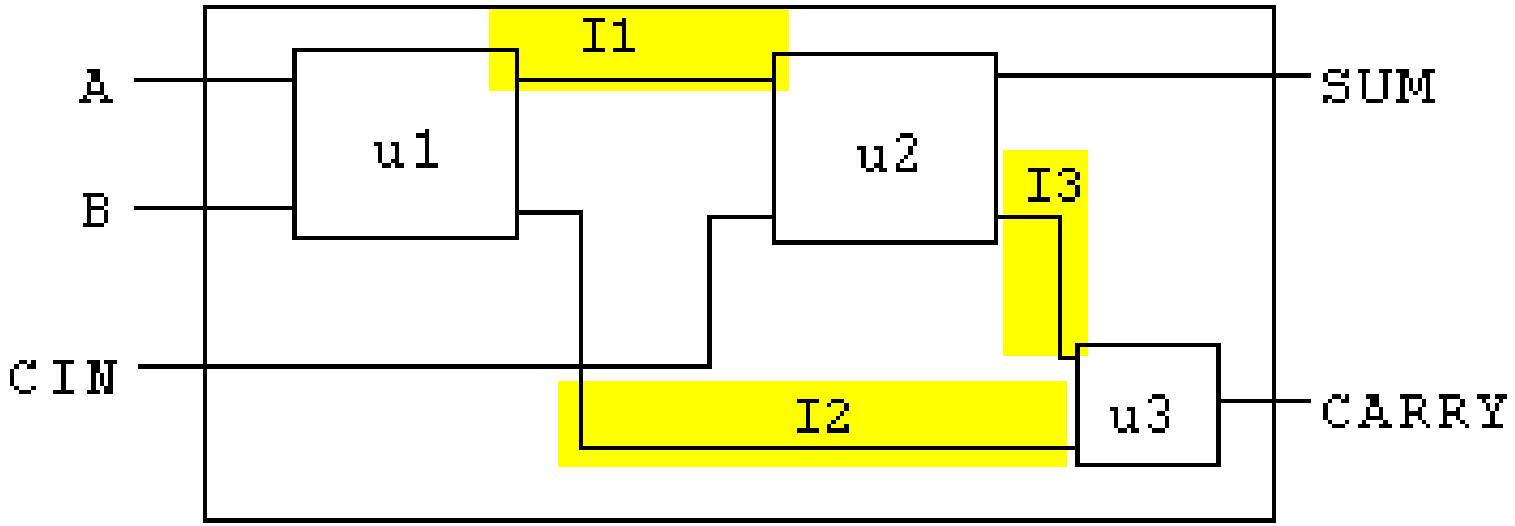
2、ARCTHITECTURE(构造体）

- 例子(HalfAdd)

```
architecture BEHAVE of HALFADD is
begin
    SUM <= A xor B;
    CARRY <= A and B;
end BEHAVE;
```

2、ARCTHITECTURE(构造体）

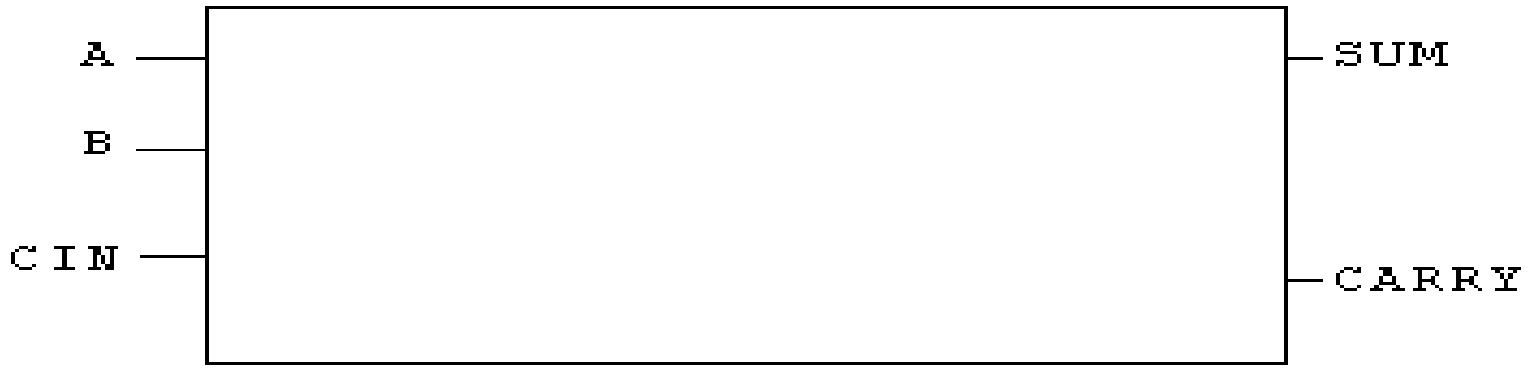
- 例子 (FullAdd) （学习如何调用现有模块）



2、ARCTHITECTURE(构造体）

- 实例(FullAdd)-entity

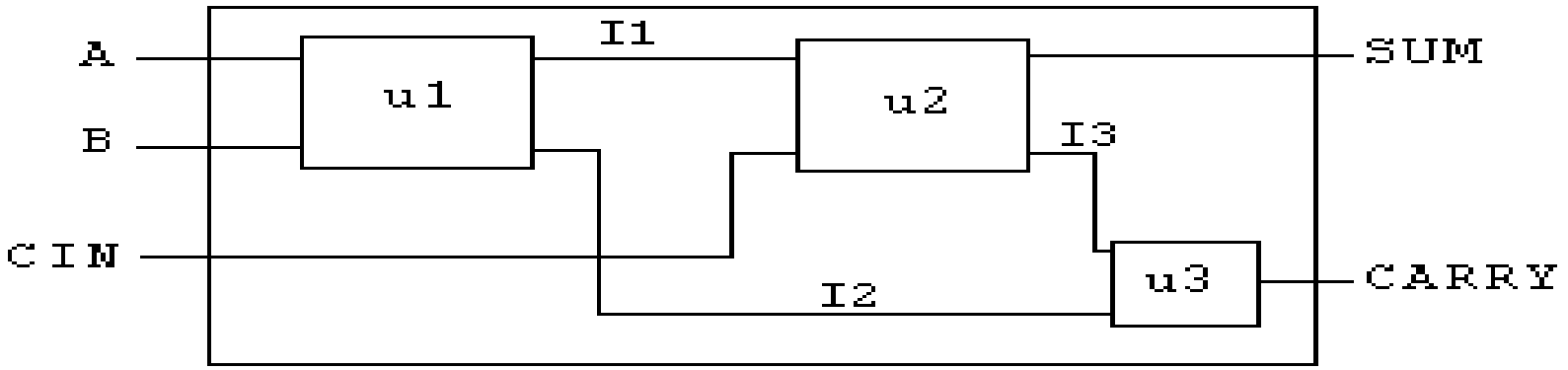
```
entity FULLADD is
port (A, B, CIN   : in bit;
      SUM, CARRY  : out bit);
end FULLADD;
```



2、ARCTHITECTURE(构造体）

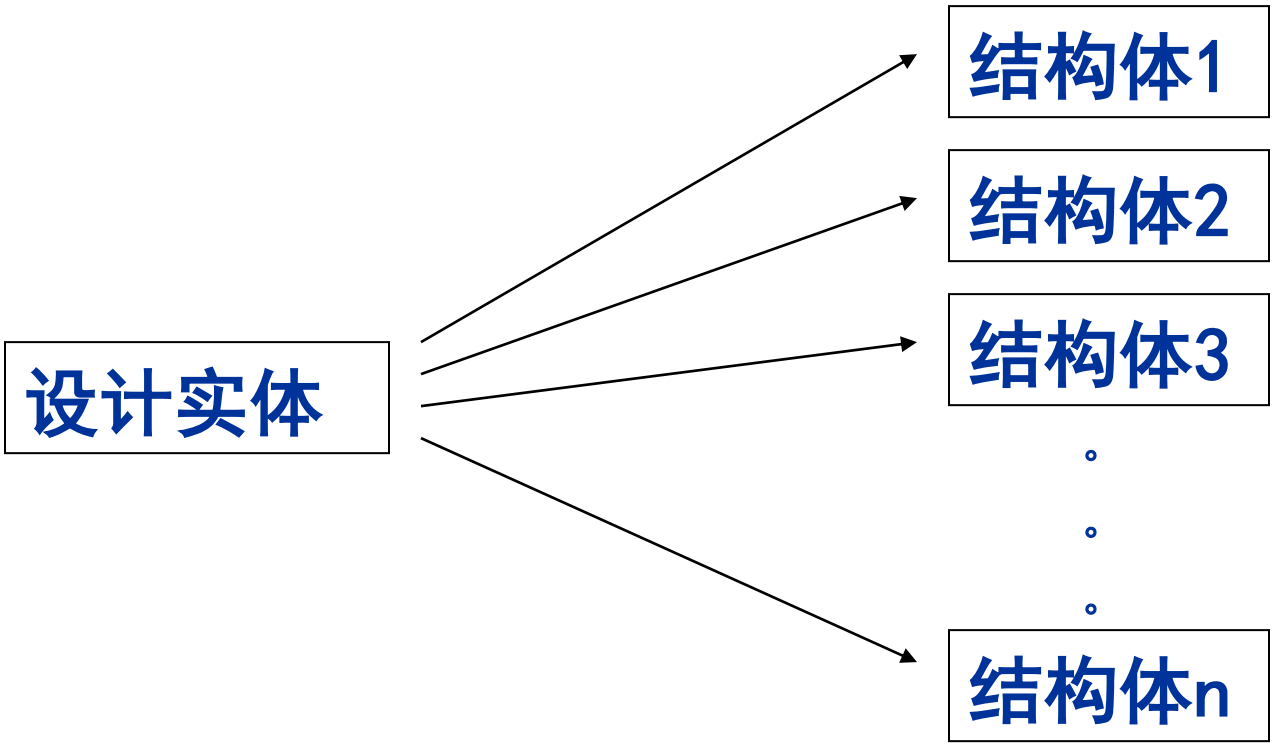
- 实例(FullAdd)-architecture

```
architecture STRUCT of FULLADD is
   signal I1, I2, I3 : bit;
    -- other declarations
begin
   u1:HALFADD port map(A,B,I1,I2);
   u2:HALFADD port map(I1,CIN,SUM,I3);
   u3:ORGATE port map(I3,I2,CARRY);
end STRUCT;
```



```
entity FULLADD is
port (A, B, CIN   : in bit;
      SUM, CARRY : out bit);
end FULLADD;

architecture STRUCT of FULLADD is
  signal I1, I2, I3 : bit;
  component HALFADD
   port(A,B : in bit;
        SUM, CARRY : out bit);
  end component;
  component ORGATE
   port(A,B : in bit;
            Z : out bit);
  end component;
begin
  u1:HALFADD port map(A,B,I1,I2);
  u2:HALFADD port map(I1,CIN,SUM,I3);
  u3:ORGATE port map(I3,I2,CARRY);
end STRUCT;
```

实体与结构体的关系

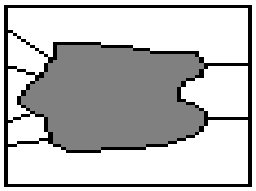


VHDL中的设计单元

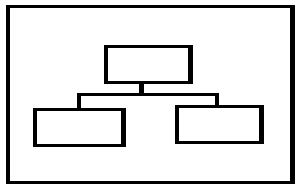
- VHDL中的设计单元（可以独立编译）

Design units...

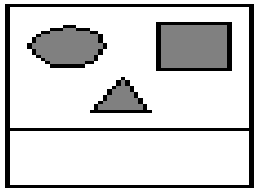
Entity



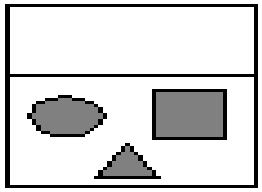
architecture



configuration



Package



Package body

3、Library 库

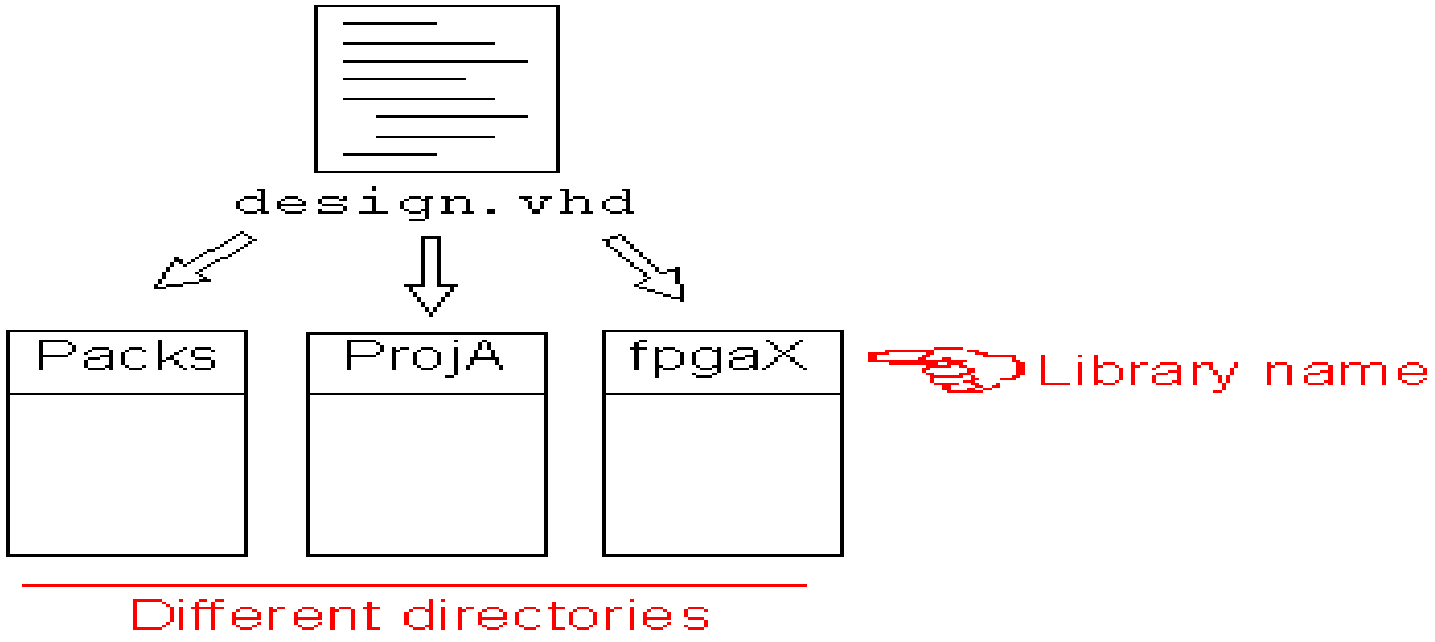
库： 数据的集合。内含各类包定义、实体、构造体等

- STD库　--VHDL的标准库
- IEEE库　-- VHDL的标准库的扩展
- 面向ASIC的库 --不同的工艺
- 不同公司自定义的库
- 普通用户自己的库

3、Library 库

- 用户自己的库

  当您的VHDL文件被编译后，编译的结果储存在特定的目录下，这个目录的逻辑名称即Library，此目录下的内容亦即是这个Library的内容。



3、Library 库—库的种类

VHDL库可分为 5种：

1）IEEE 库

定义了四个常用的程序包：

- std_logic_1164 (std_logic types & related functions)
- std_logic_arith (arithmetic functions)
- std_logic_signed (signed arithmetic functions)
- std_logic_unsigned (unsigned arithmetic functions)

3、Library 库—库的种类

Type STD_LOGIC：

9 logic value system (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’)

- **‘W’, ‘L’, ‘H” weak values (Not supported by Synthesis)**
- **‘X’ - (not ‘x’)used for unknown**
- **‘Z’ - (not ‘z’) used for tri-state**
- **‘-’ Don’t Care**

3、Library 库—库的种类

2）STD 库（默认库）

库中程序包为：standard，

定义最基本的数据类型：

Bit，bit_vector ，Boolean，

Integer，Real，and Time

注：Type Bit

2 logic value system (‘0’, ‘1’)

3）面向ASIC的库

4）WORK库（默认库）

5）用户定义库

3、Library 库—库的使用

```
library  库名;
```

程序包的使用有两种常用格式：

```
use  库名．程序包名．项目名
use  库名．程序包名．All;
```

例：

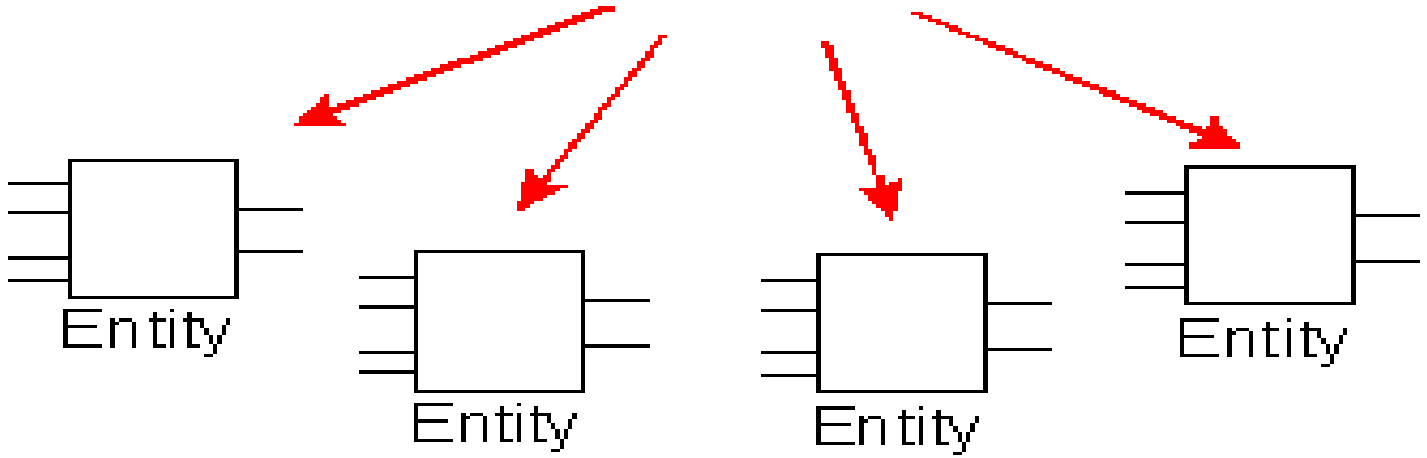
```
library  ieee;
use  ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.conv_integer;
```

2 选 1 选择器

```
library ieee;
use ieee.std_logic_1164.all;

entity mux21 is
    port(a,b:in std_logic;
         s:in std_logic;
         y:out std_logic);
end mux21;

architecture mux_arch of mux21 is
begin
    y<=a when s='0' else
       b when s='1';
end mux_arch;
```

4、Package 包

```
package PROJECT_X is

   -- definitions

end PROJECT_X;
```



程序包与库关系

程序包：

已定义的常数、数据类型、元件调用说明、子程序的一个集合。

目的：方便公共信息、资源的访问和共享。

库：

多个程序包构成库。

4、Package 包--结构

- **程序包说明的内容**

常量说明；
VHDL数据类型说明；
元件说明；
子程序说明；

- **程序包的结构包括**

程序包说明（包首）
程序包主体（包体）

4、Package 包--结构

1）程序包说明（包首）

语法：

```
package   程序包名   is
    { 包说明项 }
end   程序包名;
```

包声明项可由以下语句组成：

use 语句（用来包括其它程序包）；

类型说明；子类型说明；常量说明；

信号说明；子程序说明；元件说明。

例：程序包说明

```
package example is
    type byte is range 0 to 255;
    subtype nibble is byte range 0 to 15;
    constant byte_ff:byte:=255;
    signal addend:nibble;

    component byte_adder
       port(a,b:in byte;
            c:out byte;
            overflow:out boolean);
    end component;

    function my_function(a:in byte)
    return byte;
end example;
```

4、Package 包--结构

2）程序包包体

程序包的内容：子程序的实现算法。

包体语法：

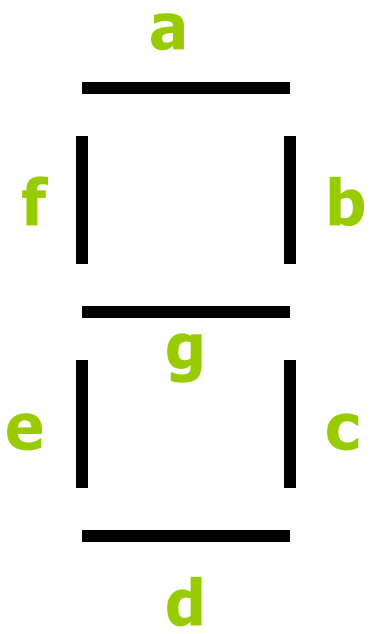
```
package body    程序包名   is
    { 包体说明项 }
end    程序包名;
```

包体说明项可含：

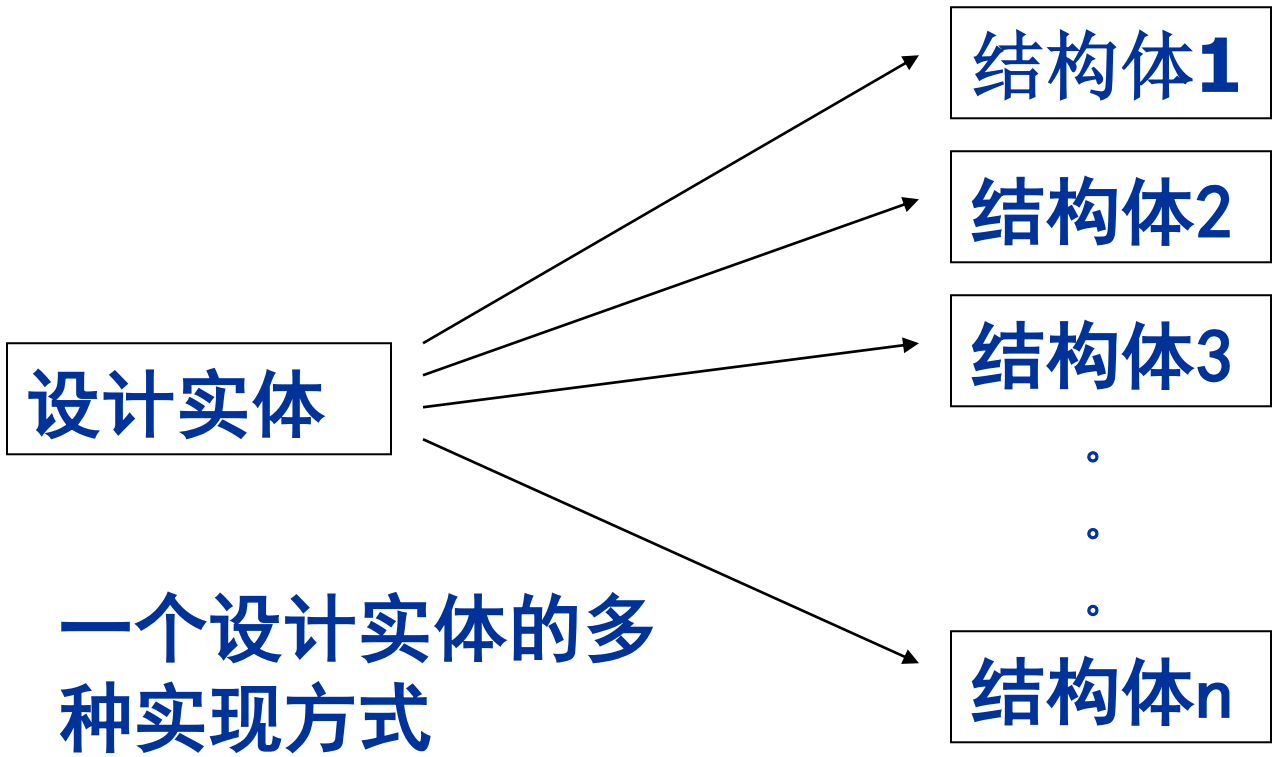
use 语句；子程序说明；**子程序主体**；类型说明；子类型说明；常量说明。

```
package seven is
    subtype segments is bit_vector(0 to 6);
    type bcd is range 0 to 9;
end seven;
library work;
use work.seven.all;
entity decoder is
    port(input: in bcd;
         drive: out segments);
end decoder;
architecture art of decoder is
begin
```

```
with input select
    drive<=B“1111110” when 0,
           B“0110000” when 1,
           B“1101101” when 2,
           B“1111001” when 3,
           B“0110011” when 4,
           B“1011011” when 5,
           B“1011111” when 6,
           B“1110000” when 7,
           B“1111111” when 8,
           B“1111011” when 9,
           B“0000000” when others;
end architecture art;
```



5、Configuration配置



一个设计实体的多种实现方式

配置：从某个实体的多种结构体描述方式中选择特定的一个。

5、Configuration配置

```
configuration  配置名  of  实体名 is
    for   选配结构体名
    end  for ;
end  配置名;
```

例：一个与非门不同实现方式的配置如下：

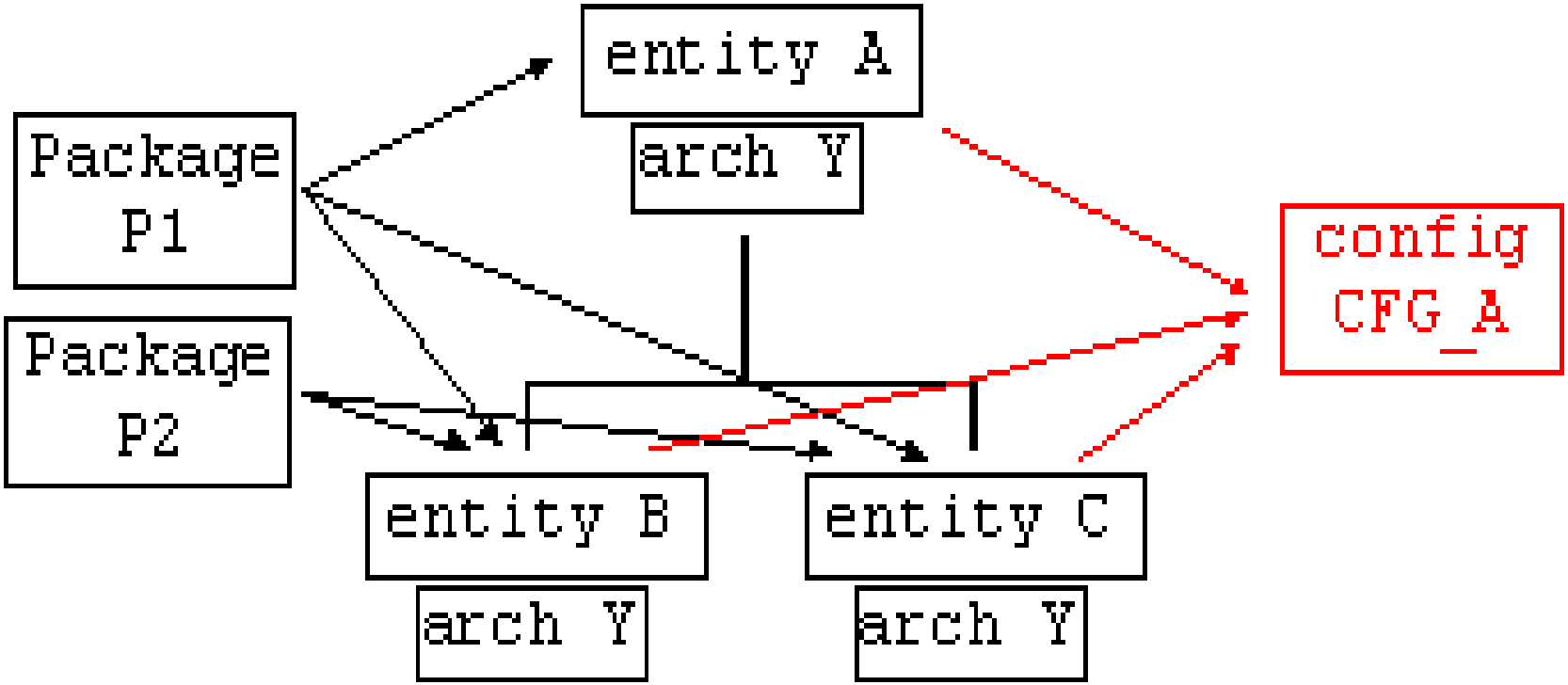
```
library ieee;
use ieee.std_logic_1164.all;
entity nand is
    port(a: in std_logic;
         b: in std_logic;
         c: out std_logic);
end entity nand;
architecture art1 of nand is
begin
     c<=not (a and b);
end architecture art1;
```

```
architecture art2 of nand is
 begin
     c<='1' when (a='0') and (b='0') else
         '1' when (a='0') and (b='1') else
         '1' when (a='1') and (b='0') else
         '0' when (a='1') and (b='1') else
         '0';
 end architecture art2;
```

5、Configuration配置

```
configuration first of nand is
    for art1;
    end for;
end first;
configuration second of nand is
    for art2
    end for;
end  second;
```

VHDL中的结构关系


entity A
arch Y
Package
P1
Package
P2
config
CFG_A
entity B
arch Y
entity C
arch Y

1.3 VHDL语言元素

1、VHDL 数据对象

2、VHDL 数据类型

3、VHDL 操作符

1.3.1 VHDL数据对象

1、Constant（常量）在程序中不可以被赋值

2、Variable（变量）在程序中可以被赋值(用“: =”)，赋值后立即变化为新值。

3、Signal（信号）在程序中可以被赋值(用“<=”)，但不立即更新，当进程挂起后，才开始更新。

4、File（文件）在程序中实现对文件的写入和读出操作。

1.3.1 VHDL数据对象

- **VHDL中的对象使用**

variable

```
x，y：integer；--定义了整数型的变量对象x，y
```

constant

```
Vcc：real；--定义了实数型的常量对象Vcc
```

signal

```
clk，reset：bit；--定义了位类型的信号对象clk，reset
```

变量与信号区别

- 注意

1、variable只能定义在process和subprogram（包括function和procedure）中，不可定以在其外部。

2、signal不能定义在process和subprogram（包括function和procedure）中，只可定以在其外部，即结构体Architecture开始部分。

变量与信号区别

①信号是全局量，可以进行进程之间的通信；而变量是局部量，只能用于进程或子程序（即函数和过程）中；

②对变量的赋值是立即发生的，对信号的赋值须经一段时间延迟才会发生；

③对变量的赋值用“:=”，对信号的赋值用“<=”；

④信号可以比拟为硬件端口之间的连接，而变量则与硬件之间没有对应关系；

1.3.1 VHDL数据对象

- **Signal 对象的常用属性**
  - **delayed[(时延值)]: 使信号产生固定时间的延时并返回**
  - **stable[(时延值)]: 返回boolean, 信号在规定时间内没有变化返回true**
  - **transaction: 返回bit类型，信号每发生一次变化，返回值翻转一次**

例子：**A<=B'delayed(10 ns)；** **--B延时10ns后赋给A；**

if（B'Stable(10 ns)）； **--判断B在10ns中是否发生变化**

1.3.1 VHDL数据对象

- 属性应用

信号的**event**和**last_value**属性经常用来确定信号的边沿

判断clk的上升沿

```
if （ (clk'event)and (clk='1')
and(clk'last_value='0')） then
```

判断clk的下降沿

```
if （ (clk'event)and (clk='0')
and(clk'last_value='1')） then
```

1.3.1 VHDL数据对象

- **初始化赋值**
- SIGNAL control： BIT:=’0’;
- SIGNAL count: INTEGER RANGE 0 TO 100;
- VARIABLE control： BIT :=’0’;

- **对信号、变量赋初值的操作是不可综合的，仅能用在仿真中。**

1.3.1 VHDL数据对象

- 例子1

Legal declarations:

```
signal Z_BUS:bit_vector(3 downto 0);
signal C_BUS:bit_vector(1 to 4);
```

Illegal declarations:

```
signal Z_BUS:bit_vector(0 downto 3);
signal C_BUS:bit_vector(3 to 0);
```

1.3.1 VHDL数据对象

- 例子2

```
signal  A,B,Z  :  bit;
signal  X_INT  :  integer;
```

signal assignment statement

```
Z  <=  A;
```

1.3.1 VHDL数据对象

- 文件（File）

```
USE STD.TEXTIO.ALL;
FILE input_file: TEXT IS IN “in.dat”;
FILE output_file:TEXT IS OUT “out.da
VARIABLE l1,l2:LINE;
VARIABLE test:bit_vector(7 downto 0);
WHILE NOT(ENDFILE(input_file))
    LOOP
        READLINE(input_file,l1);
        READ(l1,test);
        WRITE(l2,test);
        WRITELINE(out_file,l2);
    END LOOP
```

定义文件输入/输出对象

VHDL不能直接处理行变量，需要定义位变量

文件操作按行进行处理，需要定义行变量

读写操作需要将行变量转为位变量

1.3.2 VHDL数据类型

- 标准数据类型
- 用户自定义数据类型
- 数据类型的转换

1）标准数据类型

数据类型	标　识	说　　明
整数	INTEGER	$-(2^{31}-1)$ ～$+(2^{31}-1)$
实数	REAL	-1.0E+38～+1.0E+38
位	BIT	逻辑‘0’或‘1’，单引号标出
位矢量	BIT_VECTOR	BIT的组合，以双引号标出
字符	CHARACTERS	ASCII 字符，单引号标出
布尔量	BOOLEAN	逻辑“真”或“假”，TRUE or FALSE
时间	TIME	时间单位：fs,ps,ns,us,ms,sec,min,hr
错误等级	SEVERITY LEVEL	NOTE,WARNING,ERROR,FAILURE
自然数	NATURAL	整数的子集：0～$2^{31}-1$
正整数	POSTIVE	整数的子集：1～$2^{31}-1$
字符串	STRING	字符矢量，以双引号标出

IEEE库STD_LOGIC_1164

STD_LOGIC类型的数据可以具有九种取值

‘U’：初始值

‘X’：不定态

‘0’：强制0

‘1’：强制1

‘Z’：高阻态

‘W’：弱信号不定态

‘L’：弱信号0

’H’：弱信号1

’_’：不可能情况（可忽略值）

其中，‘X’方便了系统仿真，‘Z’方便了双向总线的描述。

IEEE库STD_LOGIC_1164

STD_LOGIC_VECTOR类型定义如下:

TYPE STD_LOGIC_VECTOR IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC;

```
library ieee;
use ieee.std_logic_1164.all;
entity mux4 is
port (d : in  std_logic_vector (1 downto 0);
      a0: in  std_logic;
       y: out std_logic);
end mux4;
```

基本类型应用

- 例子3

```
signal Z_BUS:bit_vector(3 downto 0);
signal C_BUS:bit_vector(1 to 4);
```

```
Z_BUS <= C_BUS;
```

Is the same as:

```
Z_BUS(3) <= C_BUS(1);
Z_BUS(2) <= C_BUS(2);
Z_BUS(1) <= C_BUS(3);
Z_BUS(0) <= C_BUS(4);
```

Assignment is by position!

基本类型应用

- 例子4

```
signal Z_BUS:bit_vector(3 downto 0);
signal C_BUS:bit_vector(1 to 4);
```

Legal:

```
Z_BUS(3 downto 2) <= "00";
C_BUS(2 to 4) <= Z_BUS(3 downto 1);
```

Illegal:

```
Z_BUS(0 to 1) <= "11";
```

- 要点：赋值语句中的方向应和声明中的方向一样

2）用户自定义数据类型

- 通用格式

  TYPE 类型名 IS 数据类型定义

- 用户可以定义的数据类型

枚举类型enumberated、整数型integer、
实数型real、数组类型array、
纪录类型record、时间类型time、
文件类型file、存取类型access

2）用户自定义数据类型

（1）整数类型（integer）、实数类型（real）

type ＜数据类型名 ＞ is ＜数据类型 ＞ ＜约束范围 ＞;

```
type digit is integer range 0  to 9;
type myreal is real range –1e3 to 1e3;
```

2）用户自定义数据类型

（2）枚举类型（enumeration）

type ＜数据类型名＞ is （＜元素1＞，＜元素2＞，...）；

```
type arith_op is (add,sub,mul,div);
type color is (red,green,blue);
type bit is ('0', '1');
type boolean is (false,true);
type std_ulogic is ('U','X','0','1','Z','W','L','H','-');
```

2）用户自定义数据类型

（3）数组类型（array）、记录类型（record）

```
type big_word is array (0 to 63) of bit; -- 一维数组
type matrix_1 is array (0 to 15,0 to 31) of bit; -- 二维数组
```

```
type instruction is record
    operator : arith_op;
    op1 : integer;
    op2 : integer;
end record;
```

2）用户自定义数据类型

（4）子类型（subtype）

subtype ＜标识符＞ is ＜基类型＞ ＜范围限制＞;

```
subtype my_int is integer range 0  to 65535;
subtype iobus is std_logic_vector(7 downto 0);
```

2）用户自定义数据类型

（5）时间类型（time）

- 格式

type 数据类型名

units 基本单位

单位;

end units

- **type time is range -1E18 to 1E18**

units

us;

ms=1000 us;

sec=1000 ms;

min=60 sec;

end units

注意：引用时间时，有的编译器要求*量*与*单位*之间应有一个空格如：**1 ns**；不能写为**1ns**；

3）数据类型转换

（1）类型标记法。用类型名称来实现关系密切的标量类型之间的转换。

例如：　　VARIABLE　x：INTEGER；

　　VARIABLE　y：REAL；

使用类型标记（即类型名）实现类型转换时，可采用赋值语句：

x :=INTEGER(y)；　　y :=REAL(x)。

3）数据

该函数由
STD_LOGIC_UNSIGNE

该函数由
STD_LOGIC_ARITH

以下函数由
STD_LOGIC_1164
程序包定义

★CONV_INTEGER ( )：将STD_LO
INTEG

★CONV_STD_LOGIC_VECTOR
类型或 SIG 类型转换成STD_LOGIC_VECTOR类型。

★TO_BIT ( )：将STD_LOGIC类型转换成BIT类型。

★TO_BIT_VECTOR( )：将STD_LOGIC_VECTOR类型转换
BIT_VECTOR 类型。

★TO_STD_LOGIC( )：将BIT类型转换成STD_LOGIC类型。

★TO_STD_LOGIC_VECTOR( )：将BIT_VECTOR类型转换成
STD_LOGIC_VECTOR类型。

注意：引用时必须首先 打开库和相应的程序包。

1.3.3 VHDL操作符

分　类	运算符	功　能
二元算术运算符	+	加
	-	减
	*	乘
	/	除
	mod	求模
	rem	求余
	**	乘方
一元算术运算符	+	正号
	-	负号
	abs	绝对值
关系运算符	=	相等
	/=	不相等
	<	小于
	>	大于
	<=	小于等于
	>=	大于等于

分　类	运算符	功　能
二元逻辑运算符	and	与
	Or	或
	Nand	与非
	Nor	或非
	xor	异或
一元逻辑运算符	not	求反
并置运算符	&	连接
赋值运算符	<=	信号赋值
	:=	变量赋值
	=>	结合

1.3.3 VHDL操作符

运算符优先级

- NOT、ABS、**；REM、 MOD、/、*；
- —(负)、+（正）；
- &、—(减)、+（加）；
- ＞= 、 ＜= 、＞、＜、/= 、= ；
- XOR、NOR、NAND、OR、AND 。
- 注意：在编写VHDL程序时，必须保证操作数的数据类型与运算符所要求的数据类型一致。

1.3.3 VHDL操作符

- “<=”赋值符：用于将数据传给信号。
- “:=”赋值符：用于将数据传给变量。

  该赋值符也用于为信号、变量、常量等指定初值。
- “=>” 符号： 在**WHEN**语句中出现，其含义是“**THEN**（则）”。

1.3.3 VHDL操作符

- 连接操作符---使用&

```
signal Z_BUS : bit_vector(3 downto 0);
signal A, B, C, D : bit;
signal BYTE : bit_vector(7 downto 0);
signal A_BUS : bit_vector(3 downto 0);
signal B_BUS : bit_vector(3 downto 0);
```

```
Z_BUS <= A & B & C & D;
```

concatenation operator

```
BYTE <= A_BUS & B_BUS;
```

1.3.3 VHDL操作符

- 集合操作---使用（）

```
signal Z_BUS : bit_vector(3 downto 0);
signal A, B, C, D : bit;
signal BYTE : bit_vector(7 downto 0);
```

```
Z_BUS <= (A, B, C, D);
```

aggregate

equivalent to:

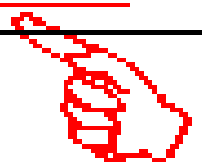
```
Z_BUS(3) <= A;
Z_BUS(2) <= B;
Z_BUS(1) <= C;
Z_BUS(0) <= D;
```

1.3.3 VHDL操作符

- 集合操作---采用序号

```
signal X : bit_vector(3 downto 0);
signal A, B, C, D : bit;
signal BYTE : bit_vector(7 downto 0);
```

```
X <= (3=>'1', 1 downto 0=>'1', 2 => B);
```



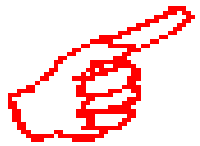
assignment by name

1.3.3 VHDL操作符

- 集合操作--采用others

```
signal X : bit_vector(3 downto 0);
signal A, B, C, D : bit;
signal BYTE : bit_vector(7 downto 0);
```

```
X <= (3=>'1', 1=>'0', others => B);
```



1.3.3 VHDL操作符

- 逻辑操作符

```
signal A_BUS, B_BUS, Z_BUS :
      std_ulogic_vector (3 downto 0);
```

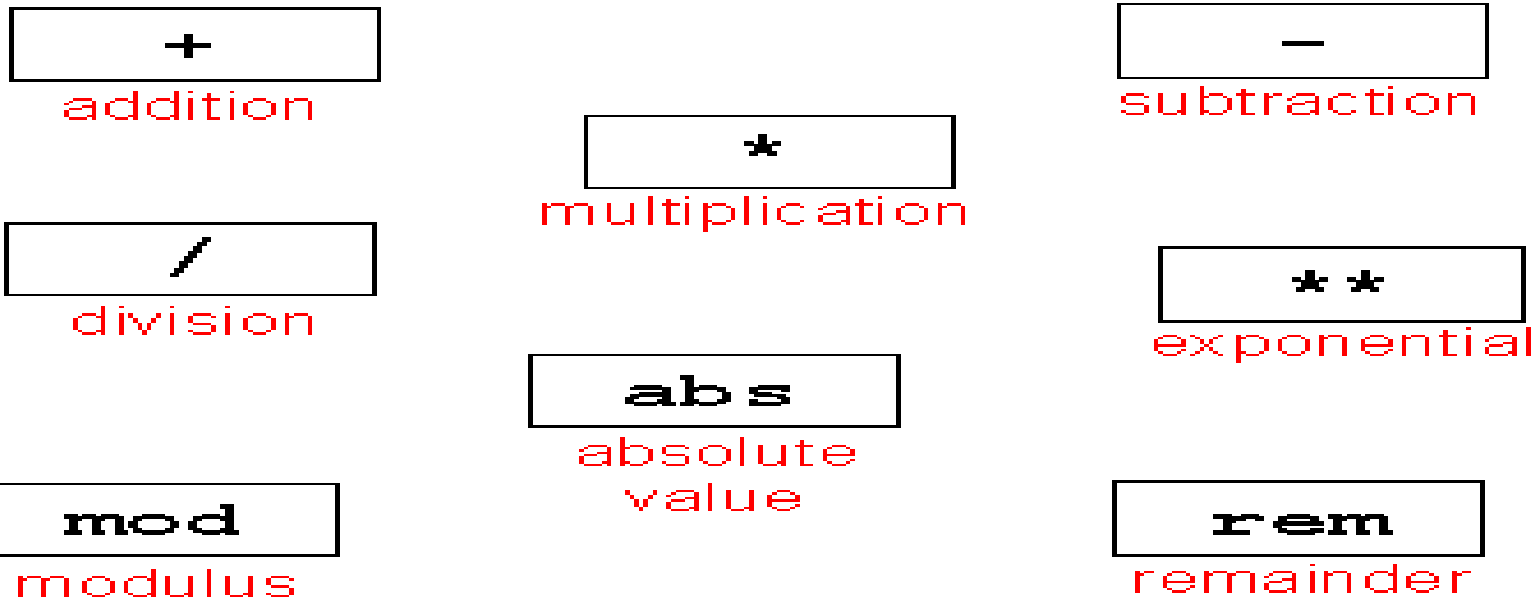
```
Z_BUS <= A_BUS and B_BUS;
```

equivalent to:

```
Z_BUS(3) <= A_BUS(3) and B_BUS(3);
Z_BUS(2) <= A_BUS(2) and B_BUS(2);
Z_BUS(1) <= A_BUS(1) and B_BUS(1);
Z_BUS(0) <= A_BUS(0) and B_BUS(0);
```

1.3.3 VHDL操作符

数学运算符



*注意:*上述运算符应用于 **integer**，**real**，**time** 类型，不能用于**vector(**如果希望用于**vector,**可以使用库**IEEE**的**std_logic_unsigned**包,它对算术运算符进行了扩展**)**

1.4 VHDL基本逻辑语句

1、并行处理（concurrent）

语句的执行与书写顺序无关，并行块内的语句时同时执行。

2、顺序处理（sequential）

语句的执行按书写的先后次序，从前到后顺序执行。

1.4 VHDL基本逻辑语句

- Architecture 中的语句及子模块之间是并行处理的

➢子模块block中的语句是并行处理的

➢子模块process中的语句是顺序处理的

➢子模块subprogram中的function和procedure是顺序处理的

1.4 VHDL基本逻辑语句

Arcthitecture（构造体）格式：（前面已讲）

Arcthitecture 构造体名 **of** 实体名 **is**

[定义语句] 内部信号、常数、元件、数据类型、函数等的定义

begin

[并行处理语句和block、process、function、procedure]

end 构造体名；

1）Block

- 普通Block
- 格式

块名：

```
BLOCK
    [定义语句]
  begin
    [并行处理语句concurrent statement]
 end block  块名
```

Architecture中的Block

- 条件Block
- 格式

块名:

BLOCK [（布尔表达式）]

[定义语句]

begin

[并行处理语句concurrent statement

[信号]**<= *guarded*** [信号,延时] ;

end block 块名

1）Block

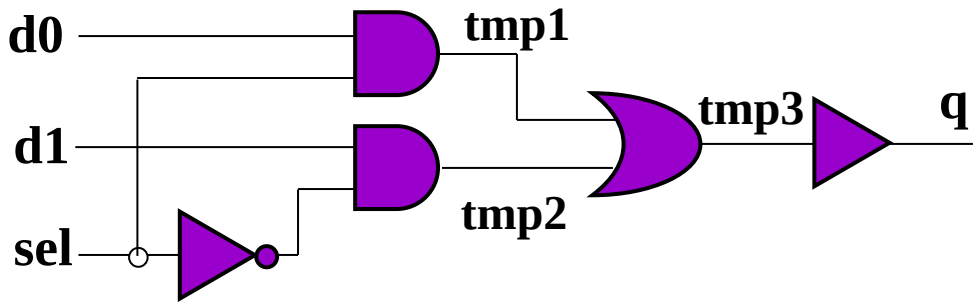
- Block 例子

```
myblock1：
 block（clk='1'）
  signal：qin：bit：='0'；
 begin
  qout<= guarded qin ;
 end block myblock1
```

```
myblock2：
 block
   begin
      qout<=qin;
   end block
myblock2
```

用BLOCK语句

```
ENTITY mux2_
    PORT(d0, d
         q
END mux2_1;
ARCHITECTUI
    SIGNAL tm
BEGIN
cale: BLOCK
    BEGIN
        tmp1<=d0 AND sel;
        tmp2<=d1 AND (not sel);
        tmp3<=tmp1 OR tmp2;
        q<=tmp3;
     END BLOCK cale;
END amux;
```

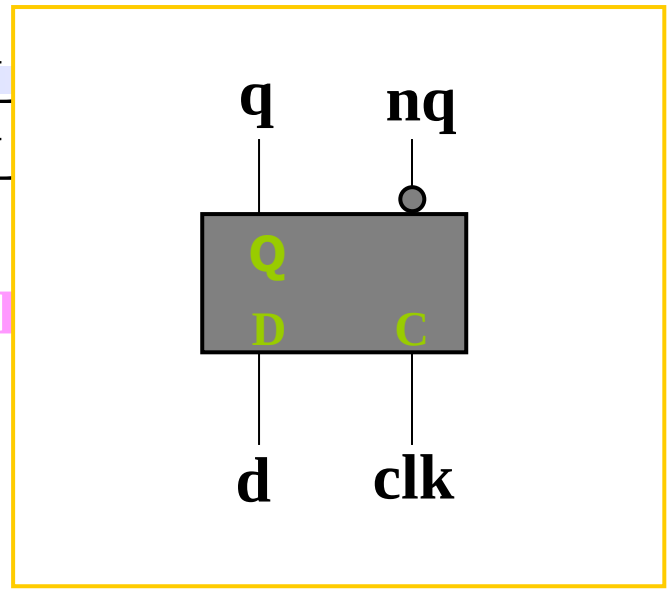


2选1 数据选择器

上述结构体中只有一个 BLOCK块，若电路复杂时可由几个BLOCK块组成。

用带保护条件的BLOCK语句描述一个锁存器的结构。

```
ENTITY latch IS
    PORT(d, clk          : IN    STD_L
         q, qn           : OUT STD_L
END latch;
ARCHITECTURE latch_a OF latch I
BEGIN
    g1:BLOCK(clk='1')
       BEGIN
          q<=guarded d after 5ns;
          qn<=guarded not(d) after 7ns;
       END BLOCK g1;
END latch_a;
```



- 在BLOCK块中的两个信号传送语句都写有前卫关键词guarded，表明只有clk=’1’为真时，这两个语句才能执行。

(*注意*：这里的综合工具不支持 guarded block 语句和 after 短语。)

2）process

- 格式

[进程名：]

process [（触发信号列表）]

Example:

```
PROCESS ( CLK )
BEGIN
    IF CLK'EVENT AND CLK = '1' THEN
          Q1<= D;
              END IF;
          Q<=Q1;

END PROCESS;
```

关于进程（ PROCESS ）的疑问？

1. 何时 PROCESS 被执行？
2. 何时 PROCESS 执行结束？
3. 可以有多个进程出现吗？
4. 多个进程之间如何通信？

何时 PROCESS 被执行？
进程敏感量

```
1.  PROCESS ( CLK )
2.  BEGIN
3.          IF CLK'EVENT AND CLK = '1' THEN
4.              Q1    <= D;
5.          END IF;
6.          Q<=Q1;

7.  END PROCESS;
```

CLK 信号 发生变化时 PROCESS 被执行

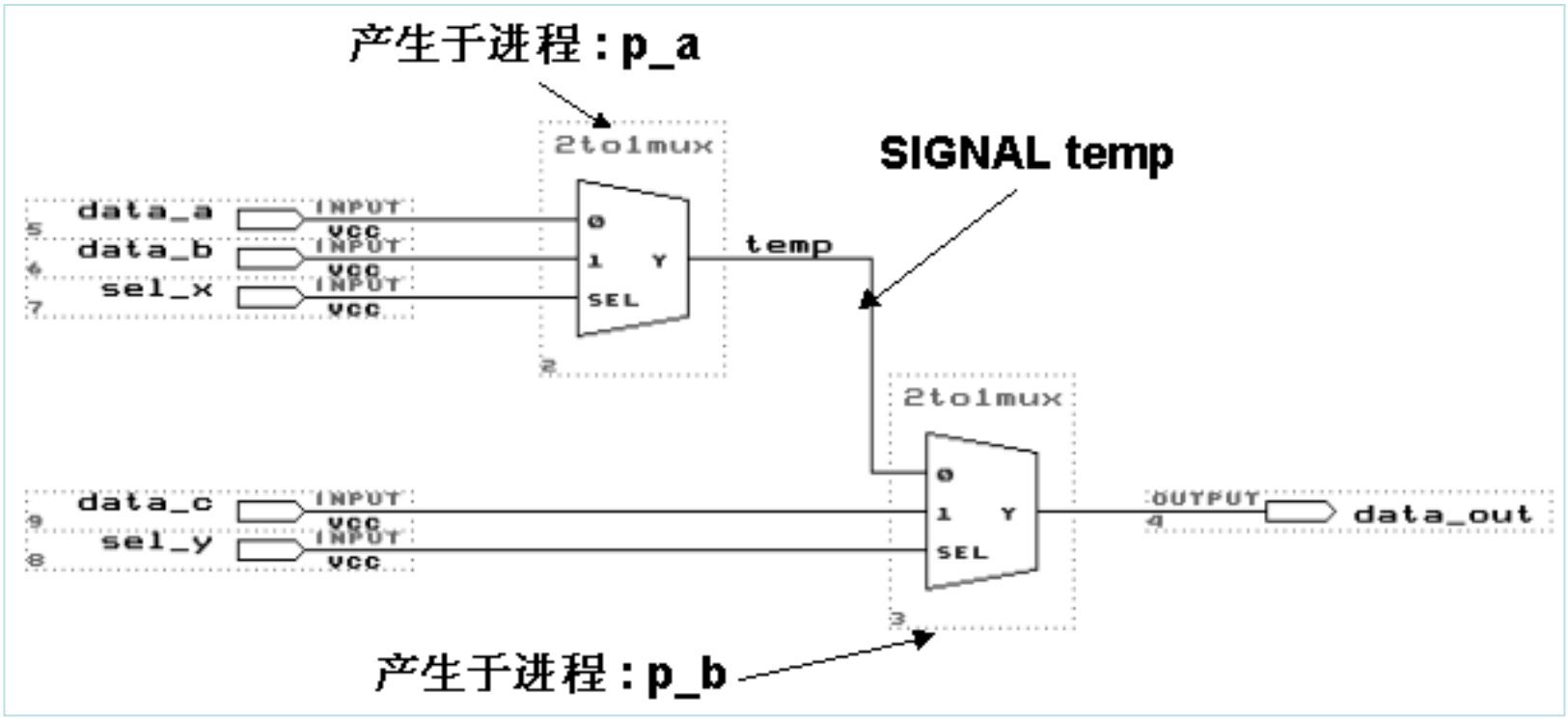
看看此PROCESS的电路？

```
1.  LIBARY IEEE;
2.  USE IEEE.STD_LOGIC_1164.ALL;
3.  ENTITY and2  IS
4.        PORT ( a        :        IN       STD_LOGIC;
5.                  b        :        IN       STD_LOGIC;
6.                  q        :        OUT      STD_LOGIC
7.        );
8.  END ENTITY and2;
9.  ARCHITECTURE bhv OF and2  IS
10.SIGNAL      Q1       :        STD_LOGIC;
11.BEGIN
12.P0:
13.       process(a,b)
14.                 begin
15.                 q <= a and b;
16.                 end process p0;
17.END ARCHITECTURE bhv;
```

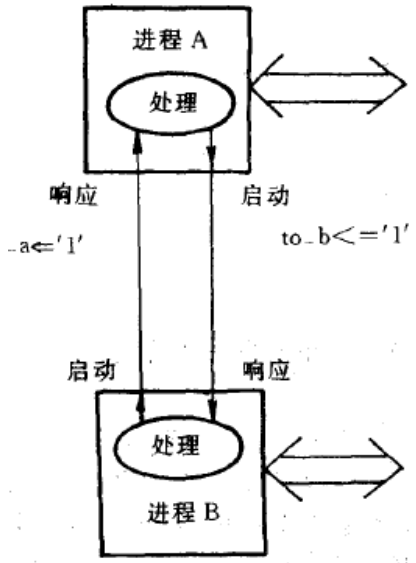
可以有多个进程出现吗？

```
1. ...
2. ARCHITECTURE BEHAV OF    mul       IS

3. SIGNAL temp              :         BIT

4. BEGIN
5. p_a:
6.         PROCESS ( a, b, selx )
7.         BEGIN

8.                 IF ( selx  ='0' ) THEN
9.                         temp      <=        a;
10.                ELSE
11.                        temp      <=        b;
12.                END IF;

13.        END PROCESS p_a;
14. p_b:
15.        PROCESS ( temp, c, sely )
16.        BEGIN

17.                IF ( sely  ='0' ) THEN
18.                        data_out<=          temp;
19.                ELSE
20.                        data_out<=          datac;
21.                END IF;

22.        END PROCESS p_b;

23. END ARCHITECTURE BEHAV;
```

生成电路


产生于进程：p_a
2to1mux
SIGNAL temp
data_a
data_b
sel_x
INPUT
VCC
0
1
Y
SEL
temp
2to1mux
data_c
sel_y
OUTPUT
data_out
产生于进程：p_b

多个进程之间如何通信？




```
ENTITY pros_com IS
PORT(event_a: IN BIT);
END pros_com;
ARCHITECTURE catch_ball OF
        pros_com IS
SIGNAL to_a, to_b: BIT:='0';
BEGIN
A:
PROCESS(event_a, to_a)
BEGIN
  IF (event_a' EVENT AND event_a='1')
       OR
     (to_a' EVENT AND to_a='1')
  THEN
    to_b<='1' AFTER 20 ns,
          '0' AFTER 30 ns;
  END IF;
END PROCESS A;
B:
PROCESS(to_b)
BEGIN
  IF(to_b' EVENT AND to_b='1')
  THEN
    to_a<='1' AFTER 10 ns,
          '0' AFTER 20 ns;
  END IF;
END PROCESS B;
END catch_ball;
```

进程 (PROCESS)的小结

- 敏感信号表所标明的信号是用来*启动进程*的。敏感信号表中的信号无论哪一个发生变化（如由’0’变’1’或由’1’变’0’）都将启动该PROCESS语句。

- PROCESS内部*各语句之间是顺序关系*。在系统仿真时， PROCESS语句是按书写顺序一条一条向下执行的。而不象BLOCK中的语句可以并行执行。

- 若构造体中有多个进程存在，*各进程之间的关系是并行关系*；进程之间的*通信*则一边通过接口由信号传递，一边并行地同步执行。

3）子程序（SUBPROGRAM）

- 过程（PROCEDURE）
- 函数（FUNCTION）

3）子程序（SUBPROGRAM）

- procedure（过程）
- 格式：

```
procedure 过程名（参数1，参数2 .....）is
  [定义语句]
begin
   [顺序执行语句]
  end 过程名
```

3）子程序（SUBPROGRAM）

- Procedure例子

```
procedure max（a，b：in  bit；
                    flag：out boolean）is
begin
   if （a=b）then
      flag<=true；
  end if
end max；
```

3）子程序（SUBPROGRAM）

例1 设计一个从两个整数中求取最大值的过程。

```
PROCEDURE max(a, b:    IN  INTEGER;
                  y:    OUT  INTEGER)  IS
BEGIN
        IF (a<b) THEN
                y<=b;
        ELSE
                y<=a;
        END IF;
END  max;
```

过程的调用：

```
max ( x,  y,  maxout ) ;
```

3）子程序（SUBPROGRAM）

- Function（函数）
- 格式：

```
function 函数名（参数1，参数2 ……）
  [定义语句]
  return 数据类型名 is  [定义语句]
begin
   [顺序执行语句]
  return [返回变量名]
end 函数名
```

3）子程序（SUBPROGRAM）

```
FUNCTION max(a:std_logic_vector;  b:std_logic_vector)
                    RETURN std_logic_vector IS
    VARIABLE  tmp :std_logic_vector(a'range);
BEGIN
    IF (a>b) THEN
            tmp:=a;
    ELSE
            tmp:=b;
    END IF;
    RETURN tmp;
END max;
```

- 函数的参数均为输入参数。
- 函数调用返回一个指定数据类型的值。

3）子程序（SUBPROGRAM）

例2 在结构体中调用求最大值的函数。

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY dpeak IS
     PORT(clk, set   : IN   STD_LOGIC;
               date       : IN   STD_LOGIC_VECTOR(5 downto 0);
               dout       : OUT  STD_LOGIC_VECTOR(5 downto 0));
END dpeak;
```

```
ARCHITECTURE rtl OF dpeak IS
        SIGNAL peak : STD_LOGIC_VECTOR(5 downto 0);
BEGIN
        dout<=peak;
        PROCESS (clk)
        BEGIN
                IF (clk'event and clk='1') THEN
                   IF (set='1') THEN
                      peak<=date;
                   ELSE
                      peak<= max(date,peak);
                   END IF;
                END IF;
        END PROCESS;
END rtl;
```

4）顺序执行语句 sequential statement

- Wait语句
- If 语句
- case语句
- for loop语句
- while 语句

4）顺序执行语句
sequential statement

- **Wait语句**
- 书写格式

  wait；--无限等待

  wait on [信号列表]　--等待信号变化

  wait until [条件]；　--等待条件满足

  wait for [时间值]；　--等待时间到

- 功能

wait语句使系统暂时挂起 (等同于end process)，此时，信号值开始更新。条件满足后，系统将继续。

4）顺序执行语句
sequential statement

- Wait语句例子

```
process（a，b）
    begin
        y<=a and b;
end process
```

等同于

```
process
    begin
        wait on a,b;
        y<=a and b;
end process
```

```
process（a，b）

    begin
        wait on a，b;
        y<=a and b;
end process
```

错误 如果process中已有敏感信号进程中不能使用wait 语句

4）顺序执行语句 sequential statement

- Wait语句例子

◆ 如果process中没有敏感信号列表，其进程中也没有wait 语句，则process中的程序代码循环执行

```
process
    begin
     clk<=not clk after 50 ns ;
end process
```

功能：产生频率为**100 ns**的**clk**信号

4）顺序执行语句 sequential statement

- If 语句格式

```
if 条件 then
   [顺序执行语句]
[else]
  [顺序执行语句]
 end if
```

```
if 条件 then
   [顺序执行语句]
[elsif]
  [顺序执行语句]
[elsif]
 [顺序执行语句]
.....
[else]
end if
```

4）顺序执行语句 sequential statement

- If 语句例子

Always executes if x = "0000"

```
process (A, B, C, X)
begin
    if (X = "0000") then
        Z <= A;
    elsif (X <= "0101") then
        Z <= B;
    else
        Z <= C;
    end if;
end process;
```

4）顺序执行语句 sequential statement

- Case 语句格式

```
Case 表达式  is
    when  条件表达式=> 顺序处理语句
    when  条件表达式=> 顺序处理语句
   .......
   when  others=> 顺序处理语句
end case
```

4）顺序执行语句
sequential statement

- Case 语句例子,条件表达式可以有多种形式

```
process (A, B, C, X)
begin
    case X is
        when 0 to 4 =>
            Z <= B;
        when 5 =>
            Z <= C;
        when 7 | 9 =>
            Z <= A;
        when others =>
            Z <= 0;
    end case;
end process;
```

range

list

others

4）顺序执行语句 sequential statement

- Case 语句例子

```
process (A, B, C, X)
begin
    case X is
        when 0 to 4 =>
            Z <= B;
        when 3 =>
            Z <= C;
        when 7 | 9 =>
            Z <= A;
        when others =>
            Z <= 0;
    end case;
end process;
```

only specify once

cover all possible values

4）顺序执行语句 sequential statement

- For loop 语句格式

For 循环变量 in 范围 loop
[顺序处理语句]
end loop

- For loop 语句例子

```
For i in 1 to 10
loop
  sum=sum+1;
end loop
```

注意：循环变量不需要定义(声明);例子中 i 不需要定义

4）顺序执行语句 sequential statement

- 在loop语句中可以用next来跳出*本次*循环，也可以用exit 来结束整个循环状态

next 格式： next [标号] [when 条件];

exit 格式： exit [标号] [when 条件];

```
For i in 1 to 10
loop
  sum=sum+1;
  next when
sum=100;
end loop
```

```
For i in 1 to 10
loop
  sum=sum+1;
  exit when
sum=100;
end loop
```

4）顺序执行语句 sequential statement

- While 语句格式

```
while 条件 loop
[顺序处理语句]
end loop
```

- While 语句例子

```
While i<10 loop
  sum=sum+1;
  i=i+1;
end loop
```

5）并行处理语句
concurrent statement

1、信号赋值操作

2、带条件的信号赋值语句

3、带选择的信号赋值语句

5）并行处理语句 concurrent statement

- 信号赋值操作
- 符号“<=”进行信号赋值操作的，
- 它可以用在顺序执行语句中，也可以用在并行处理语句中
- *注意*

  1、用在并行处理语句中时，符号<=右边的值是此条语句的敏感信号，**即符号<=右边的值发生变化就会重新激发此条赋值语句。**

  2、用在顺序执行语句中时，没有以上说法。

5）并行处理语句 concurrent statement

- 赋值语句例子

```
Myblock: Block
        begin
          clr<='1' after 10 ns;
          clr<='0' after 20 ns;
        end block myblock
```

```
process
 begin
   clr<='1' after 10 ns;
   clr<='0' after 20 ns;
end block myblock
```

程序执行**10 ns**后**clr** 为**1**，又过**10 ns**后 **0**赋给了**clr**，此时**clr** 以前的值**1**并没有清掉，**clr**将出现不稳定状态

程序执行**10 ns**后**clr** 为**1**，又过**20 ns**后 **clr**的值变为**0**，

5）并行处理语句 concurrent statement

- 条件信号带入语句格式

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4

*注意：最后的**else** 项是必须的; 满足完全性和唯一性*

5）并行处理语句 concurrent statement

- 条件信号带入语句例子

```
Block
begin
   sel<=b & a;
   q<=ain  when sel="00"
      else bin when sel="01"
      else cin  when sel="10"
      else din when sel='11"
       else xx;
end block
```

5）并行处理语句 concurrent statement

- 选择信号带入语句格式

```
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；
```

5）并行处理语句 concurrent statement

- 选择信号带入语句例子

```
Block
begin
 with  sel select
   q<=ain  when sel="00",
         bin when sel="01",
         cin  when sel="10",
         din when sel='11"
         xx; when others;
end block
```

顺序执行语句和并行处理语句

1、顺序执行语句 wait、assert、if -else 、case、for-loop、while语句只能用在process、function 和 procedure 中；

2、并行处理语句（条件信号带入和选择信号带入）只能用在architecture、block中；

元件调用

1. 元件说明：出现在architecture和begin之间

```
component <实体名>
    [generic(<类属表>) ; ]
    port(<端口名>);
end component;
```

2. 元件例化：出现在结构体的语句部分

```
<标号名>：<元件名>
[ generic map(<类属关联表>)]
  port map(<端口关联表>);
```

元件调用

（1）参数映射语句(generic map)

- 假设and2元件是一个带传输延迟时间参数的二输入与门电路：

```
entity and2 is
    generic (delay:time := 10ns);
    port(a, b:in bit;c:out bit);
end and2;
architecture behav of and2 is
Begin
    c <= a and b after delay;
end behav;
```

元件调用

现要调用u1，u2，u3　3个与门，且它们的传输延迟时间要求分别为5ns，10ns，12ns：

```
entity exam is
    port(ina, inb, inc, ind:bit;q:out bit);
end exam;
architecture behav of exam is
   component and2
     generic (delay:time);
     port(a, b:in bit;c:out bit);
   end component;
   signal s1, s2:bit;
begin
    u1:and2 generic map(5ns)  port map(ina, inb, s1);
    u2:and2 generic map(10ns) port map(inc, ind, s2);
    u3:and2 generic map(12ns) port map(s1, s2, q);
end behav;
```

元件调用

（2）端口映射语句(port map)

①位置映射方法

and2端口定义： port(a，b:in bit;c:out bit);

元件例化语句： u1:and2 port map(ina，inb，s1);

即在u1中，ina对应a，inb对应 b，s1对应c。

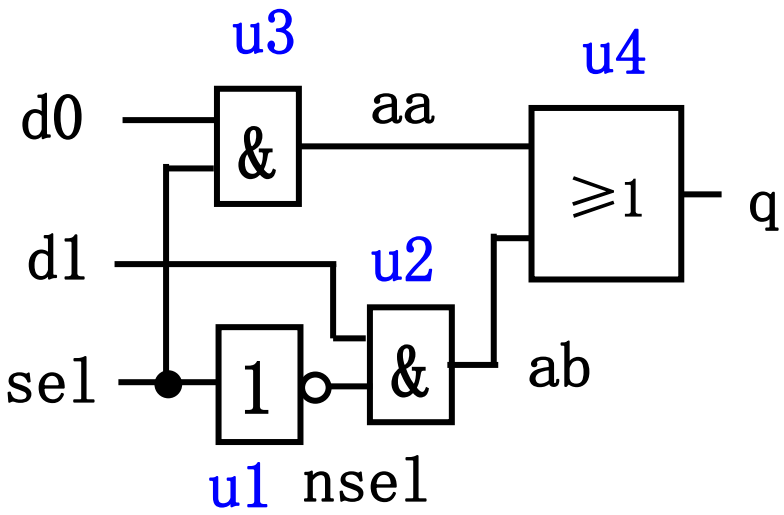
②名称映射方法

u1:and2 port map(a＝＞ina， b＝＞inb，c＝＞s1);

或

u1:and2 port map(b＝＞inb， a＝＞ina， c＝＞s1);

用VHDL描述一个二选一数据选择器。



二选一数据选择器电原理图

```
entity mux2 is
port(d0,d1,sel:in bit; q:out bit);
end mux2;
architecture struct of mux2 is
component and2
port (a,b:in bit; c :out bit);
end component;
component or2
port (a,b:in bit; c :out bit);
end component;
component inv
port (a:in bit; c :out bit);
end component;
signal aa,ab,nsel:bit;
begin
u1:inv port map (sel,nsel);
u2:and2 port map (nsel,d1,ab);
u3:and2 port map (d0,sel,aa);
u4:or2 port map (aa,ab,q);
end struct;
```

一、VHDL语法结构

- 1.1 VHDL概述
- 1.2 VHDL程序结构
- 1.3 VHDL语言元素
- 1.4 VHDL基本逻辑语句
- 1.5 VHDL描述方式

1.5 VHDL描述方式

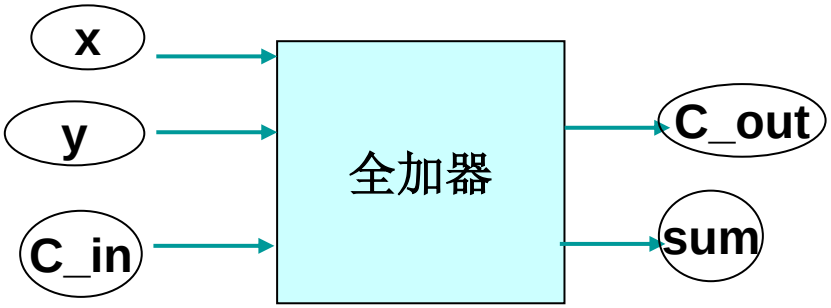
- 行为描述
- 数据流描述
- 结构描述

(1)行为描述方式

- 行为描述方式是指对系统数学模型的抽象描述，属于高级描述，只描述电路的功能，不直接指明或涉及这些行为的硬件结构。
- 行为级描述的设计模型定义了系统的行为，通常由一个或多个进程构成，每一个进程又包含了一系列的顺序语句。

(1)行为描述方式

- 1位全加器设计



输入			输出	
c_in	x	y	c_out	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

全加器VHDL行为描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
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;
ARCHITECTURE behav OF full_adder IS
BEGIN
   PROCESS (x, y, c_in)
VARIABLE  n: INTEGER;
CONSTANT sum_vector: STD_LOGIC_VECTOR (0 TO 3) :=
“0101”;
CONSTANT carry_vector: STD_LOGIC_VECTOR (0 TO 3) :=
“0011”;
```

全加器VHDL行为描述

```
BEGIN
        n := 0;
        IF x = '1' THEN
          n := n+1;
        END IF;
        IF y = '1' THEN
          n:=n+1;
        END IF;
        IF c_in = '1' THEN
          n:=n+1;
        END IF;                              --  (0 TO 3)
        sum <= sum_vector (n);               - - sum_vector初值为“0101”
        c_out <= carry_vector (n);           - - carry_vector初值为“0011”
    END PROCESS;                             -- (0 TO 3)
END behav;
```

(1)行为描述方式

- 采用行为级描述方式不是从设计实体的电路组织和门级实现来完成设计，而是着重设计正确的实体行为、准确的函数模型和精确的输出结果。
- 在行为描述方式的程序中，由于大量采用了算术运算、关系运算、惯性延时、传输延时等难于进行逻辑综合和不能进行逻辑综合的VHDL语句，在一般情况下只能用于行为层次的仿真，而不能进行逻辑综合。
- 随着设计技术的发展，Cadence、Synopsys等EDA工具能够自动完成行为综合，可以把行为级描述转换为数据流级描述方式。

(2)数据流描述方式

- 数据流描述也称为寄存器传输（RTL）描述，采用寄存器与硬件一一对应的直接描述，或者采用寄存器之间的功能描述。
- RTL描述方式建立在并行信号赋值语句描述的基础上，描述数据流的运动路径、运动方向和运动结果。
- RTL描述方式既可描述时序电路，又可描述组合电路。
- RTL描述方式是真正可以进行逻辑综合的描述方式。

(2)数据流描述方式

- 对于全加器，用布尔方程描述其逻辑功能如下：
- s = x XOR y
- sum = s XOR c_in
- c_out = (x AND y) OR(s AND c_in)

(2)数据流描述方式

- 基于全加器布尔方程的数据流描述

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY ADDER1B IS
PORT(A，B，C_IN：IN BIT;
        SUM，C_OUT：OUT BIT);
END ADDER1B ;
ARCHITECTURE ART OF ADDER1B IS
SUM<= A XOR B XOR C_IN;
C_OUT<=(A AND B)OR (A AND C_IN) OR (B AND C_IN);
END ART;
```

数据流描述注意问题

（一） “X”状态的传递：

所谓“X”状态的传递，是指*不确定信号*的传递，它将使逻辑电路产生不确定的结果。“不确定状态”在RTL仿真时是允许出现的，但在逻辑综合后的门级电路仿真中是不允许出现的。

```
PROCESS（sel）
BEGIN
        IF（sel=’1’）THEN
          y<=’0’;
        ELSE
           y<=’1’;
        END  IF;
END  PROCESS;
```

```
PROCESS（sel）
BEGIN
        IF（sel=’0’）THEN
           y<=’1’;
        ELSE
           y<=’0’;
        END  IF;
END  PROCESS;
```

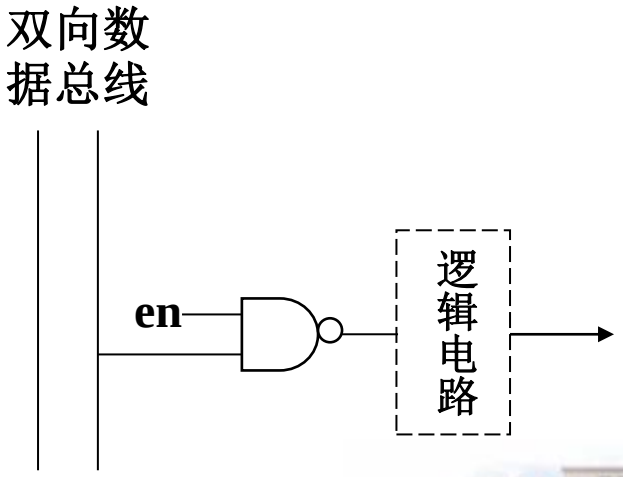
当sel=’X’时，前一个输出的y值为’1’，后一个却变成了’0’。

修改后程序

```
PROCESS（sel）
BEGIN
        IF（sel=’1’）THEN
            y<=’0’;
        ELSIF （sel=’0’）THEN
            y<=’1’;
        ELSE
            y<=’X’;
        END  IF;
END  PROCESS;
```

双向数据总线

在使用双向数据总线时，其信号取值总会出现高阻状态“Z”。当双向总线的信号去驱动逻辑电路时，就有可能出现“X”状态的传递。为了保证逻辑电路的正常工作，高阻状态“Z”应该禁止。如右图中的en信号。



数据流描述注意问题

（二）寄存器RTL描述的限制：

- 禁止在一个进程中存在两个边沿检测的寄存器描述；
- 禁止使用IF语句中的ELSE项；
- 寄存器描述中必须代入信号值。

禁止两个边沿检测

在一个进程中不允许引入两个边沿检测的寄存器进行描述！

```
PROCESS(clk1,clk2)
BEGIN
        IF (clk1 'EVENT AND clk1='1') THEN
                y<=a;
        END IF;
        IF (clk2 'EVENT AND clk2='1') THEN
                z<=b;
        END IF;
END PROCESS;
```

禁止使用ELSE

```
PROCESS(clk)
BEGIN
        IF (clk'EVENT A
                y<=a;
        ELSE                                    -- 禁止使用
                y<=b;
        END IF;
END PROCESS;
```

禁止使用ELSE！

不可能有这样的硬件电路与之对应！

必须代入信号值

```
PROCESS(clk)
VARIABLE  tmp: STD_LOGI
BEGIN
        IF (clk'EVENT AND clk= ' 1') THEN
                tmp:=a;
        END IF;
                y<=tmp;
END PROCESS;
```

必须代入信号值！

(3)结构描述方式

- 结构描述方式是描述该设计单元的硬件结构，即该硬件是如何构成的。
- 在多层次的设计中，常采用结构描述方式在高层次的设计模块中调用低层次的设计模块，或者*直接*用门电路设计单元构造一个复杂的逻辑电路。
- 编写结构描述程序可*模仿逻辑图的绘制方法*。
- 结构描述方式通常*采用元件例化语句和生成语句*编写程序。

结构描述程序设计步骤

1） 绘制框图。先确定当前设计单元中需要用到的子模块的种类和个数。对每个子模块用一个图符（称为实例元件）来代表，只标出其编号、功能和接口特征(端口及信号流向)，而不关心其内部细节。

2） 元件说明。每种子模块分别用一个元件声明语句来说明。

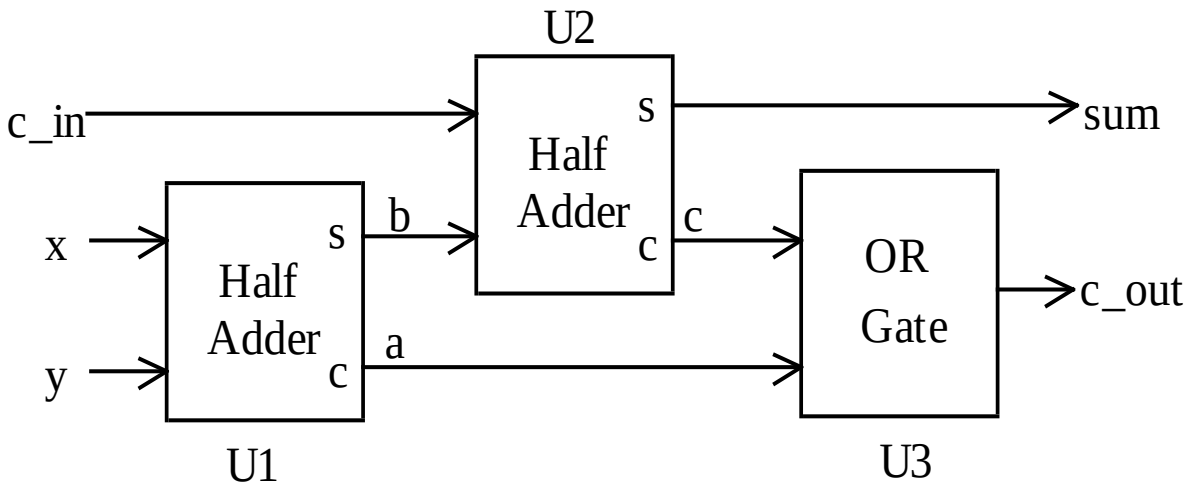
3）信号说明。为各实例元件之间的每条连接线都起一个单独的名字，称为信号名。利用SIGNAL语句对这些信号分别予以说明。

4） 元件例化。根据实例元件的端口与模板元件的端口之间的映射原理，对每个实例元件均可写出一个元件例化语句。

5） 添加必要的框架，完成整个设计文件。

1位全加器的结构描述

- 全加器由两个半加器和一个或门组成的 。



半加器设计

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY half_adder IS
  GENERIC（tpd：TIME:=10 ns）;
  PORT（in1, in2: IN STD_LOGIC;
        sum, carry: OUT STD_LOGIC);
END half_adder;
ARCHITECTURE behavioral OF half_adder IS
BEGIN
PROSESS (in1, in2)
BEGIN
   sum <= in1 XOR in2;
   carry <= in1 AND in2;
END PROCESS;
END behavioral;                          - -半加器设计完毕
```

或门设计

```
LIBRARY IEEE；
USE IEEE.STD_LOGIC_1164.ALL；
ENTITY or_gate IS
 GENERIC（tpd：TIME:=10 ns）；
 PORT（in1, in2: IN STD_LOGIC;
       out1: OUT STD_LOGIC);
END or_gate;
ARCHITECTURE structural OF or_gate IS
BEGIN
       out1 <= in1 OR in2 AFTER tpd;
END structural;                          - - 或门设计完毕
```

1位全加器的结构描述程序

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
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;
ARCHITECTURE structural OF full_adder IS
  COMPONENT half_adder
    PORT（in1, in2: IN STD_LOGIC;
      sum, carry: OUT STD_LOGIC);
  END COMPONENT;
  COMPONENT or_gate
    PORT（in1, in2: IN STD_LOGIC;
      out1: OUT STD_LOGIC);
  END COMPONENT;
```

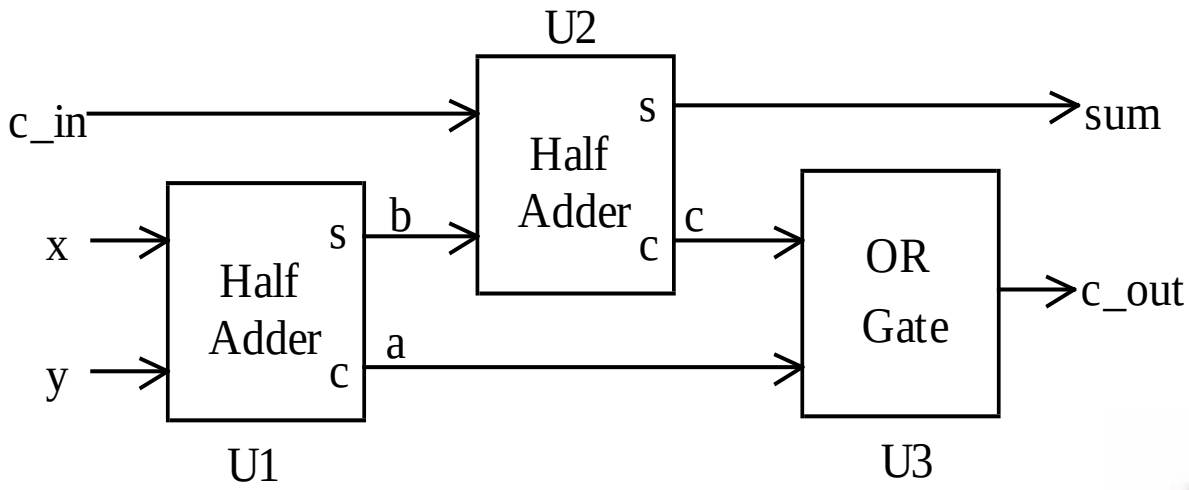
半加器声明

或门声明

模块组合

定义信号作为子模块之间的数据传递参数

```
SIGNAL a, b, c:STD_LOGIC;
BEGIN
  u1: half_adder PORT MAP (x, y, b, a);
  u2: half_adder PORT MAP (c_in, b, sum, c);
  u3: or_gate PORT MAP (c, a, c_out);
END structural;
```



(3)结构描述总结

- 对于一个复杂的电子系统，可以将其分解为若干个子系统，每个子系统再分解成模块，形成多层次设计。
- 在多层次设计中，每个层次都可以作为一个元件，再构成一个模块或系统，可以先分别仿真每个元件，然后再整体调试。
- 结构描述不仅是一种设计方法，而且是一种设计思想，是大型电子系统高层次设计的重要手段。

三种描述方式比较

描述方式	优点	缺点	适用场合
行为描述	电路特性清楚	综合效率低	大型复杂电路设计
数据流描述	布尔函数定义清楚	不易描述复杂电路，修改困难	少量门数模块设计
结构描述	连接关系清晰，电路模块化清晰	电路繁琐、复杂，不易理解	电路层次化设计

内容纲要

- VHDL语法结构
- 组合逻辑电路设计
- 时序逻辑电路设计
- 有限状态机设计

逻辑电路设计

逻辑电路

- 组合逻辑电路：现时的输出仅取决于现时的输入
- 时序逻辑电路：除与现时输入有关外还与电路的原状态有关

二、 组合逻辑电路设计

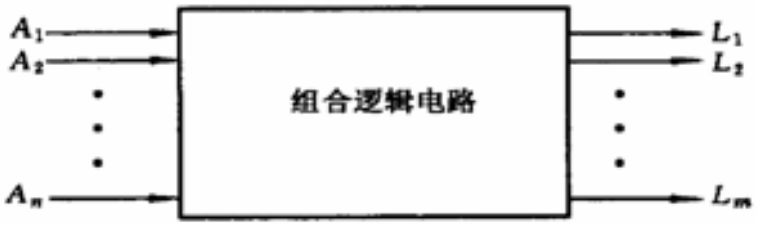
- 组合逻辑电路设计方法
- 编码器设计
- 多路选择器设计
- 比较器设计
- 数码转化电路设计

组合逻辑电路

- 任意时刻，输出状态只决定同一时刻各输入状态的组合，而与先前的状态无关的逻辑电路。

$$L_i = f(A_1, A_2, \cdots, A_n) \quad (i = 1, 2, \cdots, m)$$

式中 $A_1, A_2, \cdots, A_n$ 为输入变量。



- 特点

➢ 任意时刻的输出仅决定于该时刻的输入，输入和输出之间没有反馈延迟通路。

➢ 电路结构上不含记忆元件。

2.1 组合逻辑电路设计方法

- 基于数字电路的设计方法

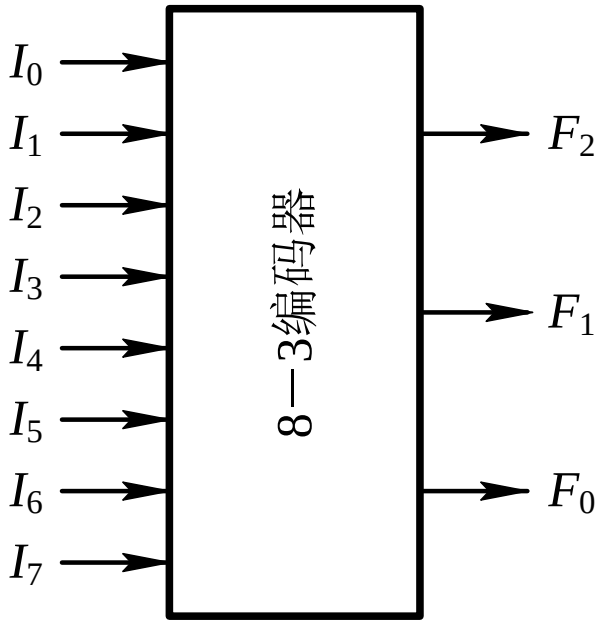
① 列出真值表
② 选择器件类型
③ 写逻辑表达式
④ 画出逻辑电路

- 基于硬件描述语言的设计方法

① 查表法
② 根据真值表，采用硬件描述语言实现功能
③ 利用EDA工具进行仿真验证
④ FPGA验证

2.2 编码器设计

n位二进制代码对$M=2^n$个一般信号进行编码的电路



输入								输出		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	F_2	F_1	F_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

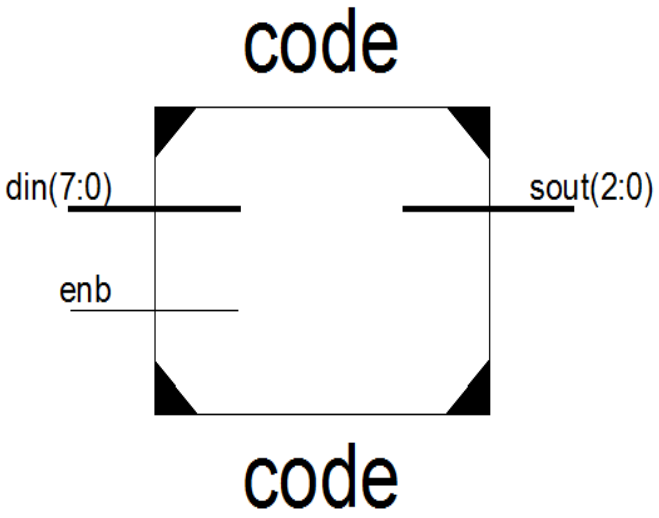
(1)VHDL描述

- 端口定义
- PORT（ enb：IN STD_LOGIC ；

  din：IN STD_LOGIC_VECTOR(7 DOWNTO 0)；

  sout：OUT STD_LOGIC_VECTOR(2 DOWNTO 0)）；

(2)编码器设计




```
------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity code is
      port( enb:in std_logic;
            sout:out std_logic_vector(2 downto 0);
            din:in std_logic_vector(7 downto 0)
          );
end code;

architecture Behavioral of code is
signal   sel :std_logic_vector(8 downto 0):="000000000";
begin

            sel(0)<=enb;
            sel(1)<=din(0);
            sel(2)<=din(1);
            sel(3)<=din(2);
            sel(4)<=din(3);
            sel(5)<=din(4);
            sel(6)<=din(5);
            sel(7)<=din(6);
            sel(8)<=din(7);

   with sel select
         sout<="000" when "000000001",
               "001" when "000000101",
               "010" when "000001001",
               "011" when "000010001",
               "100" when "000100001",
               "101" when "001000001",
               "110" when "010000001",
               "111" when "100000001",
               "000" when others;
end Behavioral;
```

(3)逻辑表达式化简

- 逻辑代数或卡诺图对逻辑表达式进行化简。
- 简化后布尔表达式为

  F(2)=I7+I6+I5+I4;

  F(1)=I7+I6+I3+I2;

  F(0)=I7+I5+I3+I1;

- 简化后VHDL结构体描述

F(2)<=(I(7) OR I(6) OR I(5) OR I(

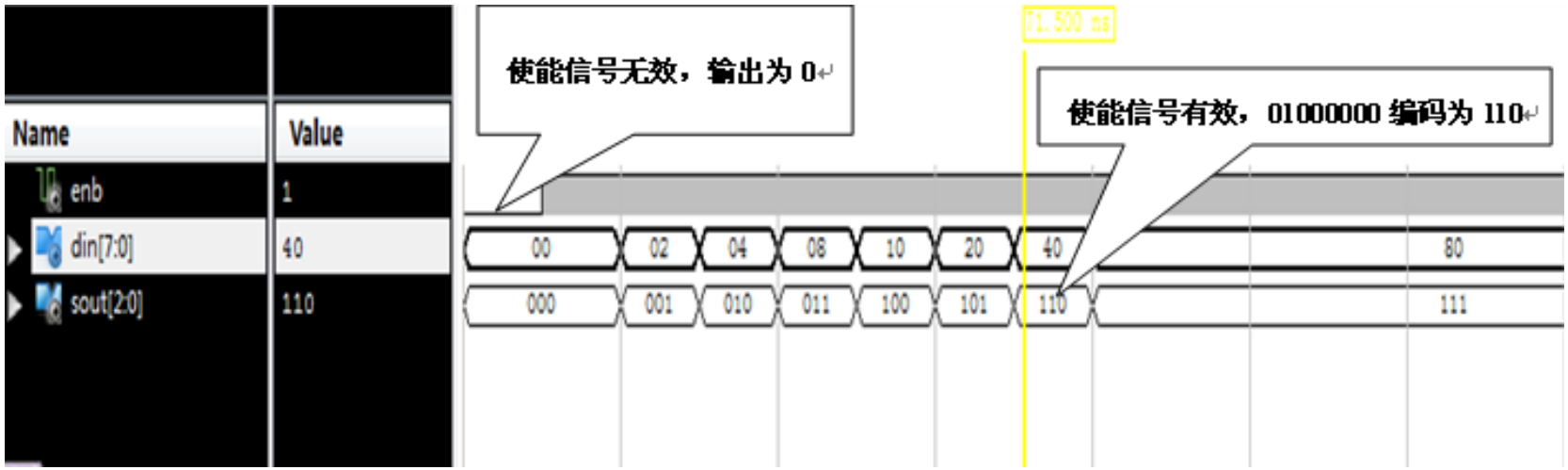
F(1)<=(I(7) OR I(6) OR I(3) OR I(

F(0)<=(I(7) OR I(5) OR I(3) OR I(

输入								输出		
I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	F_2	F_1	F_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

(4)编码器仿真结果

- 当使能信号无效时，编码器输出恒为0；当使能信号有效时，编码器能够对七位输入信号din进行正确的编码。

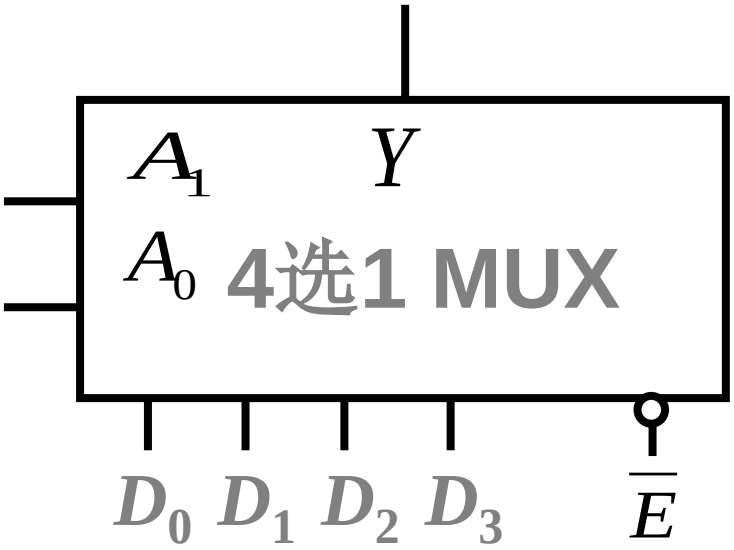


2.3 译码器设计

- N个二进制选择线，则最多可译码成2^n个数据。
- 3-8译码器VHDL 设计?

(1) 4选1数据选择器

- 从一组数据中选择一路信号进行传输的电路，称为数据选择器。



功能表

输入			输出
A_1	A_0	E	Y
×	×	1	0
0	0	0	D_0
0	1	0	D_1
1	0	0	D_2
1	1	0	D_3

(2)VHDL描述

- 端口定义
- PORT（A0：IN STD_LOGIC；
  A1： IN STD_LOGIC；
  Data：IN STD_LOGIC_VECTOR(3 DOWNTO 0);
  EN: IN STD_LOGIC；
  Y ： OUT STD_LOGIC ）；

(2)VHDL描述

- 结构体部分：WHEN…ELSE语句
- Y <= Data(0) WHEN A=“00” ELSE
  Data(1) WHEN A=“01” ELSE
  Data(2) WHEN A=“10” ELSE
  Data(3) WHEN A=“11” ELSE
  ‘0’;

(3)VHDL描述

- 结构体部分：CASE…WHEN语句
- CASE A IS

  WHEN “00” => Y <= Data(0)；

  WHEN “01” => Y <= Data(1)；

  WHEN “10” => Y <= Data(2)；

  WHEN “11” => Y <= Data(3)；

  WHEN OTHER Y <= ‘0’；

2.4 比较器设计

- 数字比较器的设计，通常依据两组二进制数码的数值来比较，即a>b，a=b或是a<b，这三种情况仅有一种值为真。

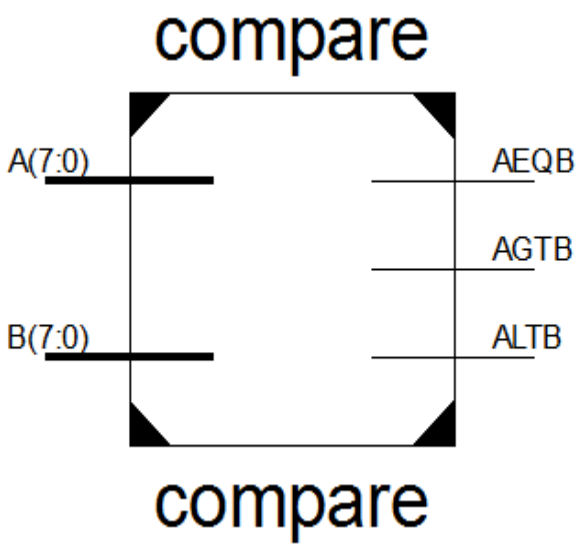
输 入	输 出		
A(7:0) B(7:0)	AGTB	AEQB	ALTB
A>B	1	0	0
A=B	0	1	0
A<B	0	0	1

(1)比较器VHDL程序

端口定义

```
PORT(A,B: IN STD_LOGIC_VECTOR (7DOWNTO 0);
     AGTB,AEQB,ALTB:OUT STD_LOGIC);
```

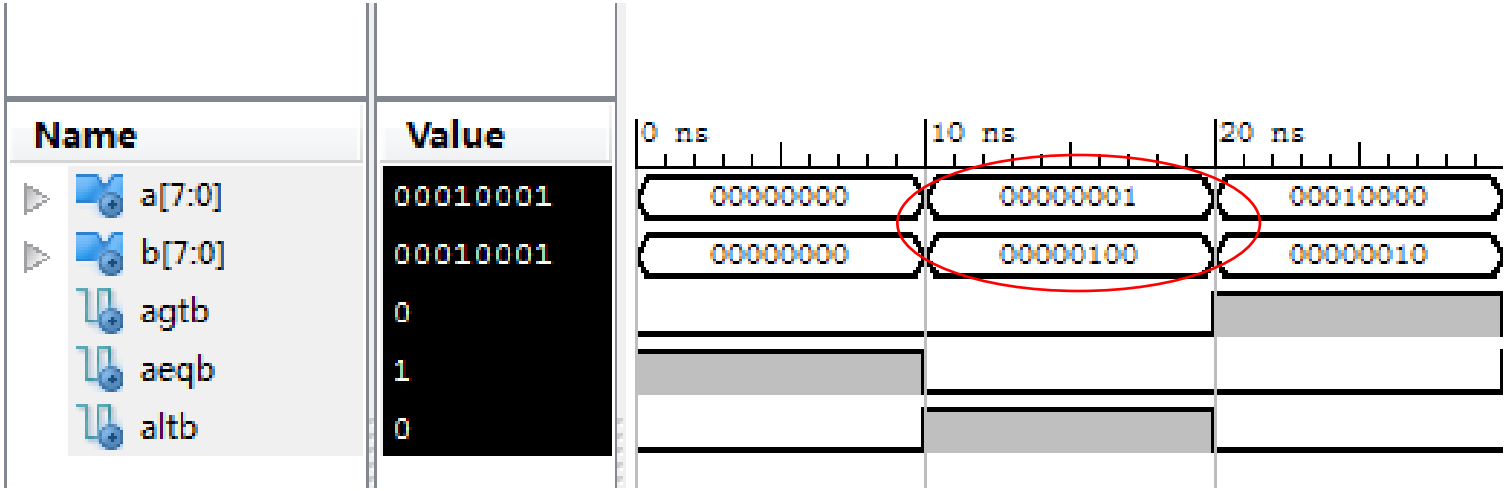
(2)比较器VHDL程序




```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity compare is
        port(A,B: in std_logic_vector(7 downto 0);
             AGTB,AEQB,ALTB:out std_logic);
end compare;

architecture Behavioral of compare is
begin
   process(A,B)
   begin
        if(A=B)then
           AGTB<='0';
           AEQB<='1';
           ALTB<='0';
        elsif(A>B)then
           AGTB<='1';
           AEQB<='0';
           ALTB<='0';
        elsif(A<B)then
           AGTB<='0';
           AEQB<='0';
           ALTB<='1';
        end if;
   end process;
end Behavioral;
```

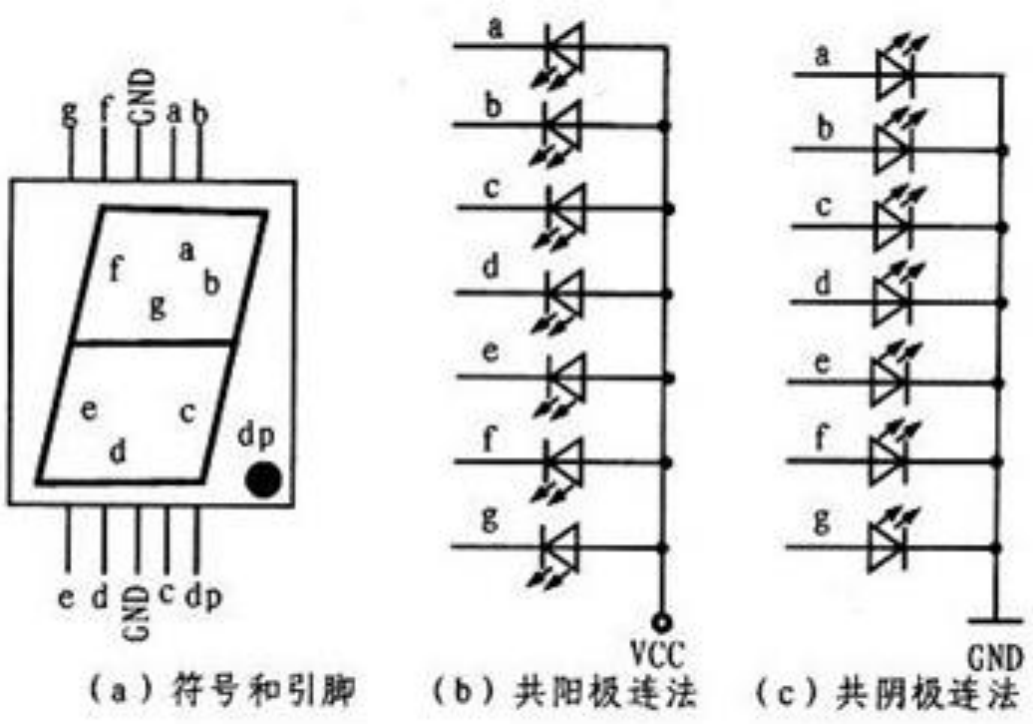
(3)比较器仿真结果

在时钟信号的上升沿，对输入数据a，b进行比较，输出相应结果。例如：在a=00000000；b=00000000时，aeqb=1。通过对仿真结果可以分析出，比较器的功能实现是正确的。



2.5 数据转换器设计

- 第一种是二进制转为十进制，第二种是BCD码转换成七段显示器码。



（a）符号和引脚　（b）共阳极连法　（c）共阴极连法

2.5 数据转换器设计

- 4位二进制编码转为十进制
- 当
- 当
- 输
- 输

二进制	十进制（十位）	十进制（个位）
0000	0	0
0001	0	1
0010	0	2
0011	0	3
0100	0	4
0101	0	5
0110	0	6
0111	0	7
1000	0	8
1001	0	9
1010	1	0
1011	1	1
1100	1	2
1101	1	3
1110	1	4
1111	1	5

是0，个位数=输入。

字是1，个位数=输入

ITO 0),因为有减法运

（BCD1）、个位数（

定义为ST

WNTO 0)。

(1)数码转换VHDL设计

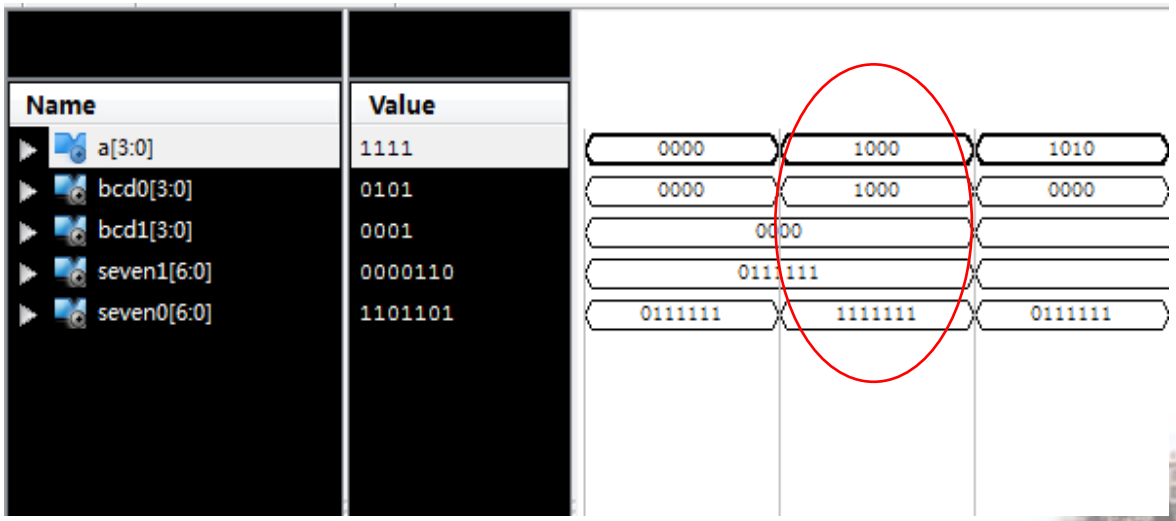
- 端口定义
- PORT(A: IN UNSIGNED (3 DOWNTO 0);
  BCD0，BCD1：OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
  SEVEN0，SEVEN1：OUT STD_LOGIC_VECTOR(6 DOWNTO 0);

(2)数据转换器程序设计

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity dataconversion is
        port( A:in UNSIGNED(3 downto 0);
              BCD0,BCD1: out STD_LOGIC_VECTOR(3 DOWNTO 0);
              SEVEN1,SEVEN0:OUT STD_LOGIC_VECTOR(6 DOWNTO 0));
end dataconversion;

architecture Behavioral of dataconversion is
signal XC:STD_LOGIC_VECTOR(3 DOWNTO 0);
begin
   Process(A)
      begin
         if (A<10)then
            BCD1<="0000";
            BCD0<=STD_LOGIC_VECTOR(A);
            SEVEN1<="0111111";
            XC<=STD_LOGIC_VECTOR(A);
         else
            BCD1<="0001";
            BCD0<=A-10;
            SEVEN1<="0000110";
            XC<=STD_LOGIC_VECTOR(A)-10;
         END if;
   end Process;
   with XC select
        SEVEN0<=  "0111111" when "0000",
                  "0000110" when "0001",
                  "1011011" when "0010",
                  "1001111" when "0011",
                  "1100110" when "0100",
                  "1101101" when "0101",
                  "1111101" when "0110",
                  "0000111" when "0111",
                  "1111111" when "1000",
                  "1101111" when "1001",
                  "0000000" when others;
end Behavioral;
```

(3)数据转换器仿真结果

- 当输入数据为1000时，相应的bcd码为：00001000；故bcd0输出1000，bcd1输出0000。相应的七段译码管高位为0，所以seven1输出0111111，低位为8，所以输出为1111111。从仿真结果可以看出功能是正确的。



三、时序逻辑电路设计

- 时序逻辑电路—任何一个时刻的输出状态不仅取决于当时的输入信号，还与电路的原状态有关。

三、时序逻辑电路设计

时序电路的特点:

（1）含有具有记忆元件（最常用的是触发器）。

（2）具有反馈通道。

输出方程

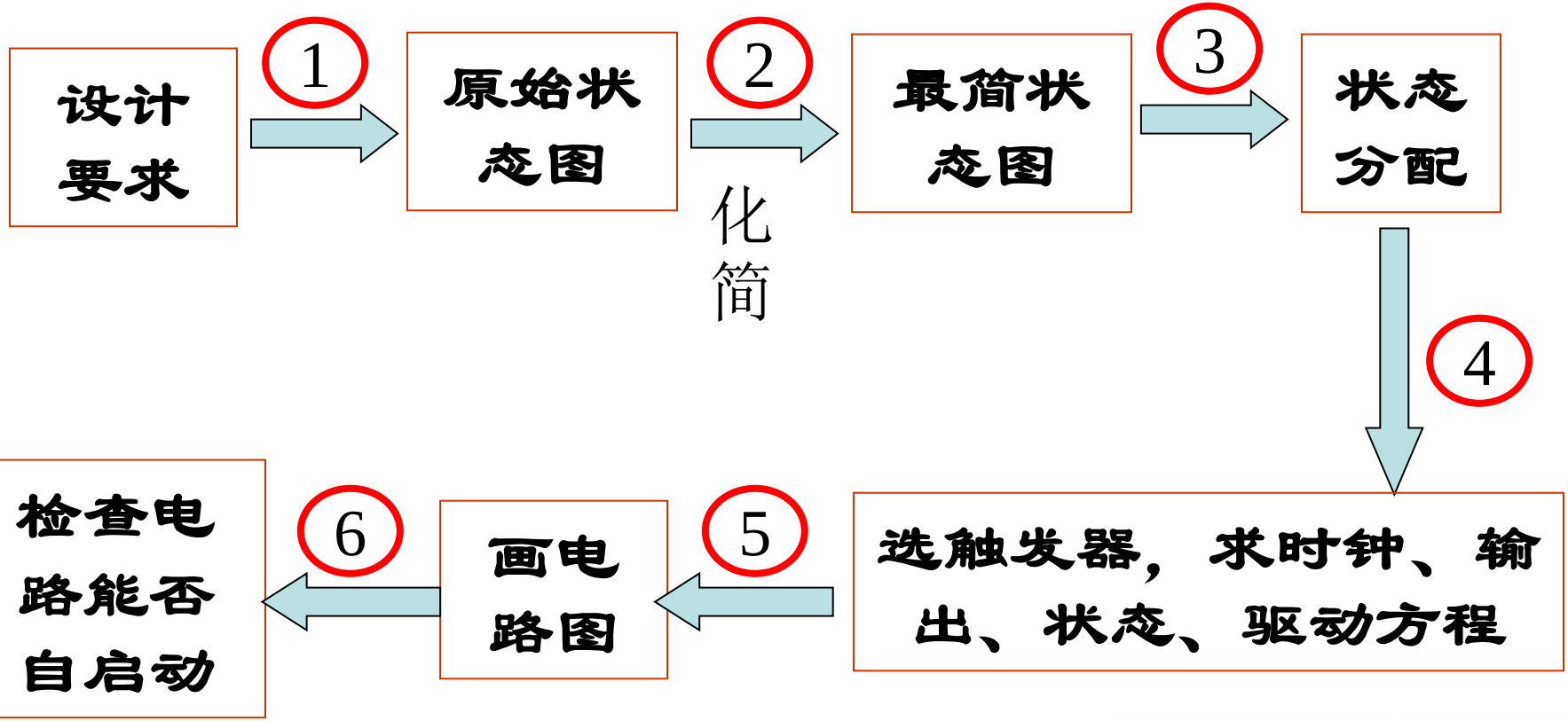
$$\begin{cases} Y_i = F_i(X_1, X_2, \cdots, X_p; Q_1^n, Q_2^n, \cdots, Q_q^n) & i = 1, 2, \cdots, m \\ W_j = G_j(X_1, X_2, \cdots, X_p; Q_1^n, Q_2^n, \cdots, Q_q^n) & j = 1, 2, \cdots, r \\ Q_k^{n+1} = H_k(W_1, W_2, \cdots, W_r; Q_1^n, Q_2^n, \cdots, Q_q^n) & k = 1, 2, \cdots, t \end{cases}$$

状态方程

激励方程

3.1 时序逻辑电路设计方法

- 基于数字电路的同步时序电路设计方法

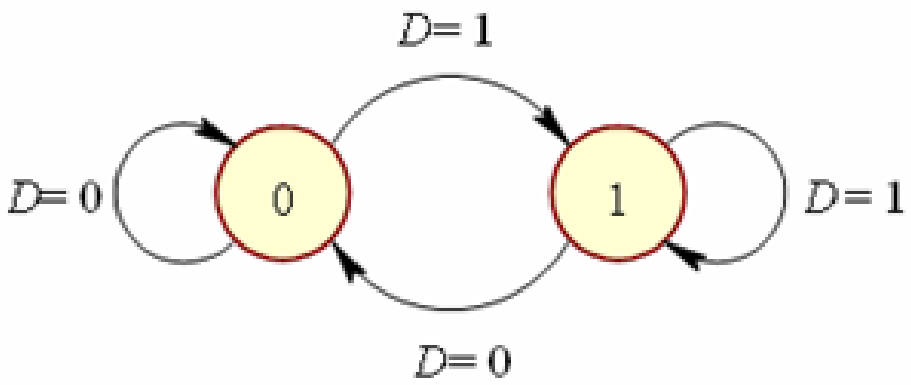


3.1 时序逻辑电路设计方法

- 基于VHDL的时序逻辑电路设计方法
- ① 确定状态数量及状态转换图
- ②按照时钟、输出和驱动状态方程编写VHDL代码
- ③利用EDA工具进行功能仿真验证
- ④FPGA验证

3.2 D触发器

- 根据D触发器状态图和状态表，确定D触发器具有2个状态，在时钟上升沿有效时，输出状态方程为 $Q^{n+1}=D$



D	Q^{n+1}
0	0
1	1

3.2 D触发器

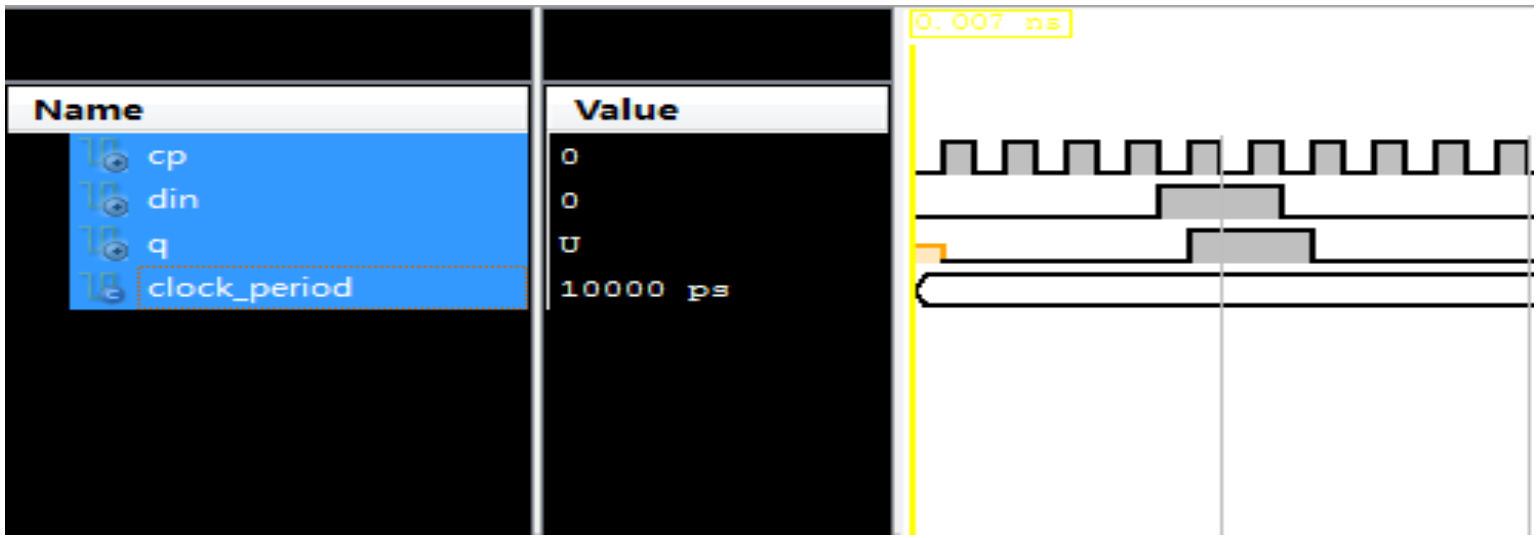
- 端口定义
- PORT（CP： IN STD_LOGIC;
  D ： IN STD_LOGIC;
  Q ： OUT STD_LOGIC）；

(1)D触发器VHDL设计

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity d is
          port(cp,din:in std_logic;
               q:out std_logic);
end d;

architecture Behavioral of d is

begin
          process(cp)
             begin
                if(cp'event and cp='1') then
                   q<=din;
                end if;
          end process;

end Behavioral;
```

(2)D触发器仿真结果

- 输出信号Q在时钟上升沿输出D信号的值，在其余时间，保持原来的值。



3.3 分频电路设计

- 基于FPGA的两种分频电路设计方法

➢ 使用FPGA芯片内部提供的锁相环电路，如：ALTERA提供的PLL（Phase Locked Loop），Xilinx提供的DLL（Delay Locked Loop）；

➢ 使用硬件描述语言，如VHDL、Verilog HDL等。

3.3 分频电路设计

- 计数器是实现分频器电路的基础。
- 计数器有普通计数器和约翰逊计数器两种。

分频电路类型

- 偶数分频器
- 奇数分频器
- 半整数分频器
- 小数分频器
- 分数分频器

(1)偶数分频器

- 占空比为50%的偶数N分频

➢ ①当计数器记到N/2-1时，将输出电平进行一次翻转，同时给计数器一个输出复位信号，如此循环下去；

➢ ②当计数器的输出为0 到N/2-1时，时钟输出为0或1，计数器输出为N/2到N-1时，时钟输出为1或0，当计数器计数到N-1时，复位计数器，如此循环下去。

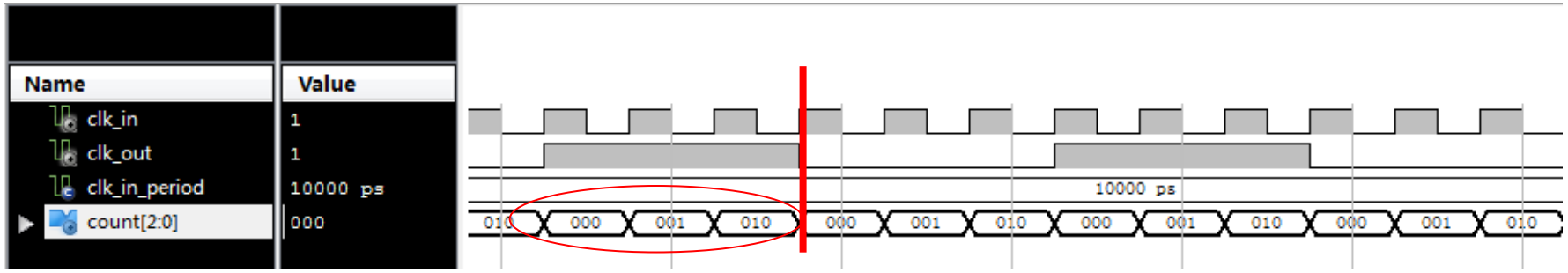
方案一

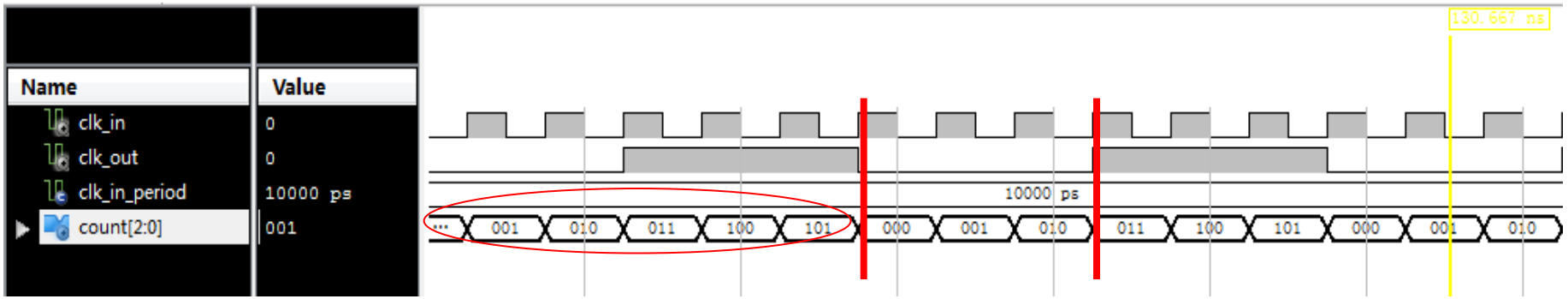
```
architecture a of div is
signal clk: std_logic:='0';
signal count : std_logic_vector(2 downto 0):="000";
begin
    process(clk_in)
        begin
            if(clk_in'event and clk_in='1')then
                if count /=2 then
                    count <=count    +1;
                else
                    clk<= not clk;
                    count<="000";
                end if;
            end if;
        end process;
        clk_out<=clk;
end a;
```

方案二

```
architecture b of   div is
signal countq: std_logic_vector(2 downto 0):="000";
begin
    process(clk_in)
            begin
            if(clk_in'event and clk_in='1')then
                if countq<5 then
                    countq <=countq+1;
                else
                    countq <="000";
                end if;
            end if;
        end process;
process(countq)
        begin
                if countq<3 then
                    clk_out<='0';
                else clk_out<='1';
                end if;
    end process;
end b;
```

6分频器仿真结果


Name
Value
clk_in
1
clk_out
1
clk_in_period
10000 ps
count[2:0]
000
10000 ps
010
000
001
010
000
001
010
000
001
010
000
001
010


Name
Value
clk_in
0
clk_out
0
clk_in_period
10000 ps
count[2:0]
001
130.667 ns
10000 ps
001
010
011
100
101
000
001
010
011
100
101
000
001
010

(2)非50%占空比的奇数分频

```
entity divo is
   port(clk_in :in std_logic;
        clk_out: out std_logic);
end divo;
architecture Behavioral of divo is
signal count :std_logic_vector(2 downto 0):="000";
begin
    process(clk_in)
        begin
            if (clk_in'event and clk_in='1') then
                if count<4 then
                    count<=count+1;
                else
                    count<="000";
                end if;
            end if;
    end process;
    process(count)
        begin
            if(count < 3) then
                clk_out<='0';
            else clk_out<='1';
            end if;
    end process;
```

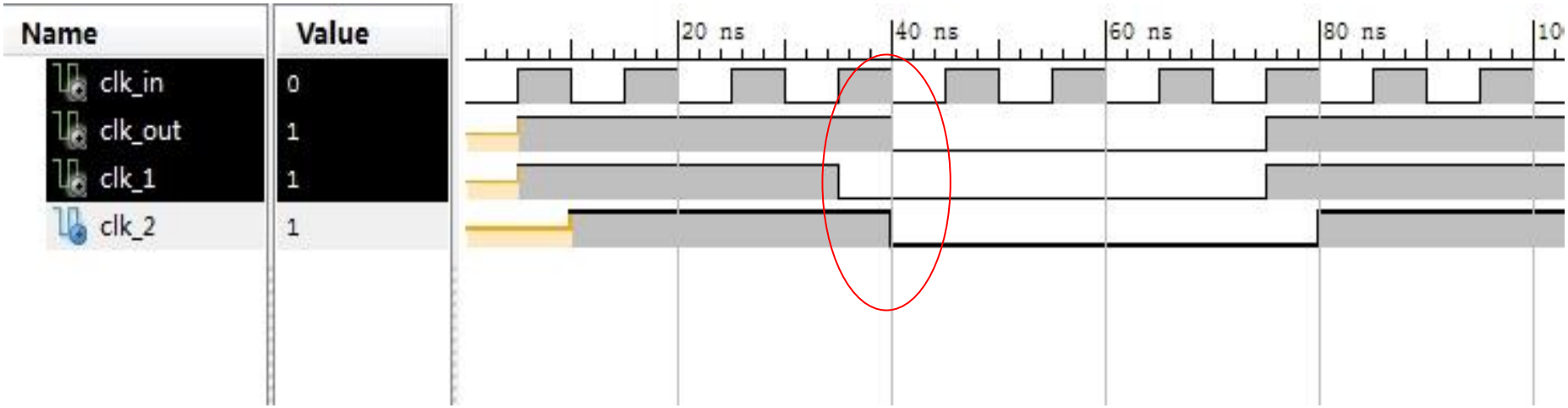
(3)50%占空比的奇数分频

- 时钟下降沿触发计数，产生一个占空比为40%的5分频器，将产生的时钟与上升沿触发的时钟相或，即可产生一个占空比为50%的5分频器。
- 推广为一般方法：要实现占空比为50%的2N+1分频器，则需要对待分频时钟上升沿和下降沿分别进行N/(2N+1)占空比分频，然后将两个分频器得到的时钟信号相或得到占空比为50%的2N+1分频器。

50%占空比的7分频设计

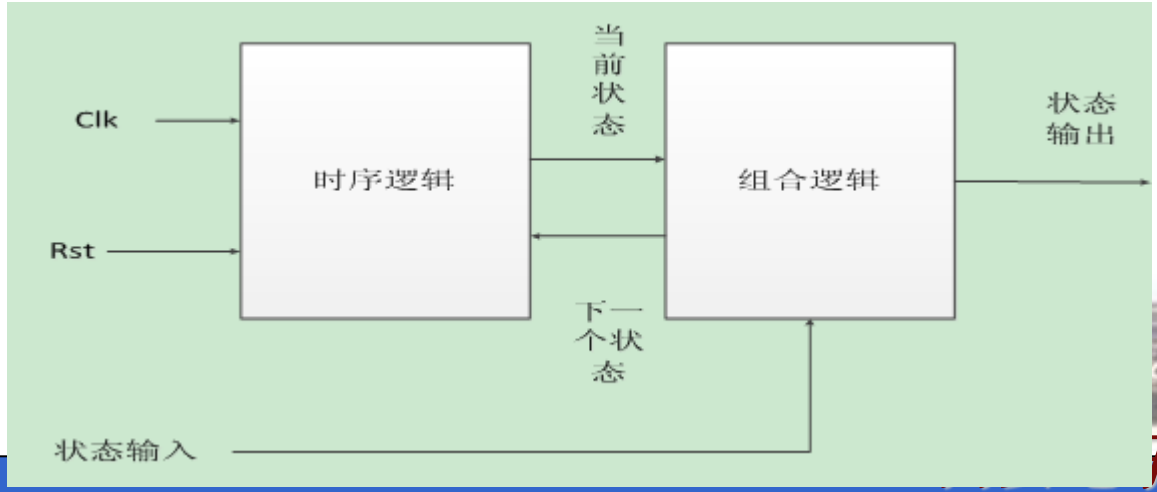
```
entity clk_div3 is
      Port ( clk_in : in    STD_LOGIC;
             clk_out : out   STD_LOGIC);
end clk_div3;
architecture Behavioral of clk_div3 is
signal cnt1,cnt2:integer range 0 to 6;
signal clk_1,clk_2:std_logic;
begin
process(clk_in)
   begin
       if(rising_edge(clk_in))then
            if (cnt1<6)then
                cnt1<=cnt1+1;
            else cnt1<=0;
            end if;
            if(cnt1 < 3)then
                clk_1<='1';
            else
                clk_1<='0';
            end if;
       end if;
end process;
process(clk_in)
   begin
       if(falling_edge(clk_in))then
            if (cnt2<6)then
                cnt2<=cnt2+1;
            else cnt2<=0;
            end if;
            if(cnt2<3)
                then clk_2<='1';
            else
                clk_2<='0';
            end if;
       end if;
end process;
clk_out<=clk_1 or clk_2;
end Behavioral;
```

仿真结果



四、有限状态机设计

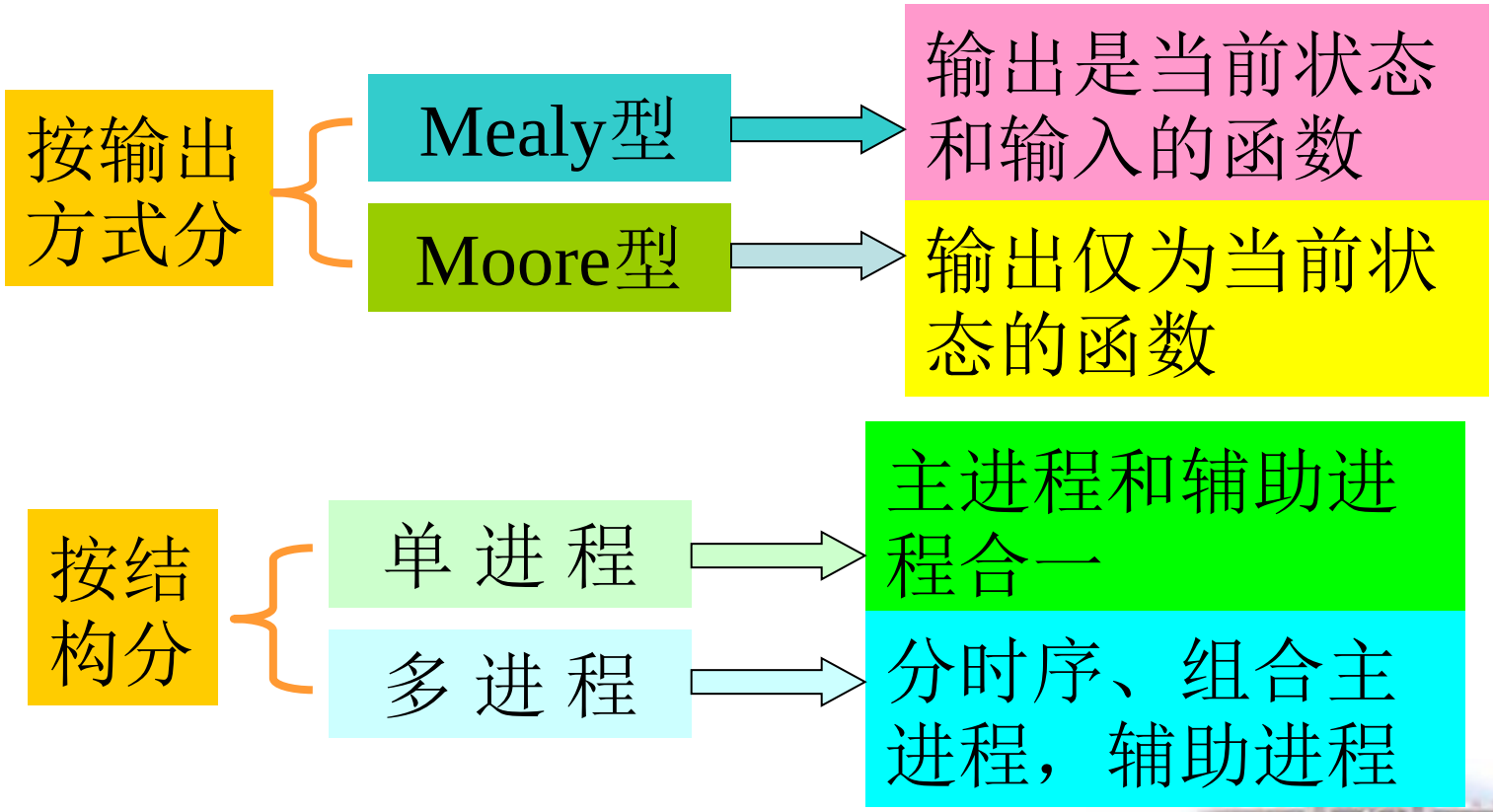
- 有限状态机（finite-state machine, FSM），是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。
- 根据当前状态和输入条件决定状态机的内部条件转换。
- 根据当前状态和输入条件确定产生输出信号序列。



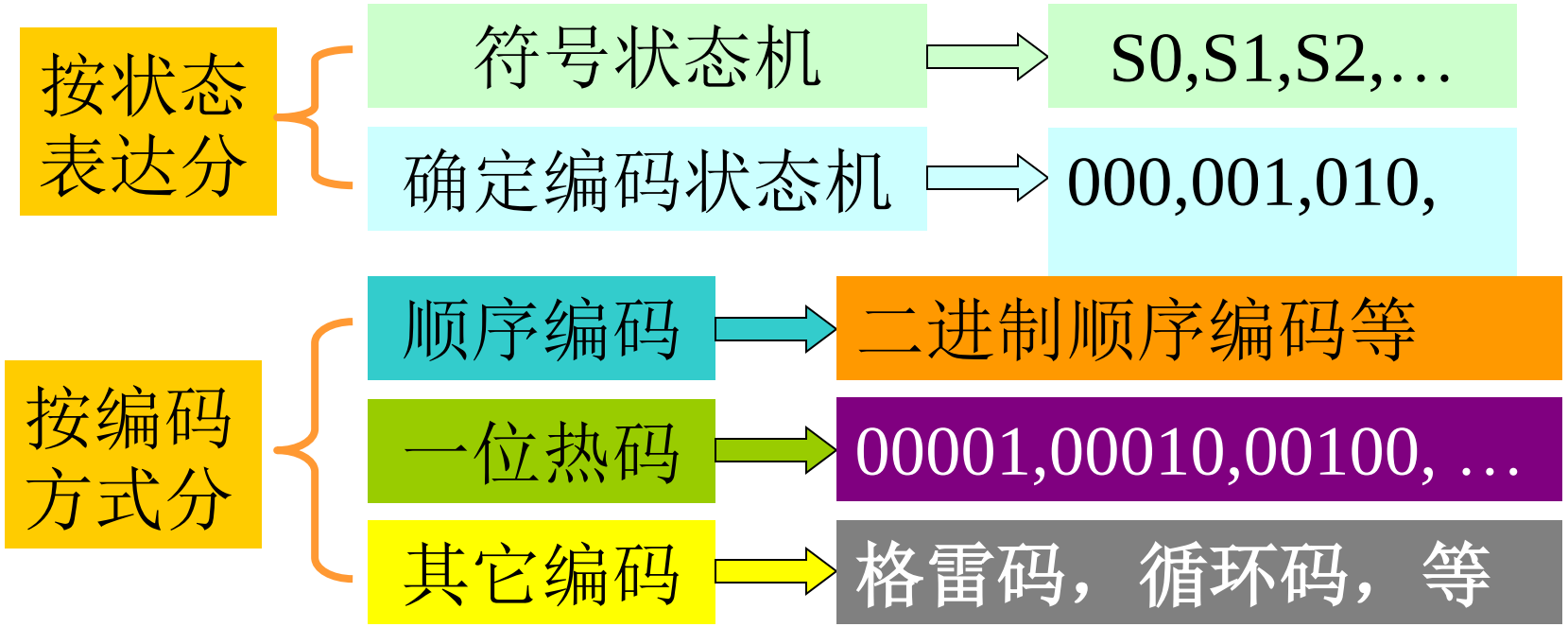
4.1 状态机编码

- 二进制编码：使用N位二进制数，表示M个工作状态，当然必须满足$2^N>M$。
- 独热码：使用N位二进制数，表示N个状态，每一位编码对应一个触发器，状态机中的每个状态都由其中一个触发器的状态来表示。
- 格雷码：使用N位二进制数，表示M个工作状态，满足$2^N>M$，并且相邻两个编码仅有一位不同。
- 约翰逊码：使用N位二进制数，表示2N个工作状态，并且相邻两个编码仅有一位不同。

4.2 有限状态机分类


按输出方式分
Mealy型
Moore型
输出是当前状态和输入的函数
输出仅为当前状态的函数
按结构分
单 进 程
多 进 程
主进程和辅助进程合一
分时序、组合主进程，辅助进程

4.2 有限状态机分类


按状态表达分
符号状态机
S0,S1,S2,…
确定编码状态机
000,001,010,
按编码方式分
顺序编码
二进制顺序编码等
一位热码
00001,00010,00100, …
其它编码
格雷码，循环码，等

4.3 FSM优势

- 设计方案相对固定，结构模式简单，可定义符号化枚举类型的状态。
- 状态机的VHDL描述层次分明，结构清晰，易读易懂。
- 基于有限状态机技术设计的控制器其工作速度大大优于CPU。
- 基于有限状态机技术设计的控制器其可靠性也优于CPU。

4.4 FSM程序结构

- 主控时序逻辑部分：任务是负责状态机运转和在外部时钟驱动下实现内部状态转换的过程。
- 主控组合逻辑部分：任务是根据状态机外部输入的状态控制信号（包括来自外部的和状态机内部的非进程的信号）和当前的状态值。Current_state来确定下一状态next_state的取值内容，以及对外部或对内部其他进程输出控制信号的内容。
- ·辅助逻辑部分：辅助逻辑部分主要是用于配合状态机的主控组合逻辑和主控时序逻辑进行工作，以完善和提高系统的性能。

4.5 FSM程序设计

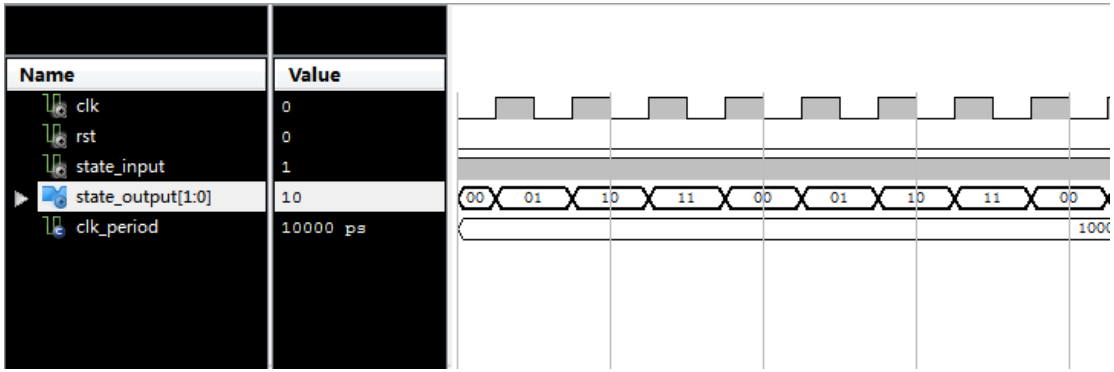
```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY two_process_state_machine IS
    PORT (clk, reset     : IN STD_LOGIC;
          state_inputs   : IN STD_LOGIC;
          comb_outputs   : OUT STD_LOGIC_VECTOR(0 TO 1));
END ENTITY two_process_state_machine;
ARCHITECTURE behv OF two_process_state_machine IS
    TYPE states IS (st0,st1,st2,st3); --定义 states 为枚举型数据类型，构造符号化状态机
    SIGNAL current_state, next_state: states;
BEGIN

REG: PROCESS (reset, clk)      --时序逻辑进程
 BEGIN
    IF reset = '1'   THEN          --异步复位
                current_state <= st0;
    ELSIF clk = '1' AND clk'EVENT THEN--出现时钟上升沿时进行状态转换
                current_state <= next_state;
    END IF;
 END PROCESS;
```

4.5 FSM程序设计

```
COM: PROCESS(current_state, state_inputs) --组合逻辑进程
    BEGIN
    CASE current_state IS
    WHEN st0 => comb_outputs <= "00"; --系统输出及其初始化
        IF state_inputs = '0' THEN --根据外部输入条件决定状态转换方向
            next_state <= st0;
        ELSE next_state <= st1;
        END IF;
    WHEN st1=> comb_outputs <= "01";
        IF state_inputs = '0' THEN      next_state <= st1;
        ELSE next_state <= st2;
        END IF;
    WHEN st2=> comb_outputs <= "10";
        IF state_inputs = '0' THEN      next_state <= st2;
        ELSE next_state <= st3;
        END IF;
    WHEN st3=>comb_outputs <= "11";
        IF state_inputs = '0' THEN      next_state <= st3;
        ELSE next_state <= st0;
        END IF;
    END CASE;
    END PROCESS;
```

FSM仿真结果

- 在复位信号无效时，state_input信号为1时，在每个时钟上升沿实现状态的转换。
- State_input信号用来控制是转为下一个状态还是维持当前状态。

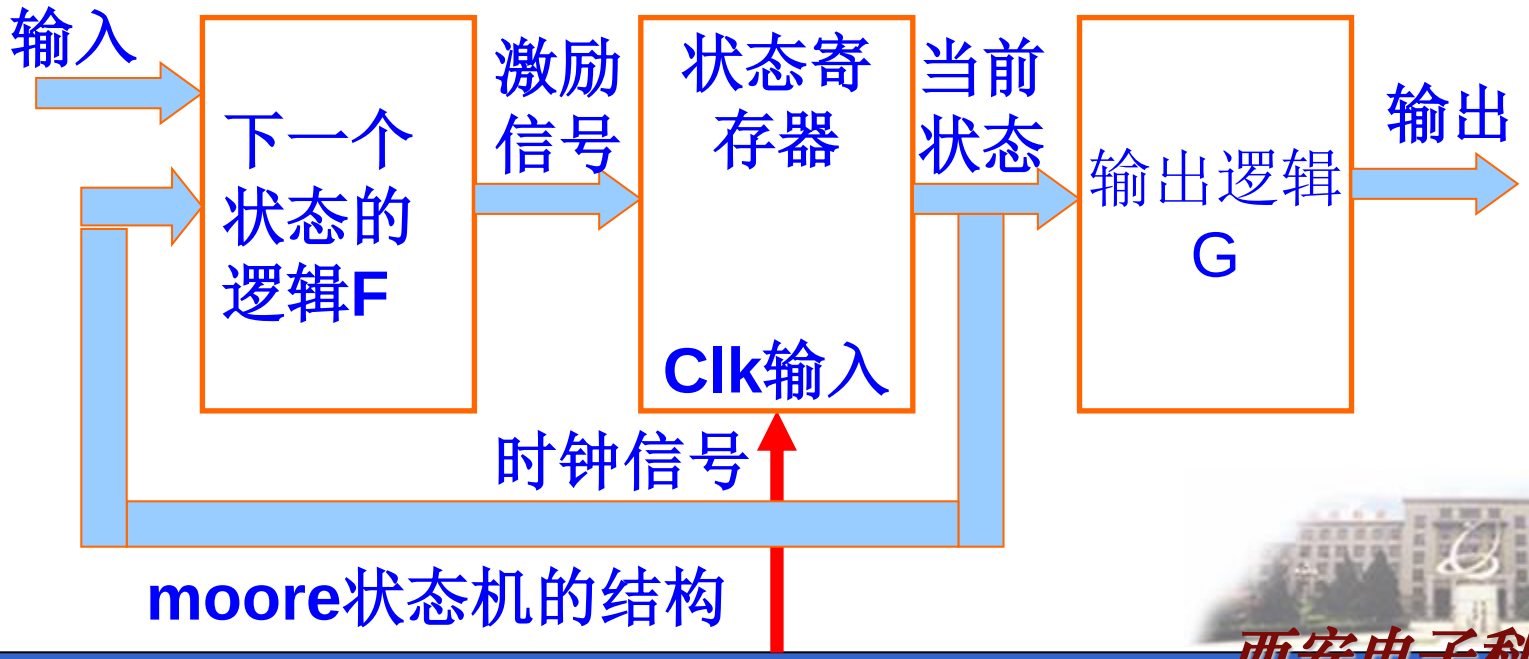


4.6 Moore状态机

- 摩尔型状态机输出只与当前状态有关，而与输入信号的当前值无关，是严格的现态函数。

下一个状态=F（当前状态，输入信号）

输出信号=G（当前状态）



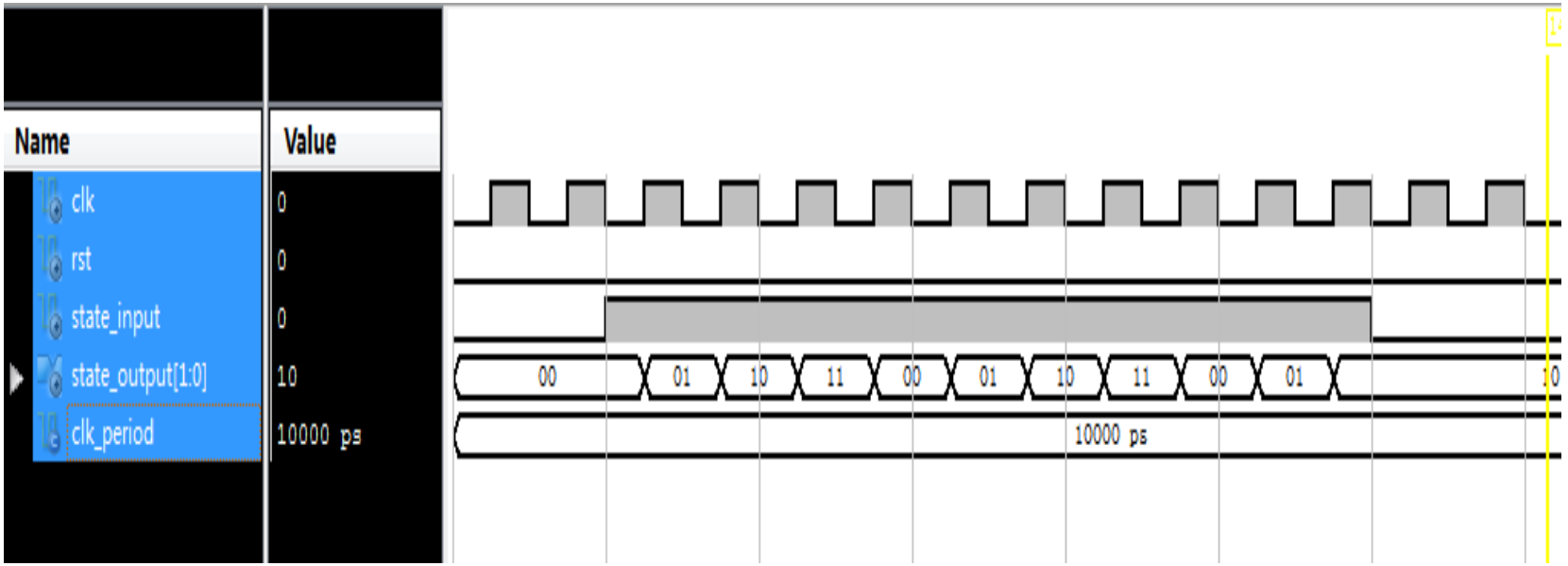
(1)Moore状态机设计

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity more is
        port( clk,rst:in std_logic;
              state_input: in std_logic;
              state_output:out std_logic_vector(1 downto 0));
end more;
architecture Behavioral of more is
type states is(st0,st1,st2,st3);
signal state:states;
begin
process(clk,rst)
   begin
      if(rst='1') then
         state<=st0; elsif(clk'event and clk='1') then
         case state is
         when st0=>
               if state_input='0' then state<=st0;else state<=st1;
               end if;
         when st1=>
               if state_input='0' then state<=st1;else state<=st2;
               end if;
         when st2=>
               if state_input='0' then state<=st2;else state<=st3;
               end if;
         when st3=>
               if state_input='0' then state<=st3;else state<=st0;
               end if;
         end case;
         end if;
      end process;
process(state)
   begin
      case state is
      when st0=>state_output<="00";
      when st1=>state_output<="01";
      when st2=>state_output<="10";
      when st3=>state_output<="11";
      end case;
end process;
end Behavioral;
```

下一个状态

输出信号

(2)Moore状态机仿真结果



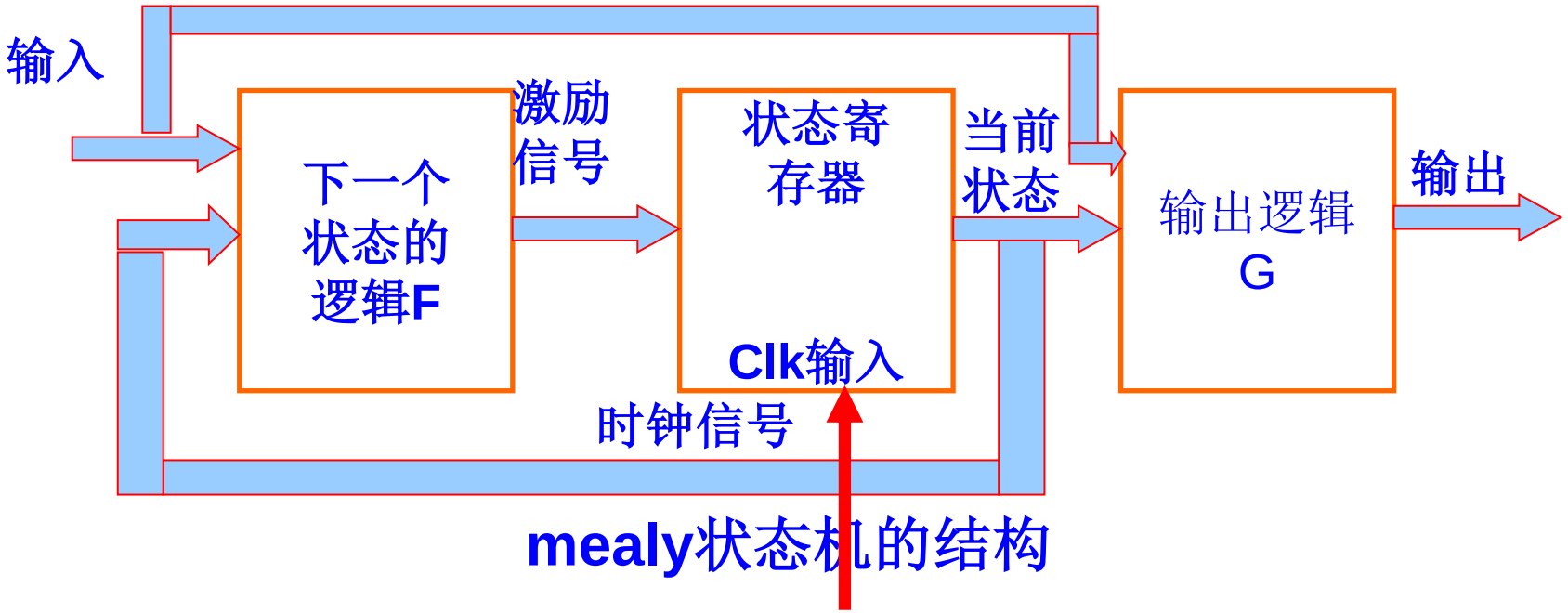
4.7 Mealy状态机设计

- Mealy状态机的输出是现在状态和所有输入的函数，输出随输入变化而随时发生变化，属于异步输出的状态机。
- 输出不依赖系统时钟，也不存在摩尔状态机中输出滞后一个时钟周期来反映输入变化问题。

4.7 Mealy状态机设计

下一个状态=F（当前状态，输入信号）

输出信号=G（当前状态，输入信号）



mealy状态机的结构

(1)Mealy状态机程序设计

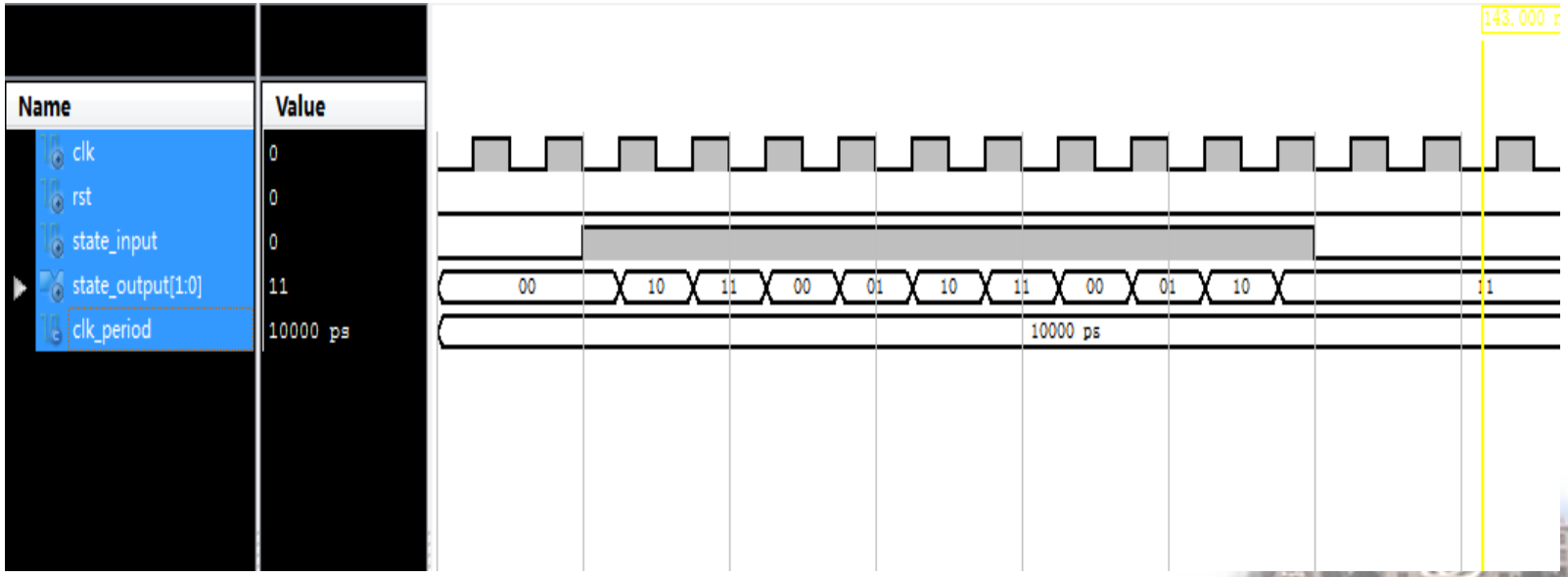
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mealy is
        port( clk,rst:in std_logic;
              state_input: in std_logic;
              state_output:out std_logic_vector(1 downto 0));
end mealy;
architecture Behavioral of mealy is
type states is(st0,st1,st2,st3);
signal state:states;
begin
process(clk,rst)
   begin
      if(rst='1') then
         state<=st0; elsif(clk'event and clk='1') then
         case state is
         when st0=>
               if state_input='0' then state<=st0;else state<=st1;
               end if;
         when st1=>
               if state_input='0' then state<=st1;else state<=st2;
               end if;
         when st2=>
               if state_input='0' then state<=st2;else state<=st3;
               end if;
         when st3=>
               if state_input='0' then state<=st3;else state<=st0;
               end if;
         end case;
         end if;
      end process;
process(state)
   begin
      case state is
      when st0=>if state_input<='0' then state_output<="00";else state_output<="01"; end if;
      when st1=>if state_input<='0' then state_output<="01";else state_output<="10"; end if;
      when st2=>if state_input<='0' then state_output<="10";else state_output<="11";end if;
      when st3=>if state_input<='0' then state_output<="11";else state_output<="00";end if ;
      end case;
end process;
end Behavioral;
```

下一个状态

输出信号

(2)Mealy状态机仿真结果

- state_out从00状态转为了10状态，那是因为当输入为1时，state_out不仅与现态（此时state=01）有关，而且与输入State_in有关，state=1，所以输出为10。



4.8 饮料自动投币售卖机设计

- 设计要求：假设饮料只有一种价格为2.5元。硬币有0.5元和1.0元两种，考虑找零，用VHDL描述其控制电路，并用FPGA实现。

(1)设计步骤

① 分析输入输出端口信号；
② 状态转移图；
③ 根据状态转移图进行VHDL语言描述；
④ 测试代码编写，仿真；
⑤ FPGA实现。

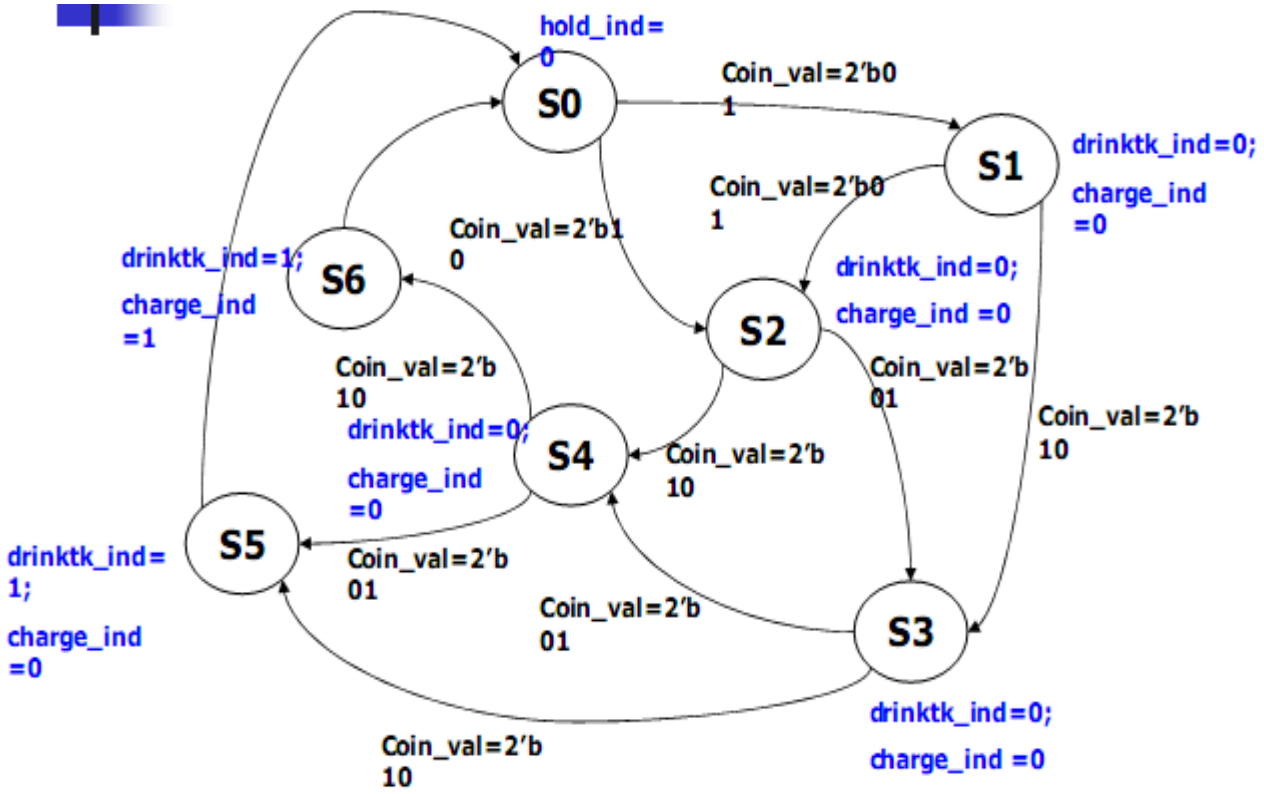
(2)输入输出接口分析

输入信号： clk，rst;
输入信号：操作开始: op_start； //定义1开始操作
输入信号：投币币值: coin_val；//定义2'b01表示0.5元；2'b10表示1元
输入信号：取消操作指示 : cancel_flag; //定义1为取消操作
输出信号：机器是否占用: hold_ind；//定义0为不占用，可以使用
输出信号：取饮料信号: drinktk_ind；//定义1为取饮料
输出信号：找零与退币标志信号: charge_ind; //定义1为找零
输出信号：找零与退币币值： charge_val;
//定义3'b001表示找0.5元； 3'b010表示找1元
// 3'b011表示找1.5元; 3'b100表示找2.0元;

(3)状态确定

S0：初始状态；
S1：已投币0.5元；
S2：已投币1.0元；
S3：已投币1.5元；
S4：已投币2.0元；
S5：已投币2.5元；
S6：已投币3.0元；

(4)状态转移图



状态说明：

S0 ： 初始态；

S1 ： 已投币0.5元；

S2 ： 已投币1.0元；

S3 ： 已投币1.5元；

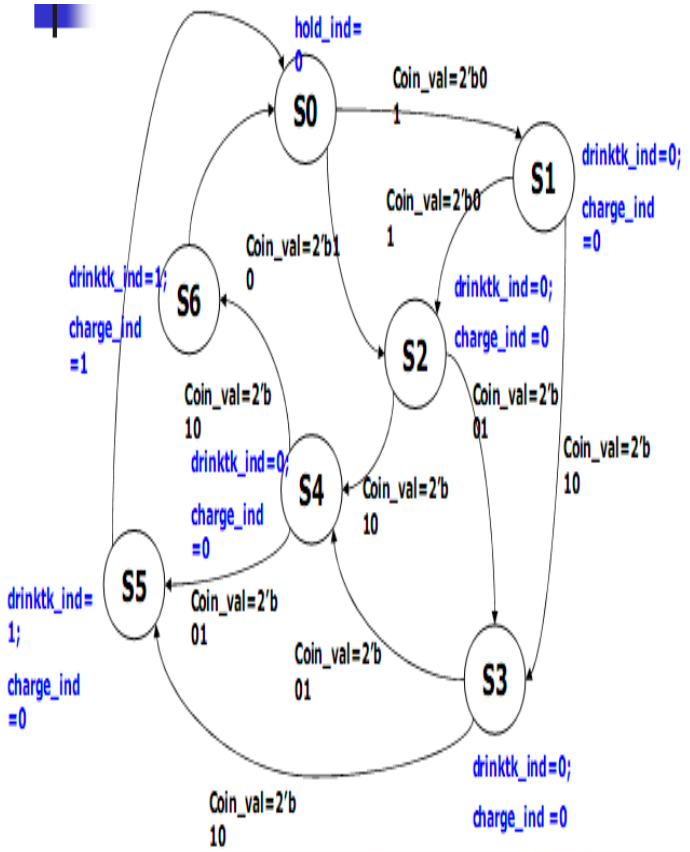
S4： 已投币2.0元；

S5 ： 已投币 2.5元；

S6 ： 已投币3.0元；

(4)状态转移图

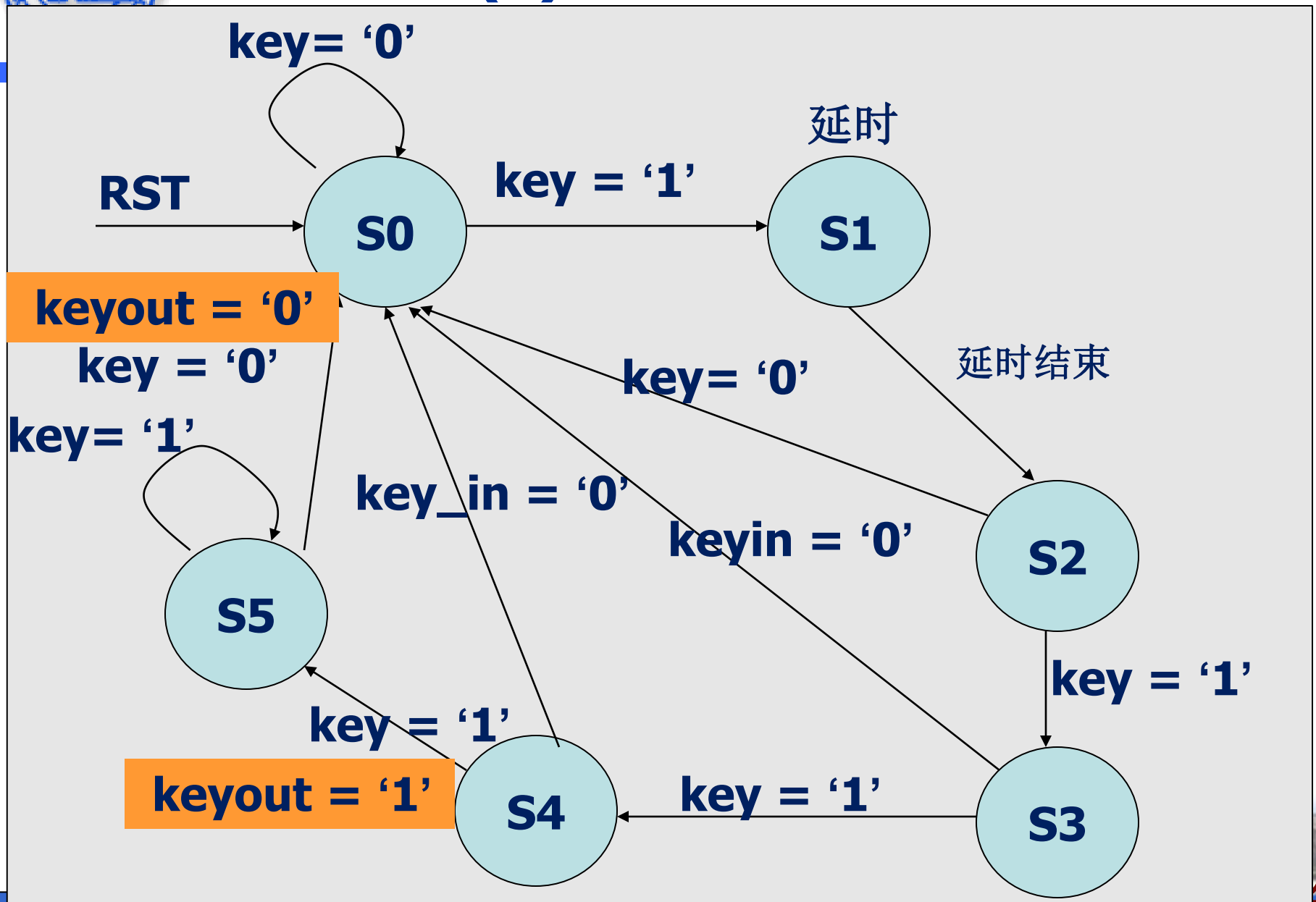
说明：1.在S0 状态下，如果检测到op_start=1，开始检测是否有投币，如果有，一次新的售货操作开始；
2.在状态S1/S2/S3/S4下，如果检测到cancel_flag=1，则取消操作，状态回S0，并退回相应的币值；
3.在状态S5下，卖出饮料不找零；在状态S6下，卖出饮料并找零；
4.在状态S5和S6 操作完后，都返回状态S0，等待下一轮新的操作开始；
5.只有在S0 状态下，hold_ind=0，可以发起新一轮操作，其它状态下都为1。



4.9 按键消抖电路设计

- 按键去抖动电路
  - 按键动作发生时,按键的输出会出现不稳定的逻辑’0’和逻辑’1’的跳变.
  - 该信号直接输入到计数器之类电路,会发生计数错误.

(1)状态机设计思路


key= ‘0’
延时
RST
key = ‘1’
S0
S1
keyout = ‘0’
key = ‘0’
key= ‘0’
延时结束
key= ‘1’
key_in = ‘0’
keyin = ‘0’
S2
S5
key = ‘1’
key = ‘1’
keyout = ‘1’
S4
key = ‘1’
S3

(2)VHDL实现

```
--实体描述
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY jitter_proof IS
        PORT (clk,rst,keyin: IN std_logic;
              keyout: OUT std_logic);
END door_control;

--结构体
ARCHITECTURE behavior OF jitter_proof IS
   TYPE states IS (S0,S1,S2,S3,S4,S5);
   SIGNAL next_state: states;
BEGIN
```

S0状态

S1状态

```
PROCESS (clk,rst,keyin)
Varible count:integer:= 1000; --延时=1000*Tclk
BEGIN
    IF (rst = '1') THEN
        next_state <= s0;
    ELSIF (clk'EVENT AND clk = '1') THEN
        CASE next_state IS
          when s0 =>
                if (keyin = ‘1’) then
                    next_state <= s1;
                else null;
                end if;
          when s1=>
                count := count -1;
                if (count = 0) then
                    count := 1000;
                    next_state <= s2;
                else null;
                end if;
```

S2状态

S3状态

```
when s2=>
    if (keyin='0') then
        next_state <= s0;
    else  next_state <= s3;
    end if;
when s3=>
    if (keyin='0') then
        next_state <= s0;
    else   next_state <= s4;
    end if;
```

S4状态

S5状态

```
        when s4=>
            if (keyin='0') then
                next_state <= s0;
            else
                next_state <= s5;
                keyout <= '1';
            end if;
        when s5=>
            if (keyin='0') then
                keyout <= '0';
                next_state <= s0;
            end if;
    end case;
  end if;
End process;
End behavior;
```